MIT/LCS/TR-209

A MACHINE ARCHITECTURE TO SUPPORT AN OBJECT-ORIENTED LANGUAGE

Alan Snyder

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract No. N00014-75-C-0661

This blank page was inserted to preserve pagination.

MIT/LCS/TR-209

A Machine Architecture to Support an Object-Oriented Language

by

Alan Snyder

Copyright (c) 1979 by Alan Snyder

March 1979

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract N00014-75-C-0661.

Massachusetts Institute of Technology
Laboratory for Computer Science

Cambridge **Massachusetts 02139**

This empty page was substituted for a blank page in the original document.

A Machine Architecture to Support an Object-Oriented Language

by

Alan Snyder

Submitted to the Department of Electrical Engineering and Computer Science
on March 9, 1979 in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Abstract

In object-oriented languages (e.g., LISP, Simula, and CLU), all (or most) data objects used by a program are implicitly allocated from a free-storage area and are accessed via fixed-size references. The storage for an object is automatically reclaimed (garbage collected) when the object is no longer accessible to the program.

This thesis presents the design of a computer system that directly supports an object-oriented machine language. The machine provides a single, large universe of objects shared by multiple processes. The design uses expected future technologies (fast-access secondary storage devices and inexpensive processors) to satisfy the goals of good performance and a simple, modular system organization.

Automatic storage reclamation is performed primarily using reference counts. The proposed reference count implementation reduces the time overhead of automatic storage reclamation and allows most reclamation processing to be performed in parallel with normal computation. In addition, the reference count scheme can be used in a multiprocessor configuration without introducing complex synchronization problems.

A proposed implementation of the machine is described in terms of a number of specialized processor modules communicating via messages. Multiple processors are used to improve performance and to achieve a more modular system structure.

Thesis Supervisor: Barbara H. Liskov
Title: Associate Professor of Computer Science and Engineering

Keywords: language-oriented machine architecture, object-oriented programming languages, garbage collection, reference counts

This empty page was substituted for a blank page in the original document.

Acknowledgments

I would like to thank Barbara Liskov for her efforts in supervising this thesis. I would also like to thank my readers, Liba Svobodova and Jack Dennis, for carefully reading the thesis and providing helpful comments. Thanks are due also to Eliot Moss, who read a draft of the thesis and pointed out a number of areas needing clarification.

This empty page was substituted for a blank page in the original document.

CONTENTS

Abstract 3

Acknowledgments 4

Contents 5

1. Introduction 9

- 1.1 Motivation 9
- 1.2 Goals 10
- 1.3 Background 12
- 1.4 Our Approach 14
- 1.5 Overview 16

2. Objects 19

- 2.1 Description 19
- 2.2 Implementation Implications 21
- 2.3 Example 23
- 2.4 Advantages 23
- 2.5 A Universe of Objects 27

3. An Object-Oriented Machine Architecture 31

- 3.1 CLU 31
- 3.2 Architectural Philosophy 33
- 3.3 The Machine Language 35
- 3.4 A Simplification 38

4. System Structure: an Overview 43

4.1 Machine Functions 43
4.2 Design Strategies 44
4.3 High-Level System Structure 46
4.4 Multiprocessing Philosophy 50
4.5 The Processing Module 55

5. The Implementation of Object References 61

5.1 The Problem 61
5.2 Our Solution 62
5.3 Associative Memory Design 64
5.4 Evaluation 68
5.5 Comparison with Other Methods 70

6. The Implementation of Automatic Storage Reclamation 77

6.1 Accessible Objects 77
6.2 Simple Garbage Collection 78
6.3 Alternatives 80
6.4 Reference Counts 81
6.5 Conventional Reference Count Implementation 83
6.6 Queued Reference Counts 84
6.7 A Note on Ordering 87
6.8 Queued Reference Count Performance 91
6.9 Garbage Collection 95
6.10 Garbage Collection Performance 98
6.11 Evaluation 105

7. The Implementation of Storage Allocation 109

7.1 Secondary Storage Allocation 109
7.2 Zoned Allocation 110
7.3 Block Allocation 114
7.4 Evaluation 116
7.5 Primary Storage Allocation 118

8. System Structure: the Memory Module 121

8.1 Vectors and Pages 121
8.2 Vector Module Specification 124
8.3 Vector Module Design 128
8.4 Page Module Specification 132
8.5 Page Module Design 135
8.6 Improvements 151

9. Conclusions 153

9.1 Evaluation 153
9.2 Further Work 160
9.3 A Perspective 162

Appendix I. A Possible Machine Language 165

1. Procedures 165
2. Procedure Activations 165
3. Instruction Set 166
4. Notes 168
5. Example 168

Appendix II. Page Module Description 171

1. Legend 171
2. Page Handler (PH) 172
3. Storage Processor (SP) 174
4. Secondary Storage Module (SSM) 176

References 179

FIGURES

Figure 1. An example of objects. 23
Figure 2. A universe of objects. 28
Figure 3. The CLU library. 30
Figure 4. Process operations. 36
Figure 5. The vector operations. 39
Figure 6. High-level system structure. 47
Figure 7. An example request and reply. 48
Figure 8. Processing Module block diagram. 56
Figure 9. IP cache of immutable object elements. 59
Figure 10. The logical structure of a set associative memory. 65
Figure 11. Set associative memory implementation. 66
Figure 12. GQ Processing Algorithm. 90
Figure 13. Recursive Mark Phase Algorithm. 97
Figure 14. Iterative Mark Phase Algorithm. 99
Figure 15. Iterative Multiprocess Mark Phase Algorithm. 104
Figure 16. Fields of a vector header word. 122
Figure 17. Memory module structure. 123
Figure 18. Vector module external operations. 124
Figure 19. Vector module internal operations. 125
Figure 20. Synchronization analysis of vector operations. 127
Figure 21. Vector module block diagram. 129
Figure 22. Page module operations. 134
Figure 23. Page module block diagram. 135
Figure 24. An example of the machine language. 169

1. Introduction

1.1 Motivation

The design of computers is strongly influenced by the characteristics of available technology. Until recently, computers have been designed under the constraint that processing hardware is expensive. The resulting desire to minimize hardware cost has had a number of unfortunate effects.

One effect is that conventional machines generally provide a rather low level machine language, thus encouraging the use of programming languages with similar low level (or "machine oriented") semantics. Although better programming languages have been developed, their implementation on conventional machines is often excessively inefficient.

Another effect is that individual processors are multiplexed to perform many different functions. For example, a single processor is often used to interpret user processes, implement the virtual memory, and control I/O devices. This multiplexing is supported by a complex, interrupt-driven operating system, characterized by considerable interactions among its various components. Such complex systems are difficult to understand or verify and are likely to be unreliable.

The cost of hardware is continually decreasing and the significant cost of software has become more and more apparent. Therefore, we believe it is appropriate to consider how hardware technology can be used to implement better programming languages and to reduce the complexity of computer systems.

1.2 Goals

This thesis presents the design of a new computer system that efficiently supports a single, large universe of *objects*. Our notion of objects appears in a number of programming languages, such as LISP [36], Simula [6], and CLU [22]. The essential characteristics of such *object-oriented* languages and their advantages over traditional *value-oriented* languages are described in detail in the next chapter. We note at this point, however, that an important implementation implication of object-oriented languages is the use of automatic storage reclamation (garbage collection).

Unlike most implementations of object-oriented languages on conventional machines, which provide a separate and usually small space of objects for each process, our computer system instead provides a single, very large space of objects shared by all of the processes in the system. This space of objects would include not only temporary objects used during the execution of programs, but also the "permanent" procedures and data normally stored in a file system.

Having a single, large *universe* of objects means that there is no distinction between objects that are local to a process and those that are stored in the permanent file system. There are no artificial barriers between different processes or between processes and the file system. Objects in the file system can be accessed directly, with no restrictions on the types of objects that may be permanently stored and no need for conversions.

The primary goal of this thesis has been to design a machine that effectively supports a large universe of objects. A second goal has been to minimize the complexity of the design. We want the machine to have a simple

and modular structure, in comparison with conventional machines and operating system kernels that provide similar functions. To accomplish these goals, we have made two assumptions about expected future technology:

The first assumption is that processors are sufficiently inexpensive that we can use a number of processors where one is used today. We use multiple processors to obtain greater modularity in the implementation of the system, as compared to current systems where processors are a scarce resource and must be multiplexed to serve many different functions. In addition, we use multiple processors to improve the performance of the system, particularly the implementation of automatic storage reclamation. Our goal is to increase the throughput of the system, although perhaps at the expense of decreased utilization of resources.

The second assumption is the existence of fast-access secondary storage devices that can be used for file storage. We are envisioning an access time on the order of 100 microseconds (compared to 10 milliseconds for current disks). Such devices are used to obtain good multi-level memory performance without introducing undue complexity. This assumption is motivated by the expected small average size of objects, based on measurements of existing programs; it is not needed if the objects supported by the system are mostly large objects, similar to pages or segments in current virtual memory systems.[1]

---

[1] Aside from a fast access time, we are not assuming any special properties of the secondary storage devices. For example, we do not assume the ability to scan the entire secondary storage in one access time, an ability provided by charge-coupled devices.

1.3 Background

We are proposing a single large space of objects whose size is similar to that of current file systems. Thus, the universe of objects will be implemented by a multi-level memory system. Current architectures cannot efficiently support such a large space of objects. The major problems involve the performance of the multi-level memory system.

Current secondary storage devices are characterized by access times that are many orders of magnitude longer than the access time of primary storage. To achieve reasonable performance in a multi-level memory system, it is thus necessary that the rate of access to secondary storage be quite low. To keep the access rate low, it is necessary to transfer a fairly large amount of *useful* data on each transfer from secondary storage. (Transferring a larger amount of data than is actually needed to fulfill a request is beneficial only if some of that additional data will be needed in the near future.)

Page sizes in current multi-level memory systems range from 128 words to 4096 words. However, programming-language objects tend to be quite small. A median size of under 20 words is not unlikely [3], and we have measured programs whose average object size is only three words (see Section 6.8). Thus, to efficiently implement an object-oriented storage model using current secondary storage devices, it is necessary to group related objects together in some manner and transfer them as a unit between primary and secondary storage.

Conventional implementations of object-oriented languages group related objects together implicitly through the use of small object spaces. Each space of objects contains the objects needed by a single process. Thus, the space can be broken into a small number of pages all of which contain objects that are likely to be used by the process. The use of a compacting garbage collector tends to

minimize the number of pages used by each space of objects, thus reducing the working sets of the processes [10]. Furthermore, since each small space of objects is garbage collected separately, the time required for garbage collection is (in most cases) tolerable.

These techniques are not easily extended to the case of a single, very large space of objects. Here, some additional grouping of objects is needed to take the place (for performance reasons) of the small object spaces in conventional systems. The mechanism for grouping objects can be either explicit (the groups are constructed by the user) or implicit (the groups are constructed automatically by the system). In addition, it must be possible to perform garbage collection on individual groups of objects. A garbage collection of an entire multi-level memory system would be quite slow. Even if the garbage collection were performed concurrently with normal system operation, it would tie up the secondary storage devices and significantly reduce the performance of the system. Furthermore, unless garbage collection is performed reasonably frequently, it will have little effect on the working sets of the active processes.

A system with these characteristics has been designed by Bishop [7]. In his system, objects are grouped into areas, which are explicitly created and manipulated by users. Objects may be explicitly assigned to areas by the programmer. However, explicit assignment of objects to areas is not necessary; a mechanism is provided that will automatically move objects to the "proper" areas. In addition, Bishop's system includes a compacting garbage collector that works on individual areas or groups of related areas.

We believe that Bishop's approach to the implementation of a large universe of objects is the correct one, given current secondary storage devices. However, his system is complex and its user interface (with areas) is more complex than

necessary. We believe that expected future technology will allow a very large universe of objects to be implemented without any notion of areas and in a relatively simple and modular way.

1.4 Our Approach

As stated above, we are making two assumptions about expected future technology. The first assumption, inexpensive processors, is fairly safe. LSI processors are being introduced today that are comparable in power to minicomputers of a few years ago. It is widely predicted that LSI processors equivalent to current mainframes will be developed in the next decade. The cost of these processors will be quite low compared to the total cost of a computer system. As a result, it will be feasible to use many processors in a single system.

The other assumption, fast-access file storage devices, is more questionable. The access time figure of 100 microseconds is within the predicted range for charge-coupled devices and electronic beam memories [33]. Electronic beam memories, however, are still in the research stage. Commercial charge-coupled devices are beginning to appear; however, the major question here is whether their cost will ever be sufficiently low to allow them to replace disks. Current projections [25] show the cost of CCDs approaching the current cost of disk memories. However, these projections also show the cost of disk memories decreasing, so that disk memories will still be an order of magnitude less expensive than CCD memories. Thus, although it is difficult to predict that CCDs will *completely* replace disks in the near future, they can be expected to be used in reasonably large quantities for file storage.

The assumed existence of fast-access secondary storage devices means that it is possible to swap individual objects between primary and secondary storage with acceptable performance. The expected decrease in the average amount of

information transferred on each secondary storage access is about a factor of 100 (10 word objects vs. 1000 word pages). We would expect, therefore, that the rate of access to secondary storage would increase by at most a factor of 100.[2] However, the expected decrease in the access time compared to current secondary storage devices is also about a factor of 100. Thus, it is reasonable to predict that the performance of a system that swapped individual objects to and from fast-access secondary storage devices would be no worse than the performance of current multi-level memory systems.

In many current systems, the time required to perform a secondary storage transfer is rivaled by the time spent by the processor in handling the page fault, locating the page in secondary storage, scheduling the transfer, switching to another process, etc. Unless this overhead can be reduced, full advantage could not be taken of the improved secondary storage access times postulated. We do not believe that a large overhead is inherently necessary. Much of the overhead in current systems represents attempts at optimization that are appropriate only in the context of a very long secondary storage delay. Other aspects, we believe, can be reduced by proper design.

We also note that a full factor of 100 improvement in the secondary storage access time may not be needed to achieve good performance. For one thing, swapping individual objects rather than pages can allow a given amount of primary storage to capture a greater portion of the working sets of executing processes, thus reducing the secondary storage access rate. In addition, the rapidly falling cost of primary storage will allow larger amounts of primary storage to be used, which can also reduce the secondary storage access rate.

---

[2]It is possible for the number of secondary storage accesses to increase by more than a factor of n when the page size is cut by a factor of n [24, 16]. However, we do not believe this anomaly to be a serious problem in practice.

However, neither of these effects reduce the number of secondary storage accesses that occur when a program or data base is initially accessed. To be conservative, we will continue to assume the factor of 100 improvement.

If individual objects are swapped, then there is no need to group objects together for performance reasons. Thus, there is no need for areas or an automatic grouping mechanism. Similarly, there is no need to have a compacting garbage collector to improve locality. (A compacting garbage collector might still be useful for reducing storage fragmentation.) Instead, we can use reference counts as our primary means of automatic storage reclamation. Infrequent garbage collection can be used to reclaim inaccessible objects that cannot be detected using reference counts.

1.5 Overview

The initial chapters present an overview of the architecture and design of the machine. In Chapter 2, we describe the object-oriented storage model in detail and discuss the advantages and implementation implications of this model. In Chapter 3, we describe the visible architecture of the machine and the philosophy of its design. In Chapter 4, we give an overview of the implementation of the machine. The system is presented as consisting of two major modules, one implementing the object-oriented memory, the other implementing processes. Our approach to the implementation of multiple processes is briefly explained.

The remainder of the thesis concentrates on the memory module. The next