DISTRIBUTED, GARBAGE-COLLECTED, PERSISTENT, VIRTUAL ADDRESS SPACES

Alvaro E. Campos

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

June 1993

A la memoria de mi padre,
a mi madre, y
a las tres Marías que iluminan mi vida.

Acknowledgements

I am greatly indebted to my advisor, David Hanson. Meeting him has been a genuine privilege; working with him has been an enlightening experience. Doubtless, Dave taught me some computer science and engineering, but it is his example as a person, teacher, and researcher that constitutes his real legacy. There are only a few of them left.

I would also like to thank the readers of this dissertation, Andrew W. Appel and Suresh Jagannathan, who could adjust to a tight schedule. Their patient perusal of the draft and insightful comments have certainly improved the presentation.

Finally, I wish to thank all people in the Department of Computer Science at Princeton. Faculty, graduate students, and staff make this department a special place to be. I have been honored with many friendships in these years, and I am saddened to leave good friends behind.

This work has been supported in part by the National Science Foundation under Grants CCR86-19811 and CCR92-00790.

Alvaro E. Campos

Abstract

Most integrated programming environments were designed for uniprocessor systems, and they require substantial processing power and storage capacity. Despite the advantages they offer — flexibility, rapid response — their appetites have limited their use. The recent increases in work-station and network computing power provide an opportunity to overcome these limitations.

This dissertation describes the design of a distributed, language-based, integrated environment and its implementation on current hardware. A distributed system allows users, who can access the system from different locations, to share values and computing power. Shared virtual-memory techniques are used to distribute the address space of *EZ*, a persistent, very high-level, string-processing programming and command language. Unlike other persistent systems, persistence pervades *EZ* and applies to both data and active objects. Distributed *EZ* runs on a loosely coupled multiprocessor, implemented as several work-stations connected by a network.

A distributed virtual-memory manager provides a shared-memory programming paradigm even though there is no shared physical memory accessible by all work-stations. The virtual address space is distributed over the secondary storage devices of the individual processors. Managers cache pages into the physical memory of the work-stations that access them and maintain coherence among the multiple copies of a page. They use replication to permit multiple readers, but permit only one writer.

A distributed, mark-and-sweep garbage collector, which works in concert with the memory manager, reclaims inaccessible objects in the distributed system. This collector is concurrent and real-time. The memory manager collaborates with the collector to avoid direct mutator assistance.

The results show that it is feasible to build an effective distributed system by using shared virtual memory to distribute the persistent address space. Performance remains a problem, due mainly to the high cost of network communication. The implementation of *EZ*'s data structures conflicts with the techniques used to maintain cache coherence and cause too much interprocessor communication. The distributed, mark-and-sweep garbage collector works well, however, and it is especially effective when the system activity is distributed among several processors.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Programming environments can be divided roughly into two categories: integrated environments for very high-level languages like LISP (Teitelman 1985, Teitelman and Masinter 1981) and Smalltalk (Goldberg, Robson, and Ingalls 1983), and environments that support more traditional programming languages like Ada, C, and FORTRAN.

The former typically offer more flexibility, superior debugging tools, and a close integration of the language, the environmental tools, and the surrounding operating system. Advocates of these kinds of environments often cite these features as their strong points, and users of these systems credit these attributes for their productivity gains. These advantages, however, come at a price: most integrated environments for very high-level languages have substantial computational requirements and can consume large amounts of memory and processor cycles, which historically may have limited their dissemination and widespread use.

The recent, rather dramatic, increases in work-station computing power provide an opportunity to overcome these past limitations. Commercial availability of integrated systems like Smalltalk is an example of the benefits that can be reaped from hardware improvements. Work-station hardware has advanced significantly during the last decade, but most work-station software has not. There are exceptions, of course; network software, user-interface software, and systems like Smalltalk are counterexamples. Nonetheless, many work-stations still run traditional systems that are little changed from those available a decade ago. They use the increased computational power to run traditional programs faster, and to run more of them.

However, this new processing power can also be applied to implementing systems that provide a richer, higher-level, computing environment. In a sense, work-station hardware is finally up to the demands imposed by integrated environments. It is thus possible to consider efficient implementations of such environments and their related programming languages. Of particular interest is the design and implementation of *persistent*, very high-level, language-based, integrated environments.

1.1 Distributed Environments

Most integrated environments were designed for uniprocessor systems. Distributed environments can make effective use of loosely coupled multiprocessor systems. In these systems, a network provides communication among several processors, which do not have access to a common physical memory. Distribution allows multiple users to access the system simultaneously regardless of their physical location, and to share the computing power available from all processors in the system. More importantly, it allows users to share data.

It is possible to exploit distribution by providing users with explicit mechanisms to communicate with different work-stations. Message passing and remote procedure calls are examples. A disadvantage of this approach is that users must be aware of distribution and must perform explicit actions to share data. An advantage is that these facilities are efficient and easy to provide because they usually reflect the underlying communication mechanisms provided by the networks.

Another approach is to provide transparent access to multiple address spaces that reside anywhere in the network. Interaddress-space pointers may be used to provide access to remote data; capability-like access rights can be added to restrict access. An advantage of this approach is that multiple address spaces can be encapsulated in the system without users being aware of distribution. But interaddress-space pointers require special treatment, for example, to be dereferenced.

This dissertation explores a third approach to distribute a persistent, very high-level, integrated environment in current hardware. Sharing is achieved by transparently distributing the entire address space of the system. The address space of the programming and command language *EZ* (Fraser and Hanson 1985) has been distributed over a network of work-stations. This single address space allows users to share values under the simple, shared-memory programming paradigm although there is no shared physical memory; the underlying system uses message passing to provide this abstraction.

EZ was chosen because it is an experimental language, and its definition and original implementation are available and can be modified. Although the distributed system has been developed specifically for *EZ*, the techniques and methods used in its design and implementation, and even large parts of the actual implementation itself, could be used for other systems. For instance, other than the interface presented by the memory manager, there is little in the virtual-memory system that is specific to *EZ*.

1.2 *EZ*

EZ is a programming environment (Fraser and Hanson 1983, Fraser and Hanson 1984, Fraser and Hanson 1985, Hanson and Kobayashi 1990) that most closely resembles those for APL (Iverson 1962), LISP, Smalltalk, and related systems. It integrates the facilities provided separately by traditional programming languages and operating systems. This integration is accomplished by casting services traditionally provided by operating systems as language features. The result is a complete computing environment that is intended to be an alternative to both conventional languages and operating systems and that also supports manipulating the environment.

From a language perspective, *EZ* is a high-level string-processing language that has its roots in Icon (Griswold and Griswold 1990) and its predecessors. *EZ* has late binding times and considerable run-time flexibility. For instance, variables can be assigned values of any type, automatic type conversions are performed for most operations, and structures are heterogeneous.

An important feature of *EZ* is the persistence of its address space; objects' values exist indefinitely, or until they are changed explicitly. Other systems also offer persistence or add persistent data types to traditional languages and environments (Atkinson and Morrison 1985, Low 1988, Vegdahl 1986). In most of these systems, however, persistence is a separate facility that is usually restricted only to data.

With persistence, strings and associative tables subsume traditional file and directory services (Fraser and Hanson 1983). Tables are also used for procedure activations permitting the traditionally separate editing and debugging tasks to be provided by a single program that edits *EZ* values (Fraser and Hanson 1985). Lightweight processes (Cooper and Draves 1988, Cormack 1988) in *EZ* and their synchronization mechanisms are also encapsulated using tables (Hanson and Kobayashi 1990). Implementing *EZ* requires specialized techniques because its semantics are very different from traditional languages and operating systems, and because it encapsulates as language features some primitive operations that usually appear as system calls in traditional systems. This encapsulation imposes constraints on the implementation that are similar to those found in operating systems.

Persistence requires a semi-permanent storage medium where values can be stored; values cannot disappear each time the system is turned off, or when there is a power failure. *EZ*'s address space is thus kept in secondary storage. A virtual-memory manager caches pages from the virtual address space into physical memory. The size of *EZ*'s virtual address space is another argument in favor of keeping it in secondary storage; its four gigabytes are too large to fit in the memory of most work-stations.

The system, however, makes no distinction between physical memory and secondary storage; values simply reside in the virtual address space. The language provides no mechanisms to force values to be written to disk or that would allow users to distinguish data that is on disk from data that, momentarily, resides only in physical memory.

1.3 The Distributed System

Distributed *EZ* adapts the recent implementations of shared virtual memory (Li 1986, Li and Hudak 1986, Li and Hudak 1989) to distribute *EZ*'s virtual address space across a network of work-stations. Communication among processors is based on reliable, order-preserving datagrams provided by the network. Campos and Hanson (1992) provide a preliminary report of the design and implementation of distributed *EZ*.

In this design, identical copies of a distributed memory manager run at each processor, and the entire virtual address space is distributed among their secondary storage devices. These software modules hide all distribution details from the rest of the system and from the users' programs, which need no change at all; both the rest of the system and the users are unaware of distribution. Unless otherwise explicitly stated, all references to the virtual-memory system and managers throughout this dissertation refer to these modules, not to the virtual-memory hardware on the processors.

The memory managers attempt to collect recently accessed pages at the processor that accesses them; each manager caches the pages that its clients are using. Managers replicate pages to allow multiple readers, but permit only one writer in order to maintain *coherence*. *EZ* is an interpreted system, and the interpreters are the only and trusted clients of the managers. Since managers yield control of pages to the interpreters, the managers cannot use invalidation (Li 1986) to maintain coherence; *confirmation* is used instead as explained in Section 4.2.

A new virtual machine has been designed and a deadlock-free interpreter has been implemented. The new virtual machine ensures atomicity of the operations and consistency of the data. Some virtual-machine instructions have been recast in terms of only three primitive operations on associative tables; these operations are atomic with respect to the tables on which they operate, but can be interrupted by other unrelated operations. The architecture of the new machine also ensures that all relevant state information is stored in the persistent virtual address space.

The original — uniprocessor — *EZ* system used an off-line garbage collector to reclaim inaccessible pages in the disk representation of the virtual address space.

This approach was fine for a prototype and, perhaps, adequate for a non-distributed, single-user, system that is subject to frequent idle periods; but an off-line collector is unsuitable for distributed, multiple-user, persistent systems.

A new, distributed, mark-and-sweep garbage collector, which works in concert with the distributed virtual-memory manager, has been implemented. This collector is concurrent and real-time, and uses the virtual-memory system instead of direct mutator assistance. For systems with a large, persistent address space, like *EZ*, efficiency is less important than minimal disruption of the mutators. Free virtual-memory pages have to be collected only fast enough to avoid long waits when the mutators allocate new pages.

This dissertation shows that shared virtual memory can be used to distribute the address space of a very high-level, garbage-collected, persistent language. Performance of the distributed system, however, remains a problem and needs to be improved. The high cost of interprocessor communication is the main problem; some implementation decisions need to be re-considered because of this high cost. The collector, however, works well with the rest of the system and interferes little with the mutators.

1.4 Organization

The *EZ* system is described in Chapter 2. This chapter is not intended to be a complete definition of the system, which, together with *EZ*'s syntax and semantics, has been described in detail elsewhere (Fraser and Hanson 1983, Fraser and Hanson 1984, Fraser and Hanson 1985, Hanson and Kobayashi 1990). The description in this chapter focuses on those features in the system that influence the implementation and design choices made in the distributed system.

Chapter 3 describes the new virtual machine, its architecture, and its instructions. It also explains the conventions that the *EZ* compiler follows in its use of the virtual machine. Special attention is given to the call and return instruction sequences and to the implementation mechanisms used to cast procedure activations as associative tables.

The distribution of the address space and, in particular, the distributed virtual-memory manager are described in Chapter 4. This chapter explains the mechanisms used to maintain coherence, and the communication between managers that takes place for this purpose. The implementation and uses of the page table and the page-replacement mechanisms are also explained.

Chapter 5 describes the distributed garbage-collection algorithm and its implementation. This chapter also describes the interaction between the collector and the memory manager and how mutators rely on the virtual-memory system to avoid direct collaboration.

Chapter 6 evaluates the performance of the system. It shows results obtained when running both sequential and parallel test programs on the prototype implementation. Results in a uniprocessor version are used as a basis for comparison. The measurements include the running time of the programs and some statistics that highlight uses of different parts of the system.

Finally, Chapter 7 draws some conclusions from the evaluation and discusses some alternative design and implementation decisions. Possible extensions to this work are also discussed.

Chapter 2

The EZ System

EZ is a programming and command language that unifies both notions by using the concepts of very high-level string-processing languages, such as SNOBOL4 (Griswold, Poage, and Polonsky 1971) and Icon (Griswold and Griswold 1990). *EZ* is designed to provide high-level language features that subsume facilities traditionally provided by operating systems. This goal is accomplished by using late binding times, execution-time scope rules, a type system in which conventionally distinct types are treated as single types, and a persistent address space in which all objects reside.

Instead of embedding a command language in a programming language, or vice versa, *EZ* is designed as a *unified* language. For the most part, the control abstractions in *EZ* are traditional; but some aspects of Icon's goal-directed expression evaluation (Griswold, Hanson, and Korb 1981) and coroutines (Hanson and Griswold 1978) are part of the language as well. The data abstractions in *EZ* are particularly simple and bridge the gap between the data abstractions of command languages and those of programming languages. This link is accomplished by moving the differences between some classes of types out of the language and into the implementation. For example, traditional strings and disk-based text files are identical at the source-language level; the use of secondary storage is hidden within the implementation, as it is in Poplar (Morris, Schmidt, and Wadler 1980).

The *EZ* language resides in a persistent environment, much like an APL (Iverson 1962) workspace, in which values exist until changed or until they are not longer accessible. *EZ* provides a garbage collector that reclaims inaccessible objects without requiring user intervention. Unlike systems that access workspaces explicitly (Atkinson *et al.* 1983), the persistence in *EZ* is implicit and a part of the language, much as in the Gem system (Irons 1976). This characteristic distinguishes *EZ* from various LISP environments, which access traditional operating systems services through functions or other explicit mechanisms (Sandewall 1978).

2.1 Programming and Command Languages

Traditional programming languages and command languages are fundamentally different. Their differences are induced by the differences in their intended application areas.

Programming and command languages differ most in the data abstractions they must support and in their principal binding times. Typical, Pascal-like, programming languages support data types that are close to those provided by most computers such as integers, reals, characters and arrays. Command languages, on the other hand, tend to support 'high-level' data types that bear little resemblance to the data types provided by most architectures. Examples are strings, files, directories, programs and processes.

Traditional programming languages tend toward early binding times. They insist on compile-time definition of many aspects of programs, such as the size and type of variables, primarily for efficiency reasons. Command languages tend toward late binding times, not only because efficiency is less important, but also because the kind of operations performed with them demand more flexibility. Indeed, compilation is delayed until execution in most command languages. Strings and lists of variable length, automatic type conversion, procedures with a variable number of arguments, and dynamic procedure definition are examples of command languages features that require late binding times.

The control abstractions of programming and command languages have superficial similarities, and the structured control constructs of programming languages appear in some command languages such as the UNIX shells (Bourne 1978, Joy 1980). Since control abstractions and data abstractions interact closely, command languages tend to offer more flexible control structures. Examples are loops that iterate over lists of items and 'generative' or pattern-matching constructs, such as file name expansion.

Unifying programming and command languages is an idea that has been examined from a number of viewpoints (Jones 1977, Levine 1980, Morris, Schmidt, and Wadler 1980, Sandewall 1978). It would make learning multiple languages unnecessary for some users, remove the distinction between 'real programs' and 'command programs,' and simplify the use and implementation of programming environments. Attempts at such unification have used LISP and traditional languages such as Pascal.

When traditional languages, such as Pascal, have been used as command languages (Brinch Hansen 1976), they have seldom offered more than procedure calls in the traditional language. Other aspects of the programming language, such as data types, appear in restricted forms or are omitted. This shortcoming can be attributed to the differences mentioned above; traditional languages lack the flexibility and data abstractions necessary for command languages.

Attempts at using high-level languages as command languages have been more encouraging. LISP and programming environments built around LISP are the prime examples (Ellis 1980, Sandewall 1978, Teitelbaum 1975). Much of this success is because LISP has many of the concepts and facilities needed in a command language (Levine 1980).

2.2 Persistence

EZ integrates programming languages and operating systems by encapsulating system services as features in a very high-level language, and by providing a large, persistent virtual address space. Everything in *EZ*, including program code, resides in a single, persistent, virtual address space, in which virtual addresses are 32 bits.

A persistent system that integrates language and operating system concepts 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 string values bound to variables. Once this 'traditional barrier' is breached, it is natural to view all objects — records, arrays, procedures — as persistent values. This view leads to a single language that supports both programming and manipulating the environment.

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 (Gelernter, Jagannathan, and London 1987, Low 1988, Paepcke 1988, Vegdahl 1986, Wile, Goldman, and Allard 1987).

Persistent data types and procedures have been added to some traditional, Pascal-like languages (Atkinson and Morrison 1985). Some other languages have mechanisms for transmitting arbitrary values between programs and, hence, they provide a form of long-term storage (Herlihy and Liskov 1982). Also, most object-oriented languages have built-in facilities for saving some objects on disk for later retrieval (Meyer 1992). Persistence is also an important aspect of some programming environments for traditional languages (Wileden *et al.* 1989), and for maintaining programming environment databases (Wiebe 1986).

The key differences between these systems and *EZ* are that most of them offer persistence as a separate facility, and that they usually restrict it to data. On the other hand, persistence pervades *EZ* and applies to both data and active objects, such as procedure activations and processes.

2.3 The *EZ* Programming Language

As a programming language, *EZ* is a high-level, string-processing language derived from SNOBOL4 (Griswold, Poage, and Polonsky 1971), SL5 (Hanson and Griswold 1978) and Icon (Griswold and Griswold 1990). It has most of the basic attributes of those languages, such as concise, expressive constructs, run-time flexibility, untyped variables and typed values, heterogeneous structures, and automatic type conversion.

Many features of *EZ* are similar to those in traditional languages, and the string-processing features are similar to those in Icon. Strings are treated as scalars, and there are numerous 'middle-level' string operations similar to those in Icon, but neither pattern matching operations nor any form of backtracking is provided. With a few exceptions, the control structures in *EZ* are identical to those found in traditional languages, such as 'while' and 'for' loops, 'if' statements, and procedure calls and returns.

EZ is an interpreted language and its flexibility comes at a cost. The current implementation is quite slow in absolute terms. For example, a program that solves the n-queens problem (see Section 6.2) takes 27 seconds to compile and run. As a comparison, an equivalent Icon program runs in 4 seconds on the same processor and a C version requires 0.7 seconds.

This section briefly describes the *EZ* programming language; it is described in varying detail elsewhere (Fraser and Hanson 1983, Fraser and Hanson 1985), and the reference manual (Fraser and Hanson 1984) contains a complete description of its syntax, semantics and built-in values.

2.3.1 Lexical and Syntactic Structure

Syntactically, the *EZ* language is similar to C (Kernighan and Ritchie 1988), with a few additional control constructs. Blanks and tabs are ignored in general, but are required to separate some lexical elements such as identifiers and reserved words, as in most languages. Comments are introduced by the sharp character (`#`), which causes the rest of the line in which it appears to be treated as white space.

EZ programs are sequences of statements and procedure declarations. Statements outside procedures are executed immediately and procedure declarations assign procedures to identifiers. Identifiers are either global or local.

Statements include the usual structured control flow constructs and expressions. Most language constructs are terminated by the end of the line on which they appear, much as in RATFOR (Kernighan 1975) and Icon. Within a construct, however, new lines may be used as desired to improve readability where it is clear that the construct is continued on the next line. Technically, when parsing, new line characters are treated as white space except in contexts where they are in the follow set (Aho, Sethi, and Ullman 1986).

EZ has a few reserved words — listed in Table 1 — that introduce language constructs and may not be used for other purposes. *EZ* is case sensitive; upper case

`break`	`end`	`in`	`return`
`continue`	`for`	`local`	`while`
`else`	`if`	`procedure`	

Table 1: Reserved words in *EZ*.

letters differ from the corresponding lower case letters in names. Reserved words appear in lower case.

2.3.2 Data Types

EZ supports fewer data types than SNOBOL4 and Icon because it treats several classes of types as a single type. *EZ* provides only four data types: numeric, string, table and procedure.

Numerics — both integers and reals — serve their conventional purposes. Strings are sequences of characters and have arbitrary length; string constants are enclosed in quote characters (`"`). The 'official' character set of *EZ* is ASCII. Tables are heterogeneous, one-dimensional arrays that can be indexed by arbitrary values — like SNOBOL4 tables — and can hold arbitrary values. Tables pervade the *EZ* system because many system services have been cast as tables. Procedures serve their conventional purposes, but are data objects that can be manipulated at the source level.

The type of a variable varies during execution and is the type of the value it currently holds. Values are converted to appropriate types automatically as necessary for most operations. Failure of any automatic conversion results in a run-time error.

For arithmetic operations, the operands are converted to numeric types if possible. Strings of digits are converted to numeric by interpreting them as numbers written in decimal notation. If the string has a period character (`.`), it is interpreted as a decimal point and the resulting number is a real; otherwise it is an integer. Tables and procedures cannot be converted to numeric values; any such attempt results in a run-time error. Integer arithmetic is performed if all of the resulting operands are integers. Otherwise, the operands are converted to real, if necessary, and real arithmetic is performed. Reals are converted to integers by truncation; integers converted to reals retain their values.

For string operations, operands are converted to strings if necessary. Integers and reals are converted using the appropriate string representation in decimal. Procedures are converted to string by returning the source code presented to the compiler when they were compiled. Tables are converted to string by converting their contents to strings and concatenating the results.

Conversions to associative tables are also provided. String, integer and real values are converted to tables by constructing a new table with the respective value associated with its index 1. Procedures are converted to tables by creating an activation record for the procedure.

For procedure operations, values other than procedures and tables are first converted to string, if necessary, and then the resulting string is compiled as if it was the body of a procedure. In this sense, compilation is just an optimization in *EZ*. Tables are converted to procedure by returning the value associated with the index `"Procedure"` in the table.

2.3.3 Operations and Expressions

The precedence and associativity of *EZ*'s operators is summarized in Table 2 in order of decreasing precedence from negation to assignment. As usual, parentheses may be used to override the built-in precedence and associativity rules. All unary operators are prefix.

The specific action of some operations depends on the types of the operands. The relational operators, for example, perform arithmetic comparison when both operands are numeric. As with other arithmetic operations, integer comparison is performed if both operands are integers; otherwise, real comparison — with the appropriate conversions — is performed. If either operand is not numeric and they have different types, an arbitrary — implementation dependent — but well defined ordering of the *EZ* data types is used to compare them. When both operands are strings, they are compared lexicographically. If both operands are procedures or if both are tables, they are compared by comparing the virtual memory addresses of their contents.

Expressions usually compute values but, as in Icon, some expressions may fail to yield values. For example, the relational operators return the value of their right

Arity	Operators	Associativity
unary	`~`	
unary	`-`	
binary	`* / %`	left-to-right
binary	`+ -`	left-to-right
binary	`||`	left-to-right
binary	`== ~= < <= >= >`	left-to-right
binary	`&`	left-to-right
binary	`|`	left-to-right
binary	`=` *op*`=`	right-to-left

Table 2: *EZ* operators.

operands only if the relation is satisfied. The absence of values is used to drive control structures; it terminates for and while loops and determines the flow of if statements. The predefined name `void` does not have a value initially.

For some operators, such as assignment, the absence of a value inhibits the execution of the operation. For example, in

```
max = max < a
```

no assignment is made to `max` if it is not less than `a`; otherwise the value of `a` is assigned. Similarly, the negation operator "`~`" returns the null string if its argument does not yield a value, and no value otherwise.

The binary `|` and `&` operators denote inclusive OR and AND, respectively. They are 'short-circuit' operators and evaluate their right operands only if the left operand is not enough to determine the outcome of the operation. The OR operator returns its left operand if it yields a value; it returns its right operand otherwise. The AND operator does not return a value if its left operand does not yield a value; it returns its right operand otherwise.

Likewise, variables do not have a value initially, and the lack of a value in a context that requires one, such as for addition, yields a run-time error. This facility is a simplification of the sequence of results that Icon's generators produce (Griswold, Hanson, and Korb 1981).

2.3.4 Substrings

Since objects' values persist until changed, assignment of a string to a variable provides the same facility as creating a file in a traditional operating system (Fraser and Hanson 1983). Substring operations provide 'random access' facilities; `s[i:j]` specifies the substring of string `s` between its character positions `i` and `j`. The positions in a substring specification can be given in either order; `s[i:j]` specifies the same substring as `s[j:i]`.

As in Icon, character positions refer to positions between characters in the string. They are numbered from the left, starting at 1 for the position to the left of the leftmost character, and continuing with positive values towards the right. They may also be specified relative to the right end of a string, starting at 0 for the position to the right of the rightmost character, and continuing with negative values towards the left. For example, `s[1:0]` is all of `s`.

Substrings may also be specified by giving a starting position and a length. The expression `s[i!l]` specifies the substring of length `l` that starts at position `i` and includes `l` characters. Positive lengths indicate `l` characters to the right of the starting position; negative values indicate characters to its left. For example, `s[0!-1]` is the string containing the last character of `s`.

2.3.5 Associative Tables

Tables are associative arrays similar to the tables of SNOBOL4 and Icon, and to Awk's arrays (Aho, Kernighan, and Weinberger 1988). They are created automatically as necessitated by subscripting; but referencing a non-existing table entry does not create the entry. Tables are as large as necessary to accommodate their contents.

Like other values, tables persist until changed and thus subsume directories in traditional operating systems. For example,

```
dissertation["title"]  = "Distributed, Garbage-Collected..."
dissertation["author"] = "Alvaro E. Campos"
```

creates a table representing this dissertation and establishes it as the value of the variable `dissertation`. Table `dissertation` is like a directory, the indices — `title` and `author` — are like file names, and the associated values are the files' contents. Tables can be used to construct a hierarchical 'file system,' such as that provided by UNIX.

The 'dot notation,' *expression.identifier*, usually used to access fields of aggregates, is equivalent to *expression*`["`*identifier*`"]`, permitting *EZ* tables to subsume records in traditional programming languages. The example above is equivalent to

```
dissertation.title  = "Distributed, Garbage-Collected..."
dissertation.author = "Alvaro E. Campos"
```

The indices and their associated values in *EZ* tables can be of arbitrary type. For example,

```
dissertation[2].heading     = "The EZ System"
dissertation[2].top         = dissertation
dissertation[2][3].heading  = "The EZ Programming Language"
dissertation[2][3].top      = dissertation[2]
dissertation[2][3].contents = "As a programming language,..."
```

creates a table for Chapter 2 of this dissertation and establishes it as the value associated with index 2 in `dissertation`. In addition, the value associated with `"top"` in the table for Chapter 2 is the table for the entire dissertation. The code also creates a table for Section 2.3, and establishes it as the value associated with index 3 in the table for Chapter 2. The latter is also the value associated with `"top"` in the table for Section 2.3 of the dissertation. Arbitrary cyclical structures, such as the ones shown in this example, are permitted and are, in fact, frequently used in *EZ*.

Entries are removed from tables by the built-in procedure `remove`, which does not return a value. This procedure accepts an arbitrary number of arguments that are considered as successive pairs. The first element of each pair is a table and the second is an index into that table; `remove` removes the index and its associated value from the table. For example,

```
remove(dissertation[2], "top")
```

removes the entry `"top"` and its associated value in the table for Chapter 2.

The substring operation can be applied to tables as well. For subtables, the 'positions' may be of any type. Given positions `i` and `j`, `t[i:j]` refers to those entries in table `t` with indices greater than or equal to `i` and less than or equal to `j`. The comparisons are made using the relational operators as described in Section 2.3.3. When the subtable operation is used in a context requiring a value, it returns a new table containing the selected entries. The second form of the operation, `t[i!j]`, is equivalent to `t[integer(i):integer(i)+integer(j)]`,where `integer` is a built-in function that returns its argument converted to an integer.

EZ strings, tables, operators, and control structures provide the facilities of files, directories, and file-system utilities found in operating systems. For example, listing the contents of a table corresponds to listing the names in a directory. In most operating systems this service is provided by a separate 'list-directory' program, but it can be provided in *EZ* with a simple loop. For example,

```
for (i in dissertation)
    s = s || " " || i
```

sequences through the table given by `dissertation`, repeatedly assigns the indices of the table to variable `i`, and concatenates the indices onto the end of `s`. Assuming that the string `s` is initially empty, its final value is `" title author..."` after the loop.

The binary operator "`||`" is string concatenation. It creates a new string by appending a copy of the second operand to a copy of the first. As usual, the operands are converted to strings if necessary.

2.3.6 Procedures and Activations

Procedures are *EZ* values that contain executable code. A procedure declaration amounts to an assignment of the procedure 'constant' to the identifier. For example, a list-directory procedure can be declared as follows:

```
procedure ls (t)
    local i, s = ""
    for (i in t)
        s = s || " " || i
    return s[2:0]
end
```

This declaration assigns to `ls` a procedure that returns a string with the list of indices in the table passed as its argument. Procedures can be invoked as usual, for instance,

```
list = ls(dissertation)
```

assigns `"title author..."` to `list`.

Local identifiers — introduced by the reserved word `local` — and parameter names are known only within the procedure in which they are declared. Undeclared identifiers are global and they are subject to the scope rules discussed in Section 2.3.7.

Conversion from a procedure to a table, through the use of the built-in function `table`, yields a table that is an activation record for the procedure. Tables created in this manner can be used as coroutines. An activation contains entries for each of the arguments and locals declared in the procedure, and entries describing the current state of the activation's execution. In addition, activations also have the index `"Procedure"`, which contains the procedure itself.

Arguments and locals in an activation can be accessed by indexing the table. For example, after creating an activation `d` for `ls`, `d.i` and `d.s` access the local variables `i` and `s`, respectively, in that activation. Activations may be invoked like procedures, so both

```
d(dissertation)
```

and

```
d.t = dissertation
d()
```

begin the execution of procedure `ls` with `dissertation` as its argument.

Since activations are just tables, entries can be added, removed or changed as desired. For example, the `"Procedure"` entry need not correspond to the procedure from which the activation was created; it can be changed to any procedure. Missing local variables and parameters are created as necessary during execution.

Treating activations and tables as a single type induces a different programming style that emphasizes activations over procedures. It is typical, for example, for tables to contain *both* data and activation information. For instance, a procedure to format dissertations can be included as the `"Procedure"` entry in the table `dissertation` described above. Then, the invocation `dissertation()` produces the formatted document.

2.3.7 Scope Rules

Scope rules, which dictate the interpretation of free identifiers, depend on the contents of tables interrogated by the compiler. Unlike most other systems, these 'symbol tables' are *EZ* tables that can be manipulated at the source-language level.

Interpretation of free identifiers is sought by searching the table that is the current value of the variable `root` for an index value lexically equal to the identifier. Initially, the entry `"root"` of table `root` contains the table `root` itself. In principle, for example, the assignment

```
sign = "do not enter"
```

is equivalent to

```
root["sign"] = "do not enter"
```

However, if the identifier is not found as an index in table `root`, the compiler searches the chain of tables given by

```
root[".."]
root[".."][".."]
root[".."][".."][".."]
...
```

until either the identifier is found, or the compiler finds a table without a `".."` entry or whose `".."` entry is not a table. If this search fails to locate the identifier, it is entered in the current table `root`.

This interpretation of free identifiers is under complete control of the user. By changing the value of `root` and altering the path given by the `".."` entries, it is possible to obtain different behaviors, such as the inheritance rules in Smalltalk (Goldberg, Robson, and Ingalls 1983), the 'search lists' of UNIX, the information-hiding aspects of modules and own variables, and the protection benefits of separate address spaces.

2.4 Processes

Processes in *EZ* are procedure activations that execute *concurrently* (Hanson and Kobayashi 1990). All processes share the same, persistent virtual address space. They are, thus, semantically similar to 'threads' or 'lightweight processes' in some operating systems (Accetta *et al.* 1986). In *EZ*, there is no analog of 'heavyweight processes' with separate address spaces as in most operating systems; some of the protection benefits of such processes can be obtained by appropriate exploitation of *EZ*'s scope rules. Processes are values. As activations, they are just associative tables and have all of the normal characteristics of other *EZ* values, including persistence; all table operations apply to processes as well.

Processes are manipulated explicitly as *EZ* values and also by a set of specific built-in functions. For instance, the expression

`create(p,` *args*...`)`

instantiates a new process by creating a new locus of control that starts execution by calling procedure `p` with arguments *args*. Specifically, the built-in function `create` creates a new process, passes *args* as its arguments, begins its execution, and returns the process itself. If `p` is a table, i.e., a procedure activation, it is simply instantiated; if `p` is a procedure, `table(p)` — an activation for the procedure — is instantiated. Other values are first converted to procedure and then instantiated as just described. For example, an 'idle' process can be instantiated by executing

```
create(procedure ()
    while (1) pause()
end)
```

The argument to `create` is compiled as an anonymous procedure, an activation for it is created, and the activation is then instantiated.

Processes terminate execution by simply returning, or by explicitly calling the built-in procedure `die`. After execution is finished, the table, i.e., the process, remains and may be used again if it is still accessible. As with any other value in *EZ*, it is accessibility that determines the lifetime of data, but not the lifetime of the locus of control.

Whenever a process is instantiated by a call to `create`, an entry describing the new process is entered into the system 'process table;' these entries are removed when the respective process terminates execution. The process table is just the *EZ* table `process` and it contains one entry per active process.

These entries are indexed by the active processes themselves and they each contain an *EZ* table with relevant information about the respective process. In particular, the entry `"process"` of these tables contains the process itself. The entry `"join"` is a linked list — threaded through the entry `"link"` of the tables — of processes waiting for this process to terminate.

Most operating systems have a 'process status' utility that gives information on the currently running processes; `ps` is such a program in UNIX. Similar facilities are easy to implement in *EZ* by sequencing through the indices of table `process`.

Semaphores are used for low-level process synchronization in *EZ*. Higher-level synchronization mechanisms, such as events and message passing, can be implemented with semaphores. A semaphore is just an *EZ* table with an appropriately initialized `"count"` entry.

The usual operations `P` and `V` on a general semaphore are built-in functions in *EZ*. Processes waiting on a semaphore `s` are linked together via the entry `"link"` mentioned above; this list emanates from the entry `"queue"` of `s`, which is created if necessary. Any table can be used as a semaphore; the entry `"count"` is added with a value of 0, if necessary. Although it is not recommended, semaphore `"count"` values and `"queue"` entries can be changed explicitly as any other value; they are just *EZ* table entries.

Chapter 3

Virtual Machine

The *EZ* compiler produces instructions for a virtual machine implemented by the *EZ* interpreter. Executable code for a procedure is stored as an array of virtual-machine instructions that is accessible from the procedure itself, as is the string that contains the source-language program. The virtual-machine instructions are executed by the interpreter to produce the results specified by the procedure. There is a close reflection of the needs of the *EZ* language on the architecture of the virtual machine and on the level and type of the instructions it provides. The instructions are simple and straightforward to execute, and they are defined in terms of a few primitive operations.

The virtual machine used by the original *EZ* system was unsuitable for a distributed system. For example, some instructions stored pointers to locations in physical memory in interpreter data structures; these pointers were used later by other instructions that, naturally, relied on the pointers being correct. In a uniprocessor system in which only one locus of control accesses the address space, using a central interpreter avoided many potential problems. In a distributed system, with several processors accessing the address space simultaneously, some problems become apparent.

One of the problems is that if pages can move in physical memory, physical pointers may no longer be valid when they are used by later instructions; with several processors, this movement of pages becomes harder to track, especially if pages can migrate among them. Another problem is that there was no guarantee as to the order in which different sequences of instructions access the pages; with several simultaneous accesses, two pages may be needed by two processors and deadlock may result if the processors access and lock the pages in different order. A third problem is that parts of the state of the virtual machine were not kept in the persistent virtual address space, but in data structures private to the interpreter; keeping all state information in the persistent address space makes it easier to recover from failure, and makes it possible to implement a fault-tolerant, distributed system that can continue to work when some processor or some link between processors fails.

In order to ensure atomicity and consistency, a new virtual machine has been defined and a new interpreter written in such a way that it is deadlock free. Of course, it is possible to write *EZ* processes that deadlock, just as it is possible to write programs with threads in Mach (Accetta *et al.* 1986, Cooper and Draves 1988) that deadlock, for example. However, the interpreter itself avoids deadlocks and guarantees that no memory pages are lost or become inaccessible even when *EZ* processes abort. Some virtual-machine instructions have been recast in terms of three primitive operations on associative tables: membership testing or look-up, insertion, and deletion. These operations are atomic with respect to the tables on which they operate, but can be interrupted by other, unrelated operations. Finally, the architecture of the new virtual machine ensures that all relevant state information is stored in the persistent virtual address space.

3.1 Virtual-Machine Architecture

The interpretation of an *EZ* process proceeds by executing virtual-machine instructions one at a time. Each process is interpreted by a C thread (Cormack 1988) that runs its own incarnation of the interpreter.

The interpreter threads cannot be preempted, but they pause after executing a small, random number of instructions (Pike 1990). All threads in the system, including those performing system tasks like garbage collection and memory management, pause after some number of operations. The pauses make it possible to interleave the execution of all threads running on the same processor, while avoiding arbitrary, asynchronous interruptions that could expose inconsistent intermediate states. Asynchronous interrupts would not only affect the execution state of processes, but would also have an impact on garbage collection (Appel 1992).

At any time during the execution of a process, there is a table that is the activation currently being executed by the process' interpreter. This table corresponds to the activation record at the top of the stack during execution of a program in a traditional programming language; it is the activation record for the 'current' procedure.

3.1.1 Frames

An activation is a table implemented using a linked list of as many 'frames' as necessary. Frames are one of the several forms used in the *EZ* system to implement tables. In particular, this form is specifically designed to hold activation records of procedures.

As shown in Figure 1, a frame is a virtual-memory page that has four fields. The field `type` is a small integer indicating the page's data type; for these pages, this tag indicates that they contain a frame. The field `type` is present in every virtual-memory pages; it allows the system to distinguish among all the different page uses and layouts.

The `next` field threads the frames in cases when more than one is needed to hold the activation. For example, Figure 1 shows a linked list of three frames for an activation. The pointers linking the frames are stored as the virtual address of the next frame in the list. As usual, to mark the end of the list, the last frame for an activation has a 0 in its `next` field.

The field `v` is an array of *EZ* values that contains information relevant to the execution of the activation. As suggested in Figure 1, the `v` arrays of all frames in the linked list for an activation are considered a single vector. The virtual machine indexes this vector with integers that increase consecutively within one frame's `v` array. For each frame, the indices start with the integer that follows the index for the last entry in the `v` array of the previous frame in the list. The first entry, number 0, for the activation's `v` vector is the first entry in the `v` array of its first frame.

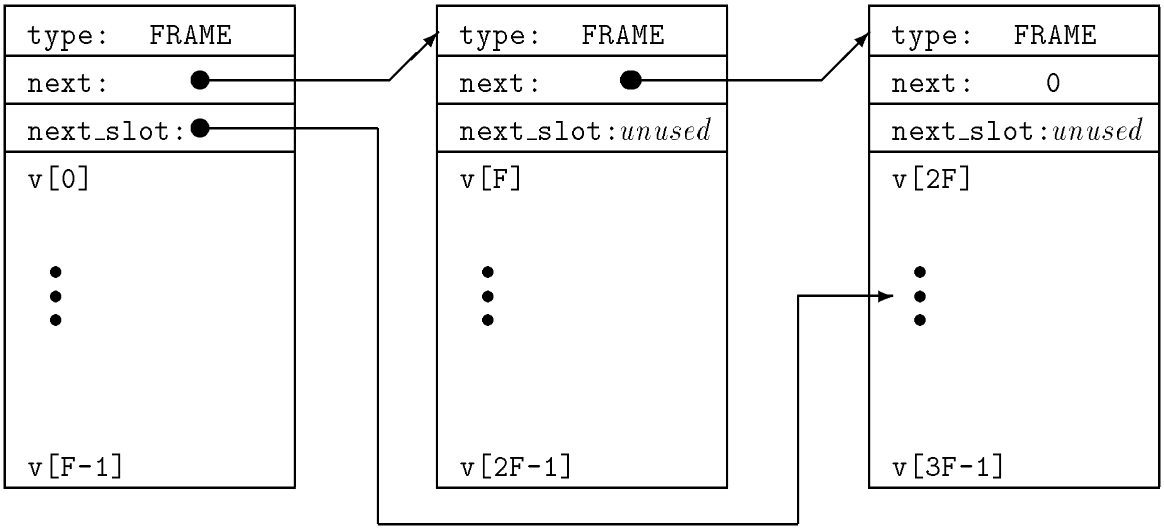

Figure 1: Frame list for an activation.

An activation's `v` vector reflects parts of the stack contents in the run-time system of a traditional programming language. It contains some control information for execution of the activation, and the arguments and locals of the procedure for which this table is the activation. The rest of the vector is used for temporary values needed for local computations, as the stack might be used in a traditional programming language. Arguments, locals, and temporaries are assigned by the compiler and are used explicitly by the instructions it generates. Figure 2 shows the layout of an activation's initial `v` vector.

Frames are allocated at run-time as they are needed; initially, when an activation is created, only one frame page is allocated. The linked list grows as it becomes necessary during run-time, when accessing an entry of vector `v` beyond the entries provided by the current list of frames. The initial size needed for vector `v` and, therefore, the number of frames originally needed by an activation for a procedure are known when the procedure is compiled. However, activations are just *EZ* tables; as such, new elements may be inserted into them during run-time. Insertions may require appending frames to the list.

In an activation's first frame, the field `next_slot` holds an integer specifying the first unassigned entry in the activation's `v` vector; in subsequent frames, this field is

`[0]:`	program counter
`[1]:`	mapping table
`[2]:`	access link
`[3]:`	procedure
`[4]:`	dynamic link
`[5]:`	return value

`[6]`
• • •
`[5+n]`
	`n` arguments

`[6+n]`
• • •
`[5+n+m]`
	`m` local objects

`[6+n+m]` • • •	temporaries

Figure 2: Layout of initial vector `v` for an activation.

not used. Initially, the `next_slot` field is set to the first entry after the last temporary required by the generated code. This field is used to assign unused entries of `v` to hold the new elements inserted into the activation.

3.1.2 Control of Execution

The first four entries in vector `v` are used by the virtual machine for execution-control purposes. As shown in Figure 2, they contain the program counter, mapping table, access link, and procedure. The predefined uses of these entries are assumed and enforced by the interpreter itself; they must be followed by any instruction sequence generated for the virtual machine, since these entries are 'hidden' arguments for several virtual-machine instructions. The compiler also follows this convention.

The program counter, `v[0]`, is the virtual address of the next instruction to be executed. Its value is incremented automatically by the virtual machine after decoding an instruction and before its execution. The magnitude of the increment depends on the number of arguments used by the current instruction; its purpose is to make the program counter refer to the next instruction. As explained in Section 3.2.2, execution of the instruction might change the value stored in the program counter.

The next entry, `v[1]`, contains the mapping table for the activation; another table that allows the system to use the activation as an *EZ* table. The mapping table maps the *EZ* values that constitute the indices stored in the activation to integer *EZ* values. These integer values indicate the entries in the activation's `v` vector where the values associated with the respective indices are stored. In particular, the mapping table maps the names of parameters and locals to their positions in vector `v`. The mapping table is accessible only by the implementation of the virtual machine through the instructions generated by the compiler; it cannot be manipulated at the source-language level.

Figure 3 shows the initial mapping table and parts of vector `v` in an activation for procedure `ls`, which was defined in Section 2.3.6. In this activation, for example, the mapping table maps the string `"i"` to the integer value 7, indicating that the value associated with index `"i"` — the local variable `i` — is stored in entry 7 of the activation's `v` vector.

When a new index-value pair is inserted into an activation, the index is stored in the mapping table, with the integer in `next_slot` as its associated value. The pair's value is stored at the `v` vector's entry specified by this integer. The `next_slot` field is then incremented by one.

The access link, `v[2]`, contains a table where free identifiers that appear in the procedure's code are to be found. It is usually the table that was the value of `root` when the procedure was compiled. This entry corresponds to the access link or static link that appears in the activation record of procedures in traditional languages.

Finally, entry `v[3]` holds the procedure for which this table is the activation.

3.1.3 Compiler Conventions

The use of the rest of vector `v` is determined by the compiler. In no way is the use of these entries enforced by the virtual machine. The virtual machine treats all these elements as just *EZ* values that it has to manipulate according to the instructions it executes.

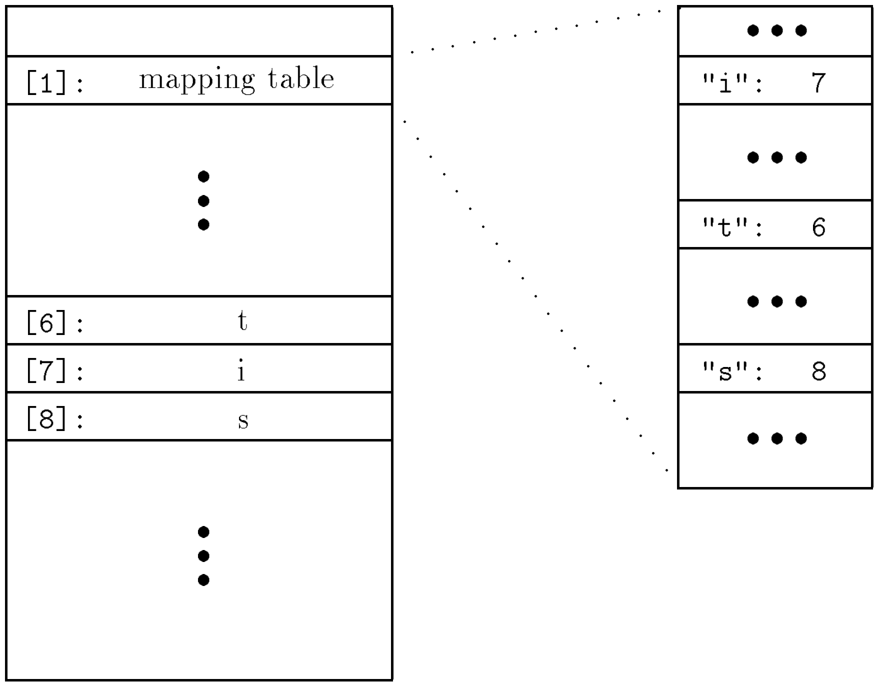

Figure 3: Initial mapping table for `ls`.

However, the *EZ* compiler does divide these other entries — from `v[4]` onward — into several groups that are used for different purposes. Mainly, this division concerns the relative positions of arguments, locals, and temporary values used by the procedure. There are also, however, a couple of values that are stored to facilitate the communication between 'caller' and 'callee' at procedure call and return boundaries. This division is just a convention that the compiler follows when generating virtual-machine instructions.

By convention, the initial entries in vector `v` after the first four are used as shown in Figure 2. They contain the dynamic link, return value, arguments, local objects, and temporaries.

The dynamic link, `v[4]`, is the activation that initiated this invocation of the activation. As in traditional programming languages, the dynamic link is used to return to the caller.

The return value, `v[5]`, stores the values returned by procedures called by the activation. The value returned by a call is stored in the activation of the caller by the callee. For this purpose, the callee accesses the caller's activation through its dynamic link.

The rest of the elements of the activation's initial `v` vector contain the argument values, the objects local to the procedure, and the temporaries needed to execute it, in that order. First, parameters are assigned to the entries starting with `v[6]`. This assignment is done in the order in which parameters appear in the source code. Local objects are assigned next, also in the order in which they are declared in the source code. Finally, the remaining initial entries are used as temporaries by the generated code.

Values associated with new indices inserted into the activation are stored in subsequent entries, starting with the first entry after the last temporary required by the generated code.

3.2 Instructions

Table 3 lists all the instructions in the virtual machine. It shows the mnemonic names for the instructions, and their arguments, together with the definition of the operations they perform. In the definitions, "`v`" represents the vector `v` of the current activation, and "`VOID`" stands for the internal representation of a non-value.

3.2.1 Code Layout

Virtual-machine instructions have a variable number of in-line arguments. Both the 'op-codes' that represent the instructions and the arguments for the instructions are integers. For most instructions, the arguments are indices into the vector `v` of the current activation. Usually, the first argument specifies the destination entry, where the value computed by the instruction is to be stored.

The compiler generates virtual-machine code as a sequence of integers, in which op-codes are followed by the corresponding number of arguments. For each procedure, this sequence is stored in a linked list of memory pages known as 'code blocks;' the list is accessible from the procedure itself.

Instruction	Definition
`LINK`	`v[0]` $\leftarrow$ address of first instruction in next page
`NOOP`	no operation
`QUIT`	terminates execution of current *EZ* process
`NEG x y`	`v[x]` $\leftarrow$ `−v[y]`
`ADD x y z`	`v[x]` $\leftarrow$ `v[y] + v[z]`
`SUB x y z`	`v[x]` $\leftarrow$ `v[y] − v[z]`
`MUL x y z`	`v[x]` $\leftarrow$ `v[y] * v[z]`
`DIV x y z`	`if (v[z] = 0)` *error* `else v[x]` $\leftarrow$ `v[y]` $\div$ `v[z]`
`MOD x y z`	`if (v[z] = 0)` *error* `else v[x]` $\leftarrow$ `v[y]` mod `v[z]`
`NOT x y`	`v[x]` $\leftarrow$ `if (v[y] = VOID) "" else VOID`
`LT x y z`	`v[x]` $\leftarrow$ `if (v[y] < v[z]) v[z] else VOID`
`LE x y z`	`v[x]` $\leftarrow$ `if (v[y]` $\leq$ `v[z]) v[z] else VOID`
`EQ x y z`	`v[x]` $\leftarrow$ `if (v[y] = v[z]) v[z] else VOID`
`NE x y z`	`v[x]` $\leftarrow$ `if (v[y]` $\neq$ `v[z]) v[z] else VOID`
`GT x y z`	`v[x]` $\leftarrow$ `if (v[y] > v[z]) v[z] else VOID`
`GE x y z`	`v[x]` $\leftarrow$ `if (v[y]` $\geq$ `v[z]) v[z] else VOID`
`JUMP a`	`v[0]` $\leftarrow$ `a`
`BEV a y`	`if (v[y] = VOID) v[0]` $\leftarrow$ `a`
`BNV a y`	`if (v[y]` $\neq$ `VOID) v[0]` $\leftarrow$ `a`
`SUBSEQ t x y z`	`v[t]` $\leftarrow$ `subsequence(v[x], v[y], v[z])`
`CATENATE x y z`	`v[x]` $\leftarrow$ `concatenation(v[y], v[z])`
`CVINT x y`	`v[x]` $\leftarrow$ `convert_to_integer(v[y])`
`CVREAL x y`	`v[x]` $\leftarrow$ `convert_to_real(v[y])`
`CVNUM x y`	`v[x]` $\leftarrow$ `convert_to_numeric(v[y])`
`CVSTR x y`	`v[x]` $\leftarrow$ `convert_to_string(v[y])`
`MKINT x y`	`v[x]` $\leftarrow$ `make_integer(y)`
`MKREAL x y`	`v[x]` $\leftarrow$ `make_real(y)`
`MKVOID x`	`v[x]` $\leftarrow$ `VOID`
`MKTABLE x`	`v[x]` $\leftarrow$ `empty_table()`
`LOAD x y z`	`v[x]` $\leftarrow$ `table_lookup(v[y], v[z])`
`STORE x y z`	`v[x]` $\leftarrow$ `table_insert(v[x], v[y], v[z])`
`LOADFRAME x y z`	`v[x]` $\leftarrow$ `frame_lookup(v[y], z)`
`STOREFRAME x y z`	`v[x]` $\leftarrow$ `frame_insert(v[x], y, v[z])`
`FETCH x a`	`v[x]` $\leftarrow$ `memory[a]`
`COPY x y`	`v[x]` $\leftarrow$ `v[y]`
`CREATE x p`	`v[x]` $\leftarrow$ `create_activation(v[p])`
`SAVE x`	`v[x]` $\leftarrow$ current activation
`RESUME t n`	current activation $\leftarrow$ `v[t]` (`n` arguments passed)

Table 3: Virtual-machine instructions.

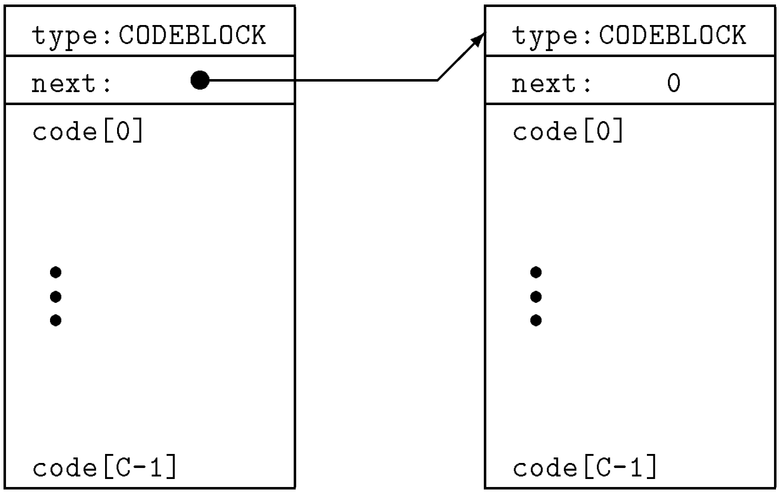

Figure 4: Code-block list for a procedure.

As shown in Figure 4, code blocks have three fields. The tag `type` indicates that the page contains a code block. The field `next` contains the virtual-memory address of the next code block for the procedure; it is the field that threads the pages of the linked list. As usual, the last code block in the linked list for a procedure has a 0 in its `next` field. The rest of the code block is an array of integers that contains the sequence of op-codes and arguments.

3.2.2 Simple Instructions

There are three special instructions defined by the virtual machine. They all have no arguments. The 'link' instruction, `LINK`, tells the virtual machine to continue executing instructions from the next code block in the list. This operation is not properly an instruction but an artifact to facilitate the implementation; they appear at the end of every code block. The compiler emits a link instruction and starts using a new code block as soon as the space left in the current code block is not large enough to hold the next instruction, its arguments, and a link instruction. Execution of a link instruction sets the program counter, `v[0]`, to the address of the first instruction in the next code block. The second special instruction, `NOOP`, does nothing; its purpose is only to allow local modifications to a sequence of instructions and to facilitate code generation. The third special instruction is `QUIT`; it terminates the execution of the current *EZ* process.

There are six arithmetic instructions. Their arguments all specify entries in the current activation's `v` vector. The first argument always specifies the destination; the others indicate the operands. All arithmetic instructions operate on numerics — both integers and reals — with the appropriate type conversions when necessary. These conversions are performed as described in Section 2.3.2; failure of any of them results in a run-time error.

There is only one logical instruction: `NOT`. It implements the operation defined by the negation operator (`~`) of the *EZ* language. The `NOT` instruction stores `VOID` or a null string in its destination, depending on whether or not the entry specified by its second argument has a value. The other logical operations of the *EZ* language — `&` and `|` — are implemented using the control-flow instructions described below.

There are six relational instructions. Following the behavior of *EZ*'s relational operators — as described in Section 2.3.3 — the specific comparison performed by these instructions also depends on the type of their operands. If the comparison succeeds, the result is the value specified by the third argument; otherwise, it is `VOID`.

In addition to the link instruction, there are three other instructions that affect the flow of control. `JUMP` is an unconditional 'jump' that always alters the execution order; it has a single argument, the virtual address of the instruction to be executed next. A `JUMP` resets the program counter, `v[0]`, to this new address. The other two control-flow instructions are the conditional branch instructions: `BEV` and `BNV`; they both have two arguments. Their first argument is a virtual address; the second specifies an entry in the vector `v` of the current activation. Both instructions conditionally change the program counter to the address given as their first argument. The 'branch-if-equal-void' instruction, `BEV`, changes the program counter only if the entry indicated by its second argument does *not* contain a value; the 'branch-if-not-void' instruction, `BNV`, changes the program counter only if its second argument *does* contain a value.

The 'subsequence' instruction, `SUBSEQ`, has four arguments that specify entries of vector `v`; the specific action it performs depends on the type of its second operand: it implements both the subtable and substring operations of the *EZ* language. The first argument specifies the destination, the second specifies the original value from which to extract the subsequence, and the third and fourth — which can be given in any order — specify its limits. If the second operand is a table, the instruction creates a new table containing all those index-value pairs in the original table whose indices are 'between' the limits, with both limits included. To determine if an index is between the limits, the same comparisons defined for the relational instructions are performed. If the second operand is not a table, the instruction attempts first to convert it to a string and the limits to integer values, if necessary. Failure of any of these conversions results in a run-time error; otherwise, the instruction creates a new string containing the characters in the second operand that are between the positions indicated by the limits.

The 'concatenation' instruction, `CATENATE`, has three arguments that specify entries of vector `v`. The first argument specifies the destination; the other two indicate the operands. If both operands are tables, the concatenation creates a new table that contains the entries that appear in either operand. Specifically, the operation is performed by first copying the first operand into a new table and then adding the entries of the second operand. If the operands have some common indices, the resulting table has the values associated with those indices in the second operand. If one or both operands are not tables, the instruction attempts first to convert them to strings, if necessary. Failure of any of these conversions results in a run-time error; otherwise, the instruction creates a new string that contains the characters of the first operand followed by the characters of the second.

There are four type-conversion instructions. They all have two arguments that specify entries of vector `v`. The first argument specifies the destination; the second specifies the value to be converted. Conversions are performed as described in Section 2.3.2. These instructions produce a run-time error if the conversion fails.

There are four value-building instructions. The 'make-integer,' `MKINT`, and 'make-real,' `MKREAL`, instructions build an integer and a real *EZ* value, respectively. The 'make-void' instruction, `MKVOID` stores `VOID` in the destination. Finally, `MKTABLE` creates and stores a new, empty table.

3.2.3 Data Movement

There are six instructions that move data. They are implemented in terms of primitive operations that perform insertion and look-up on arbitrary tables and others that perform similar tasks specifically for activations. Most virtual-machine instructions make use of these primitives; they access and modify the current activation.

Execution of the primitive operations is atomic. They are not interleaved with the execution of some other operations processing the same table; although they can be interleaved with operations on other tables. Atomicity guarantees the integrity of the data stored in tables, and also the correctness of the values returned by calls to the look-up primitives.

Atomicity of these operations is accomplished by locking some memory pages to restrict access to the same table by other processes. Locking is accomplished by gaining write access to memory pages, relying on the virtual-memory manager to guarantee that only a single writer can access a given page.

Two of the data-movement instructions — `LOAD` and `STORE` — move data from and to an arbitrary table. They both have three arguments that specify entries in the current activation's `v` vector. `LOAD` gets a value from a table and stores it in the destination. The second and third arguments specify the table and the index, respectively. If the second argument does not specify a table, or if the table does not have an entry with the given index, the instruction stores `VOID`.

`STORE` inserts an index-value pair into an arbitrary table. The first argument specifies the table where the pair is to be stored; if it is not a table, the instruction creates a new, momentarily empty, table. The second and third arguments specify the index and associated value, respectively, to be inserted. The resulting table is stored back in the destination.

Two other data-movement instructions — `LOADFRAME` and `STOREFRAME` — are similar to the previous two instructions, but they apply only to tables implemented using frames. Their purpose is to move data to and from activations, the only tables to which these instructions apply; when executed with any other value, a run-time error results. These instructions allow direct access to the entries of vector `v` of the given activation by accepting an explicit integer index.

`FETCH` moves a value directly from a specific address in the virtual address space into a temporary. It has two arguments. The first argument specifies the destination entry; the second is the virtual address of the desired value.

Finally, the instruction `COPY` moves the value stored in one temporary into another temporary. It has two arguments that specify the destination and source temporaries, respectively.

3.2.4 Activation Management

There are three instructions that create new activations and manipulate the current activation. They are used to implement calls and returns.

The 'creation' instruction, `CREATE`, has two arguments that specify entries of vector `v`. Creation instructions are used to create new activations for procedures, which they store at the entry specified by their first argument. The value stored at the second argument is converted to a procedure, if necessary. This conversion is performed as described in Section 2.3.2; its failure results in a run-time error. The procedure thus obtained is the procedure for which the activation is created. This activation may or may not be a new table. If the value stored at the entry specified by the second argument is already a table, it is converted to the internal representation for activations; otherwise, a new table, which uses this same representation, is created. Execution of a creation instruction fills in the `type`, `next` and `next_slot` fields — shown in Figure 1 — of the first frame for the new activation. It also fills in the first four entries of the new activation's `v` vector, which, as shown in Figure 2, are the program counter, mapping table, access link, and procedure. The initial values for these entries are fetched from the procedure itself, as is the initial value for the `next_slot` field of the frame.

The 'save' instruction, `SAVE`, stores the current activation in the entry of vector `v` specified by its only argument. The current activation can then be restored later, and its execution resumed, possibly after being moved to some other table. The current activation does not change: after execution of a save instruction, the current activation is still the same activation that was executing before and was saved by the instruction.

The 'resumption' instruction, `RESUME`, has two arguments and changes the current activation. It suspends execution of the current activation and activates the execution of another that was presumably stored by a save or by a creation instruction. The first argument specifies an entry of vector `v` whose value is the activation to be resumed; if this value is not an activation, a run-time error results. The second argument is an integer that specifies how many actual parameters were stored in the activation; it is needed for built-in procedures, some of which have a variable number of arguments. Execution of the resumption instruction sets the 'current activation' to be this new activation. The program continues execution at the address indicated by the program counter stored in the new current activation, i.e., execution proceeds where it was left when control was given to other activation by means of another resumption instruction. Execution of the activation does not start anew, unless it is a new activation.

3.3 Activation Invocation

The virtual machine treats calls and returns in a symmetric form. There are no special 'call' or 'return' instructions. The control flow implied by both of these constructs are implemented using `RESUME`. This symmetry also gives the flexibility required by procedure invocations in *EZ*, in which activations are tables that survive execution and can be invoked again, e.g., when using procedures as coroutines.

The virtual machine decomposes a procedure call into three primitive operations (Hanson and Griswold 1978): activation creation, argument binding, and resumption of execution.

The creation instruction, `CREATE`, creates new activations. Arguments are bound by storing the values of the actual parameters into the appropriate entries of the activation's `v` vector. Finally, the activation is executed by a `RESUME`.

A possible sequence of virtual-machine instructions for procedure calls is shown in Figure 5, which shows the instructions generated by the compiler for the call "$\texttt{P(X}_1\texttt{, X}_2\texttt{, ..., X}_k\texttt{)}$." In this sequence, `t` and `s` stand for two integers that denote temporaries in the activation of the caller; it is assumed that these temporaries are not

```
# load P into temporary s
CREATE t s
SAVE s
STOREFRAME t 4 s
# load X1 into temporary s
STOREFRAME t 6 s
# load X2 into temporary s
STOREFRAME t 7 s
...
# load Xk into temporary s
STOREFRAME t 5+k s
RESUME t k
```

Figure 5: Calling sequence.

otherwise needed. After execution of the sequence, temporary `t` holds the activation of the callee — procedure `P` — and temporary `s` holds the value of $\texttt{X}_k$ — the last argument passed.

The `CREATE` in Figure 5 creates an activation for procedure `P` and stores it in temporary `t`. This instruction sets the initial values for the fields `type`, `next` and `next_slot` of the first frame for the activation, as well as the first four entries of its vector `v`; see Figures 1 and 2. The next two instructions save the current activation — which is the caller's activation — into temporary `s`, and then store it into `v[4]` of the new activation, which sets the dynamic link for the callee's activation. The subsequent k `STOREFRAME` instructions take the k arguments successively, assuming their values are first stored into temporary `s`, and store them into `v[6]` to `v[5+`k`]` of the new activation. By convention, these entries are reserved for the arguments. Finally, the `RESUME` suspends the current activation and starts executing the callee's activation.

Notice that k cannot be greater than the number of parameters defined for procedure `P`. Otherwise, some argument values will be placed in the slots reserved for locals and temporaries in the activation. The number of arguments can be fewer than the number of parameters; unspecified arguments have whatever value was left from the

```
# load r into temporary t
STOREFRAME 4 5 t
RESUME 4 0
```

Figure 6: Returning sequence.

previous execution of the activation, or have no value in the case of a newly created activation.

A possible sequence of virtual-machine instructions for a return is shown in Figure 6, which shows the instructions generated by the compiler for the statement "`return r`." This sequence assumes that temporary `t` is not otherwise needed in the activation of the callee at the time of the return. After execution of the sequence, temporary `t` holds the value `r` returned by the procedure.

According to the conventions used by the compiler, `v[4]` in the callee's activation is its dynamic link; it contains the caller's activation, which was stored there when the call was made, as shown in Figure 5. The `STOREFRAME` instruction in Figure 6 stores the return value into `v[5]` of the *caller*'s activation. Finally, the resumption instruction suspends the execution of the callee and continues execution of the caller.

At return, only the return-value entry of the caller's activation is set, and this is precisely the only entry that is *not* set in the callee's activation at call time.

There are no 'hidden' pointers in the implementation. The activation and all its local objects become inaccessible as soon as a procedure returns. Similarly, processes are removed from the process table when they terminate; the space that they occupy becomes inaccessible. Global objects persist, but their values become inaccessible when the object is assigned a new value. Liveness analysis could help the system release the space occupied by dead values earlier (Appel 1992). The *EZ* compiler does not perform this kind of analysis. Although objects become inaccessible as explained, the precise time at which the space is reclaimed depends on the garbage-collection algorithm.

3.4 Up-Level References

Access to non-local variables is provided through the access link, `v[2]`. The access link must itself be an activation, a table implemented using frames. When a procedure is compiled, the compiler stores with the procedure the table that is the current value of variable `root`. Later, when an activation for the procedure is created, this value is used to initialize the access-link entry of the new activation.

Accessing values stored in the access link implements accesses to free variables in the source code. For instance, the instruction "`LOADFRAME t 2 s`" loads the value stored in temporary `s` of the access link into temporary `t` of the current activation.

The virtual-machine up-level referencing capabilities are more flexible than required by the language or used by the compiler. For a Pascal-like programming language with nested procedures, for example, the access link could contain the activation of the lexically enclosing procedure. This activation, in turn, would have its own access link containing other activation, and so on. The same scheme could be used in *EZ* to link the successive `".."` entries in the search path for free identifiers.

Chapter 4

Virtual-Memory Management

EZ's four-gigabyte virtual address space is too large to fit in the physical memory of most computers. Expanding the address space to use 64 bits for virtual addresses to make full use of the new 64-bit processors (Digital Equipment Corporation 1992) will exacerbate this problem. The address space must be kept in secondary storage.

There is, however, a second — and perhaps more important — reason for storing the address space on disk. *EZ* is a persistent system; objects must survive execution and cannot disappear each time the system is turned off, or each time there is a power failure.

The *virtual-memory manager* caches pages from the virtual address space into physical memory and keeps the disk image of the address space up to date. The manager isolates the interpreter, compiler, and rest of the run-time system from the actual locations of the pages and provides the system with a persistent virtual address space.

The virtual-memory manager uses a fixed number of in-memory page frames as a cache for the virtual-memory pages. When one of these frames needs to be re-used for another virtual page, and if the frame currently contains a page that has been modified, its contents must be written back to disk. Flushing the cache into secondary storage saves the persistent virtual address space and, with it, the system's state.

Currently, all memory management is done in software; the virtual-memory manager makes no use of hardware support to catch page faults or to do address translation. This decision has led to some peculiarities in the memory-mapping architecture of the system. In particular, to reduce the overhead that would result from calling the manager for every access to virtual memory, the *EZ* interpreter requests the right to access a page and then holds the page for several accesses. During these intervals, the interpreter effectively precludes the manager from accessing or preempting the page; in fact, it is the interpreter that informs the manager when it is ready to relinquish the page.

This chapter describes the design and implementation of the distributed virtual-memory manager that distributes *EZ*'s persistent address space transparently among a network of homogeneous computers. The manager adapts the techniques used in several recent implementations of shared virtual memory (Bennett, Carter, and Zwaenepoel 1990, Li and Hudak 1989, Li and Schaefer 1989a, Nitzberg and Lo 1991, Stumm and Zhou 1990) for use in *EZ*'s persistent environment. Unlike most implementations of shared virtual memory, control information is also distributed and migrates among the processors in the system.

The distributed manager has been developed specifically for *EZ*, but it can be used as the virtual-memory manager for other systems. Other than the interface presented by the manager, there is little in the virtual-memory system that is specific to *EZ*.

4.1 Shared Virtual Memory

In a distributed system, interprocess communication is based on either a 'message-passing' model or a 'shared-memory' model. While the former is a simple and convenient extension of the underlying communication mechanism of the system, the latter model provides a simpler programming paradigm that is much closer to the usual uniprocessor programming paradigm familiar to most programmers. Message-passing systems and systems that support remote procedure call (RPC) are based on the message-passing model (Bennett, Carter, and Zwaenepoel 1990).

In the shared-memory model, processes communicate through a common address space that is used much in the same way as local memory; data is accessed through familiar operations that read and write memory locations (Stumm and Zhou 1990). In this model, users need not use — or even be aware of — any communication primitives. Furthermore, the shared-memory model allows passing data by reference because there is a common memory that is accessible to all processes. In contrast, in the message-passing model, processes can pass information only by value because they operate in different address spaces.

The shared-memory model is especially suitable to be used in shared-memory multiprocessors where there is a real physical memory that is accessible to all processors. However, in loosely coupled distributed systems, where there is no shared physical memory to support it, a software layer can be used to provide a shared-memory abstraction. Several algorithms have been developed and proposed for this purpose; they use the underlying communication mechanism to implement 'shared virtual memory.'

Since it was first proposed (Li 1986), much effort has been devoted to the study of shared virtual memory (Bennett, Carter, and Zwaenepoel 1990, Gelernter and Carriero 1986, Li and Hudak 1986, Li and Hudak 1989, Li and Schaefer 1989a, Li and Schaefer 1989b, Nitzberg and Lo 1991, Stumm and Zhou 1990). Implementations of this abstraction can be made to perform as well as — and in some cases even better than — the direct use of message-passing primitives in loosely coupled distributed systems (Stumm and Zhou 1990).

In an attempt to exploit parallelism by allowing two or more simultaneous read accesses, most of these algorithms replicate parts of the shared information at several processors. As usual in the classical readers and writers problem, the objective is to let multiple readers but only a single writer access the same virtual-memory location simultaneously.

Replication forces these algorithms to maintain 'coherence' among the different replicas of a single virtual-memory page. A shared virtual memory is coherent if the value returned by a read operation is the value written by the most recent write operation to the same location. This notion is sometimes called 'strict' coherence and is used, for example, in Ivy (Li 1986). Other systems use a more relaxed notion called 'loose' coherence (Bennett, Carter, and Zwaenepoel 1990), which is implied by, but is not equivalent to, strict coherence. Under this weaker definition, the value returned by a read need only be the value written to that location by a write that could have immediately preceded the read in some legal schedule of the execution.

4.2 Distributed *EZ*

Distributed *EZ* runs on a set of homogeneous processors linked by a network through which they can communicate reliably. Homogeneity is not central to the design or to the discussion of distributed *EZ*. It does, however, facilitate the implementation because there is no need to transform from one data representation to another when sending messages between processors.

Figure 7 shows a schematic diagram of the hardware architecture used by distributed *EZ*. Each processor has its own local physical memory, and there is no global physical memory accessible to and shared by all processors. Each processor has access to some secondary-storage devices; it is unimportant whether they are local or remote. Finally, the network provides point-to-point, order-preserving, reliable communication between every pair of processors in the system.

Distributed *EZ* uses a memory manager based on shared virtual memory to distribute *EZ*'s persistent address space; all *EZ* processes share the same address space regardless where they run. The single address space allows sharing under the simpler shared-memory paradigm; the memory manager itself uses message passing to provide this service. Distributing the address space over a number of work-stations allows multiple users to access the system simultaneously from different physical locations; users can share values with one another, including source-language or executable code and active values like activations or processes.

The virtual-memory manager is distributed; identical copies of the manager run at each processor. Only the managers know about distribution; the interpreter, compiler, and rest of the run-time system are unaware of the distribution. Also, neither *EZ* processes nor users need to be aware that the system is distributed.

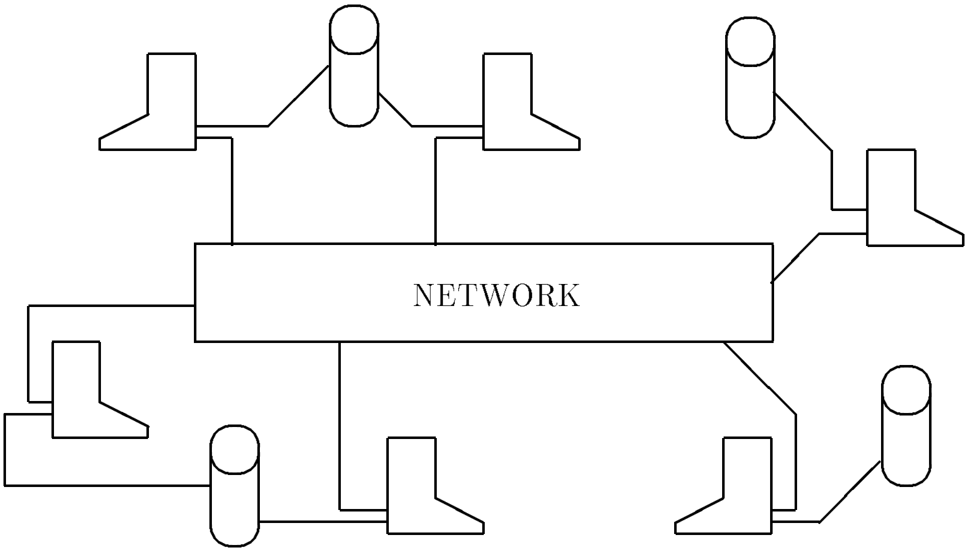

Figure 7: Hardware architecture.

As in most shared virtual memory implementations, the virtual-memory managers replicate pages at different processors in an attempt to collect recently accessed pages at the processor that accesses them; each manager caches the memory pages that the local interpreters are using. Physically, the virtual address space is distributed among the secondary storage devices of the individual processors.

Traditional implementations of shared virtual memory have used 'invalidation' techniques (Li 1986) to maintain coherence. Under this approach, a writer is allowed to access a page only after all replicas of the page have been invalidated. This behavior implies that copies that have not been invalidated are still up to date and can be accessed without additional communication. In distributed *EZ*, however, the memory manager yields control of the pages to the interpreters. This lack of control forces a different virtual-memory protocol, in which copies cannot be invalidated, and validity of the copies must be confirmed before granting access.

4.3 Memory-Manager Interface

The virtual-memory manager is a server and presents a fixed interface to its only clients, the *EZ* interpreters. The manager's services are defined by three access functions:

```
handle = allocPage()
handle = getPage(a, mode)
putPage(h, state)
```

Function `allocPage` allocates a new page of virtual memory and returns a 'handle' to its location; newly allocated pages are cleared. Function `getPage` returns a handle to the page that encompasses virtual address `a`; `mode` indicates the access mode desired for the page and can be READ, EXECUTE, or WRITE. Finally, `putPage` releases the page given by handle `h`; `state` indicates whether or not a writable page was actually modified and is either DIRTY or CLEAN. Pages requested for read or execute access must not be modified and can only be released CLEAN.

Handles are simply the physical-memory addresses of the in-cache copies of the pages. The copies occupy physical-memory frames equal in size to the size of a virtual page, and consecutive virtual addresses within a page correspond to consecutive physical addresses in its in-cache copy. Function `allocPage` returns the physical address of the first byte of the in-cache copy of the newly allocated page. Function `getPage` returns the physical location of the specific virtual address requested in the call. In both cases, the handle grants access to the entire virtual page and allows the client to access the copy by dereferencing. The handle passed to `putPage` can be the physical location of any virtual address in the page being released.

A call to `getPage` specifying write access indicates that the client wants exclusive access and may modify the page; function `allocPage` always assumes write access. A call specifying read access indicates that the client will not modify the page and that shared access is possible. Execute access mode is similar to read, but the manager is not required to get the latest version of the page. Execute access is useful when the client knows that the page is read only, i.e., it was written once and will never be modified; any copy of these pages is identical to their 'latest' version. For example, the interpreter knows that code blocks, which contain virtual-machine instructions (see Section 3.2.1), are read-only pages; therefore, it always specifies execute access mode for them.

Unlike clients of typical memory managers in operating systems, the *EZ* interpreters are 'trusted' clients of *EZ*'s memory manager and are, therefore, given complete control of the pages they request via `allocPage` or `getPage`.

Upon return from `getPage` or `allocPage`, when the interpreter has been granted the access it requested, it uses the handle to access the page as many times as it needs. Later, when the interpreter no longer needs access, it calls `putPage`. Between the call to `getPage` or `allocPage` and the matching call to `putPage`, the manager has no control over that specific page, nor is it allowed to reuse the physical frame occupied by the in-cache copy.

All interactions between *EZ* processes and the memory manager, as well as all their accesses to memory, are through the interpreters, which act on their behalf; processes never intervene directly. Thus, the interpreters are the only clients of the manager and are known to be 'reliable' and 'cooperative.' Specifically, interpreters follow the above interface protocol and do not hold pages indefinitely.

4.4 Page Ownership

Local managers grant access to pages requested by local interpreters. Before granting access, the manager caches a copy of the page into its local physical memory. The page's contents may be read from the disk image or may come through the network from another manager.

Every virtual-memory page is *owned* by one manager. The owner of a page grants access to the page to other managers, which can then grant their local interpreters the accesses they request. Other managers seeking access to a page must get permission from the page's owner. The owner also keeps the disk copy of the page up to date.

The owner of a page is not a fixed, predetermined, manager. During execution, page ownership migrates to the managers that need write access to the page. Consequently, the owner of a page is the last manager that granted write access to the page. The pages themselves migrate among the local disks of the individual processors.

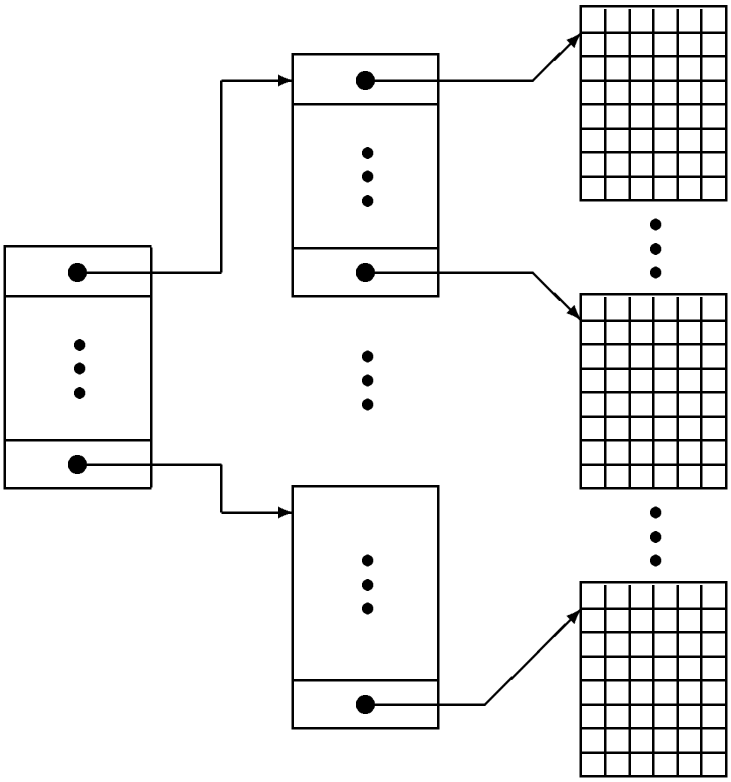

Figure 8: Set of owned pages.

Each manager has the information about its owned, *local* pages; for this purpose, each one keeps a data structure containing its set of owned pages. Figure 8 sketches the implementation. Although private to each manager, this structure resides in the virtual address space, and the pages it uses are maintained and managed by the manager itself. These local pages never migrate because they are used only by one manager and access to them is never requested by the others. They might appear only on disk because, as any page, they occupy cache frames that may be needed.

Three operations are performed on the sets of owned pages. A page is *inserted* into the set when ownership of the page is received from the previous owner, together with the permission to write the page. A page is *removed* from the set when the current owner grants write access to another manager and, therefore, relinquishes ownership of the page. Finally, when a page is cached, it is tested for *membership* in the set to determine if the manager owns it.

The current implementation uses a bit map for each of these sets. Each bit map has one bit per virtual-memory page in the address space; the bit is set if the page is owned by the manager. As suggested in Figure 8, the bits themselves are kept in a collection of virtual pages that are accessed through two levels of indirection by using another group of pages that contain their virtual addresses.

4.5 Status of Copies

A manager associates three different status values with the copies in its cache; each copy is *invalid*, *shared*, or *unique*. The status of an in-cache copy approximates the information about whether or not the copy is up to date and guides the manager when granting access to its associated page.

Invalid copies are out of date and are associated with pages owned by other managers; this value is the initial status for copies of remote, i.e., non-local, pages. Copies become invalid when the local manager learns that some other manager has been granted write access to the page, that is, when the local manager itself grants the access or when it receives a message indicating that a manager seeks write access to the page. Invalid copies must be updated by fetching a new copy from the pages' owners the next time that local access is required.

Shared copies may be up to date according to the local manager's information and other managers might also have shared copies. Shared copies of local pages are always up to date. Copies become shared when read access to the page is granted to either a local interpreter or another manager. Local accesses to shared copies can be granted if the copy is known to be up to date or after the owner confirms the copy's validity. If the owner doubts the validity of the copy, it sends a new copy; otherwise, it simply grants permission. As it is explained in Section 4.7, the owner of a page maintains a bit vector with the set of managers that have received a valid copy of the page.

Unique copies are associated with pages owned by the local manager and for which there are no up-to-date copies elsewhere; this value is the initial status for copies of local pages. Copies become unique when write access to the page is granted to local interpreters. Local requests to unique copies can be granted by the local manager without communicating with other managers. A page's owner is the last manager that had a unique copy of the page.

4.6 Messages

The virtual-memory managers communicate with each other in order to request access to remote pages and grant access to local ones. Communication is based on the network layer providing reliable datagrams — byte streams, in UNIX parlance — between any two processors. Message order is preserved: messages sent between any two pair of processors are delivered at the recipient in the same order in which they were sent by the sender.

The messages can be classified in two general groups: 'request' messages and 'acknowledgement' messages. Request messages convey a request to access memory. Acknowledgment messages are sent in response to request messages and grant access to a requesting manager.

Request messages have three fields. The first, `type`, identifies the kind of request, as explained below. The second, `sender`, contains the identification of the initiating manager — the requestor — and its processor; the identification is a small integer that is unique for each processor. The requestor is not necessarily the manager that sent the message, but rather the manager that is requesting the service; they may be different when a manager forwards the request on behalf of another manager. The third field, `address`, is the virtual address of interest in the request. Figure 9, for example, shows a request message in which processor 3 requests read access to virtual address 8074.

Acknowledgment messages, which are sent by the owner of the page, have a similar form, but they may include a copy of the virtual page itself. In acknowledgments, `sender` identifies the owner, and `type` indicates whether or not the optional page follows. Acknowledgments include a copy of the page either when the requestor specifically asks for it or when its copy might be out of date according to the information kept by the owner.

`type:`	`READ`
`sender:`	`3`
`address:`	`8074`

Figure 9: A request message.

The `sender` field helps managers distinguish between local and remote requests; local requests come from interpreters running in the same processor, remote requests come from other managers. Local requests do not travel through the network, but allow managers to handle the input uniformly, regardless of its source. The `address` fields of remote requests always specify pages, i.e., they are the virtual addresses of the first byte of the pages; local requests mirror the semantics of `getPage` and can refer to any virtual address.

There are five types of local requests. They are generated by calls to `getPage` or `putPage`. Calls to `allocPage` do not generate messages because, being a newly allocated page, the call grants its only access, and no other references can exist. Calls to `getPage` generate READ, EXECUTE, or WRITE requests, reflecting the access mode. Calls to `putPage` generate CLEAN or DIRTY requests, reflecting the state of the page.

Most requests are for locally owned pages, and handling these is straightforward because no interprocessor communication is required. Requests for pages that the local manager does not own require communication with the owner of the requested page. There are seven message types involved in this remote communication; they are refinements of the types used for local requests.

As shown in Figure 10, when an interpreter requests access to read a remote page, the local manager grants the access immediately if it knows that its in-cache copy of the page is up to date, i.e., a local interpreter is currently reading the same page. Otherwise, if the local manager has a copy of the page that it believes is up to date, i.e., it has shared status, a READ request is sent to the owner; the owner simply grants the access, but includes a copy of the page if it is possible that the requestor's copy is out of date. If the local manager does not have an up-to-date copy of the page, it sends a COPY request to the page's owner; acknowledgments to copy requests always include a copy of the page.

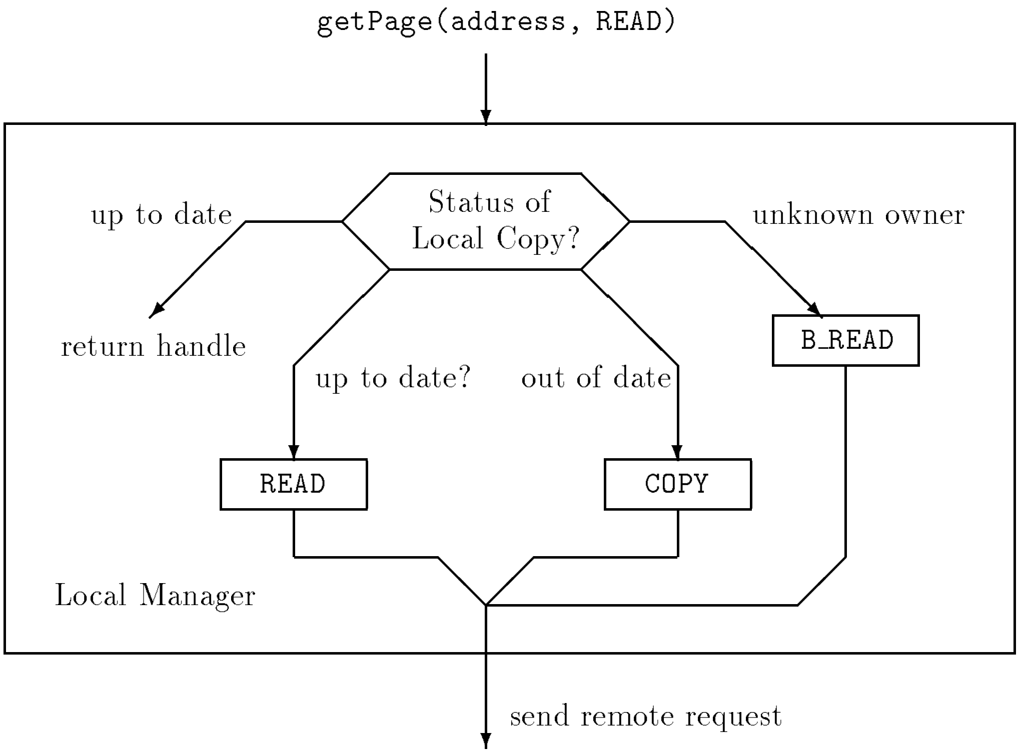

Figure 10: Remote messages for read access.

Similarly, as shown in Figure 11, when an interpreter requests access to write a remote page and the local manager believes it has an up-to-date copy, a WRITE request is sent to the owner; the owner simply grants the access, but includes a copy of the page if it is possible that the requestor's copy is out of date. If the local manager does not have an up-to-date copy, it sends a FETCH request to the page's owner; as with copy requests, acknowledgements to fetch requests always include a copy of the page. Communication with the page's owner is always required because in either case ownership of the page is transferred from the current owner to the requesting manager.

All the previous types of remote messages are sent to the owner of the desired page. There are, however, times in which the local manager does not know the owner of a page that it needs. In these cases, as indicated in Figures 10 and 11, the requestor sends a BROADCAST READ or a BROADCAST WRITE request to all managers. These messages are broadcast equivalents of the copy and fetch requests described above; a

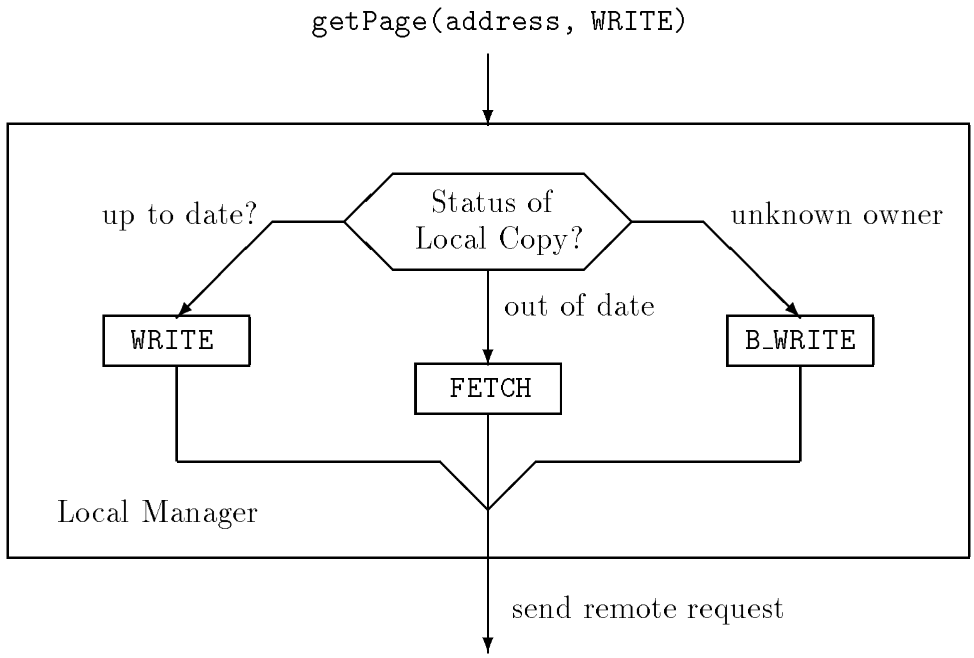

Figure 11: Remote messages for write access.

copy of the page is always included in acknowledgements to them. Only the actual owner responds to these messages, which identifies the owner for subsequent requests and reduces network traffic. Similarly, messages directed to a manager that is no longer the owner of the desired page are turned into the corresponding broadcast message by the recipient, or are simply forwarded if the recipient knows the current owner.

Requests for execute access to a remote page are simpler. The local manager grants the access immediately if it has a copy of the page; no confirmation from the owner is necessary. Otherwise, it sends a broadcast read request; as explained in Section 4.7, if there is no local copy of the page, the local manager does not have information about which manager may be the page's owner.

The last type of remote request is the DECREMENT request. It is sent to the owner of a page when the local manager, after receiving a clean request, realizes that no local interpreter is accessing the remote page. A decrement request tells the page's owner that there are no local accesses to that page in the requestor's processor.

This message is the remote equivalent of a clean message. Unlike all other remote requests, decrement requests have no acknowledgment message associated; the owner just records that one remote access to that page has been completed. There is no need for a remote equivalent to dirty requests because only owned pages can be written.

4.7 Page Table

Each manager keeps information about the pages it has cached in a *page table*. The page table is a fixed-size table implemented as an open hash table.

For each cached page there is a page-table entry that holds a pointer to the in-cache 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 values apply only to locally owned pages.

For remote pages, the reference count is the number of outstanding accesses that have been granted in response to local requests. For owned pages, this count is the number of local accesses that have been granted plus the number of other managers that have outstanding accesses to their copies of the page. If there is a write access, there is only one interpreter accessing the page in the whole system; the reference count is one at the owner and zero at any other manager that has a copy of the page. Granting access to a page increments the count; clean and dirty requests decrement the count. For remote pages, a decrement request is sent to the owner when the local count reaches zero, and the owner then decrements its count.

Requests for write access are appended to the queue of pending access requests if some interpreter is accessing the page, i.e., the reference count is positive. Similarly, requests for read access are queued if some interpreter has write access to the page, i.e., the reference count is positive and the copy has unique status. Requests waiting in the queue of pending access requests are processed when the reference count becomes zero.

The ownership hint caches information about the owner of the page. It correctly identifies owned pages and gives the probable identity of the owner of a remote page. For local pages, it contains the identification of the local manger. For remote pages, it contains the identification of the manager that was the owner when the in-cache copy was received, or of the manager that received ownership the last time that the local manager learned about write access to the page. Initially, when a remote page is added to the page table, the ownership hint is empty. The owner's identity accompanies the copies of remote pages in acknowledgments, which updates this field.

The ownership hint allows the local manager to send requests to the probable owner instead of using broadcast messages, which are needed only if there is no information about the owner. It also allows the manager to distinguish between local and remote pages without accessing the set of owned pages; membership in this set is tested only when a page is added to the page table.

The valid-copies hint is a bit vector that identifies managers to which valid copies of the page have been sent. Only the page's owner keeps this hint, which is used to avoid sending copies of pages unnecessarily. Whenever the owner receives a remote read or write request, it includes a copy of the page in the acknowledgment only if the requestor is *not* in this hint. The valid-copies hint is cleared when an entry is created for a page, and it is cleared again each time the manager grants write access. When a copy of the page is sent to another manager, the hint is updated to include the receiving manager.

Finally, the dirty bit tells the page's owner whether or not the in-cache copy of the page differs from the disk image. The bit is set whenever a dirty request is received for the page, and it is reset when the copy is written to disk.

4.8 Page Replacement

When a manager receives a request for a page about which it has no information, it assigns a new page-table entry, possibly replacing another page. If the entry selected for replacement contains an owned page that has been modified, the page is written to disk. If the new page is an owned page, its contents are read from disk.

Only entries with a zero reference count and no unprocessed requests are candidates for replacement. Entries for pages that are being accessed by some interpreter or for which there are requests are not considered.

Managers have four priority levels to classify entries that are eligible for replacement: *invalid*, *remote-shared*, *unique*, and *owned-shared*. A doubly linked list connects the candidate entries in each level. The first two levels correspond to remote pages; the last two, to locally owned pages. The status of the in-cache copy and the ownership condition of the pages are used for this classification because they are indicative of the cost associated with re-using the entry.

When the manager must replace a page, it uses the first entry on the list of invalid candidates. For these pages, the in-cache copy is useless and new copies are required before they can be accessed. Thus, there is no penalty paid when replacing these pages.

If the first list is empty, the manager uses the first entry on the list of remote-shared candidates. Doing so forces the manager to fetch a new copy of the replaced page when requested again. Being a remote page, access requires confirmation from the owner. The penalty paid by re-using these entries is that a copy of the pages is transmitted at the next access, perhaps unnecessarily.

If both of these lists are empty, the first entry on the list of unique candidates is used. Selecting an entry from this list forces the manager to reread the page from disk when requested again.

If all else fails, the first entry on the list of owned-shared candidates is used. There are valid copies of these pages elsewhere, but the manager — and owner — must 'forget' them because it is about to reuse the cache entry. Consequently, subsequent requests from other managers cause the page to be resent, perhaps unnecessarily.

Each of the above lists is arranged in increasing order by the time of the last request for the page. Entries are inserted at the end of the lists when they become available for replacement. Therefore, within each list, the least recently used entry is selected.

4.9 Implementation

The implementation of the distributed virtual-memory manager consists of about 2000 lines of ANSI C (Kernighan and Ritchie 1988). This number includes the routines that use UNIX interprocess communication primitives to provide point-to-point communication between managers. This communication is based in sockets that provide reliable byte streams (Stevens 1990).

The memory manager is a multithreaded system that uses several lightweight processes or threads (Cormack 1988). The implementation of threads consists of about 400 lines of ANSI C and about 60 lines of assembly language, but it is also used by other parts of the *EZ* system, like the interpreter and garbager collector, not only by the memory manager.

At each processor, one thread monitors the network continuously. This thread runs forever or until the whole distributed *EZ* system is shut down by a process that calls the built-in procedure `stop`. When there is no other thread running in the local processor, this thread blocks the system in that processor until an incoming message wakes it up; before blocking, it flushes the local cache to disk. Listening to the network is the only responsibility of this thread; whenever it receives a message, it creates a new thread to process it.

Each cached page for which there are requests is served by a separate, per-page thread, which allows execution to proceed even when one of the threads must wait. For instance, while one thread waits for the acknowledgment to a remote request, requests for other pages can be serviced.

Per-page threads process the requests for their pages one at a time. Each of these threads has its own FIFO queue of requests. This arrangement serializes the processing of requests for each page, which in turn ensures the consistency of the respective page-table entries and of the in-cache copies of the pages. It also simplifies programming and reduces execution time. No synchronization is necessary to access the page-table entries because all accesses come from a single thread. A per-page thread exists for as long as it has requests to serve; when its service queue becomes empty, the thread terminates.

Local requests are posted in the service queue by the interpreter threads that call `getPage` or `putPage`. Remote requests are posted by the threads that process messages received from other managers through the network. No locks are needed to post requests in the service queues because the threads cannot be preempted, and they do not pause while posting. The posting threads also create per-page threads when necessary.

Chapter 5

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 worked fine for a prototype and, as shown by the Oberon system (Wirth and Gutknecht 1989), is perhaps adequate for a non-distributed, 'single-user' system that is subject to frequent idle periods. But an off-line approach is unsuitable for a distributed system.

Distributed *EZ* uses a distributed garbage collector that works in concert with the shared virtual-memory manager described in Chapter 4. It is a mark-and-sweep collector (Augusteijn 1987, Björnerstedt 1989, Dijkstra *et al.* 1978, Schelvis and Bledoeg 1988), which has the usual two phases. First, the 'mark' phase marks all virtual-memory pages that are accessible from a few system root objects. Then, the 'sweep' phase sweeps the virtual memory and collects all those pages that remain unmarked after the mark phase.

EZ's collector is *concurrent*. It does its work in parallel with the mutators, which are the interpreter threads acting on behalf of *EZ* processes. It is also *real time*; the mutators are never interrupted for longer than a small constant time.

For systems like *EZ*, which have a large, persistent address space, efficiency of the collection algorithm is less important than concurrency. Space efficiency and minimal disruption of mutators are more important than time efficiency of the algorithm itself; *EZ*'s collector trades time for space and minimal disruption of mutators, and reduces the communication costs. Indeed, all that is required is that the collector replenish the supply of free virtual-memory pages fast enough so that applications rarely have to wait to allocate a new page. And it must collect all inaccessible pages eventually.

5.1 Distributed Garbage Collection

Most collectors proposed for distributed systems require double indirection for remote references (Lang, Queinnec, and Piquer 1992, Rudalics 1986, Schelvis and Bledoeg 1988); they use special indirect cells both at the local and at the remote processor. This restriction also requires direct mutator collaboration, and imposes a severe overhead on the mutators when creating references and, especially, for dereferencing.

Most collectors are designed for systems in which pages do not migrate (Lang, Queinnec, and Piquer 1992, Schelvis and Bledoeg 1988), or in which page migration requires mutator collaboration to update some of the collector's data structures (Björnerstedt 1989). Many proposals require global synchronization among processors (Appel, Ellis, and Li 1988, Augusteijn 1987, Björnerstedt 1989, Halstead 1984, Lermen and Maurer 1986), which in some cases implies halting the mutators. Finally, in some parallel or distributed systems collections occur in a per-process basis with no global garbage collection (Jagannathan and Philbin 1992a, Jagannathan and Philbin 1992b), or they have been serialized disallowing concurrency (Pallas and Ungar 1988).

Collectors based on reference counting are an interesting alternative because they are relatively simple and incremental. They do, however, require extra storage per reference. Reference-counting garbage collectors can be made both concurrent and real time (Rovner 1986), and have been shown to be unobtrusive even to system programs (Rovner 1984).

Distributed versions (Bevan 1987, Goldberg 1989, Lermen and Maurer 1986, Watson and Watson 1987), however, require additional data for every pointer that refers to a page on another processor, or additional synchronization between subsets of the processors at each access to remote pages.

A correct, distributed protocol for a straightforward extension of uniprocessor reference counting requires three messages per interprocessor references (Lermen and

Maurer 1986); this algorithm is concurrent but not real time because deletion of a reference requires synchronization among processors. Pages need a reference count, and each processor needs one integer per remote object that is locally referenced or one bit per remote reference.

Generational reference counting (Goldberg 1989) has lower communication overhead; it requires only one message per remote reference when the reference is removed; this collector is concurrent and real time. However, it has greater computational and space requirements; each reference has a counter and a generation field in addition to the address, and pages have a 'ledger' that keeps track of the outstanding references of each generation. Weighted reference counting (Bevan 1987, Watson and Watson 1987) is similar and also requires only one message for each interprocessor reference, but it is significantly more space efficient than generational reference counting.

Reference-counting collectors have two major weaknesses. Maintaining the counts requires direct collaboration from the mutators whenever they copy or remove a reference. Second, and more important, these algorithms cannot handle cyclical references, which makes them unsuitable for EZ, where cycles abound.

Copying collectors have the potential to be much more efficient than reference-counting and mark-and-sweep collectors; especially when the total available memory is much larger than the amount of active objects (Appel 1987, Appel 1991). These collectors avoid touching the garbage and their work is proportional to the number of accessible pages; they do, however, split the memory into two semi-spaces halving the memory available for the objects. Copying collectors can be made both concurrent and real time by using virtual-memory hardware to detect pages whose pointers have not been forwarded and may still contain references to 'from-space' pages (Appel, Ellis, and Li 1988).

Most copying collectors designed for distributed address spaces are neither concurrent nor real time. In addition, some require special hardware to be efficient or require interprocessor synchronization (Halstead 1984), and some impose restrictions on interprocessor pointers such as double indirection (Rudalics 1986). Generational collection (Lieberman and Hewitt 1983, Ungar 1984) reduces the work performed by a copying collector by scanning and copying fewer objects, but distributed versions (Pallas and Ungar 1988) present much the same problems as those of copying collectors.

Multilisp's collector (Halstead 1984), for example, is incremental, but is neither concurrent nor real time. It is based in Baker's original incremental copying algorithm (Baker 1978). Swapping of the semi-spaces requires synchronization of all processors, which must all swap at the same time; there are also some non-global synchronization requirements. This collector requires special hardware to have acceptable efficiency (Appel, Ellis, and Li 1988).

The distal-objects collector (Rudalics 1986) is also incremental, but neither concurrent nor real time. One step may take unbounded time in one processor because it may require an unbounded number of messages. Remote references require double indirection, and all objects reachable at the beginning of a collection cycle are retained; garbage produced during the cycle is not collected until the next.

Objects in *EZ*'s virtual address space do not move; this restriction greatly simplifies the implementation of the persistent address space, but disallows the use of copying collectors. It also simplifies the processing of pages by the collector itself, because not all references have to be found to be forwarded. For instance, there is no need to look at the specific virtual-machine instructions in the object code; the virtual machine guarantees that all objects to which instructions refer are accessible through other paths. Moving objects in the address space requires forwarding all pointers to point to the new locations.

In a mark-and-sweep collector, the sweep phase performs work that is proportional to the memory size; even when most of the address space contains garbage. The mark phase, by contrast, processes only the active data. Unlike copying collectors, mark-and-sweep collectors do not partition the memory.

Mark-and-sweep collectors can be made both concurrent and real time. The first such collector (Dijkstra *et al.* 1978) was single threaded, and ran in a processor that is exclusively dedicated to garbage collection while other processors run the mutators. Several distributed mark-and-sweep collectors for shared address spaces have also been proposed (Augusteijn 1987, Björnerstedt 1989, Schelvis and Bledoeg 1988).

The collector for distributed Smalltalk (Schelvis and Bledoeg 1988) works incrementally and does not use any synchronization; it is neither concurrent nor real time. Local collection is done independently, but the collectors interchange information about remote references. Objects do not migrate, and remote objects are referenced through double indirection.

The collector described by Augusteijn (1987) is concurrent, but not real time; some mutators may have to wait for a remote message exchange to take place; in particular, coloring of remote objects by the mutators requires remote communication. Each processor has its own garbage-collection process that accesses local data only. Each cycle requires global synchronization of the local collectors: three barriers per collection; and may require an unbounded number of messages. The algorithm is not completely distributed but a fully distributed version can be derived from the described collector.

Björnerstedt (1989) describes a mechanism for garbage collection that may be used in a decentralized persistent object-based systems. The algorithm requires a quiescent state, during which mutators must be stopped, to perform some phases of the collection; mutators must maintain an export-import list with interprocessor references. Recovery of local garbage does not require global communication. The distributed algorithm collects global garbage; it requires two messages from every processor to every other processor.

5.2 Garbage Collection in *EZ*

EZ's collector runs continuously without interfering with the rest of the system, and performs one collection after another. Identical copies of the collector run on each processor in the system; technically, there is one collector for each memory manager. A new collection is triggered when the number of pages allocated at any processor since the last collection exceeds a certain limit; for the prototype, this limit is 500 pages.

Since the address space is distributed among all processors, each collector can access only some pages. In particular, each local collector is in charge of processing the pages owned by its associated manager, which avoids page migration — or even remote reads — for garbage-collection purposes only.

Like uniprocessor mark-and-sweep collectors, *EZ*'s collector marks all pages that hold accessible objects starting from a few system root objects. The algorithm also marks pages referenced from within 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 (Dijkstra *et al.* 1978). It repeats the collection continuously, so it eventually reclaims all inaccessible pages.

Most pages hold only one *EZ* value or internal data structure; conversely, all aggregate values use an integral number of pages. For example, strings are implemented by lists of pages, arrays are implemented as pointers to pages of additional pointers to pages of values, and the hash tables for associative tables fit in a page and their buckets are implemented as lists of pages. These techniques produce a system in which pages cannot have both accessible and inaccessible objects; when a value becomes inaccessible, all of its pages become inaccessible. From the point of view of memory allocation, these techniques reduce internal fragmentation and simplify memory management, which deals only with single-sized pages. However, small aggregate objects produce internal fragmentation because they do not use all the memory allocated to them; a small page size reduces this effect. For the test programs used to evaluate performance of the system (see Section 6.2), a conservative estimate of the use of memory shows that less than 40% of the space is unused. From the point of view of garbage collection, those techniques allow collection in a per-page basis, with the certainty that no page can be marked solely because of a gratuitous reference from an inaccessible object that happens to reside in an accessible page.

The mark phase of each local collector proceeds in cycles. After marking the accessible owned pages, the local collector broadcasts the set of interprocessor references found in those pages to all other collectors; the collector for distributed Smalltalk (Schelvis and Bledoeg 1988) performs a similar exchange. When it receives this same information from the other collectors, the local collector learns about remote references to its local pages. This information feeds another marking cycle that expands the local collector's set of accessible pages. The exchange of information keeps the collector cycles synchronized, but the mutators need not halt. The mark phase ends when, after an exchange of information, no collector expands its accessible page set.

A concurrent garbage collector requires some form of mutator cooperation to mark the reachable objects properly; the goal of "no extra overhead for the mutator" is unattainable (Dijkstra *et al.* 1978). In *EZ*, mutator cooperation is obtained through use of the virtual-memory system as explained in Section 5.3; alternatively, it could be obtained by using the virtual-memory hardware of the processors. The mutator processes themselves need not be modified in any way.

During the sweep phase, the collector must check whether or not each local page is marked. The implementation, however, does not require access to the pages themselves. The collector accesses only its own local data structures, which fit in a relatively small number of pages, to determine which local pages are unmarked. The sweep phase requires no interprocess communication.

5.3 Invisible References

Without adequate protection, interleaving the execution of the collector with the mutators could produce some undesirable effects. 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.

Consider, for example, the three pages shown in Figure 12. Initially, page A has the only reference to page B. Suppose that page C has already been marked by the collector when the reference to B is copied into C. If A's reference is now removed before A is processed by the collector, B will not be marked because only C refers to it and C has already been processed. Therefore, at the end of the mark phase, B is unmarked and is reclaimed. The references to B have been effectively invisible to the collector, although B is never inaccessible during this process.

To avoid this problem, B must be marked atomically by the collector either when the reference was copied into C, or when A's reference was changed. The former case

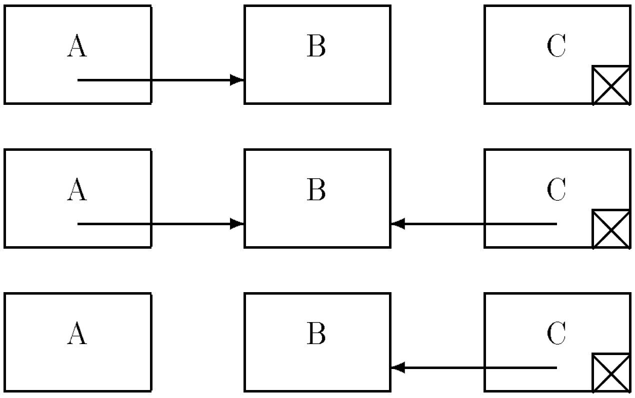

Figure 12: Occurrence of invisible references.

corresponds to marking the *new* targets of updates, the update to page C. The latter corresponds to marking the *old* targets of updates, the update to A.

Most concurrent collectors mark the new target to avoid hanging on to the page holding the old target (Dijkstra *et al.* 1978). Their main concern is to avoid floating garbage: pages that become unreachable during a collection, but are not reclaimed until a later collection (Dawson 1982).

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 the virtual-memory system to mark the pages referenced by a page before it is modified; alternatively, virtual-memory hardware could be used (Appel and Li 1991). At the start of a collection, all owned pages are set to read-only. The first write to a page causes a page fault, and pages referenced from 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.

With this scheme, no direct mutator assistance is required by the collector. The virtual-memory manager accomplishes the desired effect. However, this approach implies that garbage produced during one collection will be collected only in the next collection, as in the distal-objects collector (Rudalics 1986).

5.4 The Collection Algorithm

At every moment, each memory page is marked with one of three possible colors: 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.

Figure 13 shows pseudo-code for *EZ*'s garbage-collection algorithm. The remainder of this section relies on this figure to describe the algorithm.

At the beginning of each collection, locally owned pages have the same color, say, white. After the mark phase, pages are colored either white or black, and white pages can be reclaimed. Initially, `retain` and `gather` are black and white, respectively. The colors reverse roles in subsequent collections.

Each collection starts by setting owned pages to be read-only so that their referents are marked if they are modified, as explained in Section 5.3. Writes to these pages are caught at page faults or calls to `getPage`, and the fault handler or memory manager `scan`s the requested page before granting the access. This action is the only interaction between the mutators and the collector. As shown in Figure 13, scanning a page `shade`s its referents, colors the page black, and unprotects it.

Then, the referents of the local roots are colored by `Shade`, which colors owned white pages grey and adds remote pages to its argument `s`. When shading the referents of the local roots, the argument is the set `m`, which is empty initially and is used by the algorithm to collect all remote pages that are locally referenced and should, therefore, be marked.

Next, the mark phase 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 the local roots have been colored black, and `m` contains all remote pages that should have been colored grey.

A subset of `m` is then broadcast to the other processors. This subset is the set of pages that have not been announced in previous broadcasts. Variable `cycle` records the size of this subset, and `m` is added to `M`, which accumulates all accessible pages

```
retain, gather ← BLACK, WHITE
do forever
    M ← m ← ∅
    for every p ∈ owned do Access(p) ← READ_ONLY
    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


Scan(p, s):
    for every reference r ∈ p do Shade(Page(r), s)
    Color(p) ← retain
    Access(p) ← READ_WRITE


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

Figure 13: Garbage-collection algorithm.

in the address space. The collector then consumes similar messages from the other collectors, accumulates their sizes in `cycle`, and shades the pages mentioned in the messages. When shading the pages mentioned by other collectors, `Shade` colors owned white pages grey and adds remote pages to the set `M`.

Messages from other collectors reveal information about local pages that are referenced remotely and feeds another marking cycle, just as if the root set had been augmented. The new cycle might expand the local collector's set of accessible pages or its set of interprocessor references. The messages not only communicate the remote references between collectors, but also synchronize them and ensure that they cycle in lock-step.

As the mark phase 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.

When a page migrates, it is scanned *before* being sent to its new owner, if necessary. The memory manager triggers the scan by simulating a local write, which causes the page to be scanned as described in Section 5.3. The new owner colors the page black. The memory manager also collaborates to the garbage-collection process when allocating free pages; they are colored black before they are allocated.

No white pages are created during a collection. All accessible white pages are scanned either because there is a local reference to them from another accessible page or because there is a reference from a remote accessible page. Thus, all accessible pages are colored black after some number of cycles. Remote references are revealed by the messages exchanged by the collectors at the end of each cycle or by receiving a remote write request for the page. When all accessible pages have been colored black, the messages sent by the collectors are empty. All collectors see the same messages after each cycle of the mark phase. Therefore, all of them terminate the mark phase after the same cycle when they all receive empty messages. The sweep phase is local and requires no coordination between collectors.

5.5 Implementation

The implementation of the collector for distributed *EZ* consists of about 700 lines of ANSI C. There is a single thread per processor for the garbage-collection proper, but every message received from the other collectors is processed by an independent thread. In addition, threads from the virtual-memory manager and interpreter threads call collector routines.

The collector thread pauses periodically to reduce the time it interrupts the mutators. The mark phase pauses after shading each of the local roots, and after each grey page is scanned. When sending the messages to the other collectors at the end of each cycle, the collector pauses after broadcasting each set of references that fits in a page. The sweep phase pauses after determining the free local pages in each page of the bit map for the set of owned pages.

The only remote communication induced by garbage collection is the message exchange with the other collectors that takes place at the end of each cycle in the mark phase. There is no migration of pages or remote accesses due to garbage collection. The collector thread does get read access to local pages in order to scan them.

The collector uses reasonably efficient representations for all its data structures, which are kept in the persistent address space. The set of available pages, `free_pages`, is represented as a list of arrays of page ranges stored in pages as in UNIX (Thompson 1978). The list is accessed as a LIFO list; thus, the first page is often in the memory manager's cache.

Page sets for `k` and `m` are implemented as lists of pages containing page numbers. Management of this data structures is simple because they are merely bags of page numbers in which pages may appear more than once. The implementation is similar to that for the available-page set, but the arrays store individual page numbers instead of page ranges.

The implementation of marks differs slightly from the pseudo-code shown in Figure 13. Marking pages with three different colors is wasteful because it requires two bits per page, although two bits can distinguish four different states. Instead of being colored, pages are simply marked or not. A private bit map, similar to the one used for the implementation of the set of owned pages, is used by each collector to keep the marks for all pages in the address space. A bit is set in the bit map if the respective page is marked, i.e., it has been processed; the bit is clear if the page has not been scanned. A marked page is equivalent to a page that has been colored black. An unmarked page, which has its bit clear, can be either white or grey.

In the algorithm, grey pages are those that have been recognized as accessible, but not yet scanned. The implementation keeps a bag with the local grey pages, in the same way it keeps the bag, `m`, of remote pages that are locally referenced. When a page from this bag is scanned, the collector also sets its mark bit. Since only unmarked pages are scanned, no page is scanned more than once, even if they appear again in the bag.

The bit map for marked pages is also used to mark remote pages that, through the messages received from other collectors, are known to be accessible. With this extension, the bit map also represents the set `M` in the pseudo-code, which need not be implemented separately.

The implementation makes `Shade` simpler and faster because it does not have to determine the color of owned pages. When the collector shades a page, it inserts the page's number in one of two bags depending on whether or not the page is local. It inserts local pages into the bag of grey pages and remote pages into the bag, `m`, of remote pages that are locally referenced. Remote pages can also be marked immediately.

Pages listed in a message received from another collector, are treated similarly. Local pages are inserted into the bag of grey pages and remote pages are simply marked.

The set of owned pages, the set of available pages, and the per-page data are shared by the collector and the memory manager with appropriate locking. The data structures private to the collector — the bit map with marks, the bag of grey pages, and the bag of remote pages that are locally referenced — are not re-used, but allocated each time they are needed. The garbage collector simply loses their addresses at the end of a collection, and they become garbage to be collected by a subsequent collection.

This mechanism has several benefits. No re-initialization phase is necessary before being re-used, which would create synchronization hazards with the mutator threads and the threads processing messages from the other collectors. More importantly, the collector need not process these structures because their pages are marked when allocated and cannot be reclaimed by the current collection, and because they do not refer to other pages; thus, garbage-collection time is reduced. Finally, it eliminates the implementation problems that appear when the structures are processed by the collector itself due to mutual recursion among different part of the system.

Chapter 6

System Performance

The design and implementation of distributed *EZ* is based on a number of assumptions about the typical behavior of *EZ* programs. The purpose of the system is to share values among different users; it is *not* meant to be a multiprocessor for speeding up parallel algorithms.

Paramount is the assumption that communication between processors is expensive. For the intended architecture, a network of work-stations, this fact is well known. The system has no control over the network; it cannot, for example, change the communication speed.

Migration of pages and of their ownership is based on the assumption that a processor that writes a page tends to write it again. Migration to the writer should therefore reduce communication overhead. Most pages would move infrequently, and the cost of one migration would be amortized over several local accesses.

A similar assumption holds for read access; a page that is accessed once is likely to be accessed again soon. The local, in-cache copies reduce the number of network messages when their reference counts are positive, and allow multiple simultaneous read accesses at different processors.

Distinguishing copy messages from read messages and distinguishing fetch messages from write messages is an attempt to reduce communication costs by reducing the amount of information transmitted; the pages themselves are sent only when necessary.

An important design decision was to use confirmation instead of invalidation to maintain coherence. There was no *a priori* assumption on how well this method would work in practice. The hope was that pages would eventually migrate to a 'home' location and, therefore, few confirmation messages would be needed. Early results showed that read-only pages defeat this intuition. In particular, pages with virtual-machine instructions — code blocks — generate read requests that are sent only as confirmation because the in-cache copies are always up to date. The realization that these pages would never migrate suggested adding execute access to the original read and write alternatives. As it turns out, using execute instead of read access for code blocks saves between 10% and 20% of the execution time when the system runs in more than one processor. This savings come from reduced communication between managers; they no longer must confirm the validity of the copy of a page that cannot have changed.

The distributed garbage-collection algorithm assumes that most data structures migrate in their entirety when one of their pages migrates. Thus, the mark phase needs few cycles to process all accessible pages. In turn, fewer cycles means less network traffic induced by the garbage-collection process.

Another assumption is that most pages are scanned by the garbage-collector thread, not by the mutators. Although scanning a page takes a small amount of time, the hope is that few mutator accesses would be delayed by the scanning of a page.

This chapter assesses the performance of distributed *EZ* and attempts to determine the validity of the above stated assumptions. A prototype has been implemented on a SPARC processor — a Sun 4 work-station — running the Sun Operating System version 1, release 4.1.1-GF. The prototype runs on a single processor and uses UNIX processes to simulate different processors.

All measurements for the test programs include both the compilation and the interpretation of the compiled code and other processing inherent to the *EZ* system. The latter corresponds to inserting the processes in the process table and scheduling them on the processors. The times shown are CPU times and include both user and system time. In all cases, the page size is 512 bytes.

	Average	Minimum
`allocPage`	0.257	0.167
`getPage` (same page in cache)	0.407	0.383
`putPage`	0.042	0.033

Table 4: Times for interface functions in milliseconds.

6.1 Calls to Interface Functions

This section characterizes the CPU time used by the virtual-memory manager to satisfy requests under different circumstances. Measurements were made for 1000 calls to the interface functions. In every case, the running time was measured 12 times, the largest and smallest values were discarded, and the average of the remaining 10 values was computed. Both the average and the smallest discarded values are presented; the minimum value indicates the variability of the measurements. The values shown in Tables 4 and 5 are the times spent per access, i.e., the total time divided by 1000, and are expressed in milliseconds.

Table 4 shows the time spent by calls to each of the three interface functions. Every call to `allocPage` allocates a different page; the cache is big enough to hold all these pages, so no pages are written to secondary storage. The calls to `getPage` and `putPage` are for the same page every time; the page is local and in the cache.

Calls to `putPage` are the fastest because they only post the requests in the appropriate service queue and return immediately. There is no difference between releasing a dirty or a clean page; likewise, there is no difference when the calls are for different pages instead of the same. The pages must be in the cache because to release them the interpreter must first have access to them.

Allocation takes longer. A call to `allocPage` is also executed without yielding the processor to other threads. But `allocPage` has to get one element from the set of free pages — which requires access to another page — and has to choose a cache entry and initialize it; the latter includes clearing the physical-memory frame.

The most time-consuming function is `getPage`, even in this simple case in which the same local page is requested repeatedly and it is in the cache. The calls post the requests in the service queue as do the calls to `putPage`, but `getPage` then waits for

Page Condition	Average	Minimum
Same local page, in cache	0.450	0.433
Different local pages, in cache	1.180	1.133
Pages read from disk, no write	1.407	1.367
Pages read, frames written	1.640	1.533
Remote page in local cache	65.681	61.698
Remote pages not in local cache	70.144	64.364

Table 5: Times for accessing pages in milliseconds.

the request to be serviced by the respective per-page thread of the memory manager. In general, many threads may be active in the system, but when taking these measures there were only three threads: one making the calls, another servicing the requests, and a third listening to the network.

In summary, Table 4 shows that the simplest call to `getPage` takes about 60% longer than a call to `allocPage`, and about eight and a half times longer than a call to `putPage`.

The time spent in a call to `getPage` depends most on how the manager gets its in-cache copy, which determines how long the thread making the call has to wait for the request to be serviced. Table 5 shows the times spent under different conditions. All times shown in this table include both the calls to `getPage` and the corresponding calls to `putPage`.

The first row of Table 5 shows that for local pages that are already in the cache, repeated access to the same page takes the time expected from adding the values for `getPage` and `putPage` shown in Table 4. When the access is to different pages, the running time increases by a factor of 2.6.

When accessing local pages that are not in the cache, the pages' contents are read from disk, and the page is tested for membership in the set of owned pages. This extra processing increases the running time about 20%, or a little less that 0.23 milliseconds per page. If, in addition, the frame's contents are written to disk, the time increases about another 0.23 milliseconds per page.

Access to remote pages is much more expensive, as shown by the last two rows of Table 5. In both cases, the pages are initially in the owner's cache and no time is spent caching the owner's copy. Access to a remote page is 55 times slower than to a local page when there is a local in-cache copy; the acknowledgment does not include a copy in this case. When there is no local copy, the owner also sends a copy of the page, which increases the running time by less than 7%, or about 4.5 milliseconds per page. The time taken by remote requests is mostly due to interprocessor communication costs and is beyond the control of the system. A comparison with the time needed to access local, in-cache pages shows that over 96% of the time required by remote accesses is due to communication overhead.

6.2 Test Programs

Four *EZ* programs have been used to test the system and gather data about its performance. They are small sequential programs that are easy to understand, but exercise the system and run long enough to allow comparisons.

The first program, `hanoi`, defines a procedure that recursively solves the 'towers-of-hanoi' problem for n disks. It uses no global variables and the code is small. The test program calls this procedure to solve the four-disk problem. Its output is the sequence of disks movements, printed one per line.

The second program, `factorial`, defines a procedure that recursively computes the factorial of a non-negative integer n. Like `hanoi`, it uses no global variables, and the code is small. The test calls this procedure in a loop that computes the factorial of all integers from 0 to 13; the loop uses a global variable as its index, and prints the sequence of integer-factorial pairs one pair per line.

The next program, `random`, defines a procedure that uses the linear congruential method (Knuth 1981) to produce a pseudo-random positive integer smaller than its argument. The code is small and requires only a few arithmetic operations. The test program uses this procedure to compute a sequence of 60 pseudo-random numbers; each time, the argument is the corresponding iteration number. The output is the sequence of iteration and random-number pairs, printed one pair per line. Except for the argument to the procedure, all other values are kept in global variables, including all the constants and variables for the linear congruential method and the loop index.

	`Hanoi`	`Factorial`	`Random`	`Queens`
Allocations	184	492	333	327
Read requests	2926	6501	15206	29452
Write requests	1314	2912	4806	9962
Execute requests	174	376	557	1374
Pages modified	1445	3272	4921	9052
Time, in seconds	3	6	14	27

Table 6: Accesses to manager for one processor.

Finally, `queens` defines procedures that generate all solutions to the n-queens problem using backtracking (Hanson 1976a); the board size, n, is given as an argument. The code is larger than that of the previous tests, makes many recursive calls, and uses associative tables. The output is a sequence of n-number series; each series, printed on a separate line, represents one solution. Most values are local, but several arrays, i.e., tables with integer indices, are global variables; these arrays store the board information, which includes both a partial solution with the queens' positions, and the squares 'covered' by them.

Table 6 summarizes how the test programs use the memory manager and also includes their running times. The first row shows the number of calls to `allocPage`. The next three rows show the number of calls to `getPage` segregated according to the type of access requested. The fifth row counts the calls to `putPage` with dirty pages. Finally, the last row shows the total CPU time used by the programs. All values correspond to the system running in one processor and with a cache large enough — 1024 frames — that none of the programs required input from or output to secondary storage.

The running times in Table 6 show that each program takes approximately twice as long to run as the previous one. Similarly, the programs are increasingly more demanding in the number of pages that they request from the memory manager. Again, each one is roughly twice as demanding as the previous one. However, they all use about the same number of pages, as indicated by the number of allocations.

The numbers in Table 6 show that read accesses, which include both read and execute requests, account for at least 67% of the total number of accesses, which include calls to `allocPage` and `getPage`. The proportion of read accesses is even larger, 75%, for the two larger tests, `random` and `queens`.

About 5% of all read accesses are execute requests. The number of execute requests counts accesses to pages with virtual-machine instructions, which are the only pages requested with this access mode.

Most writable pages are returned modified; they include all newly allocated pages, which are always modified. For the first three programs, about 96% of the writable pages are modified. However, `queens` modifies only 88% of its writable pages; the difference is attributable to the use of tables for its data structures, as opposed to only activations and system tables. Table operations gain write access to some pages in order to lock them and guarantee atomicity, but do not always modify them; therefore, a smaller proportion of the writable pages are actually modified.

6.3 Effects of a Small Cache

Table 7 shows the effects of using a small cache in a single processor. The values in this table were obtained for a cache with 128 frames of physical memory.

There is no appreciable difference in the running times for the test programs with respect to those obtained for the bigger cache, which are shown in Table 6. Only `hanoi` shows a one-second increase in its running time, but this increase is within the precision of the measurements and is thus insignificant.

The real difference, as expected, is that the smaller cache fills up and pages have to be saved into secondary storage. Moving pages into disk implies that some of them have to be retrieved later when they are needed again. Newly allocated pages occupy a cache entry and might trigger writing the frame's contents to disk; however, they are not read. Allocation, then, explains why there are more pages written to than pages read from disk.

Although the cache size is below the number of pages used by the programs, even when not counting the internal pages needed by the manager itself, the disk activity is low. Few pages need to be written to disk. For each test, a page is written back, on average, less than 3.5% of the time a page is returned modified, except in the case

	`Hanoi`	`Factorial`	`Random`	`Queens`
Pages read	11	136	10	17
Pages written	53	343	173	178
Time, in seconds	4	6	14	27

Table 7: Disk activity with small cache.

of `factorial` for which this figure is about 10%. `Factorial` allocates more pages than the other programs, and these pages are used almost exclusively for activations; they are modified, but not used after the corresponding incarnation of the recursive procedure returns.

Even fewer pages are read from disk. For each test, a page is read from disk, on average, less that 0.24% of the time a page is requested for access, except again in the case of `factorial` for which this figure is about 1.3%, which is still small. The difference is explained because the many activations used by `factorial` occupy cache entries and force other pages out of the cache, although they might be needed again. For `random` and `queens`, an average number of fewer than 0.05% of pages are read for every access requested, a tiny proportion of the memory activity.

6.4 Several Processors

The tests run on more than one processor to evaluate the effects of sharing. In all cases, they are compiled in processor 1 and interpreted by processor 2. When using four processors, processors 3 and 4 do not actively contribute to the execution, but their memory managers process requests from the other processors. All results are for a 1024-frame cache.

Table 8 shows the use of the memory manager by the test programs when running in two processors and their running times. Most of the activity occurs in processor 2, where the programs run. The activity in processor 1 corresponds to system activity, the program's compilation, and the memory manager responding to requests from the other processor; no execute requests indicates that there is no local interpretation. There is a slight increase in the activity at processor 1 as the complexity of the test program increases, reflecting the increased needs of the compilation. Although

	`Hanoi`		`Factorial`		`Random`		`Queens`	
	Pr. 1	Pr. 2	Pr. 1	Pr. 2	Pr. 1	Pr. 2	Pr. 1	Pr. 2
Allocations	25	160	27	466	29	306	37	293
Read requests	160	2755	209	6329	357	14831	360	29234
Write requests	102	1235	117	2824	169	4660	186	9819
Execute requests	0	174	0	376	0	557	0	1374
Pages modified	116	1346	130	3165	176	4763	203	8889
Time, in seconds	51	52	102	104	424	432	638	649

Table 8: Accesses to manager for two processors.

the activity in processor 1 is small, the processor is running for almost as long as processor 2 does, which implies that there are enough requests to keep it active for about 98% of the time.

The running time increases between 17 and 30 times with respect to one processor (see Table 6). For `hanoi` and `factorial` this figure is 17, for `queens` it is 24, and for `random` it is 30. Most of the increase is due to interprocessor communication costs; in particular, it is due to the many remote read requests from processor 2 to processor 1. The first row of Table 9 shows the number of remote read requests sent from the processors.

There are two main reasons for the large number of remote read requests. First, access to constant values, which include pages that contain the character strings used by the system and the programs, and pages with constant values generated by the compiler; like code blocks, these pages are never written again and do not

	Processor 1	Processor 2			
		`Hanoi`	`Factorial`	`Random`	`Queens`
Read	1	579	1122	4878	7360
Write	2	12	13	13	13
Copy	2	1	1	1	1
Fetch	1	2	2	2	2
Broadcast read	0	29	30	34	39
Broadcast write	0	4	4	4	6
Decrement	3	609	1153	4913	7400
Pages received	3	38	39	43	50

Table 9: Remote requests for two processors.

migrate. Second, intermediate pages used for indirection in the implementation of tables; these pages are seldom written when elements are inserted into the tables, but have to be read to find the addresses of the pages where the new values are actually stored; consequently, these pages seldom migrate, but are read each time the table is accessed.

In the case of `factorial`, for instance, access to only two pages with constant values accounts for 739 or 66% of the remote read requests from processor 2. When these pages are moved to processor 2 before interpretation begins, the running time drops from 104 to 40 seconds. Another single page in the mapping table for the root table accounts for another 178 remote read requests; when it is moved to processor 2 before interpretation, the running time drops to 25 seconds. In summary, three pages account for 82% of the remote read requests from processor 2, which in turn account for 76% of the program's running time.

The increase in running time with respect to one processor is larger for `queens` and `random` because these tests access global tables proportionally more than the others. Therefore, they emit more requests for intermediate pages of global tables, which are created at compile time by processor 1, and used later by processor 2. `Random`'s use of global variables and, therefore, of the root table is the highest, and it presents the largest increase in running time.

The values in Table 8 indicate that the test programs show the same general behavior in two processors as in the uniprocessor (see Table 6). When adding the requests made by both processors, the numbers are similar to the one-processor case. The relative proportions are also as in the uniprocessor case. There are small differences, however, for the number of allocations, read and write requests, and pages modified; these occur because there is some system activity that occurs at each processor. And, more importantly, because being a multi-threaded system some activity might not be finished at the time of output, and because the interleaving of the threads and the processors can produce different numbers.

One case seems clear, however. As expected, the number of execute requests is zero for processor 1, and the same number as in the uniprocessor case for processor 2; interpretation occurs only at processor 2 and is for the same program. However, this simple explanation is deceiving; there is no guarantee that the same number of execute requests would be issued by both processor 2 and the uniprocessor system. The threads that interpret the programs execute a random number of instructions before pausing, and after each pause they must get the page with instructions again; therefore, the total number of execute requests depends not only on the program being executed, but also on the number of times that these threads pause. If the latter vary, the number of execute requests might well be different.

Table 9 shows the number of remote requests that each processor sends, segregated by request type; the last row shows the number of copies actually received from the owner. Processor 1 sends only a few remote messages in every case; the table shows typical values.

Comparison with the values in Table 8 shows that most requests are for local pages. In the case of write access, the total of remote write, fetch, and broadcast-write requests is less than 1.5% of the total number of write requests. In fact, this proportion is smaller than 0.7% for the three larger programs. Adding allocations to the write requests, to use all write accesses as the base, makes the percentages even smaller; allocations have not been included because they never cause remote requests, although they are part of the write accesses.

In the case of read access, the total of remote read, copy, and broadcast-read requests is less than 24% of the total number of read accesses, which includes both read and execute requests. The exception is `random`, for which this figure is about 32%, still comparatively small. Even though the proportion is small, the number itself is still large. As discussed above, this large number is mainly due to pages that are read-only or seldom written and that, therefore, do not migrate.

The number of decrement requests is equal to the total number of remote read, copy, and broadcast-read requests. Each time a manager gets read access to a page from another manager, one decrement request is sent eventually, even if the requestor grants local access more than once. Messages for write access do not contribute to the number of decrement requests because ownership is transferred to the requestor. There is no guarantee, however, that this invariant will always show in the collected numbers because of the interleaving of threads and processors.

	Hanoi	Factorial	Random	Queens
Processor 1	52	103	425	637
Processor 2	52	104	433	653
Processor 3	9	7	9	11
Processor 4	8	8	8	12

Table 10: Running time for four processors in seconds.

The number of pages sent through the network is small compared to the total number of remote requests; it is even smaller when compared to the total number of accesses sought by the processor. Less than 6% of the remote messages result in sending copies of the pages. This figure is smaller than 0.9% for the two larger programs. Also, most copies are received when the requestor asks for a copy; fewer than 5% of these pages are sent in response to a remote read or write request.

Another important observation to make, although not shown by the tables, is that no message waits in the queues of pending requests. This observation suggests that each access to a page interferes little with other accesses to the same page.

The tests were also run on four processors, although processors 3 and 4 do not actively contribute to the execution of the programs. The activity of processors 3 and 4 is due to system activity, and helps evaluate the cost of responding to broadcast requests and to messages sent to managers that no longer own the requested pages.

Table 10 shows the running times for the test programs when they run in four processors. The times for processors 1 and 2 are essentially the same as when the tests run in only two processors (see Table 8). However, the running times for processors 3 and 4 are much smaller because they do not contribute to the programs' execution.

The running times for processors 3 and 4 come from system activity only, and from the local managers responding to requests from the other managers. The latter activity is small because most pages are allocated and used by processors 1 and 2.

Processors 3 and 4 are equivalent to each other; there is no 'preference' in the system for either of them. The almost identical running time they present in every test confirms this equivalence. Recall that the numbers can hardly be exactly the same because of the multi-threaded nature of the system. Also, there is no significant difference from one test program to another.

	`Hanoi`		`Factorial`		`Random`		`Queens`	
	Pr. 3	Pr. 4	Pr. 3	Pr. 4	Pr. 3	Pr. 4	Pr. 3	Pr. 4
Allocations	3	3	3	3	3	3	3	3
Read requests	23	27	23	31	23	31	31	27
Write requests	25	27	25	29	25	29	29	27
Execute requests	0	0	0	0	0	0	0	0
Pages modified	25	24	25	25	25	25	27	24

Table 11: Accesses to manager for four processors.

The use of virtual memory by the test programs when running in four processors is described by the values in Table 11. Only the values for processors 3 and 4 are shown because the values for processors 1 and 2 are essentially the same as in the case of two processors. The number of accesses in the table is a measure of system activity because the manager itself does not call the interface functions.

Table 11, too, shows how little activity these processors have, and that there is no significant difference between them and from one test to another. Very few pages are allocated at these processors. There is a disproportionately large number of write requests, which correspond to system activity, but still only about 50% of all requests. Pages requested for write access are modified most of the time.

For completeness, all remote requests made by processors 3 and 4 are shown in Table 12. The numbers for processors 1 and 2 are similar to those in the two-processor case, and are omitted.

	`Hanoi`		`Factorial`		`Random`		`Queens`	
	Pr. 3	Pr. 4	Pr. 3	Pr. 4	Pr. 3	Pr. 4	Pr. 3	Pr. 4
Read	5	6	5	7	5	7	7	6
Write	5	5	6	7	6	6	7	5
Copy	1	2	2	4	2	3	5	2
Fetch	6	2	3	3	6	4	3	2
Broadcast read	8	8	8	8	8	8	8	8
Broadcast write	1	1	1	1	1	1	0	1
Decrement	14	16	15	19	15	18	20	16
Pages received	16	13	14	17	17	17	16	15

Table 12: Remote requests for four processors.

Processors 3 and 4 show a larger proportion of requests for remote pages than processors 1 and 2. These accesses are mainly to the global process table, which is created at processor 1. There is little opportunity for access to local pages because only a small number of pages are allocated locally, and because there is not enough activity to make more use of the cached pages.

Only a handful of requests wait in the queue of pending requests. Although there is some waiting when using four processors, the number of requests that wait does not exceed 10; each access to a page interferes little with other accesses to the same page.

Similarly, only a few requests must be forwarded to new owners of pages. Only processors 3 and 4 are mistaken as the owners of pages that they no longer own; the other two processors never forward a request. The number of forwarded request messages is smaller than 5 — usually only 1 or 2 — and the identity of the page's new owner is always known by the old owner; no broadcast messages are needed for this purpose.

6.5 A Parallel Program

A simple parallel program, `matrix`, assesses the behavior of distributed *EZ* under a different condition. All processors contribute to the execution of the program and all attempt simultaneous access to some shared data structures. The test was run using four processors, each with a 1024-frame cache.

The program implements a parallel matrix-multiplication algorithm that computes the product of two square, $n \times n$, integer matrices. Each element of the result matrix is computed by a separate *EZ* process that performs the dot product between one row of the first operand and one column of the second. The n^2 dot-product processes run in parallel.

Both operands and the result matrix are global tables, as is a global semaphore used to synchronize the processes. All other variables in the program are local. The test program uses the algorithm to multiply two 4×4 matrices; its output is the product matrix, printed one row per line.

In the actual program implementing the algorithm, the original process executes the following code.

```
mutex.count = 1 - n * n
for (i = 1; i <= n; i = i + 1)
    for (j = 1; j <= n; j = j + 1)
        create(dotprod, n, i, j)
P(mutex)
print(n)
```

The original process sets the `count` entry of semaphore `mutex` to be $1 - n^2$, creates the dot-product processes, and then calls `P` on the semaphore. After finishing its computation, each dot-product process calls `V` on the semaphore and dies. When all dot-product processes have finished and, therefore, all elements of the result matrix have been computed, the original process continues execution and prints the result.

Both operand tables are accessed by each dot-product process, which reads n elements from each of them. Each dot product is written into the result matrix only once at the end of the respective process just before it calls `V`. The global semaphore is accessed once by each dot-product process and twice by the original process; write access to some of its pages is needed each time.

As in the sequential tests, the program is compiled at processor 1 and the original process starts running in processor 2. The n^2 dot-product processes — 16 for the test — are run by the system in all processors available. The last four columns of Table 13 show the use of the memory manager at each of the four processors. For comparison, the table's first column shows the results obtained when running the program in a single processor. The number of execute requests reveals that all processors actively contribute to the execution of the program; although the activity is not exactly the same among all of them.

Table 13 shows, once again, that the running time exhibited by each processor is largely independent of the amount of local activity. All processors tend to run for a similar amount of time. Processor 4, which has the smallest running time, still runs 87% of the time that the longest running processor, processor 1, runs.

With the parallel implementation of the algorithm, the time spent at all processors contributes to the execution of the program. The running time in four processors is

	Uniprocessor	Pr. 1	Pr. 2	Pr. 3	Pr. 4
Allocations	1161	418	271	143	333
Read requests	20716	14809	3696	3141	6693
Write requests	8683	6547	1959	1262	2591
Execute requests	985	422	81	138	316
Pending requests	182	373	352	231	212
Pages modified	9089	5123	1864	1243	2710
Time, in seconds	19	137	121	136	120

Table 13: Accesses to manager for matrix multiplication.

6.2 times longer than in the uniprocessor system. This number contrasts with the much larger factors — between 17 and 30 — for the sequential test programs. The increase in running time is smaller for `matrix` because, while a process waits for a request to be satisfied, the processor can execute other processes.

The parallel implementation can be compared with a sequential program that performs the same task by simply calling procedure `dotprod` repeatedly. The sequential version runs for 18 seconds in the uniprocessor system; the parallel program is 5% slower and runs for 19 seconds. The small increase in running time for the parallel version measures the cost of creating and synchronizing its several processes.

The parallel version was also run on systems with up to four processors. Figure 14 shows values for the program's speed; they are normalized so that the sequential program running on the uniprocessor has a speed of 1. The values shown in parentheses are the running times in seconds. The drop in speed between one and two processors shows, once again, that communication costs dominate. There is, however, a slight increase in speed when the number of processors is increased above two.

Comparing the total number of accesses in four processors with the uniprocessor case, as shown in Table 13, the number of pages allocated and the number of execute requests are similar, as predicted by the experience with the other tests. The number of read and write requests increases about 37% and 42%, respectively, with respect to the uniprocessor case. Similarly, the number of pages that are modified increases about 20%. These larger figures reflect increased activity in the system, which must schedule processes in all processors; in effect, it has to move the processes from the processor where they are created to the one where they are interpreted.

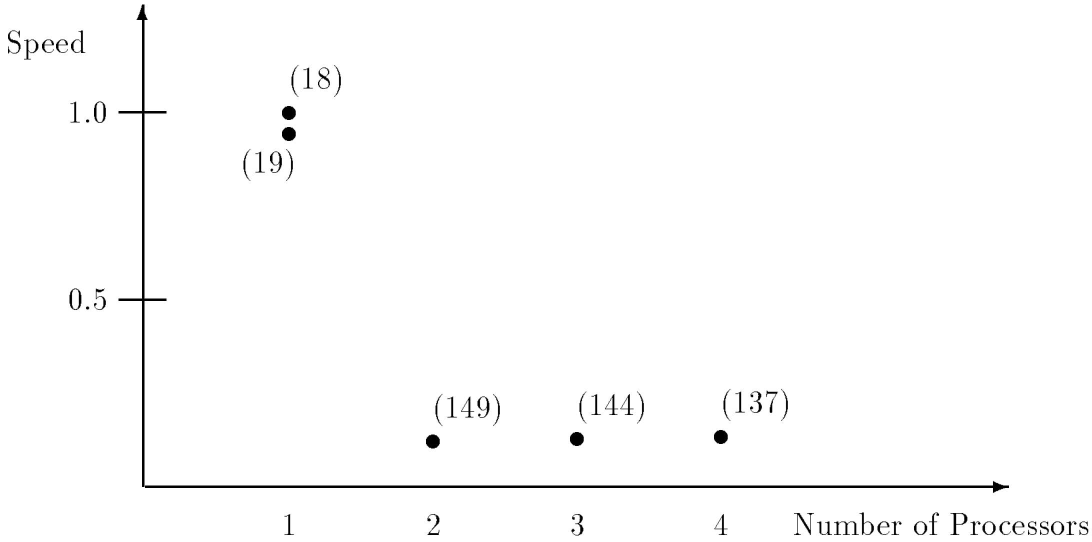

Figure 14: Speed of matrix multiplication.

Analyzing the totals for the four processors, the numbers show that the use of the memory manager is similar to the use made by the sequential programs. Read accesses, which include both read and execute requests account for about 68% of the total number of accesses, which include calls to both `allocPage` and `getPage`. About 3% of all read accesses are execute requests. Despite the increase in the number of pages modified, only about 80% of the writable pages are modified; this number is smaller that for the previous tests, and it reflects `matrix`'s more extensive use of tables, as explained at the end of Section 6.2.

One difference from the sequential programs is that, for `matrix`, more requests wait in the pending request queue. The fifth row of Table 13 shows the number of requests that could not be processed immediately and had to wait; this row is not included in previous tables because the numbers are zero or very small. Although there is an increase of 5.4 times from the uniprocessor case, the waiting requests represent fewer than 2.8% of the total number of requests. Each access to a page interferes with other accesses to the same page a little more than for the sequential programs, but the number of waiting requests is still small.

Although the numbers are not shown in the table, there was no disk activity when running `matrix` in four processors. However, the 1024-frame cache was not

	Pr. 1	Pr. 2	Pr. 3	Pr. 4
Read	811	756	812	1465
Write	165	156	113	113
Copy	231	227	151	137
Fetch	173	169	162	108
Broadcast read	81	39	70	119
Broadcast write	0	6	5	0
Decrement	1040	941	975	1662
Pages received	472	447	387	387
Copy forwarded	83	81	58	59
Fetch forwarded	39	50	49	31

Table 14: Remote requests for matrix multiplication.

enough in the uniprocessor case, which read 13 pages and wrote 200; too few to affect the running time. The multiprocessor system uses more memory due to replication and more activity, but it avoids using secondary storage because the four processors provide, in effect, four times as much physical memory as the uniprocessor.

Table 14 shows the number of remote requests that each processor sends when running `matrix`. Most requests are for local pages. In the case of write accesses, less than 9.5% of the total number of write requests generate remote messages. For the sequential programs, this number is about 1.5% or smaller, which shows that there is more interaction in the parallel program because all processes access the result matrix and the global semaphore, which they all have to write.

The write accesses imply migration of pages, and therefore a smaller proportion of the read accesses generate remote messages than for the sequential programs. The total number of read, copy, and broadcast-read remote messages is less than 17% of the total number of read accesses, which includes both read and execute requests. For the sequential programs, this figure is between 24% and 32%.

About 27% of the remote messages included a copy of the page; for the sequential programs this number is 6%. This difference is the result of more pages migrating among the different processors because they are needed at all processors. However, over 99% of these pages are received when the requestor asks for a copy; fewer than 1% are sent in acknowledgments to a remote read or write request.

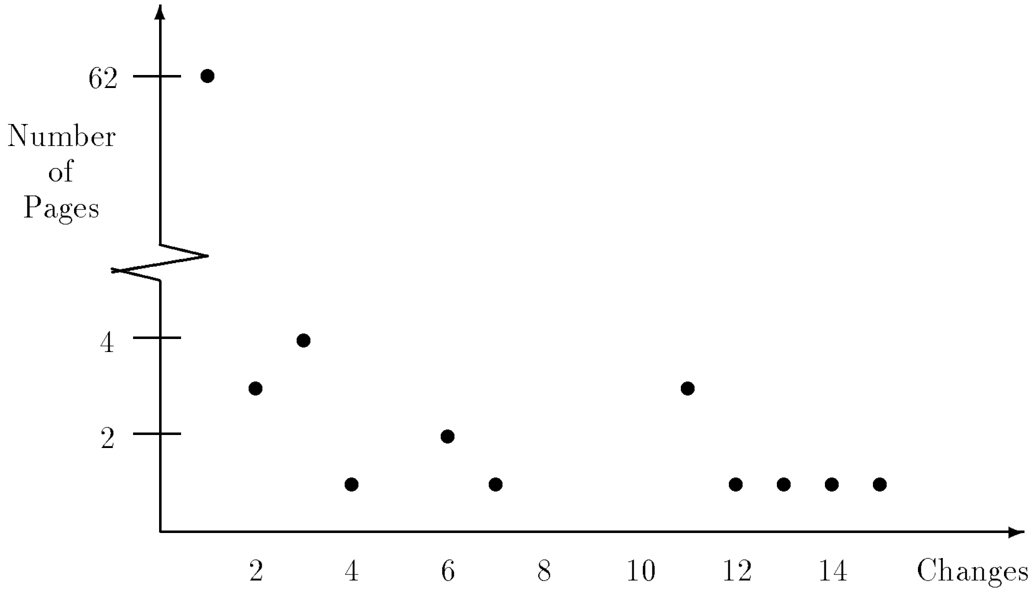

Figure 15: Number of ownership changes.

The last two rows of Table 14 show the number of messages forwarded by each processor; these numbers are not shown in the tables for the sequential programs because they are zero or very small. There is no need for broadcasts when forwarding, the old owner always knows the new owner. Requests for read access are forwarded 6% of the time. For write access, forwarding occurs 14.5% of the time because some pages needed for write access at one processor are the same ones that are needed for write access at the others and, therefore, migrate more often.

Figure 15 helps visualize the migration of pages. It plots the number of pages that change ownership versus the number of times they change ownership. Most pages, 62 of them, migrate once. Only 27 pages in all migrate more than once; 18 of them migrate fewer than 20 times.

6.6 Garbage-Collector Performance

This section evaluates the behavior of the distributed garbage collector. In order to allow several collections to take place, and to have enough activity to measure the effect of the collector, each of the sequential test programs was changed to execute five times (see Section 6.2). The change includes output, which gets repeated, and was accomplished by simply duplicating the main program's statements five times. The parallel test program, `matrix`, was not changed.

For garbage-collection purposes, all virtual pages used by the set of owned pages must be processed and must therefore be cached. These pages also help the collector determine which pages are local when it sweeps the address space to collect local free pages. The set of owned pages for the uniprocessor system uses an extra 1016 pages, which the manager reads from disk once for each test; no pages are written to disk, because the cache is big enough to hold all necessary pages. For these tests, a 4096-frame cache was used.

Table 15 shows some results obtained when running the system with garbage collection in a single processor. The first five rows in Table 15 show the use of the memory manager. There is no significant difference between these numbers and those obtained when not using the garbage collector, which are not shown. There is a small increase, however, in the number of read requests because the collector issues them to access the pages that it scans.

The next three rows show how many collections are performed during the tests, how many pages are scanned in total, and how many are scanned by the collector thread in particular. The difference between the last two is the number of pages scanned by the memory manager on behalf of the mutator processes. The last two rows of the table show the running time of the test programs with and without garbage collection, respectively.

The numbers show that the tests generate little accessible data. Only 16 pages are scanned for `hanoi`. The other sequential programs require between 65 and 80 pages scanned per collection. And the matrix-multiplication program requires scanning 111 pages per collection.

With garbage collection, the running time for the tests is the running time without garbage collection plus approximately 6 to 6.5 seconds per collection performed. Each collection takes about 6 to 6.5 seconds to execute for the number of scanned pages shown. This figure implies that the collector scans about 7 kilobytes per second, amortizing the constant cost of the sweep phase over the mark phase. The collector for

	Hanoi	Factorial	Random	Queens	Matrix
Allocations	806	2341	1541	1431	1157
Read requests	13201	32082	74546	146539	20682
Write requests	5998	14105	23417	49051	8649
Execute requests	843	1922	2759	7018	969
Pages modified	6635	15869	23999	44306	9054
Collections	1	4	3	2	2
Pages scanned	16	305	195	159	222
Collector scans	8	271	177	114	156
Time, in seconds	20	62	86	141	32
Time w/o collector	14	31	66	128	19

Table 15: Garbage collection in one processor.

Standard ML of New Jersey (Appel and MacQueen 1991), which is a two-generation, stop-and-collect collector, copies about 2 megabytes per second on the same processor.

With the exception of `hanoi`, which does not present much activity, more than 70% and as much as 90% of the scanned pages are scanned by the collector thread.

Two artificial programs were also used to test the system. The intention is to test the garbage collector under two different extreme behaviors of the mutators (Fernandez and Hanson 1992). One program, `neat`, generates almost no garbage; the other, `dirt`, generates almost only garbage. Neither program produces any output, and all variables they use are local.

Each of these programs defines a procedure that repeatedly creates a table by indexing variable `t`. `Dirt`'s procedure executes the following loop:

```
for (i = 0; i < n; i = i + 1) {
    t[0] = 0; t = 1
}
```

At each iteration, it creates table `t` and immediately loses access to the table's pages by assigning 1 to the same variable. All these pages become garbage. `Neat`'s procedure executes a similar loop:

```
for (i = 0; i < n; i = i + 1) {
    t[0] = a; a = t; t = 1
}
```

	`Dirt`	`Neat`
Allocations	4531	4531
Read requests	25990	42287
Write requests	17698	17675
Execute requests	2583	2558
Pages modified	22217	22194
Collections	9	7
Pages scanned	646	15639
Collector scans	635	15631
Time, in seconds	100	118
Time w/o collector	29	30

Table 16: Artificial programs in one processor.

where `a` is initially zero. The table created in an iteration becomes an element of the table created at the next iteration. All their pages remain accessible. The almost identical procedures do nothing else. The tests call their respective procedure to perform 1500 iterations.

Table 16 shows the measurements obtained for these artificial programs running in a single processor. Except for the read requests, both programs show almost identical behavior in their use of virtual memory. The similarity between both programs also shows in the time it takes to run them when the garbage collector is turned off. The difference in the number of read requests corresponds almost entirely to the number of pages scanned by the collector, which gets read access to them for that purpose. When the tests run with the collector turned off, the number of read requests is similar for both programs.

Both programs allocate 4531 pages, an average of three pages per iteration of the procedures. Clearly, `dirt` produces mostly garbage; only about 72 pages are scanned per collection on average; the rest is garbage. By contrast, `neat` produces almost no garbage; more than 2200 pages are scanned per collection on average.

The difference in the number of scanned pages translates into a difference in the time it takes for each collection to complete. For `dirt`, each collection takes about 7.9 seconds, which is similar to the times measured for the other tests. `Neat`, however, requires about 12.5 seconds per collection.

In turn, the difference in the time taken by each collection translates into a difference in the number of collections completed while the program runs. `Dirt` completes the nine collections that it requires because it allocates more than 4500 pages. By contrast, the collector barely keeps up with the speed at which `neat` allocates pages; it can only complete seven collections.

For both programs, most accessible pages are scanned by the collector thread. Fewer than 2% of `dirt`'s scanned pages are scanned by the mutators. In the case of `neat`, which requires scanning many more pages, the figure is only about 0.05%.

Finally, Table 17 shows the results obtained when running the test programs in four processors with garbage collection. The second row of the table specifies the total number of cycles in the mark phase, over all collections. The numbers of scanned pages, both only by the collector thread and in total, are the result of adding the values obtained for all four processors. The values that measure access to the memory manager are not included because their totals were similar to the values for one processor, which are shown in Tables 15 and 16.

For the sequential and artificial programs, most of the activity is in processor 2, followed by processor 1; processors 3 and 4 show little activity, as happens when garbage collection is turned off. For `matrix`, the activity is shared by all processors, with processor 1 taking the largest part, as happens when the garbage collector is turned off (see Table 13).

`Matrix` disperses its allocation of pages fairly uniformly among the four processors. There is a smaller number of pages allocated at any processor than in the uniprocessor system. Only one collection is triggered during `matrix`'s execution, as opposed to two collections in the uniprocessor system. Something similar happens to the artificial programs, which produce eight collections because of the smaller number of pages allocated at each processor; they are not much smaller, but enough to reduce the number of collections by one. For `neat`, the number of collections is actually eight; with four processors, the collector no longer has problems keeping up with the allocation of pages.

For the sequential programs, the ratio of the running times in four processors to the times in the uniprocessor varies from 7 to 23; these figures are much smaller than

	`Hanoi`	`Factorial`	`Random`	`Queens`	`Matrix`	`Dirt`	`Neat`
Collections	1	4	3	2	1	8	8
Total cycles	1	10	7	4	1	20	22
Pages scanned	23	303	212	176	19	634	15430
Collector scans	11	272	190	123	11	617	15422
Time, in seconds	225	458	2018	3065	135	141	144

Table 17: Garbage collection in four processors.

the ratio of 17 to 30 observed with no garbage collection. For `matrix`, the ratio is 4.2, and the running time is about the same as when running in four processors without garbage collection (see Table 13). For the artificial programs, `dirt` and `neat`, the ratio is 1.41 and 1.22, respectively. This analysis suggests that the garbage collector uses the processors without interfering with the mutators, and that it benefits from running in parallel with them.

On average, the mark phase of the collector needs fewer than 3 cycles per collection, and no collection requires more than 5 cycles. The total number of scanned pages and the proportion of pages scanned by the collector thread are about the same as in the uniprocessor system, except for `matrix`, which triggers only one collection. As in the uniprocessor, most pages are scanned by the collector thread.

Chapter 7

Conclusions

This dissertation shows that it is feasible to use shared virtual memory to distribute the address space of a very high-level, garbage-collected, persistent language.

The measurements discussed in Chapter 6 show that performance remains a problem and needs to be improved. The main obstacle is the high cost of interprocessor communication and its interaction with some of the implementation mechanisms used in distributed *EZ*. The distributed garbage collector, however, shows good performance and integration with the rest of the system.

This chapter analyzes the behavior of the system and offers proposals that might improve its performance. Further research on the system and some related questions are also presented.

7.1 Interprocessor Communication

Most accesses are to local pages and can be satisfied without interprocessor communication. The relative time required to access a local pages is small.

The measurements presented in Chapter 6 show that communication between managers through the network is expensive. The time required to gain access to a page is much longer when the local manager communicates with another manager than when the page is local, or its local copy is known to be valid. Accessing remote pages is about 55 times slower than accessing local pages.

This large extra cost of remote communication justifies almost any effort to reduce the number of remote accesses needed. Any extra costs involved in reducing the number of remote communications are recovered by the reduction in total communication costs.

A second way of reducing total communication costs is to reduce the amount of information that is sent each time that communication takes place. Transmitting copies of pages only when necessary is done for this reason.

The cost of sending a page is a small fraction of the total cost of communication, but the time saved by not sending unnecessary copies is significant. For example, the time it takes to transmit a page — about 4.5 milliseconds — is 20 times longer than the time it takes to read or write a page to disk. The cost incurred by the system to accomplish these savings is small; owners have to store and maintain valid-copies hints, and the system must distinguish messages that request a copy from messages that only ask for confirmation.

The measurements show that this scheme is effective. The number of copies that are sent represents only a small fraction of the number of remote requests, and a tiny fraction of the number of accesses sought by the programs. Furthermore, most copies sent from one manager to another are requested explicitly by the requestor and, thus, only a few are sent unnecessarily.

The measurements also confirm the assumption that pages migrate infrequently, and that the cost of a migration is amortized over several local accesses. In fact, most pages migrate only once. The fear that pages might thrash between several processors appears misplaced.

Most read accesses are for local pages. For remote pages, the intent is that the local caches would reduce communication by providing a local copy of the pages. The effectiveness of the caches is diminished, however, by the use of confirmation. Unless a remote page is already being accessed locally when read access is requested again, the confirmation protocol requires sending a message to its owner to confirm the validity of the local copy. Although proportionally few read accesses are to remote pages, the high cost of interprocessor communication makes them responsible for a large part of the running time of the test programs.

As a possible solution to this problem, the managers might delay sending the decrement request to the owner after all local accesses have finished. This delay can be defined as a minimum time between receiving access to the page from the owner and releasing the page through the decrement request. The delay might allow several local accesses to take place with no extra cost of confirmation from the owner. This solution, however, would probably require a precise tuning of the delay to be used; the chosen value has to allow multiple local accesses without interfering with other processes that might want to write the page.

Alternatively, pages that are repeatedly read by the same processor and not written could migrate to the reader. Competitive algorithms in this context are particularly promising (Karlin *et al.* 1986). The amortized cost of these algorithms is always within a constant factor of the optimal off-line algorithm. Presumably, a threshold in the number of consecutive read accesses from the same processor would trigger the migration. Roughly speaking, a page should migrate when the aggregate extra cost of the remote accesses equals the cost of migration. This criterion would suggest a small threshold in *EZ*; in practice, tuning of the threshold would be required to achieve the best possible response.

Section 7.2 proposes other solutions; they are based on specific knowledge about the pages that cause the problem.

7.2 Immobile Pages

The decision to use confirmation to maintain coherence revealed some conflicts in the implementation. This design decision was based on the belief that most pages would eventually migrate to the processor that is accessing them so that subsequent accesses would be local.

The measurements shown in Chapter 6 make it apparent that, although most pages do migrate, there are some pages that never do because they are written only once, or that seldom do because they are seldom written. Remote access to any of these 'immobile' pages triggers remote requests to the page's owner, even though the local, in-cache, copy is always up to date. Two kinds of pages are responsible for this problem, which is the single biggest factor for the large running time of the system when running on several processors.

The first kind of pages are read-only pages, which contain constant values. They include the global pages that contain the strings used by the system or by the user programs, and the pages that the compiler creates to store constants. The second kind of pages includes pages that are not strictly read-only, but are seldom modified. Pages used for indirection in the implementation of tables are examples.

The unnecessary remote requests triggered by these pages should be reduced or, better yet, eliminated. The increase in the running time of programs produced by these remote requests is too large to be ignored and severely affects the usefulness of the distributed system. Fortunately, there are ways to change the design and implementation to address this problem without sacrificing confirmation as the means to maintain coherence. Changing confirmation to, for example, invalidation would have a major impact in the design of the virtual machine and the run-time system, and would produce a system completely different to the one presented in this dissertation. The changes proposed are conceptually simple, although they may require extensive modifications to the implementation of *EZ*'s run-time system. They focus on reducing the number of remote messages, which might reduce the running time of the system dramatically.

The solution used to solve the problem presented with code blocks, i.e., the use of execute access mode, can be extended to other read-only pages. For this purpose, it is necessary to introduce some modifications to the run-time system and, in particular, to the trusted *EZ* interpreter. The virtual memory could be divided into two areas, one exclusively dedicated to store constant values, which would therefore consist only of read-only pages. For instance, the constant area could be defined as pages with addresses at the low end of the address space, which have, say, zero in their three most significant bits. Code blocks could also be allocated from this area.

Allocation of pages in either area could be requested explicitly with an argument to `allocPage`. Some modifications to the compiler and interpreter would be required.

There are two alternative ways to implement access to these pages after they are created. First, the interpreter can request execute access to these pages in the same way it does for code blocks; the desired address tells the area to which it belongs and the interpreter can decide when execute access is enough. Second, the memory manager could process requests differently according to the area in which the specific requested address lies. The first alternative seems preferable because the memory manager would be oblivious to the partitioning; only the interpreter and run-time system would know about the partitions. On the other hand, letting the memory manager make the distinction would concentrate the changes to the implementation in one place.

The second kind of pages, those that are seldom modified, reveal conflicts between confirmation, the implementation techniques used for associative tables and their primitive operations, and the granularity of migration. Execute access cannot be used with these pages because they are not read-only pages. These pages are indeed modified, although rarely, and thus an old copy is not necessarily up to date.

One possible solution, and perhaps the best and simplest one, is to modify the implementation of the primitive operations for tables to force migration of these pages. In particular, when inserting or removing elements from a table, the intermediate pages that have to be read would be requested for write access, causing them to migrate. This method would have the effect of letting the implementation increase the granularity of migration to a level between pages and whole tables, while at the same time still managing memory at the page level. Under this scheme, pages in the table still migrate only on demand; there is no prefetching of pages on the assumption that they might be accessed later.

7.3 The Garbage Collector

The distributed mark-and-sweep garbage collector works well with the memory manager and rest of the system. It is especially effective when the system activity is really distributed among several processors; the collector takes advantage of performing mostly local operations and runs without increasing the running time of the mutators significantly.

The proposed changes to the implementation of the primitive operations for associative tables may also help the collector. Inducing the migration of the intermediate pages when their predecessors and successors migrate should help reduce the number of cycles in the mark phase. The number of cycles is important because it is a measure of the amount of interprocessor communication caused by the collector.

It is evident that not only the implementation of tables plays a role in the number of cycles of a mark phase; the actual data structures used by the programs, and its interrelations, also help determine the number of cycles. The measurements show that very few cycles are required for the test programs and that, consequently, the communication overhead due to the collector is small.

Most pages scanned by the collection algorithm are scanned by the collector thread; only a few pages are scanned for the mutators by the memory manager. Consequently, few page requests are delayed by the scanning process. In most cases, the time required to gain access to a page is due to the costs of memory management itself.

Collection are reasonably fast, and many could be performed while running the test programs. Requiring a number of allocations before starting a new collection controls the number of collections performed; 500 is an appropriate number for the prototype because it allows several collections to take place without interfering with the mutators. With this value, the collector is able to perform the collections and keep up even with `neat`, which allocates only accessible pages.

However, this number seems too low for a production system, especially when compared with the huge size of the address space, which has eight mega-pages. This number needs to be tuned to allow collections to be performed often enough so that mutators seldom or never have to wait for free pages, but also sporadically enough to minimize the use of the processors for collection purposes.

7.4 Further Research

In addition to the changes proposed in Section 7.2 to improve the current implementation, there are other areas that merit further research. Some of this research can be seen as possible improvements to the current implementation, some require extensive changes to the current system, and some go beyond *EZ* itself.

For example, there are alternative ways to manage the pages used by the memory manager and collector data structures. In particular, only the local root page currently has a reserved frame in physical memory at each processor. Frames could be reserved for a number of other system pages, like the first page in each bag used by the garbage collector, the set of owned pages, and the bit map of marks. Having some of these pages always in the cache would improve the performance of the system threads and would reduce the time that processes have to wait. It may also improve cache performance by reducing the number of pages that compete for frames. Experimentation would tell how many and which system pages are worth keeping in physical memory or at fixed locations.

Another part of the system that deserves some attention is the implementation of *EZ* processes, their related primitive operations, and the scheduling mechanism. Currently, this part of the implementation is written at the associative-table level, and it probably costs more than necessary because of its repeated use of table operations. A new implementation at a lower level that takes advantage of its knowledge about the internals of tables, for example, might be more efficient. Directly accessing the pages instead of using the primitive table operations would avoid redundant page accesses and improve global system performance.

A third area relates to the use of virtual-memory hardware. In the current implementation all memory management is done in software. It is unclear how much the use of virtual-memory hardware would improve system performance, but it is an important area for further research (Appel and Li 1991). Instead of calls to `getPage`, the interpreter would simply access virtual memory. The virtual-memory hardware would provide address translation, and the page-fault handler would cache pages when necessary. Extensive changes to the virtual machine would be required to guarantee the integrity of the values that the interpreters read.

Using virtual-memory hardware would certainly reduce the time needed to gain access to local pages, to which access can be granted without interprocessor communication. The assumption is that most local accesses would not require a trap to the handler. For remote pages, however, virtual-memory hardware does not help significantly because most of the time is spent in communicating with the page's owner. Unless the number of remote requests is reduced — by the changes proposed above, for example — using virtual-memory hardware would not improve overall system performance. However, a large proportion of requests are for local pages, so reducing their processing time would help reduce the total running time of programs.

Finally, the success of a distributed, persistent system — like distributed *EZ* — hinges in part on its reliability and on how well it survives machine failures. Distributed *EZ* needs a mechanism that allows it to recover from failures. Both the distributed virtual-memory manager and the garbage collection algorithm would have to be revised to enable them to cope with failures. The persistent address space must remain coherent in the presence of failures; especial attention must be given to the possibility of missing messages.

The search for appropriate recovery mechanisms is an important area of research. Checkpoint and restart algorithms are well suited to provide reliability to object-oriented systems, main-memory based databases, and persistent programming environments (Li, Naughton, and Plank 1990). The details of reliability mechanisms and their implementation depend heavily on storage-management details, e.g., traditional disk recovery schemes depend on file-system implementation details. The disk image of the virtual address space provides the basis upon which to implement such a checkpointing mechanism in distributed *EZ*. Recent studies (Plank 1993) show how to use the virtual-memory system to provide real-time, concurrent checkpoint algorithms. These algorithms require a constant amount of working buffers, no changes to the programs running in the system, and no special hardware assistance.

7.5 More Open Questions

EZ's design assumes that a single, persistent virtual address space in which virtual addresses are 32 bits is sufficient. The rapid advances in processor speeds, memory systems, and disk technologies increase the pressure for a larger address space and distribution induces pressure for multiple address spaces.

One of the advantages of the distributed memory manager described in this dissertation is that it leaves the rest of the system — the interpreter and the run-time system — nearly untouched. One disadvantage is that it does not address the problem of multiple address spaces. Accommodating multiple address spaces motivates a second approach to distribution, which could be achieved by permitting the spaces to reside anywhere in a network. This approach can be viewed as the complement of the distributed memory manager approach; here, the original memory manager remains nearly untouched, but the interpreter and run-time system are modified.

References to other address spaces could be made through interaddress-space pointers, which are functionally equivalent to capabilities used in some distributed systems like Amoeba (Tanenbaum *et al.* 1990). 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 on these pointers. The attraction of this approach is that interaddress-space pointers can be hidden from the programmer and thus encapsulate multiple address spaces much like strings encapsulate files. The cost, however, is that the interpreter and run-time system must take special action in order to dereference these pointers; however, efficient implementation techniques for this kind of dereferencing are well known and have been used for some time in other high-level language systems (Griswold and Griswold 1986, Hanson 1976b) and for implementing implicit synchronization (Halstead 1985). Another advantage of multiple address spaces is that pages would not migrate, which simplifies memory management and also increases reliability; remote dereferencing translates essentially into remote procedure calls (Birrell and Nelson 1984, Nelson 1981) to `getPage` and `putPage`.

Another interesting line of research is to port *EZ* to a shared-memory multiprocessor. For example, the shared-memory architecture of the Firefly (Thacker and Stewart 1987, Thacker, Stewart, and Satterthwaite 1988) matches *EZ*'s large, flat, single address space. A multi-processor implementation would also test the suitability of the process mechanism and its implementation, and would require significant redesign of the virtual machine and memory-manager interface. Once again, integrity of data and atomicity of the operations are a source of concern; new ways to deal with some 'big' operations, such as compilation and concatenation of long strings, would be needed.

The research presented in this dissertation has shown the feasibility of distributing *EZ*'s address space to build a distributed *EZ* system. It shows how the idea of shared virtual memory can be applied to language-based environments, and how to provide a distributed garbage collector for these systems. Whether the techniques that have been presented are applicable to other high-level languages remains an interesting question. In principle, the techniques are applicable because there is little in the memory manager or garbage collector that is specific to *EZ*. It is unclear, however, that their effectiveness is independent of other properties of *EZ* or its implementation; some modifications to the systems and their implementations may be necessary. A LISP system, for example, with small-size objects requires a finer-grained allocation and collection algorithms, which conflicts with the desire to use larger pages for memory management. Even though the general techniques may still apply, extensive adjustments to both the designs and implementations might be needed.

References

Accetta, Michael J., Robert V. Baron, William Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, Jr., and Michael W. Young. 1986 (July).
Mach: A new kernel foundation for UNIX development.
In *Proceedings of the Summer USENIX Technical Conference*, pages 93–112, Atlanta, GA.

Aho, Alfred V., Brian W. Kernighan, and Peter J. Weinberger. 1988.
The AWK Programming Language.
Reading, MA: Addison Wesley.

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. 1986.
Compilers: Principles, Techniques, and Tools.
Reading, MA: Addison Wesley.

Appel, Andrew W. and Kai Li. 1991 (April).
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.

Appel, Andrew W. and David B. MacQueen. 1991 (aug).
Standard ML of New Jersey.
In Wirsing, Martin, editor, *Third International Symposium on Programming Language Implementation and Logic Programming*, pages 1–13, New York. Springer-Verlag.

Appel, Andrew W., John R. Ellis, and Kai Li. 1988 (July).
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.

Appel, Andrew W. 1987 (June).
Garbage collection can be faster than stack allocation.
Information Processing Letters, 25(4):275–279.

———. 1991.
Garbage collection.
In Lee, Peter, editor, *Topics in Advanced Language Implementation Techniques*, chapter 4. MIT Press.

———. 1992.
Compiling with Continuations.
Cambridge: Cambridge University Press.

Atkinson, Malcolm P. and Ronald Morrison. 1985 (October).
Procedures as persistent data objects.
ACM Transactions on Programming Languages and Systems, 7(4):539–559.

Atkinson, Malcolm P., Ken Chisholm, Paul Cockshott, and Richard Marshall.
1983 (March).
Algorithms for a persistent heap.
Software—Practice & Experience, 13(3):259–271.

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

Baker, Henry G. 1978 (April).
List processing in real-time on a serial computer.
Communications of the ACM, 21(4):280–294.

Bennett, J. K., J. B. Carter, and Willy Zwaenepoel. 1990 (March).
Munin: Distributed shared memory based on type-specific memory coherence.
Proceedings of the Second SIGPLAN Symposium on Principles & Practice of Parallel Programming, SIGPLAN Notices, 25(3):168–176.

Bevan, D. I. 1987.
Distributed garbage collection using reference counting.
In de Bakker, J. W., A. J. Nijman, and P. C. Treleaven, editors, *PARLE, Parallel Architectures and Languages Europe, Volume II: Parallel Languages (LNCS 259)*, pages 176–187. Berlin: Springer-Verlag.

Birrell, Andrew D. and Bruce J. Nelson. 1984 (February).
Implementing remote procedure calls.
ACM Transactions on Computer Systems, 2(1):39–59.

Björnerstedt, Anders. 1989.
Secondary storage garbage collection for decentralized object-based systems.
In Tsichritzis, D. C., editor, *Object Oriented Development*, pages 277–319. Genève: Centre Universitaire d'Informatique.

Bourne, Stephen R. 1978 (July).
The UNIX shell.
Bell System Technical Journal, 57(6):1971–1990.

Brinch Hansen, Per. 1976 (April).
The Solo operating system: Job interface.
Software—Practice & Experience, 6(2):151–164.

Campos, Alvaro E. and David R. Hanson. 1992 (September).
Distributed *EZ*.
In *Proceedings of the 16th Annual International Computer Software and Applications Conference*, pages 136–142, Chicago.

Cooper, Eric C. and Richard P. Draves. 1988 (June).
C threads.
Technical Report CMU-CS-88-154, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Cormack, Gordon V. 1988 (May).
A micro-kernel for concurrency in C.
Software—Practice & Experience, 18(5):485–491.

Dawson, Jeffrey L. 1982 (August).
Improved effectiveness from a real time lisp garbage collector.
In *ACM Conference on LISP and Functional Programming*, pages 159—167, Pittsburgh, PA.

Digital Equipment Corporation. 1992 (February).
Alpha Architecture Handbook—Preliminary Edition.
Maynard, MA.

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

Ellis, John R. 1980 (May).
A Lisp shell.
SIGPLAN Notices, 15(5):24–34.

Fernandez, Mary F. and David R. Hanson. 1992 (August).
Garbage collection alternatives for Icon.
Software—Practice & Experience, 22(8):659–672.

Fraser, Christopher W. and David R. Hanson. 1983 (June).
A high-level programming and command language.
Proceedings of the SIGPLAN'83 Symposium on Programming Language Issues in Software Systems, SIGPLAN Notices, 18(6):212–219.

———. 1984 (January).
The *EZ* reference manual.
Technical Report 84-1, Department of Computer Science, The University of Arizona, Tucson, AZ.

———. 1985 (January).
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.

Gelernter, David R. and N. Carriero. 1986 (May).
The S/Net's linda kernel.
ACM Transactions on Computer Systems, 4(2):110–129.

Gelernter, David, Suresh Jagannathan, and Thomas London. 1987 (July).
Parallelism, persistence and meta-cleanliness in the symmetric Lisp interpreter.
Proceedings of the SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques, SIGPLAN Notices, 22(7):274–282.

Goldberg, Adele, David Robson, and Daniel H. H. Ingalls. 1983.
SmallTalk-80: The Language and Its Implementation.
Reading, MA: Addison Wesley.

Goldberg, Benjamin. 1989 (July).
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.

Griswold, Ralph E. and Madge T. Griswold. 1986.
The Implementation of the Icon Programming Language.
Princeton, NJ: Princeton University Press.

———. 1990.
The Icon Programming Language. second edition.
Englewood Cliffs, NJ: Prentice Hall.

Griswold, Ralph E., David R. Hanson, and John T. Korb. 1981 (April).
Generators in Icon.
ACM Transactions on Programming Languages and Systems, 3(2):144–161.

Griswold, Ralph E., James F. Poage, and Ivan P. Polonsky. 1971.
The SNOBOL4 Programming Language. second edition.
Englewood Cliffs, NJ: Prentice Hall.

Halstead, Jr., Robert H. 1984 (August).
Implementation of Multilisp: Lisp on a multiprocessor.
In *ACM Conference on LISP and Functional Programming*, pages 9–17, Austin, TX.

———. 1985 (October).
Multilisp: A language for concurrent symbolic computation.
ACM Transactions on Programming Languages and Systems, 7(4):501–538.

Hanson, David R. and Ralph E. Griswold. 1978 (May).
The SL5 procedure mechanism.
Communications of the ACM, 21(5):392–400.

Hanson, David R. and Makoto Kobayashi. 1990 (March).
EZ processes.
In *Proceedings of the International Conference on Computer Languages*, pages 90–97, New Orleans, LA.

Hanson, David R. 1976a (October).
A procedure mechanism for backtrack programming.
In *Proceedings of the ACM Annual Conference*, pages 401–405, Houston, TX.

———. 1976b (April).
Variable associations in SNOBOL4.
Software—Practice & Experience, 6(2):245–254.

Herlihy, M. and Barbara H. Liskov. 1982 (October).
A value transmission method for abstract data types.
ACM Transactions on Programming Languages and Systems, 4(4):527–551.

Irons, Edgar T. 1976 (October).
Software for a graphics editing machine.
In *Proceedings of the Fifth Texas Conference on Computing Systems*, pages 13–19, Austin, TX.

Iverson, Kenneth E. 1962.
A Programming Language.
New York, NY: John Wiley & Sons.

Jagannathan, Suresh and Jim Philbin. 1992a (July).
A customizable substrate for concurrent languages.
Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation, SIGPLAN Notices, 27(7):55–67.

———. 1992b (June).
A foundation for an efficient multi-threaded scheme system.
In *ACM Conference on LISP and Functional Programming*, pages 345—357, San Francisco, CA.

Jones, Anita K. 1977.
The narrowing gap between language systems and operating systems.
In *Proceedings of the IFIPS Conference*, pages 869–873, Montreal.

Joy, William N. 1980 (November).
An introduction to the C shell.
Technical report, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA.

Karlin, Anna R., Mark S. Manasse, Larry Rudolph, and Daniel D. Sleator. 1986 (October).
Competitive snoopy caching.
In *Annual Symposium on Foundations of Computer Science*, pages 244–254, Toronto.

Kernighan, Brian W. and Dennis M. Ritchie. 1988.
The C Programming Language. second edition.
Englewood Cliffs, NJ: Prentice Hall.

Kernighan, Brian W. 1975 (October).
RATFOR — a preprocessor for a rational fortran.
Software—Practice & Experience, 5(4):395–406.

Knuth, Donald E. 1981.
The Art of Computer Programming: Volume 2, Seminumerical Algorithms. second edition.
Reading, MA: Addison Wesley.

Lang, Bernard, Christian Queinnec, and José Piquer. 1992 (January).
Garbage collecting the world.
In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 39–50, Albuquerque, NM.

Lermen, Claus-Werner and Deiter Maurer. 1986 (August).
A protocol for distributed reference counting.
In *ACM Conference on LISP and Functional Programming*, pages 343–350, Cambridge, MA.

Levine, John. 1980 (May).
Why a Lisp-based command language?
SIGPLAN Notices, 15(5):49–53.

Li, Kai and Paul Hudak. 1986 (August).
Memory coherence in shared virtual memory systems.
In *Proceedings 5th Annual Symposium on Principles of Distributed Computing*, pages 229–239, Calgary, Alberta.

———. 1989 (November).
Memory coherence in shared virtual memory systems.
ACM Transactions on Computer Systems, 7(4):321–359.

Li, Kai and Richard Schaefer. 1989a.
A hypercube shared virtual memory system.
In *Proceedings of the International Conference on Parallel Processing*, pages 125–132, University Park, PA.

———. 1989b (April).
Shiva: An operating system transforming a hypercube into a shared-memory machine.
Technical Report CS-TR-217-89, Princeton University, Department of Computer Science, Princeton, NJ.

Li, Kai, Jeffrey F. Naughton, and James S. Plank. 1990 (March).
Real-time, concurrent checkpoint for parallel programs.
Proceedings of the Second SIGPLAN Symposium on Principles & Practice of Parallel Programming, SIGPLAN Notices, 25(3):79–88.

Li, Kai. 1986 (September).
Shared Virtual Memory on Loosely Coupled Multiprocessors.
PhD thesis, Yale University, New Haven, CT.

Lieberman, Henry and Carl Hewitt. 1983 (June).
A real-time garbage collector based on the lifetimes of objects.
Communications of the ACM, 26(6):419–429.

Low, Colin. 1988 (August).
A shared, persistent object store.
In Gjessing, Stein and Kristen Nygaard, editors, *ECOOP'88: European Conference on Object-Oriented Programming (LNCS 322)*, pages 390–410, Berlin. Springer-Verlag.

Meyer, Bertrand. 1992.
Eiffel: The Language.
London: Prentice Hall International.

Morris, James H., Eric Schmidt, and Phillip Wadler. 1980 (January).
Experience with a string processing applicative language.
In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 32–46, Las Vegas, NV.

Nelson, Bruce J. 1981 (May).
Remote Procedure Call.
PhD thesis, Carnegie Mellon University, Pittsburgh, PA.

Nitzberg, Bill and Virginia Lo. 1991 (August).
Distributed shared memory: A survey of issues and algorithms.
IEEE Computer, 24(8):52–60.

Paepcke, Andreas. 1988 (August).
PCLOS: A flexible implementation of CLOS persistence.
In Gjessing, Stein and Kristen Nygaard, editors, *ECOOP'88: European Conference on Object-Oriented Programming (LNCS 322)*, pages 374–389, Berlin. Springer-Verlag.

Pallas, Joseph and David Ungar. 1988 (July).
Multiprocessor smalltalk: A case study of a multiprocessor-based programming environment.
Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation, SIGPLAN Notices, 23(7):11–20.

Pike, Rob. 1990 (July).
The implementation of Newsqueak.
Software—Practice & Experience, 20(7):649–659.

Plank, James Steven. 1993 (June).
Efficient Checkpointing on MIMD Architectures.
PhD thesis, Department of Computer Science, Princeton University, Princeton, NJ.

Rovner, Paul. 1984 (July).
On adding garbage collection and runtime types to a strongly-typed, statically-checked, concurrent language.
Technical Report CSL-84-7, Palo Alto Research Center, Xerox Corp., Palo Alto, CA.

———. 1986 (November).
Extending Modula-2 to build large, integrated systems.
IEEE Software, 3(6):46–57.

Rudalics, Martin. 1986 (August).
Distributed copying garbage collection.
In *ACM Conference on LISP and Functional Programming*, pages 364–372, Cambridge, MA.

Sandewall, Erik. 1978 (March).
Programming in the interactive environment: The Lisp experience.
ACM Computing Surveys, 10(1):35–71.

Schelvis, Marcel and Eddy Bledoeg. 1988 (August).
The implementation of a distributed Smalltalk.
In Gjessing, Stein and Kristen Nygaard, editors, *ECOOP'88: European Conference on Object-Oriented Programming (LNCS 322)*, pages 212–232, Berlin. Springer-Verlag.

Stevens, W. Richard. 1990.
Unix Network Programming.
Englewood Cliffs, NJ: Prentice-Hall, Inc.

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

Tanenbaum, Andrew S., Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. 1990 (December).
Experiences with the Amoeba distributed operating system.
Communications of the ACM, 33(12):46–63.

Teitelbaum, Warren. 1975 (December).
Interlisp reference manual.
Technical report, Palo Alto Research Center, Xerox Corp., Palo Alto, CA.

Teitelman, Warren and Larry Masinter. 1981 (April).
The Interlisp programming environment.
Computer, 14(4):25–34.

Teitelman, Warren. 1985 (March).
A tour through Cedar.
IEEE Transactions on Software Engineering, SE-11(3):285–302.

Thacker, Charles P. and Lawrence C. Stewart. 1987 (October).
Firefly: A multiprocessor workstation.
In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–172, Palo Alto, CA.

Thacker, Charles P., Lawrence C. Stewart, and Edward H. Satterthwaite, Jr. 1988 (August).
Firefly: A multiprocessor workstation.
IEEE Transactions on Computers, 37(8):909–920.

Thompson, Ken. 1978 (July).
UNIX implementation.
Bell System Technical Journal, 57(6):1931–1946.

Ungar, David. 1984 (May).
Generation scavenging: A non-disruptive high-performance storage reclamation algorithm.
Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices, 19(5):157–167.

Vegdahl, Steven R. 1986 (November).
Moving structures between Smalltalk images.
OOPSLA'86 Conference Proceedings, SIGPLAN Notices, 21(11):466–471.

Watson, Paul and Ian Watson. 1987.
An efficient garbage collection scheme for parallel computer architecture.
In de Bakker, J. W., A. J. Nijman, and P. C. Treleaven, editors, *PARLE, Parallel Architectures and Languages Europe, Volume II: Parallel Languages (LNCS 259)*, pages 432–443. Berlin: Springer-Verlag.

Wiebe, Douglas. 1986 (November).
A distributed repository for immutable persistent objects.
OOPSLA'86 Conference Proceedings, SIGPLAN Notices, 21(11):453–465.

Wile, David S., Neal Goldman, and Dennis G. Allard. 1987 (August).
Maintaining object persistence in the Common Lisp framework.
In *Proceedings of the Second International Workshop on Persistent Object Systems*, pages 382–405, Appin, Scotland.

Wileden, Jack C., Alexander L. Wolf, Charles D. Fisher, and Peri L. Tarr. 1989 (February).
PGRAPHITE: An experiment in persistent typed object management.
Proceedings of SIGSOFT'88: Third Symposium on Software Development Environments, SIGPLAN Notices, 24(2):130–142.

Wirth, Niklaus and Jurg Gutknecht. 1989 (September).
The Oberon system.
Software—Practice & Experience, 19(9):657–693.