DIGITAL ACCESS TO
SCHOLARSHIP AT HARVARD
DASH.HARVARD.EDU

HARVARD LIBRARY
Office for Scholarly Communication

R-Code a Very Capable Virtual Computer

Citation
Walton, Robert Lee. 1995. R-Code a Very Capable Virtual Computer. Harvard Computer Science Group Technical Report TR-37-95.

Permanent link
http://nrs.harvard.edu/urn-3:HUL.InstRepos:26506458

Terms of Use
This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

Share Your Story

The Harvard community has made this article openly available.
Please share how this access benefits you. Submit a story .

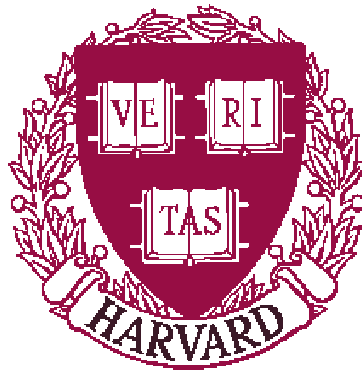
Accessibility

R-Code a Very Capable Virtual Computer

Robert Lee Walton

TR-37-95

October 1995



Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

R-CODE
A Very Capable Virtual Computer

A thesis presented

by

Robert Lee Walton

to

The Division of Applied Sciences

in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
in the subject of
Computer Science

Harvard University
Cambridge, Massachusetts

October 1995

© 1995 by Robert Lee Walton
All rights reserved.

ABSTRACT

This thesis investigates the design of a machine independent virtual computer, the R-CODE computer, for use as a target by high level language compilers. Unlike previous machine independent targets, R-CODE provides higher level capabilities, such as a garbage collecting memory manager, tagged data, type maps, array descriptors, register dataflow semantics, and a shared object memory. Emphasis is on trying to find universal versions of these high level features to promote interoperability of future programming languages and to suggest a migration path for future hardware.

The memory manager design combines both automatic garbage detection and an explicit "manual" delete operation. It permits objects to be copied at any time, to compact memory or expand objects. It traps obsolete addresses and instantly forwards copied objects using a software cache of an object map. It uses an optimized write-barrier, and is better suited for real-time than a standard copying collector.

R-CODE investigates the design of type maps that extend the virtual function tables of C++ and similar tables of HASKELL, EIFFEL, and SATHER 0.6. R-CODE proposes to include numeric types and sizes in type maps, and to inline information from type maps by using dynamic case statements, which switch on a type map identifier. When confronted with a type map not seen before, a dynamic case statement compiles a new case of itself to handle the new type.

R-CODE also investigates using IEEE floating point signaling-NaNs as tagged data, and making array descriptors first class data.

R-CODE uses a new "register dataflow" execution flow model to better match the coming generation of superscalar processors. Functional dataflow is used for operations on register values, and memory operations are treated as unordered I/O. Barriers are introduced to sequence groups of unordered memory operations. A detailed semantic execution flow model is presented.

R-CODE includes a shared object memory design to support multi-threaded programming within a building where network shared object memory reads and writes take several thousand instruction-execution-times to complete. The design runs on exist-

ing symmetric processors, but requires special caches to run on future within-building systems.

ACKNOWLEDGMENTS

For an older student, with a family to support, financing is very important. I would like to thank the following people and institutions that made this thesis financially possible: my wife, my mother, and my wife's family; the Division of Applied Sciences of Harvard University; the Advanced Research Projects Agency (Contract Nr. F19628-92-C-0113); and Professor Thomas Cheatham.

Contents

1 Introduction **1**

1.1 Background . . . 1

1.1.1 S-CODE . . . 4

1.1.2 R-CODE . . . 7

1.1.3 V-CODEs . . . 9

1.2 Methodology . . . 11

1.3 Context . . . 12

1.4 Hardware Trends . . . 13

1.5 R-CODE Feature Selection . . . 15

1.5.1 Memory Management . . . 15

1.5.2 Data Types . . . 16

1.5.3 Execution Flow . . . 18

1.5.4 Shared Object Memory . . . 19

1.6 Summary of the Thesis . . . 21

1.7 Some Related Work . . . 21

2 Memory Management **22**

2.1 Goals . . . 22

2.2 Requirements . . . 22

2.3 Definitions 25
2.3.1 Memory and Reachability 25
2.3.2 Marking, Scavenging, and Sweeping 25
2.3.3 Mutators and Frames 26
2.3.4 Copying and Spaces 27
2.3.5 Forwarding 28
2.3.6 Swizzling 28
2.3.7 Manual Deletion and Type Change 30
2.3.8 Conservative Pointer Components 30
2.4 Two Level Addressing 31
2.4.1 Related Work and Alternatives to Object Maps 33
2.4.2 Address Registers 34
2.4.3 Multi-Process Single-Processor Systems 36
2.4.4 Address Register Loads and Stores 36
2.4.5 Shared Memory Multi-Processors 38
2.4.6 Copying Stops 38
2.4.7 Address Register Overhead 39
2.4.8 The Interrupt Check Alternative 41
2.4.9 Other Work Related to Address Registers 42
2.5 Concurrent Garbage Detection Algorithms 43
2.5.1 The Standard Marking Algorithm 43
2.5.2 Ephemeral Marking 44
2.5.3 Concurrent Marking Conditions 46
2.5.4 Snapshot and Non-Snapshot Detectors 47
2.5.4.1 Non-Snapshot Detectors 47
2.5.4.2 Snapshot Detectors 48
2.5.5 Read and Write Barriers 49

2.5.5.1 Read Barrier Detectors 50
2.5.5.2 Write Barriers Detectors 50
2.5.5.2.1 The Write Barrier End Game 50
2.5.5.2.2 Write Barrier End Thrashing 51
2.5.5.3 Snapshot Detector Barriers 53
2.5.6 Comparison of Read-Barrier, Write-Barrier, and Snapshot Detectors 53
2.5.7 Copying Collectors 55
2.5.8 Forwarding 57
2.5.8.1 Read Barrier Forwarding 57
2.5.8.2 Write Barrier Forwarding 58
2.5.8.3 Replication Forwarding 59
2.5.8.4 Two Level Addressing 60
2.5.8.5 Comparison of Read-Barrier-Forwarding, Write-Barrier-Forwarding, Replication-Forwarding, and Two-Level-Addressing Detectors 60
2.5.9 The R-CODE Write Barrier 61
2.5.9.1 The Bitstring AND Write Barrier Test 61
2.5.9.2 Local and Global Heaps: A Future Use for the Write Barrier Test 62
2.5.9.3 Deferred Action Buffers 63
2.5.9.4 Write Barrier Mutator Overhead 64
2.5.9.5 Related Work on Write Barriers 67
2.6 Non-Mutator Overhead 67
2.7 Summary 70

3 Data Types **71**
3.1 Goals 71

3.2 Requirements 72
3.3 Basic Data Types 73
3.3.1 Memory Units 74
3.3.2 Tagged Values 76
3.3.3 Pointers 78
3.3.3.1 The Pointer Type Field 79
3.3.3.2 Scalar Types 81
3.3.3.3 Loads and Stores 82
3.3.3.4 Related Work on Dynamic Compilation 84
3.4 Type Maps 84
3.4.1 Issues to be Analyzed 86
3.4.2 Type Map Related Work 87
3.4.3 Type Matching 88
3.4.3.1 Static Type Matching 88
3.4.3.2 Dynamic Type Matching in R-CODE 89
3.4.3.3 Dynamic Type Matching via Formal Types 91
3.4.3.4 The Formal Type ANY 93
3.4.3.5 Formal Component Descriptors 93
3.4.4 Constancy of Type Maps 94
3.4.4.1 Adding Component Descriptors to Type Maps 94
3.4.4.2 Type Variables 95
3.4.5 Actual Type Specification 95
3.4.5.1 An Actual Type Specification Example 96
3.4.6 Component Descriptor Contents 98
3.4.7 R-CODE Component Descriptors 100
3.4.7.1 R-CODE Actual Component Descriptors 100
3.4.7.2 R-CODE Formal Component Descriptors 102

3.4.8 Subobjects . . . 103
3.4.9 Examples of Deficiencies in C++ . . . 104
3.4.10 Type Maps in Other Languages . . . 106
3.5 Array Descriptors . . . 107
3.5.1 Summary . . . 109

4 Execution Flow 110

4.1 Goals . . . 110
4.2 Requirements . . . 111
4.3 Principals . . . 114
4.3.1 Virtual Computer Representation . . . 114
4.3.2 Dataflow for Register Values . . . 115
4.3.3 RAM Memory is an Input/Output Device . . . 115
4.3.4 Based on Cases, Calls, Returns, Barriers, and Exception Throws 116
4.3.5 Barriers are at Routine Top Level . . . 116
4.3.6 Return Terminates Subsequent Partitions . . . 116
4.3.7 Exception Catches are Cases Attached to Call Instructions . . 117
4.3.8 Exception Throws are like Returns with Termination . . . 117
4.3.9 Out-of-Line Equals Inline . . . 117
4.3.10 Blocks are Routines . . . 118
4.3.11 Loops are Tail-Recursive Routines . . . 118
4.3.12 Traps are like Subroutine Calls . . . 118
4.3.13 NaNs instead of Traps . . . 118
4.3.14 Either Strict or Non-Strict . . . 119
4.3.15 Eager but for Traps . . . 120
4.3.16 Memory Reads have No Side Effects . . . 120
4.3.17 Case Statements Propagate Permissions . . . 120

4.3.18 Flexible Value Lists 122
4.3.19 Second Class Frame Pointers 122
4.3.20 Controllable Inlining 123
4.4 Related Work 123
4.5 R-CODE Machine Language Control Flow Semantics 125
4.5.1 R-CODE Basic Register Dataflow 125
4.5.1.1 Registers 125
4.5.1.2 Instructions 126
4.5.1.3 Type Codes 128
4.5.2 Frame Memory 128
4.5.2.1 Register Frames 129
4.5.2.2 Frame Heap 130
4.5.2.3 Value Lists 131
4.5.3 Execution State 133
4.5.3.1 The Execution Tree 133
4.5.3.2 Tree Node States 135
4.5.3.3 State Transition Rules 135
4.5.3.4 State Maintenance 136
4.5.4 Cases 137
4.5.5 Barriers 138
4.5.6 Calls and Returns 139
4.5.7 Exception Throws 141
4.5.8 Blocks 142
4.5.9 Loops 143
4.5.10 Inlining 144
4.5.11 Localizing 144
4.5.12 Coroutines 145

4.5.13 Non-Signaling-NaNs 146
4.5.14 Traps 147
4.5.15 Continuations 148
4.5.16 Recording State 148
4.6 Implementation Challenges 149
4.6.1 Simulating the R-CODE Computer 149
4.6.2 Non-Signaling-NaN Outputs 150
4.6.3 Trap Implementation 150
4.7 Summary 151

5 Shared Object Memory 152
5.1 Goals 152
5.2 Requirements 154
5.3 Main Problems 155
5.3.1 The Cache Coherency Problem 155
5.3.2 The Thread Synchronization Problem 157
5.3.3 The Write Delay Problem 159
5.3.4 The Atomic Transaction Problem 160
5.3.5 The Hotspot Problem 162
5.4 Principals 163
5.4.1 The Ordered Partition Model 163
5.4.2 Commutative/Associative Operations 163
5.4.3 Type Change 164
5.4.4 Per Component Access Disciplines 164
5.4.5 Volatile Component Caching 164
5.4.6 Probabilistic Error Detection 165
5.5 Related Work 165

5.6 R-CODE Shared Object Memory Semantics 165
5.6.1 Access Disciplines 165
5.6.2 Access Specifications 166
5.6.3 Access Specification Combination 168
5.6.4 The Volatile Cache 171
5.6.5 The Read-Only Access Discipline 172
5.6.6 The Write-Only Access Discipline 172
5.6.7 The Volatile Access Discipline 172
5.6.8 The Write-Once Access Discipline 172
5.6.9 The Accumulate Access Discipline 173
5.6.10 The Atomic Transaction Access Discipline 173
5.7 Summary 175

List of Figures

1.1 Language Interfaces 5
1.2 S-CODE Advanced Example 7
1.3 Example AT-CODE Information Graph 10

2.1 Addressing Data 32
2.2 Address Load and Store RISC Pseudo-Code 40
2.3 Write Barrier RISC Pseudo-Code 65
2.4 Deferred Buffer Processing Loop RISC Pseudo-Code 66

3.1 A Vector of 8 5-Bit Elements 75
3.2 64-Bit and 128-Bit Tagged Values 77
3.3 128-Bit Tagged Pointer 79
3.4 The Pointer Type Field 80
3.5 Copy Types 81
3.6 Type Maps 85
3.7 Static Type Matching 89
3.8 R-CODE Default Type Matching 90
3.9 Dynamic Type Matching by Formal Types 92
3.10 R-CODE Non-Constant Component Descriptors 101
3.11 Array Descriptors 108

4.1 Tagged Register Values 126
4.2 Example Arithmetic Instruction 127
4.3 Example Instruction Execution Tree 134

5.1 R-CODE Pointers and Types 167
5.2 Combining R-CODE Pointers and Component Descriptors 169
5.3 Access Specification Combination 170

Chapter 1

Introduction

1.1 Background

After a long career in computer software development, I returned to graduate school a few years ago specifically to develop a programming language such that:

G_α: A high school student knowing a bit of algebra and geometry can investigate and change commercially written games, editors, and simulators.

G_β: The language is a suitable foundation for enhancements that merge the best of

C, Fortran,
C++, Ada (Eiffel, ...),
SML, Haskell (Id, ...),
Lisp, Emerald (Smalltalk, ...),
MatLab, Mathematica, Latex.

I did this for two reasons. First, having a family made me aware how difficult it is for average people to learn existing commercial programming languages. In particular, I became ambitious to design a language in which companies would write computer games and high schoolers would enjoy modifying them.

Second, I have had an extensive career specializing in high performance software that glues other software together, via communications or common data structures, and

this gave me insight into what made existing programming languages good or bad from the interfacing point of view. In other words, I have written in the neighborhood of 200,000 lines of code, much of it glue software, in assembly language, C, LISP, Ada, and C++, and I have developed opinions on how existing languages should be changed to make interfacing easier.

One's first idea, of course, is to design a "neat little language", and after doing a partial definition of one of these[1], and looking at the fate of SCHEME and EIFFEL, two good neat little languages, I decided that a more potent method is required.

The problem is that it is difficult to convince people, such as the commercial companies mentioned in $\mathbf{G}_\alpha$ above, to switch to a new language, without offering some very good reason to do so. The only good reason I can think of is that the new language is a good candidate for a standard that would be much better than the previous standards.

I suspect that many aspects of modern computer languages cannot be successfully standardized. But there are a few that might be, and these form the basis of an approach.

The first things that lend themselves to standardization are *high performance operations*. Any operation that is high performance is short and does not have very many degrees of freedom. Therefore it is possible to work through all possibilities and choose a best one. Thus we have been able to standardize on the 8-bit byte, and on two's complement integer arithmetic. It is not clear that two's complement integer arithmetic is actually better than one's complement, but is it not noticeably worse, and there is little chance that someone will come up with something significantly better.

Conversely, low performance operations are difficult to standardize. There is always some twist that works much better in some special case important to someone. Consider, for example, trying to standardize on a single universal sort algorithm.

The second kind of thing that lends itself to standardization is "*universal glue*". By this I mean some format for communication that many programmers adhere to so their programs will interoperate with other programs.

There are several examples of data structures used as universal glue. One is LISP: symbols, cons cells, and various types of numbers. This data structure permits many

[1] Called, for the record, "The Game Language", and unpublished

different functions to interoperate.

A similar example is MatLab. Here the data structure is a 2D matrix of complex floating point numbers, stored as two matrices of floating point numbers, one for the real part and one for the imaginary part. The imaginary matrix may be omitted. One dimension size may be set to 1 to get a vector. Both dimension sizes may be set to 1, and the result is called a "scalar". And there is a "text flag", which if set means the numbers are to be interpreted as ASCII character codes, i.e. text is represented. A vector with the text flag set is therefore a character string. And that is the whole show, data-structure-wise, upon which a large collection of interoperating MatLab subroutines is built.

The other kind of "universal glue" is syntax. The LISP S-expression syntax provides a way of interfacing programs to each other. COMMONLISP enhances this syntactic glue with its "lambda list" extensions, such as optional arguments and keyword arguments.

Universal glue pays for itself not because it is optimally efficient in terms of computer cycles or human effort, but because without it programs cannot interoperate. The more potent forms of universal glue make it easier to program certain classes of interoperating programs, which is why these forms of glue have been successful. Most successful pieces of universal glue have a fairly simple structure.

None of the lower performance universal glues to date, e.g. LISP and MatLab, has more than a restricted audience. However, the C language has a very wide audience, because it both plays the role of universal glue and has high performance.

So what parts of a programming language might we successfully standardize?

Under the guise of universal glue, one can standardize some syntax, type checking, and calling conventions: an interface that basically upgrades the LISP S-expression and COMMONLISP lambda list by adding strong typing and infix operators. I introduce such an interface, which I call *S-CODE*, for "*surface code*".

All programming languages execute programs in some abstract machine model. For example, the machine model for C includes a stack containing function frames. For another example, the model for COMMONLISP includes a memory with COMMONLISP objects and garbage collection. The runtime system is part of the machine model, but so are the hardware conventions used to make subroutine calls and maintain memory.

Standardizing a machine model for use by a set of programming languages seems to be necessary if you want the languages to interoperate. So I propose a standard abstract machine model called *R-CODE*, which stands for "*register code*", because it supports a register data flow model of program execution. R-CODE supports garbage collection and other advanced programming language features. R-CODE is in effect an interface to a virtual computer, on which many programming languages may run together. R-CODE tends to standardize high performance interfacing, and therefore is both universal glue and high performance.

Lastly, programming languages often have input/output support: e.g. `printf/scanf` in C and `<</>>` in C++. Because I want to support games and simulations, I expect to need something more potently visual. My root idea is to define data structures that are simple and logical and easy for programs to manipulate, but which can be displayed visually using predefined and complex visualization programs. I call these data structures *V-CODEs*. A simple example of a V-CODE consists of ASCII text buffers, pointers into these buffers, and character string labels, all organized into a directory information structure. This particular V-CODE has the name "*AT-CODE*", standing for "*ASCII Text Code*".

The word V-CODE itself stands for "*visualization code*", but the visualization programs that display V-CODEs can be replaced by vocalization programs that permit blind people to hear the data, and so V-CODE can equally well stand for "*vocalization code*".

The relationship of the three interfaces, S-CODE, R-CODE, and V-CODE, is indicated in Figure 1.1.

This thesis describes the rationale for a design of R-CODE. In order to give a more complete background, I introduce S-CODE, R-CODE, and V-CODEs in more detail immediately below, but then discuss nothing but R-CODE for the rest of the thesis.

1.1.1 S-CODE

S-CODE is a surface syntax which adds to the capabilities of COMMONLISP lambda lists. One important feature to be added is strong typing. To my mind there are two reasons for doing this. First, many programmers cannot write efficient code in COMMONLISP, even though in theory one can do so by adding suitable declarations. It is too hard to tell when COMMONLISP declarations are needed for efficiency

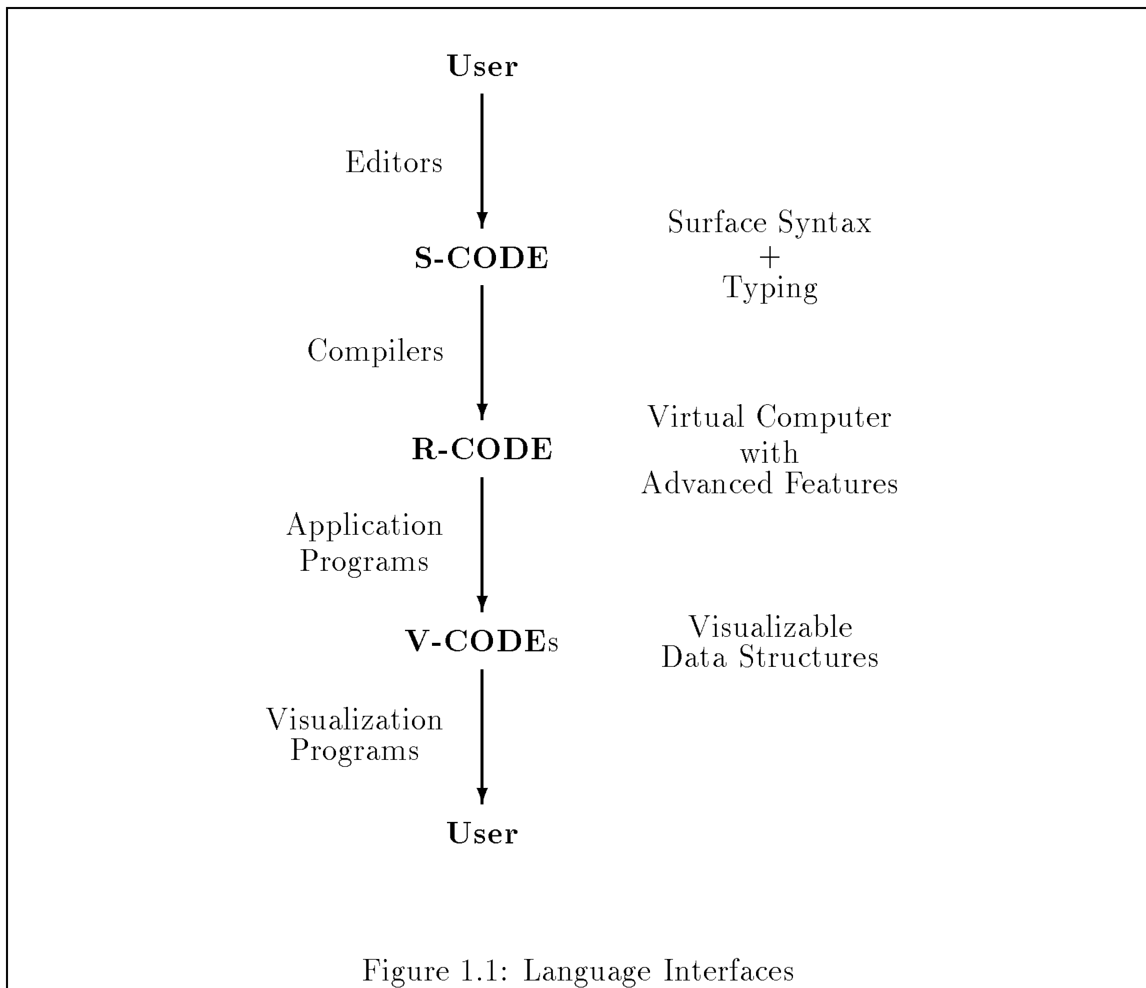


Figure 1.1: Language Interfaces

purposes. Second, overloading, as in C++ and Ada, solves many naming problems that arise when writing large systems of programs. The programmer need only worry about getting unique type names, if the names of most functions and variables include types implicitly.

Other important features to be added to S-CODE are purely syntactic features like infix operators and an improved scheme for lexical analysis.

The following is the definition in a possible version of S-CODE for the prototype of the addition/subtraction expression:

```
define (v) = (x) ?  + (y) ...  # - (z) ...
where
     x, y[.], z[.], v are each an (n)
and an n is a number
```

Here the `define` statement is defining the “form”

```
(x) ? + (y) ... # - (z) ...
```

S-CODE expressions are sequences of clauses, each consisting of some keywords followed by some arguments. In a form, arguments are indicated by identifiers in parenthesis: e.g. `(x)`, `(y)`, `(z)`. The keywords in the form we are defining are `+` and `-`. The first clause, defined by `(x) ?`, has no keyword, and the `?` indicates that this clause can be omitted. The second clause, defined by `+ (y) ...`, has the keyword `+`, and the ... indicates this clause can be repeated zero or more times. Thus `y` is really a vector of values. The third clause, defined by `- (z) ...`, is similar, but the keyword is `-`. The `#` between the second and third clause indicates that instances of these clauses can be switched in order. The lack of a `#` after the first clause means any first clause must come before any later clauses.

Forms are S-CODE expressions with “form variables” such as `(x)` in various places where subexpressions would be placed. Types are also S-CODE expressions, and `(n)` above is a form variable for a type. Form variables are indicated by surrounding them by parentheses in at least one of their uses within the form. Typing is essentially a process of matching expressions and assigning expressions as form variable values. This is just an application of the unification algorithm, analogous to unification in PROLOG.

I call this possible version of S-CODE a “*unifying clausal grammar.*”

Example Function Prototype:

```
define sort (x) into (y) such that (e{y[i],y[j]})
     for all (i) < (j)
where
     x, y are each a vector from (type1) to (type2)
and y is a write-only variable
and i, j are each a type1 variable
and e is a boolean
and type1 is an enumeration
```

Example Use of Above Prototype:

```
sort company.employees into v such that
         (v[i].name < v[j].name) for all i < j
```

Figure 1.2: S-CODE Advanced Example

Without getting into more details, we give another example of a function prototype in Figure 1.2.

Although I have done some preliminary work on S-CODE, it is not close to being formally defined.

1.1.2 R-CODE

Languages with garbage collection have problems interoperating with languages that do not have garbage collection. Even worse would be the case of two languages with different garbage collectors trying to interoperate with each other.

Therefore, it seems that developing a standard garbage collecting *memory manager* is very important to the future of computing.

If you standardize on a memory manager that includes garbage collection, then you must standardize on other aspects of memory layout as well, including program stack and frame organization, and how arguments are passed. You may also have to standardize on the way memory loads and stores are handled. In the end, you have what

amounts to a virtual computer with builtin memory management.

One might as well build machine independence into this computer too. Proposals to develop a machine independent computer, or equivalently a machine independent low level programming language, are quite old. One of the original proposals was to make a language called *UNCOL*, or Universal Computer Oriented Language[2], which was a suitable target language for every compiler, yet was machine independent. UNCOL was proposed specifically to reduce the burden of producing $M \times N$ compilers for M source languages and N types of hardware. Instead, there would be M *source analyzers* compiling the languages to UNCOL, and N *code generators* compiling UNCOL to machine language, for a total of $M + N$ compilers.

The UNCOL idea did not work originally, perhaps in part for the following two reasons. First, machines were not very standard: some had 36 bit words, some 32 bit words, some supported 8 bit bytes, some had 6 or 9 bit bytes, and so forth. Now machines are very standardized as far as data structuring is concerned, which makes it much easier to define an UNCOL, since now only instruction set differences need be hidden. The second difficulty with UNCOLs was that they were not very different from higher level languages such as C and FORTRAN. So people who wanted to work on developing an UNCOL were often told to use C instead. In fact, C has been used as an UNCOL successfully to implement C++[Str94, section 3.3.1], COMMONLISP[YHS], EIFFEL[M$^+$91, section 6.3.3], and other languages. But of course C does not include a standard garbage collecting memory manager.

R-CODE is an UNCOL that includes a garbage collecting memory manager and is packaged as a virtual computer. It has its own data organization, instruction set, etc. Implementations on real computers must promise to make the virtual computer look real and be "efficient".

The R-CODE virtual computer looks something like a RISC machine. It has 65,536 virtual registers per routine execution, so there is no need to map more than one

[2] A modern survey of UNCOLs can be found in [Maca]. One study committee commented in 1958 that the concept "had been discussed by many independent persons as long ago as 1954. It might not be difficult to prove that 'this was well-known to Babbage'."

variable to a single register. The registers are 128 bits and can hold tagged data, including 128 bit floating point numbers. The instructions are 64 bits, and have plenty of space for option flags. However, both the registers and instructions are virtual: R-CODE is compiled to real machine languages that take advantage of of the fact that the types of values in registers are known at compile time, so R-CODE registers can be mapped to real untagged registers, R-CODE instructions can be mapped to real untagged instructions, and R-CODE will execute efficiently. Nevertheless, when R-CODE routines are stopped during debugging, the debugging interface of the R-CODE virtual computer makes things look as if R-CODE were being directly executed.

Besides a garbage collecting memory manager, R-CODE includes other features not found in normal computers. These features are all high performance universal glue features that we feel will be necessary for programming languages in the future. Included are type maps, array descriptors, tagged data, register data flow, and shared object memory. Later in this chapter we will introduce these in more detail.

R-CODE gets it name, "register code", from one of its features: register data flow.

1.1.3 V-CODEs

Some of the programs we are interested in having high school students modify are games, simulations, spreadsheets, word processors, and picture processors. All of these contain some data structured in a way natural to the program, and some fairly large piece of programming that permits users to visualize this data.

The V-CODE idea is to define data structures natural to classes of programs such that the programming necessary to visualize these data structures can be a prewritten fixed *visualization program* plus a few simple application specific rules. Then the application programmer will only have to worry about manipulating the data structures, which are natural to the application, and will not have to worry about programming visualization.

A side benefit of this approach is that the visualization program can be replaced by a *vocalization program* that will make the data structures accessible to blind people. Since the data structures are natural to the application, this should be an efficient way of proceeding.

Therefore, a *V-CODE* is a data structure natural to a class of programs such that

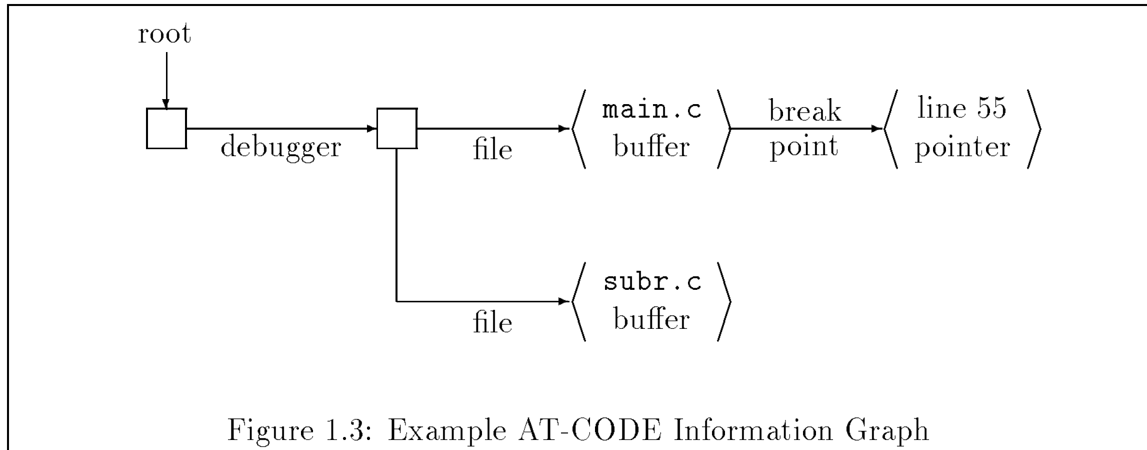


Figure 1.3: Example AT-CODE Information Graph

a visualization (and vocalization) program can be prewritten to provide a visual (vocal) interface between a person and the data structure, with the help of some simple application and person specific rules.

Work has been done on the simplest V-CODE, which is built around ASCII text buffers, and is called *AT-CODE*, or "*ASCII Text Code*." The AT-CODE information structure is an information graph whose nodes are buffers, pointers at characters in buffers, pointers at regions in buffers, and symbols. The symbols are just character strings used as labels, as in LISP (but without any associated values or functions). An example information graph is given in Figure 1.3.

In this example there can be several programs running, one of which is the `debugger`. The `debugger` has several `file` attributes, each of which is an ASCII text buffer mapped onto a file. These buffers can have `breakpoint` attributes, which are each a pointer to a line in the file where a breakpoint has been set by the debugger.

The visualization program can display the `file` buffers just like an editor. The visualization program may be given the rule:

```
.debugger.file<buffer>.breakpoint<pointer> ===> highlight green
```

which causes the visualization program to display any `debugger file breakpoint` line in green. The visualization program may be given the rule:

```
.debugger.file<buffer> ===> on b send breakpoint <cursor>
```

which causes the visualization program to send the command

`breakpoint` *copy-of-cursor*

to the debugger program whenever the user types a 'b' while a `debugger file` is the current edit buffer.

Other V-CODEs could represent data that could be visualized as spreadsheets. More advanced V-CODEs could represent abstract syntax trees that could be rendered into displays with variable-width fonts and mathematical formulas; or could represent data in a simulation that could be rendered into 3D pictures as in a flight simulator.

1.2 Methodology

Clearly I am interested in defining general purpose interfaces. In order to do so, one may engage in the following kinds of activities:

$\mathbf{M}_1$: Define interfaces and analyze them. Analysis consists of listing options and giving reasons pro and con for each option. A more or less exhaustive search of the "reasonable option space" is desirable.

Analysis might use tools such as rough timing estimates and small pieces of pseudocode expressing either implementation or usage.

$\mathbf{M}_2$: Build a system employing the interface and test it on a small scale.

$\mathbf{M}_3$: Build a system employing the interface and attempt to "market" it to the world at large. This may consist of giving the system away for free and attempting to get a large number of customers. One could also charge a fee for the system, but since we are dealing with general purpose interfaces, this is not so likely as in other areas of software.

My approach is to mine method $\mathbf{M}_1$, analysis, until it starts running dry. I believe it yields more meaningful results for general purpose interfaces than method $\mathbf{M}_2$, small scale implementation. It is also cheaper.

In the case of R-CODE there is enough analysis work for at least one thesis, and this thesis contains only analysis.

1.3 Context

Programming languages take about 10 years to develop and popularize, and then last perhaps 30 years, so:

> $\mathbf{C}_1$: Our interfaces are for use primarily in the years 2005-2035 A.D.

We have a strong interest in what should be, rather than what is. This is an appropriate attitude for a Ph.D. thesis, and it causes no problems on the software side, for that is under our control. But hardware is a different matter. We need to develop the art of getting computer manufacturers to put special hardware into every computer sold to high school students.

A hint on how to proceed comes from the current relationship of RISC to CISC computers. Digital Equipment Corporation is phasing out production of their VAX computers. Instead they "emulate" the VAX on their new RISC ALPHA's. This "emulation" technology actually consists of compiling binaries most of the time, and typically has an inefficiency factor of 3[Dig94, section 4.4]. Thus a 150 MIPS ALPHA runs VAX code at the rate of 50 MIPS, which is a very fast VAX.

Therefore our method will be to design software that will run with some inefficiency factor, possibly up to 2 or 3 but hopefully much lower, on existing hardware, and will run optimally, without inefficiency compared to current software, if new hardware is added to current computers. This consideration only affects R-CODE, and not S-CODE or V-CODE, which are not hardware sensitive.

What must then happen is for the software to become popular before any hardware is built. After the software becomes widespread, computer manufacturers will find it beneficial to make the optimizing hardware.

In order to become popular while being inefficient, R-CODE will have to offer capabilities worth the inefficiency. In some cases this my be done by simply making the new capabilities optional additions to existing efficient capabilities, and making the new capabilities as efficient as possible. Then commercial programmers can chose whether to use the new capabilities for a piece of code, or stick with the old capabilities. Usually over half the lines of code in a program are not executed enough to make efficiency a problem, and so there may be a lot of use for advanced capabilities even if they are inefficient. An example of this is the several hybrid LISP/C systems,

such as EMACS, in which LISP is used for control, where efficiency is not required, and C is used for operations that must be efficient.

In other cases the new capabilities will not be optional. In these cases they must be worth something significant.

So in summary:

$\mathbf{C}_2$: R-CODE must be popularized on existing hardware, but may require new hardware to run optimally. R-CODE may be inefficient by a factor of as much as 2 or 3 when run without new hardware, but should be fast enough to become popular.

$\mathbf{C}_3$: An inefficient feature in R-CODE must either be optional, or must have worth sufficient to overcome the adverse value of its inefficiency. Optional features must interoperate well with standard features.

1.4 Hardware Trends

Since I am attempting to develop programming languages for the future, in particular for 2005-2035 A.D., we must pay attention to the likely development of hardware in this time frame. By this I mean we must consider current hardware trends; <u>not</u> new hardware one might hope for. We are interested mostly in personal computer hardware: what every high school student will have.

From the point of view of programming languages, computer hardware is changing more rapidly now than at any time in the last several decades. Three major themes are increased parallelism, the "memory wall", and increased network speed.

The increase in parallelism means that sequential execution is no longer a viable assumption, even for personal computers. Some consequences for our $\mathbf{C}_1$ time frame (2005-2035 A.D.) are:

$\mathbf{H}_1$: Even personal computers are switching to superscalar processors, which simultaneously execute two or four instructions[E^+95]. A superscalar processor needs to execute instructions out of order so that it can keep its multiple pipelines full of useful work.

H_2: Shared memory multi-processors have made their appearance for desktop workstations. I expect to see 1 to 16 processors per personal computer in our time frame.

The "*memory wall*" is the name given in [WK95] for the phenomenon that processor speeds are increasing at the rate of 80% per year while DRAM memory speeds are only increasing at the rate of 7% per year. The faster existing microprocessors, such as the ALPHA with a peak speed of 300 MIPS, are already suffering secondary cache miss times of 30-100 instructions[F+95], with corresponding adverse impact on timing[Sit92, Appendix A]. Some consequences for our time frame are:

H_3: Organizing lists using small discontiguous CONS cells is becoming very inefficient due to large cache miss times, and so lists will probably be represented in the future by variable length vectors, analogously to character strings.

H_4: Subroutine calls will have a high overhead, and routines should be inlined much more frequently. Documentation for the ALPHA already advises that routines shorter than 20 instructions should be inlined[Sit92, A.2.3].

The memory wall also provides an additional reason for wanting to reorder instructions, as per **H_1** above. Because memory is becoming slower relative to processors, processors need memory read instructions to be executed as early in the instruction stream as possible.

The increase in network speed is indicate by the progression from Ethernet to FDDI to ATM. Inexpensive board-mounted lasers with gigabit per second speeds are becoming available. The latency limit for getting from one side of a building to another at the speed of light is of the order of 1 microsecond, and this is likely the only limit on network communications that will be operative during our time frame. We expect to see within-building networks evolve somewhat in the direction of shared memory buses, though the latency will be on the order of 1,000 instruction executions, and networks will have to operate somewhat differently from normal shared memory because of this.

Although I do not expect high school students to have very high speed network connections at home, even in our time frame, they will have such in school. Therefore the consequences for us are:

H_5: A large number of processors connected by networks capable of "paging in" an object in 1,000 instruction execution times will be available sometimes. These should at least be usable by parallel compilers and text processors.

Of course there will be other hardware developments, such as networks that replace the current phone system. But these are not high performance, so they have little impact on R-CODE, and I will ignore them.

1.5 R-CODE Feature Selection

R-CODE defines a virtual computer such that the assembly language for this virtual computer is a good target language for compilers, but still machine independent.

The purpose of R-CODE is to get everyone to standardize on a common garbage collector and some similar very capable features. The next question is, what exactly should these very capable features of R-CODE be?

1.5.1 Memory Management

Clearly one feature should be garbage collection. However, when I thought about trying to get commercial programmers in general to accept garbage collection (as per $\mathbf{G}_\alpha$, page 1), I became uncomfortable. Garbage collection has not been sellable to that bunch for 30 years. The most commonly given reason is the desire to eliminate pauses, but the technology to eliminate pauses has been around for 10 years, at least.

I concluded that commercial programmers would not want to buy into a memory manager that did not permit manual deletion, via the good old fashion "delete operation". There are many instances in programming when you know something should be deleted and you do not want to have to track down and kill all the pointers to it. However, once an object is manually deleted, use of any dangling reference to it should be flagged as an error, especially when students are modifying code. So in this sense, manual deletion should be "strongly typed".

I also concluded that ability to move an object at any time is an extremely strong selling point in a memory manager. One use of this is to compact memory very slowly

to avoid interfering with the time response of the application code. Another use is to expand a table when needed.

Lastly I feel that commercial programmers will want the memory manager to have good timing characteristics, and perform in real-time when necessary.

Thus I come to our first R-CODE features:

$\mathbf{F}_1$: R-CODE will have a memory management system with automatic garbage collection, a strongly typed manual delete operation, and an operation to move an object at any time.

$\mathbf{F}_2$: R-CODE will have a memory management system with timing characteristics suitable for real-time applications.

For the last few decades, long term data has been stored in files. But this is changing, and in the future data will be stored in databases that store objects which point at each other. When such a object oriented database is input to main memory, there are two problems. First, all the object pointers must be adjusted to point where the objects land in memory. Second, objects must not be read until they are needed, as the entire database will often be much too big to put into main memory.

The operation of adjusting pointers in this situation goes under the name of "swizzling" the pointers. Since objects are not read into memory until they are needed, swizzling pointers must be delayed until the pointers are used. Thus memory management gets involved, and we are led to our next feature:

$\mathbf{F}_3$: R-CODE will have a memory management system that supports delayed swizzling of pointers so external databases can be input efficiently into main memory.

The chapter on memory management proposes ways of providing the above features.

1.5.2 Data Types

The next issue I faced, after garbage collection, was what aspects of data types should be present at run time and directly supported by R-CODE. Associated with every type of object must be some function to perform garbage detection activities.

Specifically, there must be a "*scavenger*" function for every type that discovers the pointers in any object of that type, and also discovers for each pointer the type of the object it points at.

Modern object oriented languages associate with every type of object a list of functions that can do various things to the object, such as print it or display it graphically. In the high performance object oriented languages, such as C++, this list is organized as a vector, and the compiler always knows at compile time the index of the element it wants from the vector to perform a given operation.

We call such a vector of information about a type a "*type map*", and we make type maps visible in R-CODE. The garbage collector is our first customer for type maps, but there are other important uses.

In modern programming languages it has become important to distinguish between the type a routine thinks an object has, which we call the object's "*formal type*", and the type the object actually has, its "*actual type*". Type maps may be used to make objects of different actual types appear to have the same formal type. Then one routine can be used on many actual object types.

There are many uses for this. For example, programs written at one time can be applied to data defined at a different time if a type map can compensate for the differences between what the program expects and the actual data format. Or "polymorphic" routines can be written to handle any type of object that has certain components and operations.

There are two reasons for making type maps part of R-CODE. The first is to be sure that different programming languages can interoperate. The way type maps are created and passed between routines needs to be standardized, and certain components of type maps, such as the scavenging functions mentioned above, need to be standardized. Other components could, however, be language dependent.

The second reason for making type maps part of R-CODE is efficiency. The functions specified by type map components should often be inlined. Also, many of these functions reduce to a single load or store instruction, so that the only information really in the type map is the displacement, size, and numeric type of a component. Special hardware can make type maps containing this kind of information efficient, but before such hardware is built, software techniques are needed. One such technique uses "*dynamic case statements*" that switch on the type map input to a particular piece of code, and dynamically compile new cases when the case statement sees a new

type map.

Thus the feature we want R-CODE to support is:

> **F_4**: R-CODE will support type maps, and will efficiently handle execution of very small functions selected from a type map, and also efficiently handle loads and stores in which only displacement, numeric type, and size are stored in the type map.

Just as the map from an actual object to a formal object (i.e. the object as seen by a routine) is encoded in a type map, a linear map from a list of subscripts to an address within an array is encoded in a map we call a "*array descriptor*". One of the things I learned during my decades writing glue software is that array descriptors are the right way to treat array accesses. This is because there are many instances when pieces of an array need to be treated as arrays in their own right. Thus:

> **F_5**: R-CODE will support array descriptors, treating them as first class data, except that they may not be modified once created.

Lastly, LISP and other languages permit use of tagged data whose type is not known at compile time. There are times when this is indispensable, as when LISP S-expressions are used to describe linguistic information. Therefore:

> **F_6**: R-CODE will support tagged and untagged data and an efficient transition between the two kinds of data.

The chapter on data types proposes ways of providing these R-CODE data type features.

1.5.3 Execution Flow

We are free to develop completely new languages, and thus escape the baggage of the past. Two problems suggest our languages should be more functional.

First, as indicated by **H_1** above (page 13), compilers need to be able to reorder instructions to get hardware efficiency. Functional languages are insensitive to order of execution, as long as computations terminate.

Second, in order to make it possible for high schoolers to understand programs well enough to change them, we need debuggers that do a very good job of displaying what a program is doing. Functional languages are single assignment languages, in which each "variable" gets only a single value during its lifetime, and this makes it easier to understand the program and to display what the program is doing.

Functional languages can handle most programming tasks easily and efficiently. But, on the other hand, they usually cannot express everything required in a single program. They cannot handle objects and arrays with changing state. They have trouble with algorithms that require state maintenance, such as histogram computation or graph tracing.

The extra efficiency one might get from functional languages derives from the fact that they can be executed in data flow order: each primitive operation (e.g. add) can execute whenever its inputs are ready. To maximize the parallelism this permits, special hardware is needed to detect when the inputs to an operation are ready (see, for example, the work of Burton Smith[Smi78, ACC+90] and Arvind[Nik91, Pap90]). But R-CODE must run efficiently without special hardware.

Therefore R-CODE uses a compromise. It is possible to compile a functional language efficiently for a sequential computer as long as all the variables are register-like, in that they are local to a routine, cannot be aliased at runtime using pointers, and therefore the compiler can figure out at compile time when they will receive their values. But variables that are global or are array elements addressed by general subscripting cannot be handled efficiently using data flow execution order on existing computers. So the compromise is:

> $\mathbf{F}_7$: R-CODE will be a functional data flow language with register-like variables that treats RAM memory as an I/O device.

R-CODE gets its name, "*register code*", from this feature, which is investigated in the chapter on execution flow.

1.5.4 Shared Object Memory

As mentioned above ($\mathbf{H}_5$, page 15), sometimes student computers will be connected in a building-wide network capable of "paging in" an object in 1,000 instruction

execution times. Some means of easily using this feature to write parallel compilers, text processors, simulators, calculators, and games is needed.

The memory of the computers on the network becomes, with the help of the network, a kind of disk-like memory that can store objects and retrieve them more than 1000 times faster than current disks. The question is: what kinds of operations can be performed on objects in this "*shared object memory*?"

First, many objects will turn out to be read-only after they are created, just as many files are. The operations on these are creation, reading an object, and reading part of a large object.

For functional programming, it is also convenient to have objects that go through an initial write-only phase, during which many processes write them but cannot read them. Then the objects are switched to read-only, after which they cannot be written.

A histogram is similar but undergoes an accumulate-only phase, during which many processes may add to its elements. Then, when it is finished, it is switched to read-only.

This thinking leads to the following:

> $\mathbf{F}_8$: R-CODE will support a shared object memory in which individual object components can be be marked as read-only, write-only, write-once, or accumulate-only as part of the component type. Also, objects may change types dynamically, so their components can, for example, switch from write-only to read-only.

However, this approach is not always enough, so:

> $\mathbf{F}_9$: R-CODE shared object memory will support atomic multi-object transactions.

Special hardware will be necessary to make a real networked shared object memory, but this is not available, even approximately, today. However, shared memory symmetric multi-processors have been available for some time, and are working their way toward student computers (see $\mathbf{H}_2$, page 14). This leads to:

> $\mathbf{F}_{10}$: R-CODE shared object memory will be efficient on existing symmetric multi-processors, but may require specially designed hardware for efficiency on a within-building network.

The chapter on shared object memory explores these features.

1.6 Summary of the Thesis

The remainder of the thesis consists of four chapters, matching the four groups of features just introduced. Each of these chapters may be read independently of the others.

There is also a separate R-CODE Architecture Description [Wal] which maps the ideas of this thesis onto a specific architecture. However, this thesis and the Architecture Description are independent of each other.

1.7 Some Related Work

Each of the following chapters cite work related to the topic of the chapter. But because the chapters are on ambitious advanced features, many historical efforts to provide student usable languages and UNCOLs are not listed. Therefore we list some of these briefly here.

Some of the more popular student oriented languages are BASIC, SCHEME, and MATLAB. Others are SMALLTALK and MATHEMATICA. Some more recent attempts in these directions are EIFFEL[Mey92] and SATHER[Int94], which develop type dispatching ideas related to type maps, SELF[U^+93], which develops dynamic compilation ideas, and DYLAN[App94], which combines strong typing and LISP.

The original UNCOL proposal is described in [Maca, CPW74]. Some efforts over the years to build UNCOL languages are JANUS[CPW74, HW78] and ANDF[Macb, Def94], which are machine independent languages with approximately the same capabilities as C, but with lower level data types and only basic operations.

Chapter 2

Memory Management

2.1 Goals

The main goal of this chapter is to:

> $\mathbf{G}_\mu$: Find a design for a universal memory manager that everyone can share.

The main motivation for this goal is that it is very hard for two pieces of code to interoperate if they each assume a different memory manager. Thus, however difficult our goal may be, there is considerable value in reaching it.

The method for achieving this goal is to examine all the reasonable alternative ways of building a memory manager that might meet the requirements of the next section, and pick the alternatives that seem best. There are quite a few alternatives, leading to a lengthy analysis.

2.2 Requirements

The R-CODE memory manager is to support new programming languages meeting the requirement (see page 1):

> $\mathbf{G}_\alpha$: A high school student knowing a bit of algebra and geometry can investigate and change commercially written games, editors, and simulators.

Clearly one memory management feature should be garbage collection. However, when I thought about trying to get commercial programmers in general to accept garbage collection, as per $\mathbf{G}_\alpha$, I became uncomfortable. Garbage collection has not been sellable to that bunch for 30 years. The most commonly given reason is the desire to eliminate pauses, but the technology to eliminate pauses has been around for 10 years, at least.

I concluded that commercial programmers would not want to buy into a memory manager that did not permit manual deletion, via the good old fashion "delete operation". There are many instances in programming when you know an object should be deleted and you do not want to have to track down and kill all the pointers to it. The object may have substantial memory resources, and you do not want to wait until an automatic garbage detector has run before these are recovered and reused. However, once an object is manually deleted, use of any dangling reference to it should be flagged as an error, especially when students are modifying code. So in this sense, manual deletion should be "strongly typed".

After being manually deleted, memory would be returned to some free list for immediate reuse. Users would be able to design their own free memory lists and memory allocators.

For example, an object may be deleted when it ceases to be in some "directory" and is not "open". In this case there may be one reference count for the directory and another for being open, and the object is deleted when both counts go to zero. The directory count may be used separately to initiate cleanup action.

Or as another example, a communication system may use fixed size message buffers that are allocated from queues of free blocks of memory of the appropriate size, and returned to these queues as soon at they are no longer needed. In this case there might only be one reference count of the number of queues a buffer is on plus the number of times the buffer has been "opened" by processes.

In general, users of the memory manager, and not the manager itself, should maintain reference counts. Compilers may automate this where appropriate.

I also concluded that ability to move an object at any time is a strong selling point in a memory manager. One use of this is to compact memory very slowly to avoid interfering with the time response of the application code. Another use is to expand a table when needed.

Lastly I feel that commercial programmers will want the memory manager with good overall timing characteristics, that can perform in real-time when necessary.

Thus I come to our first R-CODE memory management features:

> **F_1**: R-CODE will have a memory management system with automatic garbage collection, a strongly typed manual delete operation, and an operation to move an object at any time.

> **F_2**: R-CODE will have a memory management system with timing characteristics suitable for real-time applications.

The other memory management feature concerns object oriented databases, which I expect to be used increasingly in the future. When such a object oriented database is input to main memory, there are two problems. First, all the object pointers must be adjusted to point where the objects land in memory, an operation called "swizzling" the pointers. Second, objects must not be read until they are needed, as the entire database will often be much too big to put into main memory.

Since objects are not read into memory until they are needed, swizzling pointers must be delayed until the pointers are used. Thus memory management gets involved, and we are led to our last memory management feature:

> **F_3**: R-CODE will have a memory management system that supports delayed swizzling of pointers so external databases can be input efficiently into main memory.

This chapter is organized into four parts. The first part contains definitions. The second part discusses two level addressing, my solution to the need for strongly typed manual deletion and dynamic object movement. The third part discusses how to structure garbage detection in order to minimize timing impact on application processes, and introduces a bit string AND write barrier test to this end. The fourth part discusses other overheads of automatic garbage collection.

Below I will reference other work that impacts on specific parts of my analysis. Some more general surveys of memory management are Wilson's modern survey of garbage collection[Wil92], the introduction to Hayes's recent thesis[Hay94], Cohen and Nicolau's 1981 survey of garbage collection[Coh81], Cohen's 1983 survey of compaction[CN83], and Hickey and Cohen's 1984 analysis of performance[HC84].

General references can be found in the surveys just cited.

2.3 Definitions

In this section I will define an abstract model of memory management suitable for investigating the design of the R-CODE memory manager. As befits an abstract model, I omit details which I do not believe will affect the outcome.

2.3.1 Memory and Reachability

Memory is a sequence of *objects*, *free blocks*, and *gaps*. A gap is an unimplemented piece of memory; a free block is piece of memory that is free to be allocated to objects. Each object, free block, or gap has a non-zero positive integer *length*, and each has an *address* which equals the sum of the lengths of the previous objects, free blocks, and gaps in memory.

Objects contain *pointers* to other objects. A pointer to an object is in effect the address of the object. The different places where pointers can be stored within an object are called *pointer components*. A pointer component may be empty, meaning it contains no pointer. Such a pointer component is said to be *null*. Two pointer components in the same object may not overlap.

An object is said to be *reachable* if it is one of a particular set of objects, called the *root set*, or if it can be *reached* from the root set by following the pointers in objects. In order to be accessed, an object must be reachable; and furthermore, once an object becomes unreachable, it stays unreachable. Unreachable objects may therefore be garbage collected.

2.3.2 Marking, Scavenging, and Sweeping

Marking is the process of putting a mark on each object that is reachable, and leaving the mark off of most objects that are unreachable. To this end, each object may be thought of as having a *marked bit* which is set if the object is marked, and clear otherwise. To *mark* an object is to set its marked bit.

To *scavenge* a pointer component is to mark the object pointed at by the component if the component is not null and the object pointed at is not already marked.[1] To

[1] Sometimes this is called *scanning* the component.

scavenge an object is to scavenge all its pointer components. All objects that have been marked must be scavenged, so that it is convenient to think of each object as having a *scavenged bit* which is set when the object is scavenged, and clear otherwise.[2] There are questions of timing in concurrent algorithms: do you set the scavenged bit just before or just after scavenging an object.

To *sweep* means to find all unmarked objects and turn them into free blocks. To *sweep* an object means to turn the object into a free block if and only if its marked bit is off.

2.3.3 Mutators and Frames

A *mutator* is an application program process that is trying to do useful work, and for whose benefit the garbage collector algorithm is being run. There may be any number of mutators running on any number of hardware processors. Each mutator has a *stack*, which is a sequence of objects called *frames*. The frame at the top end of the stack is called the *top frame*. A mutator may perform the following actions:

Create a new object all of whose pointer components are null, and write a pointer to this new object into the top frame.

Write a pointer into a pointer component. The pointer written must be the value of a pointer component in the top frame, and the object containing the pointer component to be changed must be a non-frame object pointed at by the top frame.

Read a pointer from a pointer component into the top frame. The pointer component must be in a non-frame object pointed at by the top frame, and the value of that pointer component will be written into the top frame.

Load-root: load a pointer to a non-frame root object into the top frame.

Reference an object. Some non-pointer-component part of the object is read or written. The object must be a non-frame object pointed at by the top frame.

[2] In the literature[Wil92] the marked and scavenged flags are often represented by assigning one of three colors to an object: white = unmarked and unscavenged; grey = marked and unscavenged; black = marked and scavenged.

> *Push* a new frame into the top of the stack. The new top frame is created as a new object with null pointer components. Then some pointer components of the previous top frame may be copied into the new top frame.
>
> *Pop* the top frame from the stack. The top frame must not be reachable after it has been popped. Some pointer components of the top frame being popped may be copied into the frame that becomes the new top frame.

Objects (including frames) may not contain pointers to frames.

The frames in every mutator stack are part of the root set.

A stack may only be changed by its own mutator, and that mutator may only change the top frame of its stack.

In the R-CODE virtual computer, mutator stacks may share frames that are not top frames, and so the abstract model of this chapter does not exactly fit the model of R-CODE in other chapters. But as stated above, I have intentionally omitted this and other extra complexities from my memory model because they make no essential difference to the analysis of this chapter.

2.3.4 Copying and Spaces

To *copy* an object means to move it in memory, changing its address. To *compact* memory means to copy objects so as to glue free blocks together to make larger free blocks. Objects may be copied for reasons other than compacting memory. They may be moved because their size has increased and they cannot expand where they are.

A *space* is a subset of memory such that every object and every free block is either completely outside or completely inside the space. Typically a space is a contiguous region of memory addresses, but it may also be a set of pages.

Objects are sometimes copied between spaces. Some spaces may be memory accessible by only some processors in a multi-processor system. Database files that are not actually in memory may be viewed as if they were a space in memory for our purposes. Therefore a mutator may not be able to reference or mutate objects in some spaces, and the objects may have to be copied to another space in order for the mutator to access them.

A *copying collector* is a kind of garbage collector that works by copying all reachable objects in one space, called the *from-space*, into a second space, called the *to-space*. After all reachable objects are copied, the from-space is freed. This kind of garbage collector naturally compacts memory.

2.3.5 Forwarding

Consider the situation where an object has been copied one or more times and there is a single most recent copy of the object, along with possible older copies. In this situation, which I call the *forwarding scenario*, it is desirable to adjust each pointer component pointing at the object to point at the most recent copy. This adjustment is called *forwarding* the pointer component. A pointer that points at the most recent copy of the object it points at is said to be *forwarded*.

As an aid to forwarding, a pointer to the new location of a copied object is often left at the previous address of the object.[3]

The question arises: which copy of an object does a mutator read or write. If the mutators all access only the most recent copy of an object, I say the memory management system employs *eager forwarding*. If the mutators may read a non-recent copy of an object, and must write all copies of the object identically, I say the memory management system employs *lazy forwarding*.

In an eager forwarding system, pointers may be forwarded when they are read into the top frame. In many systems, eager or lazy, pointer components are forwarded when they are scavenged.

2.3.6 Swizzling

Consider the situation where there are two spaces, one inaccessible to mutators and one accessible to mutators. The inaccessible space has other desirable properties: it might be permanent disk storage, for example, or it might be accessible to mutators other than those we are considering. Objects may have copies in either space or in

[3]Storing this address is sometimes called "forwarding the object," but I carefully avoid this terminology, since in my scheme it might mean instead that all pointer components in the object were forwarded.

both, and may be copied back and forth between the spaces. However, it is desired that an object copy in one space only point at object copies in the same space.

In this situation, which I call the *swizzling scenario*, adjusting a pointer component in one space to point at a copy of its target object in the same space is called *swizzling* the pointer component. To swizzle all pointer components in an object is called *swizzling* the object. A pointer stored in one space that points to the same space is said to be *swizzled*.[4]

An object that has a copy in one space may have an empty place reserved for it in the other space. Such an empty place is called an object *slot*.

When a pointer is swizzled, a check is made to see if the object pointed at has already been copied, and if yes, the pointer is swizzled to point at this copy. If no, a slot is allocated for the object, and the pointer is swizzled to point at this slot.

In the swizzling scenario, it is necessary to delay copying objects from the inaccessible space to the accessible space until the objects are actually accessed by the mutators, because if all objects were copied, there would be too many copies. There would even be too many copies if all objects pointed at by a copied object were copied, for then all objects reachable from the root of a database would be copied when the root was copied. In order to delay copying, it is necessary to delay swizzling.

Pointer components might not be swizzled until they are used to access the objects they point at. This is called *demand pointer swizzling*. To implement this, something must be done to distinguish unswizzled pointers from swizzled pointers.

Or all the pointer components of an object may be swizzled when the object is first accessed, with the swizzled pointers possibly pointing at empty slots into which other objects will be copied. This is called *demand object swizzling*. In this situation, the act of using a pointer to access an object will cause the object to be copied if the pointer pointed at an empty slot, and will cause the object to be swizzled if the pointer pointed at an unswizzled object. To implement this, something must be done to distinguish empty slots or unswizzled objects from swizzled objects.

Swizzling and forwarding should not be confused with each other. The same memory management system might implement each by a different method. The two level addressing scheme used in R-CODE does exactly this.

[4] A history of swizzling is given in [SKW93, section 2.5].

2.3.7 Manual Deletion and Type Change

Manual deletion refers to deleting an object and most of its memory in response to an operation executed by a mutator. *Strongly typed manual deletion* requires that all accesses to a manually deleted object be trapped as errors. Strongly typed manual deletion of an object is conceptually similar to copying the object into an inaccessible region of memory and instantly forwarding all pointers to the object.

Type-change refers to changing the "type" of an object in a strongly typed system, in response to an operation executed by a mutator. For example, a write-only object might be made read-only. *Strongly typed type-change* requires that the addresses used to access the object under the old type must be immediately invalidated, and the object must be given a new address. This is similar to deleting an object manually and allocating a new object that is a copy of the original. Attempts to access the object using its old address result in detected errors, just as in the case of strongly typed manual deletion.

2.3.8 Conservative Pointer Components

Sometimes it is not known what parts of an object are pointer components, though it is possible to list all places in the object that might be pointer components. Places that might be pointer components are called *conservative pointer components*. For purposes of finding all reachable objects, knowing all conservative pointer components may suffice, if one can tell a valid object pointer from random data. What one does is to try to interpret each conservative pointer component value as a pointer to an object, and if this fails, ignore the value, while if it succeeds, declare the object pointed at to be reachable.

A value might be interpretable as a pointer and yet not really be a pointer. Thus an unreachable object might accidentally be declared to be reachable. More importantly, one must never forward or swizzle a conservative pointer component.

A number of papers[Zor93] have been written on *conservative garbage collectors*, which are just garbage collectors that use conservative pointer components in lieu of knowing where all the pointer components are. However, there does not seem to be any reason why R-CODE should not be able to correctly identify pointer components, so I will not mention conservative collectors or conservative pointer components again in this thesis.

2.4 Two Level Addressing

Feature $\mathbf{F}_1$ (page 24) requires strongly typed manual deletion and instant forwarding of pointers to objects whenever the objects are moved. The only way to provide strongly typed manual deletion seems to be to associate a trap flag with each object, such that when the trap flag is set, all accesses of the object are trapped. Similarly, when an object is copied, the only way to immediately forward all pointers to the object seems to be to make all accesses of the object use indirect addressing through a table location that holds the current address of the object.

These ideas lead to the scheme I call *two level addressing*, that seems to be the only way of satisfying the requirements of $\mathbf{F}_1$. Two level addressing provides a trap flag that can also be used to provide demand object swizzling, as required by feature $\mathbf{F}_3$(page 24).

In the two level addressing scheme, pointers contain an *object number* and a within-object byte displacement. The object number specifies an entry in a table called the *object map*, and this entry has the base address of the object (see Figure 2.1). The base address and the within-object displacement are added to get the address of the byte accessed. The object map entry also contains a *trap flag* that may be set to trap all references to the object.

To move an object dynamically, a process does the following:

1. Set the trap flag of the object to stop all processes accessing the object.
2. Move the object.
3. Update the object map entry to point at the new address of the object.
4. Clear the object trap flag.
5. Restart any processes that stopped because they tried to access the object.

Strongly typed manual deletion of an object merely requires setting the trap flag in the object map entry, and deleting the memory used by the object, without deleting the object map entry. The object map entry is automatically collected by a mark and sweep garbage collector at some later time.

Stopping processes that try to access a moving object is not optimal. It is possible to stop only processes that write an object, as long as the source and destination copies

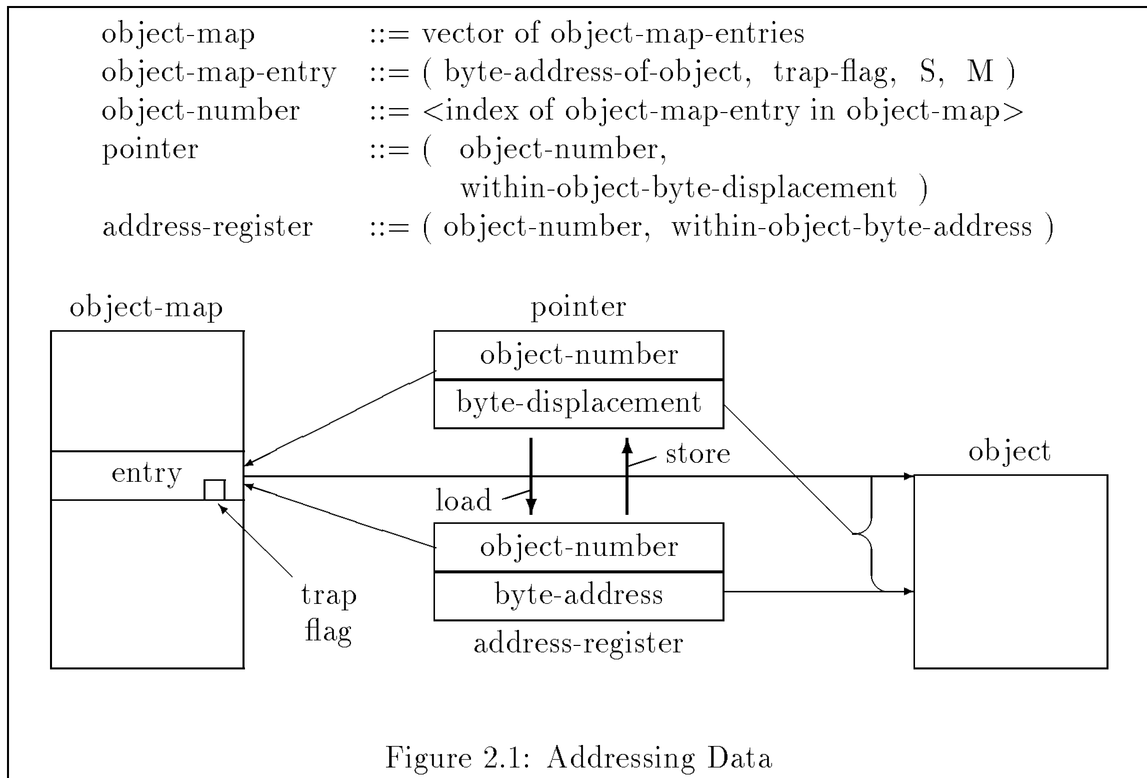


Figure 2.1: Addressing Data

of the object do not overlap. If processes that write an object write both copies of the object when the object is being copied, it is possible to copy the object incrementally, only stopping writing processes while copying each increment (to avoid write/copy conflicts), and running writing processes for a while between copy increments.

The trap flag can also be used to implement the strongly typed type-change operation. The object whose type is being changed is given a new object number with a new object map entry, and the old object number is invalidated by setting the trap flag in its object map entry.

The trap flag can be used to implement demand swizzling by allocating object map entries with their trap flags set to serve as slots where objects are "to be copied" when they are first accessed. When an object that has only such a map entry is first accessed, the trap allocates memory for the object, copies the object from external

memory, and swizzles the pointers in the object so that they point either at other objects already copied into memory, or at object map entries serving as slots.

Such a system, in which an object map entry serving as a slot has no associated memory for the object itself, approximates demand pointer swizzling, because only a small amount of memory, the object map entry, is allocated for unaccessed objects.[5] An alternative system associates a block of memory sufficient to contain the object with the object map entry of a slot, and initiates copying the object into that memory. Then when the object is first accessed, the trap waits until object copying is finished, and then swizzles all pointers in the object. This alternative system implements demand object swizzling.

2.4.1 Related Work and Alternatives to Object Maps

Object maps seem to be the only way to quickly obsolete the old address of an object that has been deleted and to quickly forward addresses of objects that have been moved. Both the trap flag and address indirection through the object map entry seem necessary for these functions.

Hardware has been built to implement object maps, but no such hardware survives today among common computers.

One of the earliest references to such hardware was the following piece of folklore that the author ran into in the mid-1960's:

> A large military system was built on a computer whose core memory was so small that data and program had to be copied constantly between it and magnetic tape. The computer had a very large number of index "registers," which were in core memory, and could be used to let data and instruction blocks move around in memory freely while the program ran. But using them doubled the memory access time, so the system designers decided not to. They built the system and it ran. They then assigned a small group to rewrite the system using the index registers. As expected, the new system was much smaller and simpler than the original. Unexpectedly, the new system ran faster.

The Intel i432[Org83] was another system that built object maps into hardware.

---

[5] A similar "hybrid" system is proposed in [VD93].

Brooks proposed in [Bro84] to implement object maps without trap flags in software by performing the required indirect addressing on (almost) every memory access. I will describe this work in more detail below.

If quick obsolescence of old addresses is not required, two software systems that do not continually indirect through object maps for reading can be used.

In the Pegasus system of North and Reppy[NR87], two or more copies can exist simultaneously, and any copy can be read without further checking. Write operations are made atomic and update all copies. Writes must be atomic to avoid race situations with copying. Obsolete addresses are not completely flushed until the end of the garbage collection cycle.

Atomic write operations take a lot of time because of synchronization overhead, and are not very practical for applications such as updating each element of a large array. So an alternative is proposed by Nettles, O'Toole, Pierce, and Haines[NOPH92], in which a complete set of copies of all used objects is produced while the application processes continue to use the originals of the objects. The copies point only at copies, while the originals point only at originals. A write log is produced by the application processes that is used to update the copies. The system switches over to the copies at the end of a garbage collection. This system cannot handle multiple processors writing to the same object component because each processor would have an independent unsynchronized write log.

These software solutions that do not continually indirect through object maps do not support manual deletion, type change, or swizzling; do not support use of object copying to increase the size of the object; do not permit an object and its copy to overlap; and do not support immediate reuse of memory vacated by a copied object.

2.4.2 Address Registers

In order to get to the point where special hardware for object maps is built, we first need to popularize software that needs this hardware. To do this we need a software approach that does the same thing with usable efficiency. The rest of the discussion of two level addressing will center on the solution to this problem that I am proposing for R-CODE. A different solution of similar quality is described near the end of this discussion.

The key idea is to introduce the *address register* as a software implementation of an

object map cache entry. An address register is very analogous to a machine register, but has special properties. To access an object, one must load a pointer pointing somewhere within the object into one of the processor's address registers. Address registers can be saved and restored like any other register.

Pointers not stored in address registers are stored in the form of an object number plus a byte displacement. Such pointers do not have to be changed when an object is moved.

Pointers stored in address registers are stored in the form of an object number plus a byte address of a byte within the object (see Figure 2.1, page 32). When an object is moved, each address register pointing into the object must be updated by adding the offset of the move into the byte address in the register.

When an address register is saved, it is stored as a pointer in the form that contains the displacement. Loading an address register converts a byte displacement to a byte address, and saving the register converts a byte address to a byte displacement. Thus the only byte addresses pointing into an object are in the object's object map entry and in address registers.

An object map entry also contains a trap flag that is checked whenever an address register is loaded, and causes the processor to trap if it is on. The fact that the trap flag is checked only when the address register is loaded means that on a multi-processor system, where one processor may set a trap flag while another is using the object, the second processor will not "see" the trap flag until it has been interrupted and saved its address registers.

An address register that is no longer in use needs to be cleared by the application code so it will not cause inadvertent traps if the trap flag of the object it points at is turned on. Interrupts can save and restore an address register at any time, so if the trap flag is on, a process can trap at any time during an interrupt restore operation.

In most implementations, object numbers can be the direct addresses of object map entries. The byte address part of an address register is typically stored in an actual machine register. The object number part of the address register is stored in global memory, where it can be seen by other processors in a multi-processor system (see below).

Above I said the byte address in an address register has to point "within" an object. However, this is not literally true. The byte address can point beyond either end of the object, as long as any address actually used to access part of the object points

within the object.

2.4.3 Multi-Process Single-Processor Systems

In a multi-process single-processor system, the single processor should have only one set of address registers, which are shared among processes. Each process that is not running has its address registers saved in the form of pointers. When the process is resumed, it will reload the processor address registers with these saved pointers, and trap if the trap flag of any object pointed at is set.

An example use of this system is moving an object on a single processor, multi-process system. The trap flag for the object is set by the process that is going to move the object at a time when no address register is pointing at the object. At this time, all other processes have saved their address register contents as pointers. If the moving process is interrupted while it is moving the object, the interrupting process runs until it loads an address register with a pointer to the object. Then the interrupting process is trapped, saves its address registers as pointers, and stops, waiting for the object move to finish. When the moving process finishes the move, it clears the trap flag and restarts any processes waiting for the move to finish. These processes then reload their address registers with the new location of the object.

From the point of view of a process addressing an object but not moving it, the direct byte-address part of any address register pointing at the object is spontaneously adjusted whenever another process moves the object.

This method can also be applied to a single user process under a standard operating system to support user process interrupts. The standard operating system does not have to be modified, as the user process can be treated as a virtual computer with its own interrupt system.

2.4.4 Address Register Loads and Stores

There are some problems loading and storing address registers in a system where interrupts can occur between any two instructions.

Loading address registers is not a problem if one makes the rule that the byte address in an address register can be meaningless garbage. Then the object number part of the register can be loaded first, before any meaningful byte address is loaded. To

make the rule work, adding address offsets to garbage byte addresses must not cause failures. Generally computers have unsigned integer add instructions that ignore overflows and can be used to add address offsets without trapping, even if they are adding the offsets to garbage values.

Making an appropriately atomic address register store operation can be trickier. This operation subtracts the byte address of an object in its object map entry from the byte-address in an address register to produce a displacement. The displacement cannot be stored in the same location as either argument, however. If an interrupt occurred and the object was moved during the interrupt, any location holding a byte-address should be adjusted, but any location holding a displacement should not, so the two kinds of location must be distinct. What is needed is an atomic three address subtract instruction, even on a two address computer.

One can make an atomic three address subtract instruction on a two address computer without any extra normal execution overhead by the following trick. The rule is enforced that no interrupt can occur before a register-to-register subtract instruction (unless the previous instruction is also a register-to-register subtract). If this rule can be enforced, the atomic subtraction needed to store an address register X can be done as follows:

$$
\begin{aligned}
&\mathrm{D} = \text{map-entry-address(X.object-number)}\\
&\mathrm{D} = \mathrm{D} - \text{X.byte-address}\\
&\mathrm{D} = -\,\mathrm{D}
\end{aligned}
$$

where D and X.byte-address are in registers and no interrupt is allowed just before their subtraction.

The rule can be enforced by programming the interrupt routine to check if the next instruction after the interrupt is a register-to-register subtract instruction. If it is, the interrupt routine emulates that instruction before completely saving the interrupted process state. The only overhead is per interrupt, and it is small since both the time to check the next instruction and the time to emulate a register-to-register subtract instruction are small.

If we are applying this method to support user process interrupts in a single user process under a standard operating system, only the user process interrupt trap routine needs to be modified. Again the standard operating system does not need to be modified.

2.4.5 Shared Memory Multi-Processors

In a symmetric multi-processor shared memory system the simple approach to getting all processors to recognize a newly set trap flag is to interrupt them all and get each to check its address registers. However, this is inefficient for the processors not setting the trap flag, and it is desirable to move some of the work off to the trap flag setting processor.

This is done by putting the object number part of each address register in global memory, so the trap flag setting processor can find out which other processors are referencing the object in their address registers. Only these processors need be interrupted. Note that the address register load operation must write the object number being loaded into global memory before reading the trap flag for that object. This requires a memory barrier operation between the object number write and the trap flag read on newer faster processors[Sit92].

This approach does not scale well to large numbers of processors. For such systems special hardware is indicated for this and other reasons (e.g. cache coherency).

This method can also be applied to multiple user processes under a standard operating system, provided any process that sets an object trap flag can get any other process that shares the same memory to interrupt "sufficiently promptly." The standard operating system does not have to be modified, as the user processes are treated as virtual processors. Note that strict priority scheduling of user processes would normally not be allowed, as a trap flag setting process might not be able to get lower priority processes to interrupt.

2.4.6 Copying Stops

Stopping processes that try to access an object being copied disrupts real-time response. However, the effect is no worse than I/O interrupts if the objects being copied are small enough to be copied in approximately the time taken by an I/O interrupt routine. If the object moving process is part of the garbage collection system, it can detect when processes have been stopped, and slow down copying to avoid too many such stoppages in a short period of time.

It is important to copy each object at a priority level appropriate for the processes accessing the object, since these processes will have to wait for the copying process to finish.

Large objects can be copied if they are only accessed by low priority processes (stopping is OK). Or large objects can be fixed in memory and not moved. Another option is to be sure the source and destination of the copy do not overlap, and only stop writing processes. And yet another idea is to preserve the page alignment of a large object when it is copied, and copy most of the object by merely copying page table entries, without actually copying the contents of the pages.

Thus the worst case is when high priority processes must write a large object that is itself one of a set of objects dynamically created and destroyed on a slower time scale, and the large object cannot be copied by copying page table entries. If this case cannot be avoided, the following scheme can be used to copy the large object incrementally. Readers are not stopped during the copy, but are interrupted at the end of the copy to get the correct new object location. Writers are stopped during each copy increment, but are allowed to run between increments to maintain their real-time response. Writers write both copies of the object while it is being copied, so at the end of the copy both copies of the object are identical. Writers must be stopped during copy increments to avoid write/copy conflict.

2.4.7 Address Register Overhead

The overhead of the address register implementation of two level addressing consists mostly in the time taken to load and store address registers. With ordinary address registers, load and store would each take one RISC instruction. With two level addressing, load takes 9-18 RISC instructions and store takes 5-10 instructions. The lower bounds on these estimates are the instruction counts from Figure 2.2, and the upper bounds are simply double the lower bounds.

On modern computers it may be more important to count secondary cache misses than it is to count instructions, in order to determine timing. Use of address registers tends to add one secondary cache miss, for the object map entry, for every address register load. Presumably address register stores would not add further misses most of the time.

For two address instruction computers, there is an additional per interrupt overhead to check whether the next instruction is a register to register subtract, and emulate it if it is.

While these overheads are significant, for most programs they should average much

In the following `P` is a pointer in memory and `A` is an address register. The pointer is being loaded into the address register, or the address register is being stored in the pointer. `A.object-number` is the object number part the address register stored in global memory, and `A.byte-address` is the byte address part stored in a hardware register. Object numbers are addresses of object map entries. Each pseudo-instruction below is intended to map to one RISC instruction on a typical RISC computer.

Address Load:

```
temp-object-number-reg = P.object-number
A.object-number = temp-object-number-reg
memory-barrier-instruction
temp-trap-reg = map-flag-word(temp-object-number-reg)
temp-trap-reg =
        temp-trap-reg AND trap-flag-mask-constant
if (temp-trap-reg non-zero)
    then call trap-subroutine
    // subroutine argument is in temp-object-number-reg
    // subroutine may return to this point to resume
    //     process
A.byte-address = map-address(temp-object-number-reg)
temp-displacement-reg = P.byte-displacement
A.byte-address =
        A.byte-address + temp-displacement-register
```

Address Store:

```
temp-object-number-reg = A.object-number
temp-byte-address-reg =
        map-address(temp-object-number-reg)
    // interrupts adjust temp-byte-address-reg when
    //   temp-object-number-reg object moves
temp-displacement-reg =
        A.byte-address - temp-byte-address-reg
P.object-number = temp-object-number-reg
P.byte-displacement = temp-displacement-reg
```

Figure 2.2: Address Load and Store RISC Pseudo-Code

less than the factor of 3 overhead of emulating a VAX on an ALPHA mentioned on page 12. If a 150 MIPS ALPHA can emulate a 50 MIPS VAX, it should be able to emulate an R-CODE virtual computer running much faster than 50 MIPS, at least as far as the memory manager is concerned.

Note that, very importantly, the overhead is zero for stepping addresses through an array after the address register is loaded. Byte addresses in address registers can be incremented or decremented directly.

The address register load overhead may appear to be a high price to pay for languages like LISP that have small cons cells. However, because memory speeds are falling behind processor speeds[WK95], modern computers are suffering very large overheads for secondary cache misses, so small cons cells are becoming inefficient anyway. Thus address register load overhead may be less significant on future computers.

2.4.8 The Interrupt Check Alternative

There is a classical way of handling object map like structures in LISP implementations. Interrupts are not permitted except when a special *interrupt check* operation occurs in the code. Between such operations, where interrupts cannot occur, direct addresses to objects may be put into any register, as long as trap flags are checked when addresses are loaded from object maps. When an interrupt check operation is executed, all address information outside the object maps becomes invalid if and only if an interrupt occurs, and in that case any subsequent use of addresses must reread them from object map entries and recheck the entry trap flags.

Interrupt check instructions can be placed at the beginning of subroutines and loops. This scheme can be used when LISP is compiled into a language like C. The interrupt check operation checks a global interrupt flag, and if on, calls an interrupt processing routine, which according to the rules of C, may change any object map entry, as these are globally accessible.

The interrupt check scheme has operations very similar to address register loading, done at similar times. However, instead of storing address registers, the interrupt check scheme does interrupt check operations.

In the interrupt check scheme, addresses outside object maps become invalid when a subroutine is called, whereas in the address register scheme, it is possible to have address registers which remain valid across subroutine calls.

In a multi-processor system, setting an object's trap flag does not take effect until all processes that are accessing the object interrupt, just as for the address register scheme. Therefore, just as for the address register scheme, the object numbers of objects being accessed need to be put into global variables where they can be seen by the process setting the trap flag. Then that process can interrupt only other processes that are accessing the object whose trap flag it is setting.

Thus on a multi-processor system, the interrupt check scheme needs to implement the object number part of address registers. These can be treated like their byte address counterparts, becoming invalid when an interrupt occurs or a subroutine is called.

In a serious real-time system, where processes sharing a common garbage collected memory can indefinitely preempt each other with priority scheduling, the operating system scheduler must be modified for both the address register and interrupt check schemes. For the address register scheme the scheduler must save address registers in the appropriate manner. For the interrupt check scheme, the scheduler must wait for a process to reach its next interrupt check operation, before interrupting the process to switch to another process sharing the same memory. This requires setting a timer in case the user process gets caught in an infinite loop.

In a system that is merely interactive, and not real-time, processes can be treated like independent processors if none can preempt any other sharing the same memory, and the operating system need not be modified.

It is not clear whether the address register scheme is better than the interrupt check scheme, but both implement object maps with manual object deletion and dynamic moving. For R-CODE I have proposed the address register scheme because it seems a better match to future hardware and does not require as intrusive tinkering with the way interrupts are scheduled.

2.4.9 Other Work Related to Address Registers

Brooks[Bro84] introduces a scheme that has an object map equivalent and performs an indirect address operation through the object map for almost every memory access. To implement this on the Motorola 68000, a single hardware register, named `ipr`, is given the special function of holding the address of the beginning of the object currently being accessed. This register holds the only direct address outside the object map.

What serves as the object map in this scheme is a word in front of each object copy that holds the address of the beginning of the current version of the object. Given the address in `ipr` of some copy of the object, the address of the current copy can be loaded into `ipr` by loading the word just before the location pointed at by `ipr`. This is done when returning from an interrupt, so a process can be interrupted by a second process that moves an object.

The 68000 computer has addressing modes in which two registers are added to form an address, so this computer has no need for any register to address any point in an object other than the beginning of the object.

There is no trap flag facility with this scheme, no ability to do manual deletion, and no ability to interrupt a process copying an object with another process that tries to access the object and is then stopped. Nevertheless, there is a version of an object map, and `ipr` is a primitive version of an address register.

2.5 Concurrent Garbage Detection Algorithms

In addition to manual deletion and dynamic object moving, our memory manager must provide for automatic garbage collection of objects, and also collection of the object map entries for manually deleted objects as required by $\mathbf{F}_1$ (page 24). This leads us to a study of various concurrent object marking algorithms.

In this section I will exhaustively investigate the options for concurrent marking algorithms, and pick the variants that seem to be most promising. There are a number of options, resulting in a lengthy analysis.

2.5.1 The Standard Marking Algorithm

Garbage detection algorithms all attempt to mark reachable objects (see sections 2.3.1 and 2.3.2). The standard algorithm is the following:

1. Initialize. Turn off the mark and scavenged bits of all objects.

2. Root set. Turn on the marked bits of the root set.

3. Scavenge. For each object whose marked bit is on and whose scavenged bit is off, scavenge the object and set its scavenged bit.

This algorithm maintains an important invariant: no scavenged object points at an unmarked object. This invariant, which need not be maintained precisely at every instant, must hold at the instant marking finishes.

A problem with this algorithm is how to maintain a list of all objects that have been marked but not scavenged.

This algorithm assumes that after it is done, all objects will be swept, so unmarked objects will be returned to the free list. Compactification may be done as part of sweeping. Whenever compactification is done, forwarding must be done. Discussion of forwarding is deferred till later.

2.5.2 Ephemeral Marking

Studies have shown ([Hay94, Chapter 3]) that 85% of all objects allocated have fairly short lifetimes. Also, some systems have initial program loads that contain tens of megabytes of objects that need never be garbage collected.

Garbage collection can be adapted to these facts by dividing objects into two kinds: *permanent objects* and *ephemeral objects*. Then an *ephemeral garbage collection* is just a standard garbage collection in which the permanent objects are treated as part of the root set.

By a *full garbage collection* I will mean one that can collect permanent and ephemeral objects, i.e. one that ignores the distinction between the two kinds of objects, or more specifically, one that has as small a root set as possible.

In order to do ephemeral garbage collection efficiently, a list of all permanent objects that might point at ephemeral objects needs to be kept at all times, so the many permanent objects that do not point at ephemeral objects can be ignored. The objects on this list are called *ephemeral root* objects. Permanent objects that are not ephemeral root objects are called *not-ephemeral-root* objects. Thus the efficiency of ephemeral garbage collection depends upon the propensity for permanent objects to be not-ephemeral-root objects, or in other words, for the list of ephemeral root objects to be short.

One can add two bits to each object: the *permanent bit* to indicate that the object is permanent; and the *not-ephemeral-root bit* to indicate that the object is a permanent object that does not point at ephemeral objects. An important invariant must be maintained: no not-ephemeral-root object points at a non-permanent (ephemeral)

object. This invariant is strictly analogous to the invariant which says no scavenged object may point at an unmarked object: the permanent and not-ephemeral-root bits are respectively analogous to the marked and scavenged bits; and the list of ephemeral root objects is analogous to the list of unscavenged marked objects.

Thus the analogy:

$$\text{scavenged} \equiv \text{not-ephemeral-root}$$
$$\text{marked} \equiv \text{permanent}$$

is quite precise.

This analogy is not an accident. At the start of an ephemeral garbage collection, it is possible to set the marked bit of each object equal to the object's permanent bit, and set the scavenged bit equal to the not-ephemeral-root bit. However this is not what is generally done. Instead, the marked bits of all permanent objects are permanently set, and never cleared, to cause pointers to permanent objects to be ignored for marking. Similarly, the scavenged bits of all not-ephemeral-root objects are either permanently set or permanently cleared, depending on the type of garbage collector, non-snapshot or snapshot (see below), to prevent any action when pointers are stored into not-ephemeral-root objects, except, of course, for the possible action of turning off the not-ephemeral-root bit of the object and putting it on the ephemeral root object list.

During an ephemeral marking garbage detection every ephemeral root object is scavenged. This scavenging operation can be programmed to discover whether the ephemeral root object, which might point at an ephemeral object, does in fact point at any ephemeral object. If not, the ephemeral root object can be declared to be a not-ephemeral-root object, and taken off the list of ephemeral root objects.

Sometimes, instead of keeping a list of ephemeral root objects, a list of the exact permanent object pointer components that might point at ephemeral objects is kept. These are called *ephemeral root pointers*. However, I do not propose this for R-CODE because it does not fit into the efficient testing scheme given below in section 2.5.9.1.

Often several ephemeral garbage collections are run "simultaneously", using a sequence of successively smaller permanent object sets, each consisting of successively older objects. This scheme is referred to as *generational garbage collection*.

2.5.3 Concurrent Marking Conditions

It is desirable to run garbage detection concurrently with mutators. It is also desirable to run several ephemeral garbage detections concurrently with each other, as in a generational scheme where one garbage detection detects garbage among very new objects, with all older objects in its permanent object set, and another detects garbage among relatively old objects, with only objects in the initial program load in its permanent object set.

In order to run the mutator concurrently with marking it must maintain the invariants:

$\mathbf{I}_s$: No scavenged object points at an unmarked object.

$\mathbf{I}_e$: No not-ephemeral-root object points at an ephemeral object.

Both these invariants have the same structural form. Suppose we associate two bits, M for "marked" and S for "scavenged", with each object. Then invariant $\mathbf{I}_s$ says

"If object X points at object Y, then $X.S \wedge (\neg Y.M) = 0$."

Invariant $\mathbf{I}_e$ says exactly the same thing if we reinterpret M to mean "permanent" and S to mean "not-ephemeral-root".

A non-ephemeral marking process strives to meet three conditions at some single point in time:

1. All root objects are marked.
2. All marked objects are scavenged.
3. Invariant $\mathbf{I}_s$ above holds.

An ephemeral marking process strives to meet the following alternate set of conditions at some single point in time, while ignoring pointers to permanent objects, ignoring not-ephemeral-root objects, and maintaining the list of ephemeral root objects.

1. All root objects are marked.

2. All ephemeral-root objects are marked.
3. All marked objects are scavenged.
4. Invariants $\mathbf{I}_s$ and $\mathbf{I}_e$ above hold.

2.5.4 Snapshot and Non-Snapshot Detectors

The marking conditions described above need not hold at all times. Rather, they must hold at some particular specified time. There are two common choices: the end of the garbage detection, and the beginning of the garbage detection. Detectors using the later choice, the beginning of garbage detection, are called *snapshot* detectors.

In this section, I will talk about "*storing*" a pointer into an object X. If X is a top frame, this "store" is actually done by a "read operation" (as defined in section 2.3.3, page 26). If X is not a top frame, this "store" is done by a "write operation".

2.5.4.1 Non-Snapshot Detectors

In a *non-snapshot* detector the conditions given above must hold at the end of garbage detection.

In order to make invariant $\mathbf{I}_s$ hold at the end of the garbage detection, the mutator, when it is about to store a pointer to Y into the object X and discovers that $X.S \wedge (\neg Y.M) \neq 0$, can merely arrange that either $Y.M$ be turned on, or $X.S$ be turned off, sometime before the end of the garbage detection. Thus although the invariant may be violated temporarily, it will be re-established before the end of garbage detection.

Note that for a non-snapshot detector, the bit $X.S$ should be turned on as soon as any pointer components in X are scavenged. Thus in general it is turned on at the beginning of scavenging X. Efficiency aside, no harm is done by turning $X.S$ on prematurely.

In order to make invariant $\mathbf{I}_e$ hold at the end of the garbage detection, the same procedure may be followed, but with $X.S$ always turned off sometime before the end of garbage detection. This indicates that X is no longer a "not-ephemeral-root" object, but may now point at an ephemeral object.

2.5.4.2 Snapshot Detectors

In a *snapshot* detector the conditions given above must hold at the beginning of garbage detection. Objects allocated since the beginning of garbage collection cannot be collected, and are neither marked nor scavenged.

I define an object to be "marked at the beginning of garbage collection" if and only if it is marked at the end of garbage collection, and similarly define an object to be "scavenged at the beginning of garbage collection" if and only if it is scavenged at the end.

In order to make invariant $\mathbf{I}_s$ hold at the beginning of the garbage detection, the mutator, when it is about to store over a pointer to Y that was previously stored in the object X, and both objects were allocated before the beginning of garbage detection, and $(\neg X.S) \wedge (\neg Y.M) \neq 0$, can arrange that $Y.M$ be turned on sometime before the end of the garbage detection. Note that the invariant itself is not checked, but clearly X was reachable at the beginning of garbage detection, or else we would not be storing into it.

Note that for a snapshot detector, the bit $X.S$ should not be turned on until after all pointer components in X have been scavenged. Thus in general it is turned on at the end of scavenging X. Efficiency aside, no harm is done by turning $X.S$ on late.

Then it can be proved that all the required conditions hold at the beginning of garbage detection. Briefly, if object X points at object Y at the beginning of garbage detection, and X is scavenged at the "beginning" (i.e. by the end) of detection, then we must show that Y is marked at the "beginning" (i.e. by the end) of detection. There are two cases. First, the pointer component in X pointing at Y is not changed until after $X.S$ is set. Then when X is scavenged, Y will be marked. The second case is when the pointer component is changed before $X.S$ is set. But then Y will be marked as a consequence of the pointer to Y being stored over while $X.S$ is off.

In order to make the ephemeral invariant $\mathbf{I}_e$ hold, we can assume that it holds at the beginning of the garbage detection, and arrange, as in the non-snapshot case, for it to hold at the beginning of the next detection. More specifically, when the mutator is about to store a pointer to Y into the object X and discovers that $X.S \wedge (\neg Y.M) \neq 0$, it arranges that $X.S$ be turned off sometime before the beginning of the next garbage detection. Here $X.S$ is on if X is a permanent object that is not an ephemeral root, and $Y.M$ is on if Y is a permanent object.

In theory objects allocated after garbage collection starts are ignored by a snapshot detector. In practice they can have both their M and S bits set and will be effectively ignored.

It is possible to replace any single test $(\neg X.S) \wedge (\neg Y.M) \neq 0$ by the simpler test $(\neg Y.M) \neq 0$, i.e. $Y.M$ is off, whenever a pointer to Y in an object X is stored over, without marking any more objects than one would otherwise mark, assuming objects allocated after detection began are marked. Suppose $X.S$ is on when the pointer from X to Y is stored over. Then if the pointer to be stored over existed when X was scavenged, $Y.M$ will be on. But if not, then the pointer was stored after garbage detection started. Therefore Y was reachable after detection started, and hence either Y was allocated after detection began, or Y was reachable when detection began.

2.5.5 Read and Write Barriers

In the discussion of ephemeral, non-snapshot, and snapshot detectors, three different tests were mentioned:

$\mathbf{T}_{ns}$: Non-snapshot scavenge: $X.S \wedge (\neg Y.M) \neq 0$

$\mathbf{T}_{ss}$: Snapshot scavenge: $(\neg X.S) \wedge (\neg Y.M) \neq 0$

$\mathbf{T}_{e}$: Ephemeral root: $X.S \wedge (\neg Y.M) \neq 0$

When a pointer is read into a top frame, these tests must be applied where X is a top frame. A test applied during a read operation called a *read barrier*.

When a pointer is written into an object that is not a top frame, these tests must be applied where X is not a top frame. A test applied during a write operation is called a *write barrier*.

Non-snapshot detectors come in two flavors. Those that enforce $\mathbf{I}_s$ with the aid of $\mathbf{T}_{ns}$ read barrier tests, but use no write barrier tests, are called *read barrier detectors*. Those that enforce $\mathbf{I}_s$ with the aid of $\mathbf{T}_{ns}$ write barrier tests, but use no read barrier tests, are called *write barrier detectors*.

Snapshot detectors, on the other hand, must always use a $\mathbf{T}_{ss}$ write barrier test. It is possible to program them to also require a $\mathbf{T}_{ss}$ read barrier test, but this should never be done as it is obviously inefficient.

Invariant $\mathbf{I}_e$ must always be maintained with the help of a $\mathbf{T}_e$ write barrier test.

2.5.5.1 Read Barrier Detectors

A *read barrier detector* is a non-snapshot detector that scavenges all top frames before it scavenges any other object. More specifically, the rule is enforced that after the S bit is set for any object that is not a top frame, then no top frame will contain a pointer to an unmarked object. Therefore, write barriers will not be necessary to maintain invariant $\mathbf{I}_s$, as any pointer written will point at a marked object.

Write barriers are still necessary to maintain $\mathbf{I}_e$.

A read barrier detector sets all frame S flags at the very beginning of detection, so the read barrier test $\mathbf{T}_{ns}$ reduces to $(\neg Y.M) \neq 0$, i.e., $Y.M$ is off. The frames do not have to be immediately scavenged; the mutators may run while they are being scavenged. But they must be completely scavenged before the $\mathbf{I}_s$ invariant S flag of any non-frame object is set.

Note that the read barrier need not actually mark an unmarked object. It need merely schedule the object to be marked at some time before the end of garbage detection.

2.5.5.2 Write Barriers Detectors

A *write barrier detector* is a non-snapshot detector that does not scavenge frames until the end of detection, and uses no read barrier test whatsoever. Therefore the top frame may contain pointers to unmarked objects, and a $\mathbf{T}_{ns}$ write barrier test is needed to support invariant $\mathbf{I}_s$. A similar $\mathbf{T}_e$ write barrier test is needed to support invariant $\mathbf{I}_e$.

2.5.5.2.1 The Write Barrier End Game. Finishing a write barrier detection is not trivial. The following is a reasonable ending algorithm:

1. Wait until a time when all non-frame root objects have been marked, and all marked non-frame objects have been scavenged.
2. Scavenge stacks from the bottom up. To scavenge a frame, set its S bit and scavenge it. But if a frame being scavenged becomes the top frame, clear its S bit and stop scavenging the frame's stack.

   If there are any unscavenged marked non-frame objects at the end of this step, go back to step 1.

3. Stop all mutators and scavenge all unscavenged frames.

   If any objects are newly marked by this scavenging, restart mutators and go back to step 1.

4. Terminate the detection and restart the mutators.

The time needed when mutators are stopped in step 3 can be a problem. Use can be made of code optimized to check each frame to see if it points at any unmarked objects.

2.5.5.2.2 Write Barrier End Thrashing. If the write barrier detection ending algorithm above must repeatedly go back to step 1 from steps 2 or 3, this is known as *write barrier end thrashing*.

There are theoretical examples where such thrashing can be severe. Suppose a mutator has built a very long LISP list reachable only from its top frame, and then applies the LISP NREVERSE function to destructively reverse the elements of the list. Suppose the garbage detector reaches step 3 of the ending algorithm just after NREVERSE starts.

During the NREVERSE function, the top frame maintains pointers to the head of the unreversed part of the list and the head of the already reversed part. An element is moved from the head of the unreversed part, and the element is changed to become the head of the reversed part. Once changed, it is no longer possible to reach the yet unreversed part of the list through the moved element.

So step 3 will mark the heads of the reversed and unreversed parts of the list. Then the mutator will be restarted, and move the head of the unreversed part to the reversed part, and change it. The garbage detector will then not be able to reach the rest of the unreversed part of the list through already marked objects. The detector will run until it reexecutes step 3, without marking the unreverse remainder of the list. Then at step 3 the whole process will repeat, and may continue doing so as long as the NREVERSE function continues.

Note that this sort of behavior cannot happen unless mutators destroy the connectivity of the part of the object graph reachable only from the stack. The NREVERSE function is particularly perverse in this respect.

The garbage detector can detect such thrashing by counting the number of times it reaches steps 2 and 3. Furthermore, by keeping a count of how many times unmarked

objects were discovered for each mutator in steps 2 and 3, the detector can identify which mutators are causing thrashing.

The only danger is that there will be so little extra work for the detector discovered by step 2 or step 3 that the cost of the end algorithm will not be properly amortized over the extra detection work discovered. If this begins to happen, the detector may take one of the following special steps:

1. Switch into a mode where the allocator always sets new objects as being marked and scavenged. This prevents thrashing caused by simply allocating new objects.

2. Switch into a mode where the mutators causing trashing are stopped while garbage detection attempts to finish. These mutators may be restarted after some significant amount of work has been done by the garbage detector, even if detection has not finished.

3. Switch into a mode where some mutators have a read barrier that marks any object when a pointer to it is copied into the mutator's top frame.

Whether any of these steps are necessary, and which are best to use, is a very probabilistic and experimental subject. Clearly mode 3 will work well, but it requires considerable extra code, though this extra code is not executed most of the time.

Write barrier end thrashing is significant because it is the only component of write barrier detector overhead that cannot be rationally bounded. One would hope that the overhead of a detector could be bounded in terms of reasonable parameters such as number of used objects, rate of object allocation, rate of byte allocation, number of pointer writes, and so forth. And in fact every overhead of a write barrier detector can be so bounded, except write barrier end thrashing.

Write barrier end thrashing is likely to be similar to other kinds of thrashing that can happen in a computer system. For example, cache thrashing, and virtual memory page thrashing. It will be unlikely in practice, but unpredictable in theory. But it is possible to detect write barrier thrashing, and take some easy countermeasures, such as modes 1 and 2 above.

2.5.5.3 Snapshot Detector Barriers

A *snapshot detector* begins by scavenging all top frames, and thereafter, whenever a frame becomes a top frame, stops its mutator and scavenges the frame before permitting the mutator to continue. Then no $\mathbf{T}_{ss}$ read barrier test is needed.

In a snapshot detector, an object's $\mathbf{I}_s$ invariant S bit cannot be set until after the object is scavenged, and it is the setting of this bit for frames that makes the read barrier test unnecessary. So each top frame must really be scavenged with its mutator stopped before the mutator can be permitted to continue.

However, scavenging a top frame for a snapshot detector can be done by making a copy of the frame with the mutator stopped, and then scavenging the copy while the mutator resumes.

A snapshot detector always requires both $\mathbf{T}_{ss}$ and $\mathbf{T}_e$ write barrier tests. The first can be simplified to test $(\neg Y.M) \neq 0$, i.e. $Y.M$ is off.

2.5.6 Comparison of Read-Barrier, Write-Barrier, and Snapshot Detectors

The following are some comparisons between these three kinds of detector:

1. Read-barriers make pointer reads more expensive, while write barriers make pointer writes more expensive. There are typically many more pointer reads than writes, so the total overhead for read barriers can be much greater.

2. If a garbage detection takes a long time, a write barrier detector may detect significantly more garbage than a snapshot detector or a read barrier detector. This is because objects created during garbage detection, but which are never reachable except from a mutator stack, and which become unreachable before the end of garbage detection, are usually classified as garbage by a write barrier detector.

   Both a read-barrier and a snapshot detector, however, effectively mark any newly create object.

3. The tests $\mathbf{T}_{ns}$ and $\mathbf{T}_e$ needed to maintain invariants $\mathbf{I}_s$ and $\mathbf{I}_e$ when a pointer is written into an object are so similar in a write barrier detector that they can be bundled together and done in parallel by executing a single bit-string AND

operation. So both kinds of tests can be done in the same amount of time it would take to do one kind of test.

But in a snapshot detector the previous value of the pointer component must be tested to maintain invariant $\mathbf{I}_s$, and the new value of the pointer component to maintain invariant $\mathbf{I}_e$, so the totality of both kinds of write barrier tests takes twice as long as for a write barrier detector.

4. If we examine our invariants

   $\mathbf{I}_s$: No scavenged object points at an unmarked object.

   $\mathbf{I}_e$: No not-ephemeral-root object points at an ephemeral object.

   we see that only $\mathbf{I}_s$ can be maintained by a read barrier. $\mathbf{I}_e$ must be maintained by a write barrier, because the invariant is maintained by clearing the S flag of the object into which the pointer is being stored (making it an ephemeral-root), rather than setting the M flag of the object pointed at (making it permanent).

   Thus a read-barrier ephemeral detector must also use a separate write barrier too.

5. Write barrier detectors have some trouble getting the stacks scavenged at the end of detection without using operations that disrupt real-time performance.

   Snapshot detectors have some trouble getting stacks scavenged at the beginning of detection without using operations that disrupt real-time performance.

   Read barrier detectors have no such problems since they merely need to start scavenging stacks at the beginning of detection, and can take as long as they like while mutators are running to actually scavenge the stacks.

All but the last comparison is a reason to favor write barrier detectors. Therefore, R-CODE uses a write barrier detector. This means that R-CODE must live with the possibility of write barrier end thrashing (section 2.5.5.2.2), and control such thrashing with measurement tools and other means. I do not anticipate this thrashing to be much of a problem in the real world.

2.5.7 Copying Collectors

A *copying collector* organizes some of memory into two disjoint *spaces*, the *from-space* and the *to-space*, and assuming all objects it wants to detect as garbage are initially in the from-space, attempts to copy all reachable objects in the from-space into the to-space.

Specifically, a copying collector copies a from-space object to to-space either when the object is marked or when it is scavenged or in between. To distinguish these three options, I refer to *copy-on-mark* detectors, *copy-on-scavenge* detectors, and *mark-copy-scavenge* detectors.

Thus for a copy-on-mark detector, being marked is synonymous with being in to-space, and being unmarked is synonymous with being in from-space, for the set of objects under consideration.

Copying collectors have been used with read barrier detectors, and proposed for write barrier detectors.

Some advantages and disadvantages of copying collectors are:

1. A copy-on-mark read barrier detector can do forwarding at the same time as the read barrier check. Therefore all pointers in the top frame point at copied objects. This is one of only two ways of doing forwarding that has mutators using only forwarded pointers for objects that have been copied, so there are no problems with two copies of an object becoming inconsistent (see "Forwarding" below).

   Note that if pointers used by mutators are to be forwarded when read using a read barrier, marking cannot be delayed in the way it otherwise could be. If marking were delayed, the mutator might end up using a non-forwarded pointer, which would be a problem if the two copies of the object became inconsistent.

   If a copy-on-mark detector forwards pointers both during the read barrier and when a pointer is scavenged, then at the end of garbage detection all pointers will be forwarded.

   Write barrier and snapshot copy-on-mark detectors, all copy-on-scavenge detectors, and all mark-copy-scavenge detectors do not have the nice property of having mutators only access to-space copies of objects and not the from-space originals.

2. Compaction happens automatically without sweeping.

3. For copy-on-mark detectors, the list of marked, unscavenged objects is easily maintained by simply scavenging objects in the same order in which they were copied. If copies are allocated in to-space in order of increasing addresses, then objects that are marked can be scavenged in order of increasing addresses, and it is merely necessary to keep the address of the boundary between scavenged and unscavenged objects in to-space to know which objects have been scavenged.

   For mark-copy-scavenge detectors, a separate list of objects to be copied must be maintained, but the list of copied objects to be scavenged is easily maintained as just described.

4. For copy-on-mark detectors, the cost of marking an object becomes high, because the object must be copied.

   In real-time systems it is better to perform copying at the convenience of the system, in order to stretch the overhead out evenly over time. But mutators do marking via read or write barriers on their own schedule, and with copy-on-mark detectors, this means copying is done on the mutator's marking schedule, which may bunch copies at awkward moments. In other words, the mutator overhead of a copy-on-mark detector is not "rationally bounded".

   Both the copy-on-scavenge and mark-copy-scavenge detectors solve this problem.

5. Copying collectors require a separate to-space at least as big as the set of reachable objects in from-space, and cannot simply compact a single memory space by sliding objects to one end.

6. The order in which objects appear in memory changes, and this might adversely impact time efficiency, amount of memory required, or predictability, depending on the total situation. There is much debate on the memory order objects should appear in for best efficiency. However, allocation order appears to be one of the better orders.

Because R-CODE uses two level addressing, it does not need to worry about forwarding pointers in other ways. Therefore, the principal advantage of a copy-on-mark detector with read barrier forwarding does not apply to R-CODE.

The principal disadvantage of copy-on-mark collectors, the possibility of uncontrolled mutator delay copying a set of objects referenced by the mutator, conflicts with requirement $\mathbf{F}_2$ (page 24) that R-CODE be suitable for real-time applications. Therefore R-CODE does not use a copy-on-mark collector algorithm. There is no reason for R-CODE to associate copying and scavenging, so R-CODE does not use a copy-on-scavenge or a mark-copy-scavenge algorithm.

2.5.8 Forwarding

When objects are copied by garbage collectors, a principal problem is making sure all mutators access the same copy or consistent copies. There are four ways of doing this: read-barrier-forwarding, write-barrier-forwarding, replication-forwarding, and two level addressing. All four ways of doing this are discussed in this section for completeness, although for reasons other than just forwarding, R-CODE uses the two level addressing scheme described in section 2.4 above.

2.5.8.1 Read Barrier Forwarding

A *read-barrier-forwarding* garbage collector is a special case of a copy-on-mark copying collector with a read barrier detector. A read-barrier-forwarding collector ensures that all object pointers in top frames point at object copies in to-space, and not at the originals in from-space. Therefore, mutators only access to-space copies.

With a read barrier detector, all pointers in top frames have been marked. With a copy-on-mark collector, all objects that have been marked have also been copied to to-space. The read barrier that is part of the garbage detector is augmented to forward all pointers to objects which have already been copied, with the result that all pointers in top frames are to to-space.

Therefore all mutators access only the to-space versions of copied objects, and the collector is of the eager forwarding variety.

The read barrier in this detector may not delay marking, but must mark and copy any unmarked object immediately, so it can forward pointers to the object immediately.

Pointer components are also forwarded when they are scavenged. Thus at the end of garbage detection, all objects have been copied to to-space, and all pointers have been forwarded and point at to-space.

A read-barrier-forwarding collector does not have to begin by copying all objects pointed at by top frames. Instead, it can run mutators and frame scavenging simultaneously until all the frames are scavenged, as long as no non-frames are scavenged until all frames have been scavenged.

One of the standard kinds of garbage collector in use today is the read-barrier-forwarding collector, which is generally known[Wil92] as "*The Copying Collector.*"

2.5.8.2 Write Barrier Forwarding

A *write-barrier-forwarding* garbage collector is a special case of a copy-on-mark copying collector with a write barrier detector. A write-barrier-forwarding collector may have some mutators accessing the to-space version of a copied object while other mutators are still accessing the from-space version.

The scavenger and write barriers are augmented to forward all pointers to objects which have been copied. Therefore, by the end of garbage detection, all pointers will have been forwarded and point at to-space.

Because mutators may access either copy, write operations must write all copies of an object. To do this they must synchronize with each other and with processes copying objects. This synchronization tends to impose a high cost in time on write operations.

Read operations may access any copy, and the collector is of the lazy forwarding variety.

The write barrier in this detector may not delay marking, but must mark and copy any unmarked object immediately, so it can forward pointers to the object immediately.

It is possible to avoid the extra write overhead on objects being initialized for the first time, and therefore it is possible to avoid it for all writes in a purely functional language. Thus it is possible to avoid the overhead for most writes in a mostly functional language.

The Pegasus system of North and Reppy[NR87], described on page 34 is a write-barrier-forwarding garbage collector. The garbage collector proposed by Brooks described on page 42 mixes the write-barrier-forwarding collector and object map ideas.

2.5.8.3 Replication Forwarding

A *replication-forwarding* garbage collector does not require objects to be copied at any given time, nor does it require the detector be of any particular type, such as read barrier or write barrier.

The mutators in a replication-forwarding collector always access the originals of the objects in from-space. At some point, not necessarily during detection, a copy of all marked objects is made in to-space, by a background process. All the pointers in the to-space copies are forwarded, but none of the pointers in the from-space copies are forwarded.

The goal of copying is to copy all marked objects. Copying can start any time after marking starts, and finish any time after marking finishes and before the next marking cycle starts.

New objects allocated after a write barrier detection ends but before copying ends can be allocated in to-space, and never be copied, so no pointers to them will require forwarding. New objects allocated after a read barrier or snapshot detection begins can likewise be allocated in to-space.

The stack frames are not copied till the end of the copying. Until this time they contain only pointers to from-space. Then at the end of the copying, mutators are stopped, the stack frames are copied, the pointers in the stack frames are forwarded, and the mutators resume with the copied stacks.

This algorithm for copying the stacks can be made more sophisticated along the same lines as the detection ending algorithm for write barrier detectors, in order to stop mutators for a shorter period of time. However there is no analog of write barrier end thrashing, as nothing is left to do at this point except copy and forward the stacks.

In order to keep the copies up to date, each process makes a write log of all the writes it makes into objects that might have a copy. These write logs are processed by the copying process.

Because the write logs can consist of log buffers private to the writing process while the log buffer is being filled, writes do not have to synchronize with the copy process. This makes writes much more efficient than they would be if they had to synchronize.

A deficiency of this system is that two mutators may not write the same object component because of time race condition problems updating the copies. This could be addressed by having special write operations that were synchronized with each

other, at the cost of extra synchronization overhead per write. In a single processor system it might be addressed by building a fast atomic operation to write a log common to all processes.

The system can be designed so new objects do not have copies until after they are initialized, and therefore writes done to initialize an object need not be logged.

The collector of proposed by Nettles, O'Toole, Pierce, and Haines[NOPH92] described on page 34 is a replication-forwarding collector.

2.5.8.4 Two Level Addressing

A *two level addressing* memory manager requires that pointers consist of two parts: an object number which identifies an object, and a within object byte displacement, that identifiers a byte location within the object. All accesses to the object use the object number to look up the current address of the object in a table called the object map. The object map also contains a trap flag for each object, that can be used to trap all accesses to the object.

With two level addressing an object can be moved at any time. The trap flag is set to stop processes that try to access the object, the object is moved, the trap flag is cleared, and any processes stopped by the trap flag are restarted.

There is a more detailed description of two level addressing above in section 2.4.

Two level addressing makes object copying completely independent of garbage detection or collection.

2.5.8.5 Comparison of Read-Barrier-Forwarding, Write-Barrier-Forwarding, Replication-Forwarding, and Two-Level-Addressing Detectors

The following are comparisons of the forwarding mechanisms.

1. Read-barrier-forwarding can stop a mutator for as long as it takes to copy the objects it is referencing. It may be necessary to copy many such objects at one time, and this makes it difficult to bound mutator execution time.

2. Write-barrier-forwarding can make non-initialization writes take much longer than normal because of synchronization problems.

3. Replication-forwarding does not easily permit two distinct processes to write the same component of an object, unless special synchronized writes are used that take much longer than normal writes.

4. Two-level-addressing is implemented with address registers into which pointers must be loaded to be used, and in software implementations, operations to load and store address registers take much longer than normal.

5. Two-level-addressing permits a number of other things to be done besides just forwarding. For example, objects can be copied to increase their size, or an object can be deleted and the object trap flag set to detect dangling pointers. The other forms of forwarding do not support such operations.

R-CODE uses two level addressing because it supports manual deletion and object size increases as well as garbage collection forwarding.

2.5.9 The R-CODE Write Barrier

One of the advantages of a write barrier detector listed on page 53 is that the tests $\mathbf{T}_{ns}$ and $\mathbf{T}_e$ can be combined into a single test using a bit string AND operation. In this section we will describe the *R-CODE write barrier* and analyze its effect on mutator performance.

2.5.9.1 The Bitstring AND Write Barrier Test

.

The $\mathbf{T}_{ns}$ write barrier test to ensure that no scavenged object points at an unmarked object, and the $\mathbf{T}_e$ test to ensure that no not-ephemeral-root object points at a non-permanent (ephemeral) object, are operationally identical if we make the equivalences:

$$\text{scavenged} \equiv \text{not-ephemeral-root}$$
$$\text{marked} \equiv \text{permanent}$$

To perform these tests for several garbage detections running in parallel, we may associate two bit vectors with each object (e.g. in its object-map-entry), S and M.

Then when a pointer to object X is stored in object Y, all the write barrier tests are combined into the single test: $Y.S \wedge (\neg X.M) \neq 0$. This test is called the *bitstring AND write barrier test*. If this test is true, special action must be taken that may mark objects and update lists.

If the matching bits of S and M that were respectively on and off have the interpretations "scavenged" and "marked", the action is to mark object X, i.e. set the $X.M$ bit, and to put X on the list of objects to be scavenged. This action need not be done immediately; it may be postponed until any time before the end of garbage detection.

If the matching bits of S and M that were respectively on and off have the interpretations "permanent object that is not an ephemeral root object" and "permanent object", then the action is to clear the $Y.S$ bit, put object Y on the list of ephemeral root objects, and mark Y. Again this need not be done immediately: it may be postponed until any time before the end of garbage detection.

Several garbage detections can run asynchronously using the same bit vectors. Each detection is assigned a different pair of matching bits to mean "marked" and "scavenged". Several different sets of permanent objects can also be maintained at the same time by assigning other pairs of matching bits to mean "permanent" and "not-ephemeral-root".

Most computers do not have a single bit string "x AND NOT y" instruction, but do have an "AND" instruction. For these, the bit vector M can be stored complemented in memory, so that each write requires only a single bit string AND instruction to perform all write barrier tests.

2.5.9.2 Local and Global Heaps: A Future Use for the Write Barrier Test

Here I will mention a possible additional future use for the write barrier test just described.

Because memory speeds are falling behind processor speeds[WK95], memory local to one processor is becoming more efficient than memory shared between several processors. This is partly just a matter of not having to maintain secondary cache coherence for local memory.

As a consequence, it may become expedient in the future to have separate local and

shared heaps. Each processor would have a *local heap* only it could access, and several processors would share a separate *shared heap*. The rule would be enforced that the shared heap could not point at local heaps, so that the several processors would not have troubles accessing objects pointed at by the shared heap.

To enforce this rule a pair of S and M bits would be assigned to enforce the invariant:

$\mathbf{I}_l$: No shared heap object points at a local heap object.

The M and S bits would both be off for local objects, and both be on for shared objects, so the write barrier test would have the standard form, $X.S \wedge (\neg Y.M) \neq 0$, to trap on storing a pointer to a local object Y into a shared object X.

I do not expect this to be very useful in the immediate future, but it may be within the decade.

There is, however, a technical difficulty with using the $X.S \wedge (\neg Y.M) \neq 0$ test to detect illegal stores of local pointers into shared heap objects. The difficulty is that for all other applications of this test, the pointer needs to be written in the object before the test is made,[6] whereas for the use we are describing here, we would like the pointer to be stored only after the test has been passed.

For efficiency, we will have to live with storing the pointer before the test, but this will mean that for a short time after an illegal pointer is stored the shared object will actually contain a pointer to a local heap. This situation can be detected and fixed when the deferred action buffer is processed (see section 2.5.9.3 below for deferred action buffers). Also, it may be possible to use separate ranges of object numbers for different local and shared heaps, so that illegal uses of local pointers will be detected no matter how they were communicated. Then the write barrier test will be just a convenience for detecting the guilty party.

2.5.9.3 Deferred Action Buffers

In a concurrent system, there is a problem with setting M bits, clearing S bits, and manipulating related lists: these actions must be atomic. I will assume hardware with

[6]Because there is a scavenger process that turns an object's S bit on and then looks at pointers in the object, it is necessary for the mutator to put the pointer in the object for the scavenger process to see before the mutator looks at the S bit to see whether the scavenger has already processed the object, and therefore will not see the pointer.

a relatively high atomic synchronization overhead for these actions, and therefore introduce a method of delaying and batching them.

Our store pointer instruction will do just the following. On storing a pointer to object Y into object X, if $X.S \wedge (\neg Y.M) \neq 0$, pointers to X and Y will be written into a *deferred action buffer*. Each process will have its own buffer, and so will not need to interlock these buffer writes. When the buffer is full, its process will synchronize with other processes to set M bits, clear S bits, and maintain lists.

There are two ways to keep the time needed to process a full deferred action buffer reasonable for fast high priority processes. First, the buffer may be made just long enough to amortize the cost of synchronizing, in which case the cost of processing it should not be more than several times the cost of synchronizing. Or second, full buffers may be shipped to a lower priority process for processing.

2.5.9.4 Write Barrier Mutator Overhead

The mutator write barrier pseudo-code is shown in Figure 2.3. Based on this the write barrier takes 6-12 RISC instructions best case and 10-20 instructions worst case, where the lower bounds are from the figure and the upper bounds are double the lower bounds. The best case is when the write barrier test indicates no further action is required. The worst case is when a deferred action buffer entry is written. Here we assume that in the worst case the end of the deferred action buffer is not reached, so the subroutine to process this buffer is not called.

When the end of the deferred action buffer is reached, the deferred action buffer end processing routine is called. This routine locks out the scavenger processes and executes the loop in Figure 2.4 to process the buffer entries.

For each entry, either some $X.S$ bits are turned off or some $Y.M$ bits are turned on. It is important to do this soon after the write barrier test indicates it is necessary, so it will normally be done by the mutator. Otherwise, a single unmarked Y object may produce many deferred action buffer entries before it is marked, for example.

Then an action is sent to a scavenger that will perform the actual processing. The scavengers run in background with different priorities: there might be a faster high priority scavenger for ephemeral garbage collection, and a very low priority scavenger for a full garbage collection. The action is sent to a buffer associated with the destination scavenger. An operation code is sent as part of the action to tell the scavenger

In the following `Y-object-number-reg` is a register holding the object number of an object Y, and `AX` is the address register pointing at an object X that contains a `pointer-component` into which Y's object number is to be stored. The current entry in the deferred action buffer is pointed at by the `deferred-entry-reg` register, and the point just after the end of the buffer is pointed at by the `deferred-end-reg`. An `action-entry` in this buffer has two components: `X` and `Y`. Other considerations are as in Figure 2.2, page 40.

```
pointer-component (AX.byte-address) = Y-object-number-reg
memory-barrier-instruction
temp-reg-1 = map-S-bitstring(AX.object-number)
temp-reg-2 = map-not-M-bitstring(Y-object-number-reg)
temp-reg-1 = temp-reg-1 AND temp-reg-2
if (temp-reg-1 non-zero) then
        X(deferred-entry-reg) = AX.object-number
        Y(deferred-entry-reg) = Y-object-number-reg
        deferred-entry-reg += action-entry-size-constant
        if (deferred-entry-reg >= deferred-end-reg)
                call deferred-action-buffer-end-subroutine
```

Figure 2.3: Write Barrier RISC Pseudo-Code

what to do, i.e. which bits of $X.S$ were cleared and which bits of $Y.M$ were set. If more than one bit is changed, there may be work for more than one scavenger, but the mutator only sends the action to the highest priority of these scavengers, and after that one scavenger has done its part, it forwards the action to the next highest priority scavenger.

The time needed to process a deferred action buffer entry is 30-60 RISC instructions, where as before the lower bound is from the pseudo-code and the upper bound is double the lower bound, and in this case the assumption is made that the per entry overhead for starting and stopping the processing of one buffer is half the per entry loop overhead (which is 20-40 instructions from Figure 2.4). The overhead for starting and stopping the processing of one buffer includes the time to lock out scavengers before processing the buffer, and reset this lock after processing the buffer. The buffer

The following loop processes the action entries in a deferred action buffer. For each entry containing the object numbers of X and Y, some $X.S$ bits may be turned off and some $Y.M$ bits may be turned on (the complement of M is called `not-M` and its bits are turned off). Then an operation is sent to one of several (1 to 3) scavengers with different priorities. In the case given below, the second scavenger is sent to using a buffer whose current entry is pointed at by `scavenge-2-entry-reg`.

```
loop:
    X-object-number-reg = X(deferred-entry-reg)
    Y-object-number-reg = Y(deferred-entry-reg)
    S-reg = map-S-bitstring(X-object-number-reg)
    not-M-reg = map-not-M-bitstring(Y-object-number-reg)
    temp-reg = S-reg AND not-M-reg
    jump to dispatch-table(temp-reg)
            .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
        case-....:
            S-reg = S-reg AND case-S-mask-constant
            map-S-bitstring(X-object-number-reg) = S-reg
            not-M-reg = not-M-reg AND case-not-M-mask-constant
            map-not-M-bitstring(Y-object-number-reg) = not-M-reg
            scavenge-opcode(scavenge-2-entry-reg) =
                    case-opcode-constant
            scavenge-X(scavenge-2-entry-reg) = X-object-number-reg
            scavenge-Y(scavenge-2-entry-reg) = Y-object-number-reg
            scavenge-2-entry-reg += scavenge-entry-size-constant
            if (scavenge-2-entry-reg >= scavenge-2-end-reg)
                call scavenge-buffer-2-done
            jump to end-of-dispatched-code
            .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
end-of-dispatched-code:
    deferred-entry-reg += action-entry-size-constant
    if (deferred-entry-reg < deferred-end-reg)
            jump to next iteration of loop
end-of-loop:
```

Figure 2.4: Deferred Buffer Processing Loop RISC Pseudo-Code

need only be long enough to amortize this time.

The mutator write barrier overhead can be viewed as having two components: the time for the test, 6-12 instructions, and the time for the mutator to process an action, 35-70 instructions. I will call the first component the "*mutator test time*", and the second component the "*mutator action time*." The total mutator action time during any complete garbage detection is proportional to the number of used objects plus the number of objects added to the ephemeral root list. Thus the total action time is bounded by considerations similar to those that bound the total scavenger process time. These are discussed below in section 2.6.

For high priority processes, the deferred action buffer may be sent to a lower priority process for handling, moving the action time off to lower priority.

It is important to note that when a new object is initialized, only the write barrier test time, 6-12 instructions, occurs. This is because the new object's scavenged flags are not set (including the flag that really means "not-ephemeral-root"). No action is written in the deferred action buffer, and no buffer processing occurs. However, there may be exceptions to this during the "Write Barrier End Game": see section 2.5.5.2.1 above. Also, the test must still be made for two reasons: first, illegal stores of local pointers into global storage must still be detected (see section 2.5.9.2, "Local and Global Heaps", above), and second, the new object may have its scavenged flag set during the write barrier end game.

The importance of a fast write barrier test should be clear.

2.5.9.5 Related Work on Write Barriers

Write barrier tests are described by Hosking, Moss, and Stefanovic in [HMS92], which measures various write barrier schemes used to maintain ephemeral root-set lists of permanent objects that reference ephemeral objects. The tests there involve comparing generation numbers, "trapping" if one is less than the other. Unlike our system, there is no ability to suppress a trap if one has already occurred for the object being stored into (S bit already cleared). Also, there is no ability to have the $\mathbf{T}_{ns}$ write barrier test be "for free" in the presence of the $\mathbf{T}_e$ ephemeral root write barrier test.

2.6 Non-Mutator Overhead

In a concurrent system, garbage collection processes run concurrently with mutators.

These collection processes may do operations such as scavenging marked objects, sweeping unused objects into free blocks, and compacting used objects to make a single large free block.

The garbage collection processes require a certain amount of CPU time to process:

1. Each used object (for marking/scavenging/sweeping/compacting).
2. Each ephemeral root object (for marking/scavenging).
3. Each pointer in a used or ephemeral root object (for scavenging).
4. Each byte of a used object (for compacting).
5. Each unused object (for sweeping).
6. Each pointer to a root object (for marking).

I will call one garbage collection algorithm execution a *gc cycle*. Thus for each used object there is a certain amount of CPU time required during each gc cycle, which I refer to as the *gc time overhead* of the used object. This is a linear function of the number of pointers in the object and the size of the object. There is a similar linear function for ephemeral root objects, but the size in bytes of such objects does not contribute. There is a gc time overhead for each unused object, which is generally a small fixed time regardless of the contents and size of the object. And lastly there is a small fixed gc time overhead for each pointer to a root object, to mark the root object.

To keep the total gc time overhead bounded, one must have:

1. An upper bound for the number of used objects.
2. An upper bound for the number of pointers to root objects.
3. An upper bound for the number of ephemeral root objects.
4. An upper bound for the number of pointers in used and ephemeral root objects.
5. An upper bound for the number of bytes in used objects.
6. An upper bound for the allocation rate of new objects.

7. An upper bound for the rate of allocating bytes in new objects.

8. A lower bound for the number of bytes of memory available for used and unused objects.

9. A lower bound on the number of used plus unused objects allowed.

These items can be used to find bounds for the time taken by gc cycles, and this with the various upper bounds gives a minimum efficiency for the garbage collector.

A garbage collection cycle also needs memory for lists of objects to be scavenged and lists of ephemeral root objects. This memory has a size proportional to the number of used objects and the number of ephemeral root objects, respectively.

There is no problem bounding all the above appropriately for real-time applications as long as the allocation rates are not too high and there is several times as much memory as required to hold the used objects.

There is one part of the write barrier garbage collector non-mutator overhead that is not bounded by the above considerations. This is the overhead resulting from write barrier end thrashing that was discussed in section 2.5.5.2.2 above.

There is also a question of how to define "*used object*." Bounds on the number of used objects and the number of bytes they consume are experimentally measured for most systems, so any experimentally measurable definition will work. We therefore chose to define an object as used during a garbage collection cycle if it exists and is not collected at the end of the cycle. Similar practical definitions can be made for the other quantities above.

Use of ephemeral garbage collection helps keep the amount of memory needed down by not counting permanent objects in any of the above bounds, except that ephemeral root objects count slightly in the time bound.

Real-time systems often use queues of free blocks of several fixed sizes to get good real-time allocation and deallocation times. This can also be done with our memory manager, using manual deletion to return an object's memory to a free queue. Then the byte allocation rate for the garbage collection cycle does not have to count bytes allocated and freed in this manner. Therefore cycles can be very much longer than they would otherwise have to be, and the cycle can be very efficient. The only part of an unused object that the cycle must collect is the object map entry, which is relatively small.

2.7 Summary

I believe that a memory manager based on two level addressing as implemented in software by address registers, and on the bitstring AND write barrier test as discussed above, will be a strong candidate for a universal memory manager that will permit many different languages to interoperate.

Chapter 3

Data Types

3.1 Goals

As an intermediate programming language, the R-CODE virtual machine language provides only data types matched to the hardware. Thus the data types of R-CODE are various forms of integer and floating point numbers, and various forms of pointers.

The main goal of this chapter is to:

> $\mathbf{G}_\iota$: Standardize data types so that different high level languages, different computer hardware, and programs written by different people at different times can interoperate.

This is not a new goal. It has been pursued by the computer hardware community for decades, and has resulted in standardizing on the 8-bit byte, the two's complement integer, and the IEEE floating point number format. As far as number types are concerned, the differences between big and little endian computers and between the alignment requirements of different computers are the only significant remaining issues.

R-CODE follows in this trend, but is very careful about pointer data because it must support the garbage collecting memory management of Chapter 2. R-CODE is also very careful about big and little endian and alignment issues.

In addition, R-CODE introduces other ways of formatting information that make it easier for programs written by different people at different times to interoperate. The

first of these is tagged values, which are used in many LISP implementations. The second is including type information in pointers, so machine code discovers at run time the exact size of numbers pointed at. The third is type maps, so machine code discovers at run time both the size and displacements of structure components. The fourth is a type matching system, so machine code can translate at run time the actual types of data, as expressed by type maps, into the formal types expected by the code, also expressed by type maps. The fifth is array descriptors, so that machine code discovers array layouts at run time.

The method of this chapter is simply to extend the basic hardware data types that have become standard in the last several decades. R-CODE includes all the extensions I have thought of that address $\mathbf{G}_\iota$ above and that have a reasonable chance of being passably efficient on existing hardware (see $\mathbf{C}_2$, page 13) and being very efficient on potential future hardware. Only occasionally does this chapter discuss alternatives that have not been included in R-CODE for one reason or another.

3.2 Requirements

The basic R-CODE goal (see page 1):

> $\mathbf{G}_\alpha$: A high school student knowing a bit of algebra and geometry can investigate and change commercially written games, editors, and simulators.

leads to programming languages of the "strongly typed LISP" variety. Basic LISP computes with tagged values, but if the user provides sufficient strongly typed language style declarations, LISP can deal with untagged values, and be much more efficient on existing computers. Thence the requirement (quoted from page 18):

> $\mathbf{F}_6$: R-CODE will support tagged and untagged values and an efficient transition between the two kinds of data.

We also want programming languages that support mixing code written by different people, some commercial, some students, and some non-commercial professionals. To get this code to interoperate properly, it is desirable to delay until runtime matching the actual type of data to the formal type expected by functions, to the extent

allowed by efficiency goals. One way of doing this is to represent types by type maps that tell the sizes and displacements of components of objects of the type, and the addresses of functions that operate on these objects. A type map is a vector organized analogously to a page table, and can be efficiently implemented in future hardware. R-CODE attempts to make type maps as capable as they reasonably can be, and therefore strives to meet the following requirement:

> $\mathbf{F}_4$: R-CODE will support type maps, and will efficiently handle execution of very small functions selected from a type map, and also efficiently handle loads and stores in which only displacement, numeric type, and size are stored in the type map.

Just as the map from an actual object to a formal object (i.e. the object as seen by a routine) is encoded in a type map, a linear map from a list of subscripts to an address within an array is encoded in a map we call a "array descriptor". One of the things I learned during my decades writing glue software (see "Background", page 1) is that array descriptors are the right way to treat array accesses. This is because there are many instances when pieces of an array need to be treated as arrays in their own right. Thus:

> $\mathbf{F}_5$: R-CODE will support array descriptors, treating them as first class data except that they may not be modified once created.

The rest of this chapter is organized into three main sections. The first describes basic R-CODE number, pointer, and tagged types, and explains how R-CODE pointers containing numeric type information are used. The second analyzes type maps. The third describes R-CODE array descriptors.

3.3 Basic Data Types

Integer and *floating point number* types are standardized by existing hardware. R-CODE supports signed two's complement integers of sizes from 1 to 64 bits, unsigned binary integers of sizes from 1 to 64 bits, and floating point numbers of four sizes: 16, 32, 64, and 128 bits[1].

[1]The format for a 128 bit floating point IEEE number can be found in [Inc92]. For 16 bit IEEE numbers the only sensible parameters are 10 bits of mantissa and 5 of exponent. If denormalized

One set of basic data type issues concerns how to transmit data efficiently between big and little endian computers, and between computers with different alignment requirements. A second set of issues concerns how to represent the tagged values required by LISP. A third set concerns how to represent and use pointers. This last includes whether or not to include type information in pointers, what information to include, and how to use this information efficiently at run time. The following subsections consider these three sets of issues in turn.

3.3.1 Memory Units

A *memory unit* is a data component that preserves its addressability when transported without reformatting between computers of different data formats. By addressability we mean merely the ability to compute the address of the memory unit. By computers of different data formats we include, in particular, computers of different endianhoods. An aligned 8-bit byte is a memory unit.

On the other hand, if we transmit a byte-aligned vector of 8 5-bit elements from a big endian computer to a little endian computer, we get a result in which the elements cannot reasonably be addressed unless all 5 bytes of the vector are reversed in memory and the direction in which the elements are access is also reversed: see Figure 3.1.

Clearly the 5-bit elements of this vector are not memory units. The entire 40-bit vector might be considered a memory unit, because even without reversing its bytes the vector itself is addressable after transmission.

R-CODE requires that all memory units have a size in bits that is a power of two, and that all memory units be *aligned* (have an address that is an exact multiple of their length). Addressing memory units of this kind is the same for big and little endian computers, except for units of size less than 8 bits. For these, R-CODE requires big endian addressing. It is possible to insist on big endian addressing of memory units, even on little endian computers, by just complementing some of the low order bit-address bits when accessing the unit, because memory units are aligned on power of 2 boundaries. Big endian is preferred because that is what I/O devices usually

numbers are supported, this gives a precision of 0.1% or 10^{-7}, whichever is greater, and a range in excess of -65,000 to 65,000. Only load and store operations, and not arithmetic, would be supported for 16 bit floating point numbers. Implementations might have to do 128 bit floating point computations to only 64 bit precision, for efficiency.

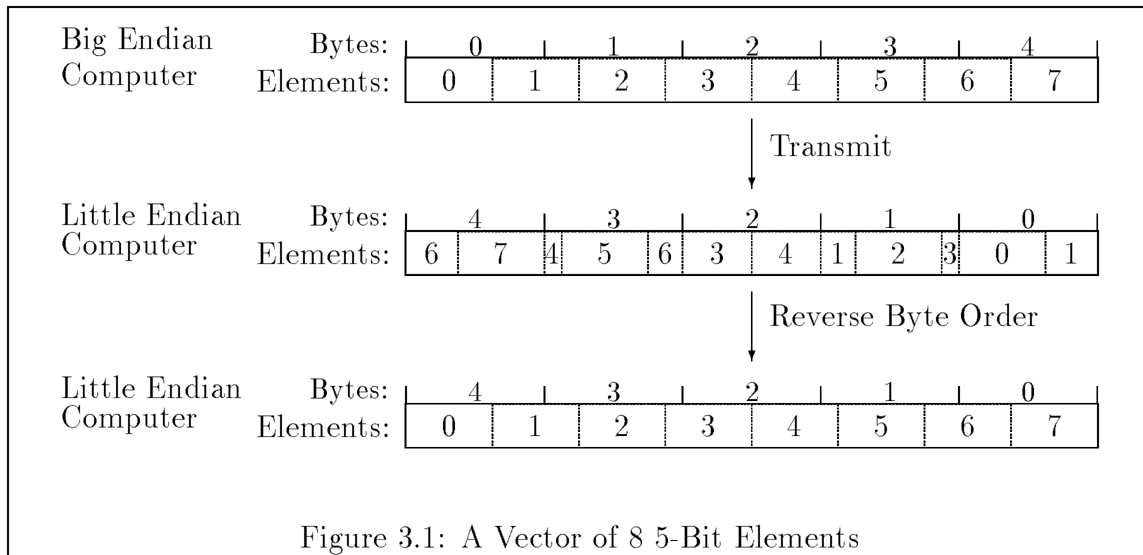


Figure 3.1: A Vector of 8 5-Bit Elements

prefer.

R-CODE requires that objects which are to be copied consist of contiguous disjoint memory units, with the displacement of each memory unit from the beginning of the object being a multiple of the size of the memory unit. The *alignment* of the object is then the size of the largest memory unit.

With these R-CODE requirements, when an object is transmitted between a big and a little endian computer, the only "endian conversion" required is reversing the bytes of each memory unit larger than 8 bits. This solves the problem of efficiently transmitting data between computers of different endianhood.

R-CODE does not permit arbitrary *bit fields*, e.g. a 5-bit quantity or unaligned 2-bit quantity, to exist outside memory units. Such fields can only exist inside memory units, and can be extracted from these memory units by operations such as integer shifts. Therefore, such bit fields can be ignored when transmitting data between computers and reformatting memory units to meet the needs of different computers.

The reasons for this are:

1. As discussed above, arbitrary bit fields are not memory units, and cannot sensibly be copied between computers of different endianhood.

2. Many modern computers do not have instructions for loading and storing bit fields directly into memory.

The consequence of the R-CODE requirements is that to access a component, one must first access the memory unit that contains it, and then access the component within the memory unit. Displacements of memory units can be used to load or store memory units. Displacements of components within memory units can be used to extract components from the memory units, or merge components into the memory units.

The *displacements* of the memory units are computationally like one would expect displacements to be; but the displacements of components within memory units are more like the numeric type of the component, and need to be encoded in instructions by a compiler to be most efficient on many existing computers.

One can generalize R-CODE's restrictive notion of memory units rather easily to include any value that is a multiple of 8 bits in length and is aligned on a byte boundary. When transmitted between different endian computers, the bytes of the unit are reversed. The vector in Figure 3.1 would be an example. The major disadvantage is that unaligned data are much, much slower to access than aligned data on the coming generation of computers (a factor of 10-100 slower on the Alpha: [Sit92, p. A-6-7, and A-2]).

R-CODE memory units also have *formats*: signed integer, unsigned integer, floating point, and tagged value. These can be used to translate memory units when copying data between different kinds of computers. For example, floating point data might be converted. Format specific translation of memory units is not needed when both computers have IEEE format floating point numbers.

3.3.2 Tagged Values

In order to support LISP, R-CODE provides *64-bit tagged values* modeled on the IEEE 64-bit floating point number. Floating point numbers are represented as per the IEEE format. Tagged integers and pointers are represented as signaling-NaNs, IEEE values that will cause a trap if input to a floating point operation. The non-signaling-NaN is reserved for use in indicating the output of an instruction whose inputs were illegal: see Chapter 4 (particularly section 4.3.13, page 118).

Figure 3.2 gives the basic format of R-CODE 64- and 128-bit tagged values. For

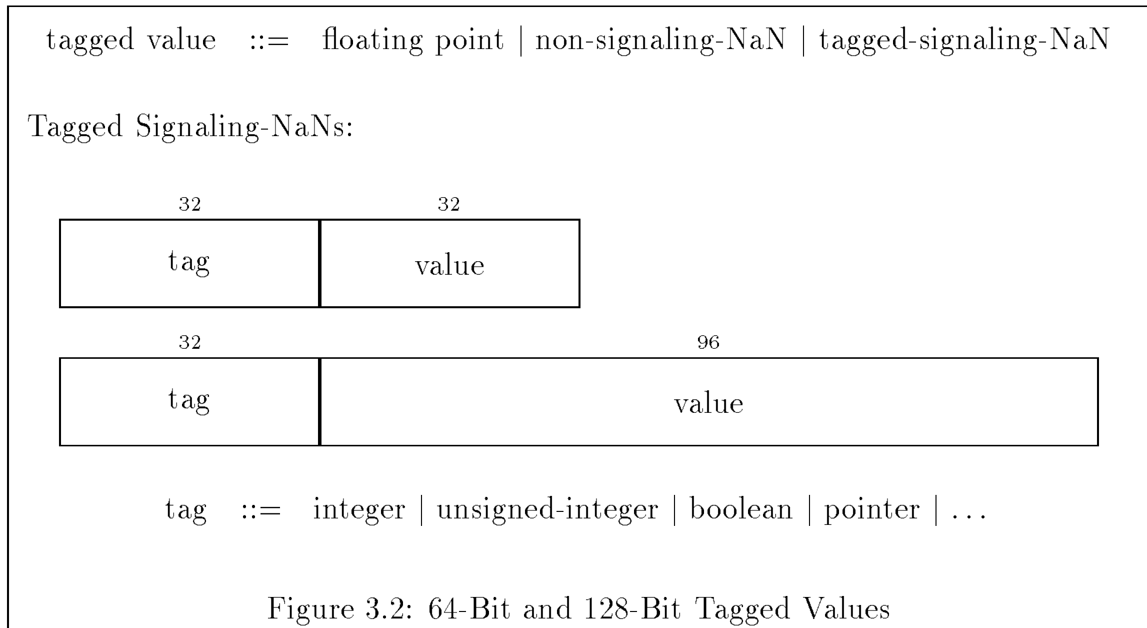


Figure 3.2: 64-Bit and 128-Bit Tagged Values

64-bit tagged values, 32-bit quantities can be represented by a 32-bit *tag* and a 32-bit value, where the tag is chosen to make the whole an IEEE signaling-NaN. This is used to represent 32-bit signed and unsigned integers. It is also used to represent special values such as the *OMITTED* value, that can be passed to indicate an omitted optional argument, and the *ERROR* value, that contains a 32-bit error code and can be passed to indicate an error.

When a *pointer* is stored in a 64-bit tagged value, the 32-bit value part is used to hold the *object number* that designates the object being pointed at. This object number references an object map entry in a table, and the object map entry holds the address of the object: see section 2.4 in Chapter 2 for details (particularly Figure 2.1 on page 32). The 32-bit tag in the pointer tagged value is used to hold a 19-bit *type map number* that references a type map which describes the apparent type of the object. Type maps are discussed below in section 3.4 of this chapter. 19 bits is the largest number of bits that can reasonably be allocated for a type map number while still making the value a 64-bit IEEE signaling-NaN and leaving room for other tagged values.

A LISP system has been built by Umemura[Ume91] that represents LISP tagged

data as 64-bit VAX floating numbers. Pointers (without type map numbers) are represented by invalid floating point numbers that cause traps when input to arithmetic operations. In this LISP system, the integer 1 and the floating point number 1.0 are not distinguished, and floating point hardware instructions are used for LISP integer arithmetic. Arithmetic efficiency is comparable to FORTRAN.

R-CODE register dataflow execution, as described in Chapter 4, is supported by *128-bit tagged values* that are used as the values of virtual registers. The registers are virtual in that they all have an associated type code that restricts their range of values and permits them to be stored more compactly in actual implementations. For example, the type code of a register might restrict the register's values to 32-bit signed integers, so the implementation would store the register in a 32-bit hardware register, but the R-CODE debugging interface would present the value to a source language debugger as a 128-bit tagged value. So the human user can think of all register values as being 128-bit tagged values, while the hardware can view them more efficiently.

One of the dangers of using a virtual computer model as an intermediate representation for program code is that the constraints of packing information into the fixed size data units of a virtual computer will corrupt the semantics of the intermediate representation. R-CODE avoids most of this problem by using very large virtual registers and very large virtual instructions. Thus, after some experimentation, 128 bits was settled on as large enough for a virtual register, though there is little reason one could not go to 256 bits.

The 128-bit tagged value format is based on the 128-bit IEEE floating point format[2], just like the 64-bit tagged value format. Floating point numbers, including the infinities, are represented in IEEE format. Non-signaling-NaNs are reserved to indicate the outputs of instructions with illegal inputs. 64-bit quantities, such as signed and unsigned integers, are prefaced with a tag. 128-bit tagged pointer data are discussed in the next section.

3.3.3 Pointers

A full pointer in R-CODE is a *128-bit tagged pointer* formatted as in Figure 3.3. The object number designates an object, the unsigned displacement designates the

[2]See [Inc92, Table 3-5]. Its not completely clear to me whether this is an official IEEE standard yet for the 128-bit size.

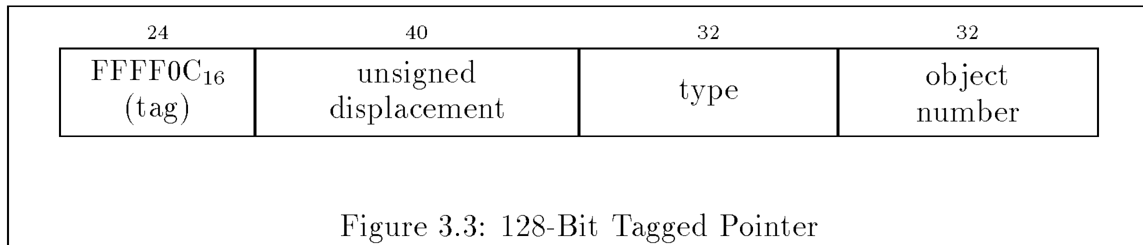


Figure 3.3: 128-Bit Tagged Pointer

displacement in bits of a memory unit within the object, and the type designates the type of the component stored in the memory unit or the component that begins with the memory unit. Types will be explained below.

128-bit pointers are usually stored in virtual registers, and therefore are not stored by implementations directly in the format given. In particular, implementations usually store only a 32 bit displacement, which is in units of either bits or bytes, depending upon the type field in the pointer. This means that implementations have permission to implement only 32 bits of displacement for bit aligned memory units and implement only 35 bits of displacement for byte aligned memory units.

3.3.3.1 The Pointer Type Field

There are three different formats for the 32-bit *type field* of a 128-bit tagged pointer, formatted as in Figure 3.4. The *scalar type*, which we will explain below, is used for simple number and pointer components. The *mapped type* is used for components associated with a type map, and will be explained later within section 3.4. A mapped type contains a *type map number* that points at a type map. The type map has the scalar type of the component in element 0, so even a mapped type has a scalar type.

Lastly the *routine type* is just a means of expanding the 32-bit type field to hold extra information, namely pointers to three routines that perform the component load, store, and load-address functions. The routine type contains a *routine map number* that points at a routine map which holds this extra information, and also holds the 32-bit type that would have been stored where the routine type was stored if we did not need the extra information. This 32-bit type cannot itself be a routine type, but can be a scalar or mapped type. In either case one eventually finds a scalar type, so even a routine type has a scalar type.

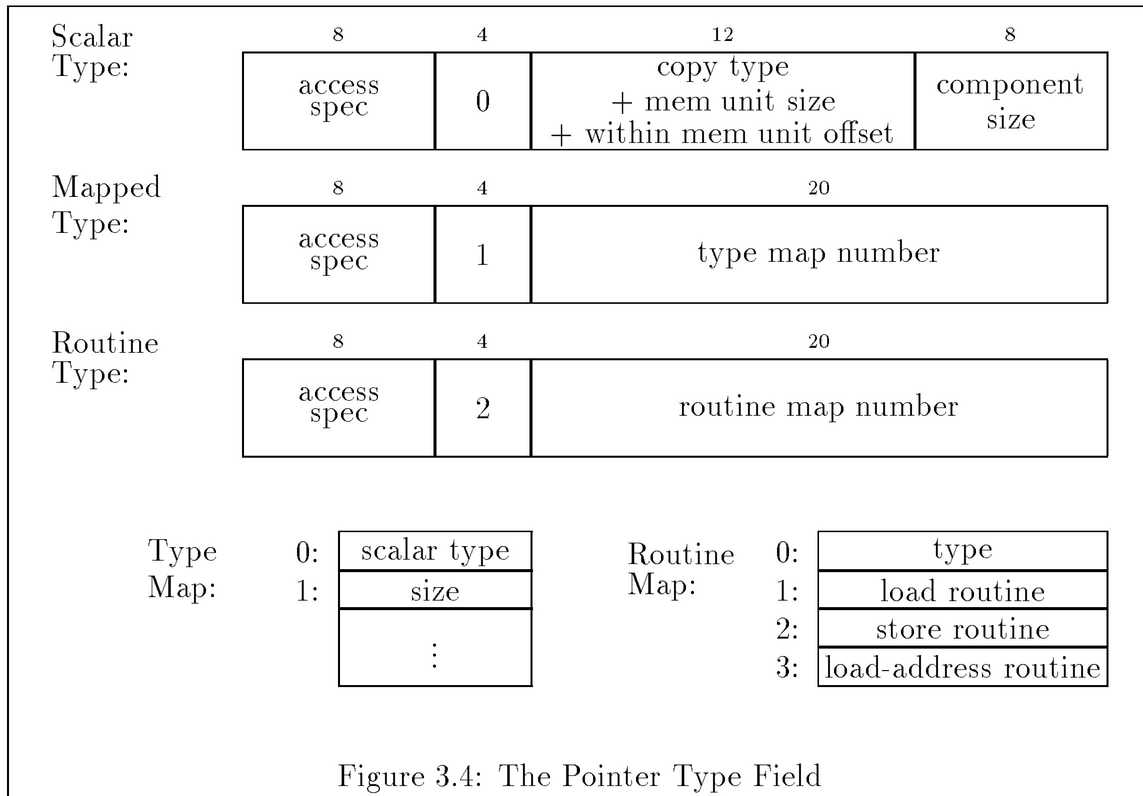


Figure 3.4: The Pointer Type Field

The *access specification* field of a scalar, mapped, or routine type specifies whether the component is write-only, read-only, read-write, or something more complicated. On existing computers access specification information needs to be compiled into instructions to be efficient.

In parallel computing it is desirable to break code into blocks within which memory operations may be reordered by the compiler or hardware. Memory read operations on a read-only component can be reordered without problem, but read-write and even write-only operations on the same component cannot be. Thus it is desirable to introduce other more order-independent access specifications such as "write-once", "accumulator", and "atomic-transaction". These are discussed in Chapter 5.

3.3.3.2 Scalar Types

The *scalar type* contains a *copy type* and a *component size*. These two pieces of information are just what is needed to copy a scalar value between registers. In addition the scalar type contains a *memory unit size* and a *within-memory-unit offset* that are needed to copy the scalar value between a register and memory.

Copy types are listed in Figure 3.5.

integer	Signed integer.
unsigned	Unsigned binary integer.
unsigned$N$	Unsigned binary integer that is an exact multiple of N for $N = 2, 4, 8, 16, 32, 64$, or 128
float	Floating point number.
tagged	64 or 128 bit tagged value.
pointer	64 or 128 bit tagged pointer.
object number	32-bit object number.
type map number	32 bit type map number.
routine map number	32 bit routine map number.
reduced pointer	64 bit pointer containing a 32-bit object number and a 32-bit unsigned displacement.
contiguous subobject	Object that is a contiguous set of memory units.
discontiguous subobject	Object that is scattered in memory.

Figure 3.5: Copy Types

Besides the normal number types there are special types for storing numbers that are exact multiples of small powers of two. An "*unsigned8*" copy type value, for example, can be used to store an unsigned integer that is an exact multiple of 8, without storing the low order 3 bits of the integer. The value actually stored in memory is multiplied by 8 to get the true value. Such a value can store a size that appears to be in units of one bit when in fact it is in units of one byte.

There are a variety of pointer formats. A *reduced pointer* contains just the information an implementation would store for a 128-bit tagged pointer, but without the type field (see Figure 3.5). The displacement is stored as 32 bits and is in either bit or byte units,

depending on the missing type field, which must be added later by an instruction.

A *contiguous subobject* is a contiguous sequence of memory units that contain subobject components accessed via a type map. A *discontiguous subobject* is a set of components accessed via a type map, with the components being possibly scattered in memory. Subobjects are discussed more below in section 3.4.8.

The scalar type stores the size of the memory unit containing the scalar component and the offset of the component within that memory unit. For integers the *memory unit size* can be 1, 2, 4, 8, 16, 32, 64, or 128 bits, and the *offset within the memory unit* is up to 7 bits long. The *displacement* stored in the pointer is the displacement of the memory unit, and must be an exact multiple of the memory unit size, since all memory units in R-CODE are aligned. Pointer components, including object and type map numbers, are all aligned, and their memory unit size and component size are equal. Pointer components only come in 32, 64, or 128 bit sizes.

3.3.3.3 Loads and Stores

The next question is what happens when an R-CODE load or store instruction inputs a 128-bit tagged pointer for the purpose of reading or writing a memory location.

R-CODE has an instruction to check whether the type field of a pointer matches a particular value exactly. If this instruction has been applied to the pointer which the load or store instruction is going to use, the R-CODE load or store can be compiled for existing computers into a single machine load or store instruction containing the specified type information.

It is also possible that the pointer was computed in the current subroutine by some instruction that specified the pointer type field, or the type field will be known at compile time for some other similarly straightforward reason.

But if the type field is not known, the load or store must cope with many different types. I propose that R-CODE implementations do this by making the load or store instruction into a case statement that switches on the type field of the pointer. Furthermore, the cases of this case statement are dynamically compiled: whenever a new value for the type field is seen by the case statement, it compiles a new case of itself to handle the new type field value. Thus load and store instructions compile into *dynamic case statements*.

If load and store instructions using the same type field can be batched, this will

be more efficient. It may even be appropriate to batch such instructions with non-memory reference instructions, as when the batch forms a tight loop. Such batching is facilitated by the fact that R-CODE tries to permit instructions to be as reorderable as possible (see Chapter 4). This kind of batching is clearly experimental, and considerable experience will be required to quantify its efficiency.

When a routine receives an argument that is a pointer to an object or subobject, the routine typically applies a type matching operation to get a new pointer with a match type. Then this new pointer is used with load and store instructions. These load and store instructions are dynamic case statements, switching on the match type map number. But the implementation can streamline this situation by having the load and store dynamic case statements switch directly on the type map number from the original pointer argument, so the type matching operation is never actually executed, except while compiling a new case.

An advantage of the dynamic case statement approach is that one is not limited to compiling cases that contain single load or store instructions, but may compile inline whole small routines to perform loads and stores. Routine types take advantage of this by specifying routines to perform component load, store, and load-address operations. These routines can be compiled inline if they are small, or calls to them can be compiled if they are large. Inlining is becoming increasingly important; for example, one manufacturer of newest generation computers recommends that routines of 20 or fewer instructions be inlined for efficiency ([Sit92, Section A.2.3, Number 2.]).

Load and store instructions operate on the scalar type of a pointer. A pointer may have a mapped type which in turn has a scalar type stored in a type map. Thus the pointer may point at scalar numeric elements of an array, for example, while there is still a type map associated with the type of these elements. This type map may contain function pointers for operations on the array elements, such as the "max" operation mentioned in section 3.4 below. Calls to such functions may be separately treated as dynamic case statements, or may be batched with load and store instructions that are so treated.

Dynamic case statements have two differences from the hash table based dynamic dispatching common to object oriented languages. First, the table searched by a dynamic case statement is very much smaller than the hash tables: often it may be only 1 to 3 elements, and sometimes 10 or so. Second, the code being dispatched to can be inlined at the call site by the dynamic case statement.

3.3.3.4 Related Work on Dynamic Compilation

Hölzle and Ungar have implemented a compiler for the SELF language in [UU94] that is very similar to dynamic case statements. The difference is that the SELF system does not compile new cases dynamically at runtime, but rather feeds back to a compiler a histogram of the frequency with which various types are encountered at runtime by a particular object oriented method call. If a case has not been compiled inline, it is handled by an general object oriented method dispatcher. Their system handles 75-80% of all method invocations with cases compiled inline.

There has been considerable additional work on dynamic or adaptive compiling in general, and not just compiling case statements switching on types. The work on SELF referenced in [UU94] can be used as a starting point for accessing this literature.

The advantage of R-CODE should be that compilation of new cases should be very fast, because R-CODE is like machine language. Thus dynamic on-the-fly compilation should be doable with little run-time penalty. This remains to be tested in practice. And, of course, if one wants full optimization, then recompilation of full routines using knowledge of the types seen so far would be necessary and take time.

Lastly, some way of handling particular situations where classical object oriented hash table dispatch is really the best approach, because a particular dynamic case statement would have too many cases that would not benefit from inlining, is also needed by an R-CODE implementation.

3.4 Type Maps

A *type map*[3] is a vector of *component descriptors* (see Figure 3.6), where each component descriptor tells how to access a component within a block of memory that represents an object. Type maps are used to complete the definition of a type at run time.

For example, suppose a subroutine is passed an object that it knows to be of formal type "animal". It knows that all animal objects have a "weight" component, but it does not know at compile time where the weight component is within the object. However, it does know that at run time a type map will be provided to enable the object to be viewed as an animal, and that element 3 of this type map will be the

[3]Type maps in some form were invented at least as early as 1966 [Str94, p. 258].

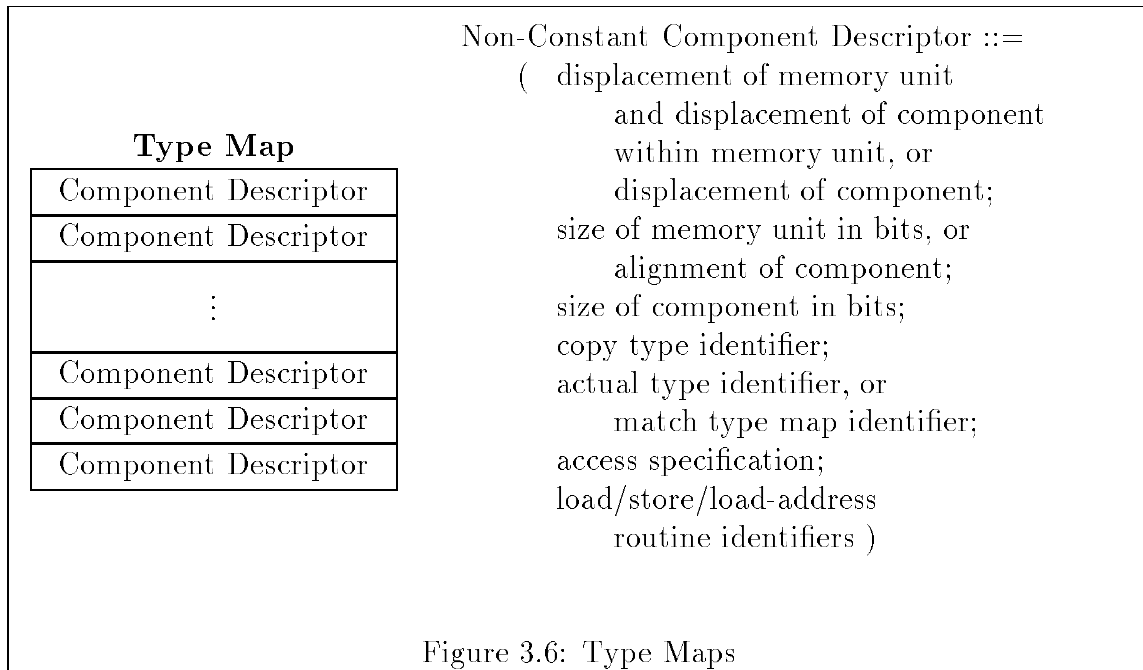


Figure 3.6: Type Maps

displacement of the weight component in the object.

A type map may also contain constants. These may be thought of as descriptions of "components" that depend only on the type of an object. Such constant components can be addresses of functions that tell how to perform operations on objects of a given type. Or constant components can be addresses of type-related variable data, such as free lists for allocating objects of a given type.

For example, suppose a subroutine is passed an array whose elements have formal type "lattice element". The subroutine knows that all lattice element types have a "max" operation, but does not know at compile time how to perform this operation. However, it does know that at run time a type map will be provided to enable the array element values to be viewed as having values of type lattice element, and that element 2 of this type map will be the address of a function that will implement the max operation for these values.

Type maps are computed as follows. Every object or value has an *actual type* that denotes the object format and contents. Every piece of code that references the object or value considers the object or value to have a *formal type*, which denotes

what the code knows about the format and contents of the object or value. There is a *type matching operation* that inputs an actual type and a formal type and outputs a type map. This type map is called the *match type map*, and assists the code in interpreting the object or value as having the desired formal type, as the above two examples illustrate.

Note that languages usually permit every actual type to be used as a formal type. Also, formal types that are not also actual types are often called *abstract types*, and are supported only by more recent languages.

Below I will discuss type maps somewhat abstractly. It may help the reader to known that in the end R-CODE non-constant component descriptors turn out to be just like 128-bit tagged pointers (page 78 above) without the object number.

In R-CODE, type maps provide a notion of a *virtual object* that is mapped onto an *actual object*. The actual objects and the code using them may be created by different programmers at different times, and the type maps fix up the difference by presenting the code with the correct virtual objects. Or many different types of actual object with something in common may be processed by the same code by making all these objects appear to have the same formal type using type maps. All of this promotes the goal $\mathbf{G}_\iota$ of permitting programs to interoperate.

Type maps are required to be actual vectors: they may not be conceptual vectors implemented by hash tables. If we were to allow type maps to be hash tables, some of the analysis would be quite different. Another way of putting this is to say that any use of hash tables is confined to the type matching operation, and the output of this matching operation is a type map that is a true vector that will be accessed using element indices known at compile time. Furthermore, as we will see below in sections 3.4.3.3 and 3.4.3.1, type matching can often be done using only vector element accesses with indices known at compile time.

3.4.1 Issues to be Analyzed

A basic principle of R-CODE is to provide maximally capable type maps. The method of this section is to analyze what is possible with type maps, and include the best of these possibilities in R-CODE. The issues I will discuss are:

1. How might type matching be done?

2. Should type maps be constant?

3. How should actual types be specified?

4. What information can be put into a component descriptor?

5. What is in an R-CODE component descriptor?

6. How are subobjects handled?

7. Examples of deficiencies in existing languages.

One issue not analyzed is how to architect future hardware that will optimally implement more capable type maps.

3.4.2 Type Map Related Work

Although limited type maps have been used by various languages for decades, no one seems to have attempted a systematic analysis of their potential[4]. Note that type maps in our sense are intended to be high performance; we exclude the many low performance dispatch table methods and methods that always make out-of-line subroutine calls.

Detailed information has been published on the type maps used in C++[Str94, p. 266], EIFFEL[MS94, page 7], SATHER 0.6[MS94], and HASKELL[Jon94, section 7.5.1].

Languages that have made some use of type maps have not been comprehensive about doing so. Probably this is because type maps are introduced only to implement certain features of the languages.

However, we believe that the most successful programming languages provide relatively complete control over their target hardware and runtime systems, that is, over their computer model. Examples are C and LISP. Proceeding in this manner, we

[4] The paper by Milton and Schmidt[MS94] comes close. It contains an analysis of EIFFEL and SATHER dispatching, but does not contain topics, such as general type matching and inclusion of type information in component descriptors, that arise when one attempts to analyze type maps independently of any language.

believe languages should provide complete control over type maps, permitting the latter to do all they can.

C++ uses type maps that it calls virtual function tables which contain displacements and function addresses. These tables could easily contain data addresses and addresses of other type maps, but C++ does not support this. Thus C++ does not provide complete control of its computer model. This assertion also applies to other languages.

We discuss such issues in more detail in sections 3.4.9 and 3.4.10 after our analysis of type maps.

In order to design R-CODE, a comprehensive analysis of the possibilities of type maps is required. Since none is available, I have produced my own, the current version of which is reported in the following subsections.

3.4.3 Type Matching

The *type matching operation* inputs an actual type and a formal type and outputs a type map.[5]

There are two kinds of type matching: *static type matching*, for which enough is known at compile time to permit the compiler to ensure that matching will work if there are no compile errors; and *dynamic type matching*, for which the actual type is sufficiently unknown at compile time that the match might fail. There are several ways to do dynamic type matching.

3.4.3.1 Static Type Matching

A language such as HASKELL performs type matching by applying one of the following cases:

1. The compiler knows the actual type. In this case, the compiler computes the type map. Note that the compiler always knows the formal type.

[5] In HASKELL[HF92, H^+92] the actual type is called a "type", the formal type is called a "class", and the type map is built rather explicitly by an instance declaration.

2. The compiler does not know the actual type, but does know a formal type X, and knows that formal type X inherits from the formal type Y, for which we desire to obtain the type map. The compiler also knows the location of an "X type map" resulting from matching the formal type X with the actual type (see Figure 3.7). And furthermore, when the X type map was created, a pointer to the desired "Y type map" matching Y to the actual type was stored in in the X type map, along with pointers to type maps for any other formal type from which X inherited. So the compiler merely outputs code to read the pointer to the Y type map from the X type map.

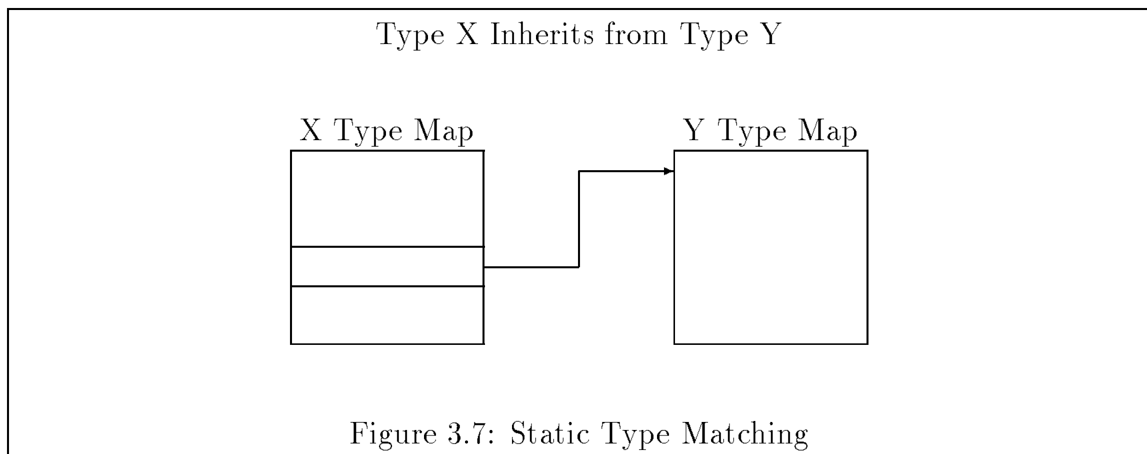


Figure 3.7: Static Type Matching

3.4.3.2 Dynamic Type Matching in R-CODE

Communication systems and general databases transmit and hold data of arbitrary type. When extracting data from such subsystems, code is compiled that does not know the actual type of the datum until run time, and needs to test whether the actual type can be matched to various formal types.

There are several ways to do this dynamic type matching. The following system is proposed for R-CODE.

In R-CODE both actual and formal type are represented by type maps, called the *actual type map* and the *formal type map*. The R-CODE *type matching operation*

produces a *match type map* from an actual and a formal type map. This operation is memoizing: it remembers its former results and replays them. When it has no former result, it traps to software that can be supplied by the compiler or user.

R-CODE supplies default trap software for performing matches that have not been done before (see Figure 3.8). This software expects formal type maps to have *formal component descriptors*, that give information about corresponding match type map descriptors. Each component descriptor in an actual or formal type map has an associated 32-bit *component-ID*. If the i'th component descriptor in the formal type map has component ID x, the actual type map component descriptor with component ID x is found, and this becomes the i'th component descriptor in the match type map.

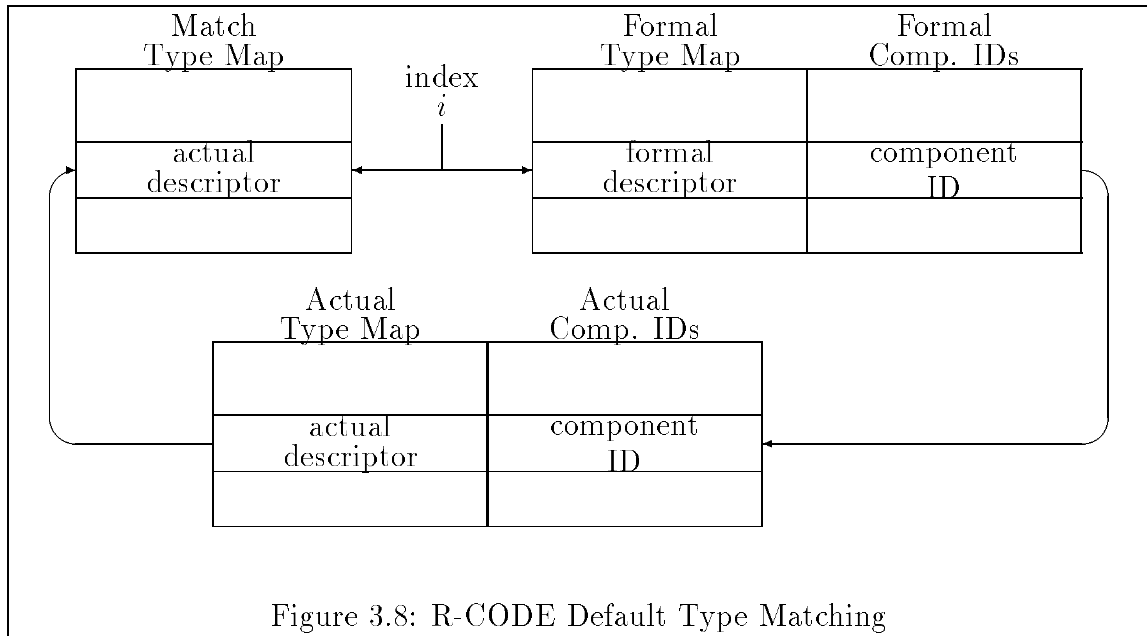


Figure 3.8: R-CODE Default Type Matching

An error occurs if there is no actual component descriptor with ID x, or if the actual descriptor with ID x violates any specification that is encoded in the formal component descriptor of ID x (see "Formal Component Descriptors" below, page 93).

As mentioned in the section on "Loads and Stores" above (page 82), the type matching operation can be optimized away when its match type map result is used by a dynamic case statement, by having the case statement switch on the actual type input to the type matching operation, instead of on the match type output. Thus although the

type matching operation may need a hash table to look up memoized results, the hash table will not be used during normal execution, but only when compiling a new dynamic case.

3.4.3.3 Dynamic Type Matching via Formal Types

Suppose type X inherits from type Y, and we have some actual type T whose objects may contain an X object with its Y subobject, or may contain just a Y object with no X object (see Figure 3.9). The result of matching T with a formal type Y is some "Y type map", and the result of matching T with type X is some "X type map" or an error. If objects of type T contain an X object with its Y subobject, then we could put a pointer to the X type map in some particular element of the Y type map, and easily obtain a pointer to the X type map from a pointer to the Y type map. But if objects of type T just contain a Y object outside any X object, then there is no X type map, and we can put a "null" in the Y type map element were the pointer to the X type map should be, to indicate that the actual type does not match with X.

However, it may be that objects of type Y are not at displacement zero when included in objects of type X, so in order to pass from objects of type Y to objects of type X, one must also subtract the displacement of Y objects in X objects from the address of the Y object. So in addition this displacement needs to be stored in all type maps for formal type Y.

Thus when we derive a type X from a type Y, we add to all Y type maps two constants: first the address of an X type map or null, and second the displacement of Y objects in X objects. In R-CODE it turns out that these two pieces of information can be combined in a single component descriptor that makes the X objects look like components of some Y objects. If a Y object is in fact in an X object, this X component is present; otherwise it is "null".

A disadvantage of this approach occurs if there are very many types X that inherit from one type Y. Then the number different formal type maps corresponding to Y goes up as the number of types X inheriting from Y, and the number of constants in each of these type maps goes up as the number of types X inheriting from Y, and so the total number of constants in type maps goes up as the square of the number of types X inheriting from Y.

An alternative approach just stores the identifier of the actual type, from which a type map was derived, in the type map, permitting matches to other formal types

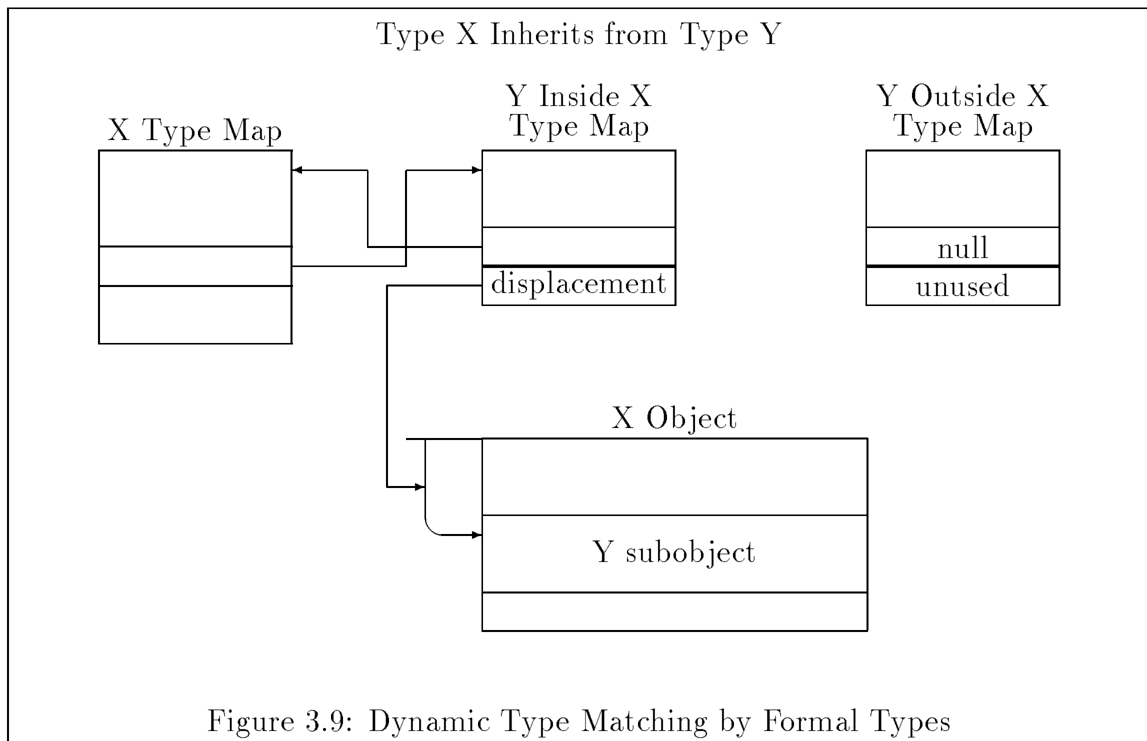


Figure 3.9: Dynamic Type Matching by Formal Types

using this identifier. However these matches would presumably have to be done using a hash table.

Suppose all components of Y are accessed by obtaining their displacement from the appropriate Y type map. Then it is possible to insist that the displacement of Y in X always be zero, and any actual non-zero displacement be compensated for by adding the true displacement of Y in X to the displacements of the Y components in the "Y inside X type map" (see Figure 3.9). This would permit the displacement of Y in X to be omitted from this type map, as it could be assumed to be zero. It is only necessary to include this displacement if it is needed, as when some Y components are at fixed offsets from the start of Y, or when the start of a contiguous Y is needed to copy Y to another location in memory. It is also necessary to include this displacement in any system (see "Actual Type Specification" below, page 95) in which objects begin with an actual type identifier specifying their position within other larger objects. Usually the displacement will be needed for one of these reasons.

3.4.3.4 The Formal Type ANY

Conceptually it is possible to have a formal type "*ANY*" from which all other types inherit. As suggested in the last section, a match type map for some actual type and ANY would contain pointers to all type maps resulting from matching the actual type. Such a match type map is conceptually equivalent to an actual type identifier, and can be used to implement the dynamic match operation by simply reading a vector element.

"ANY" match type maps would be very sparse vectors, and would take a lot of memory unless they were stored as dispatch tables, instead of as type map vectors. But such dispatch tables are similar to and less flexible than the R-CODE type matching memoizer.

The R-CODE default type matching algorithm specified above is conceptually at least as general as "ANY"-based type matching. If each formal component descriptor had its own unique component ID, then we could add to each actual type that had a corresponding component the correct actual component descriptor with that component ID.

Whether this is economical depends upon how types are actually used. The R-CODE model tends to view actual and formal types as subsets of the universe of all possible kinds of components, where a particular kind of component, say the "weight of a body", is shared by many different types. If this view is close to the user's, the R-CODE model will be economical.

3.4.3.5 Formal Component Descriptors

R-CODE represents formal types by formal type maps containing formal component descriptors. Formal component descriptors contain information about corresponding match type map component descriptors. Formal type maps and formal component descriptors are always known at compile time.

For example, a formal component descriptor may specify the corresponding match type map component descriptor completely, permitting code using this match descriptor to compile into instructions that contain all information from the descriptor. This is just what is needed to compile the C programming language.

Or the formal component descriptor may specify everything but the component displacement, permitting the instructions to contain all information but the displacement, which they get from the match type map at run time.

If a formal component descriptor leaves too much unspecified, efficient component load instructions cannot be compiled in the normal way. Instead, each load instruction is compiled as a case statement switching on the identifier of the match type map. Whenever a new match type map that the instruction has never seen before is encounterred, another case of the instruction is dynamically compiled.

Similar *dynamic case statement* methods are used for store instructions and for load-address instructions used to locate components. It may be possible to batch such "cased" instructions together when they are dependent on the same match type map, and produce a single case statement for the entire batch, thus reducing case branch overhead.

When the type map components are binary functions implementing operations such as "max", these operations may be made into dynamic case statements switching on the type map, with the functions pointed at by the type map being dynamically compiled inline.

Dynamic case statements and batching may be used for such operations just as for load and store instructions.

Thus formal component descriptors in R-CODE can specify enough to compile efficient code directly as in a C compiler, or leave enough open to require dynamic case statements for loads, stores, and other operations.

3.4.4 Constancy of Type Maps

R-CODE assumes that type maps do not change during the execution of a program, so that code can be dynamically compiled using information in type maps. Thus type maps are like inlineable code; changing them forces recompilation of all code that inlined information from the type maps.

Similarly R-CODE assumes the match operation does not change, and can be memoized.

3.4.4.1 Adding Component Descriptors to Type Maps

In the section "Dynamic Type Matching via Formal Types" above (page 91) we noted that deriving a new formal type X from an existing formal type Y might add component descriptors to the existing Y type maps. Adding component descriptors to the end of existing type maps does not compromise the constancy of type maps. It also permits new formal operations to be defined for existing formal types when

new code is loaded, which can be useful.

3.4.4.2 Type Variables

An object type (e.g. C++ "class"), may have variables associated with it, as well as constants.

There are two reasons not to store such variables in a type map.

1. The same formal object type may require many match type maps to accommodate different object layouts, i.e. actual types, and these should all share the same type variables. So each match type map should have a pointer to the shared type variables.

2. It is important to know that type map information is constant, so compilation, initial or dynamic, can trust the information not to change (any more often than code changes).

3.4.5 Actual Type Specification

An actual type "identifier" can be put into three places:

1. In machine instructions.

2. In the data.

3. In a pointer to the data.

These options are roughly in order of increasing flexibility.

An actual type identifier may be the address of a type map or a numeric code. When actual type identifiers are stored in objects, there are reasons for using numeric codes instead of addresses:

1. The data is to be copied directly between RAM memory and an external medium.

2. The data is in RAM memory shared by several different programs that are linked separately and may need different versions of type maps. For example, a particular function may be at different addresses in the code space of different links, so that function addresses stored in type maps are determined by the type and the program.

3. The data needs to be compact, and cannot afford the space of a full type map address.

Reasons 2 and 3 are also valid when actual type identifiers are stored in pointers.

R-CODE uses *type map numbers* to identify type maps, and uses type maps to identify actual types. R-CODE can store type map numbers in pointers or in data. Numeric codes are used for all three reasons above, and also because the R-CODE memory management system indirects all addresses anyway to permit objects to be moved at any time, so the indirection needed to look up a numeric code in a table would be done by R-CODE in any case.

3.4.5.1 An Actual Type Specification Example

Reason 2 arises in large real-time systems, and we will sketch an example, because most language designers are unfamiliar with such systems. Our example large real-time system has a few hundred thousand lines of code loaded at one time into the same symmetric multiprocessor shared memory computer. Communication is by message passing between separate programs running at different priority levels. This is because no one has figured out how to efficiently debug large real-time systems without breaking them up into modest sized programs that take in a single stream of messages and send out messages to several streams. Also, it is uneconomical to try to link too much code together, so each modest sized program is linked separately.

In spite of the fact that communication is by message passing, the messages are stored in shared memory, and only pointers are actually passed (the messages become read-only once sent). This is because systems that actually copy messages, even just memory to memory, use so much computer time doing the copying, and cause such worries for programmers trying to optimize message size, that they are not competitive.

So far we have described systems that actually exist. Now let us hypothesize an object oriented approach to organizing the code.[6] Most code will operate on objects that are packed into messages. When a message arrives in the input of a program, the program control code figures out what to do, constructs output messages, and then passes pieces of input and output messages, expressed as objects, to a subroutine library to perform computations.

The objects will be denoted by addresses of actual type identifiers stored in messages. Given the address of an actual type identifier, a subroutine can locate the type maps it needs by type matching operations, and these tell how to access everything else in the object, as the object is stored in that particular message, starting from the address of the actual type identifier. The same formal type of object may be packed in different ways into different messages by varying the actual type identifier and type maps.

In this hypothesized paradigm for developing programs, separate programmers called system integrators establish which objects are packed into which messages, and write the program control code. The mission of these integrators is to get each computation done at the right priority level and within the proper latency. To do this they move information and computation around in ways that would upset a mathematical programmer who just looked at the computations that needed to be done. The paradigm will work if the object subroutine libraries give the system integrators enough room to maneuver.

Note that what we have just described is not currently done because support for type maps is too weak in any existing or proposed language we know of. Instead the details of the structure of each message are written into the algorithm code for processing that message. This limits code reusability, and makes it harder to move computations around, making system integration more difficult. It is not automatically clear, however, whether the system we have just proposed would improve reusability and system integratability enough to be economical.

The system we have just proposed stores type identifiers in the data and passes pointers to these identifiers to subroutines. But this system is not as flexible as one that stores type identifiers in the pointers, and not in the data, as we shall see.

[6]This approach was suggested to the author by a comment of Dr. Richard M. Elkin of Raytheon Corporation.

Consider the following problem with the system proposed above that stores type identifiers in the data. The system integrators find that an existing message needs to be viewed as containing several new objects, which it did not previously need to appear to contain, but whose components are already in the message. This means new actual type identifiers need to be added to the message, but nothing else. In a real-time system this might not be a problem, but if we add long term data storage of "messages" to the system, then we may find it very inconvenient to change "messages" just to view them in a new way.

Therefore we propose a different system in which each message begins with an actual type identifier that is the only actual type identifier in the message. Matching this to an application specific formal type obtains a type map that contains for each object in the message a subobject component descriptor telling the actual type and relative location of that object. From these descriptors, pointers to objects are obtained that contain the actual types of the objects. These pointers with their contained actual types are passed to subroutines that compute with objects.

This hypothetical example indicates some of the possibilities of type maps, and some of the variations that can be played on the type map theme. R-CODE encodes actual types as type map numbers in either 64- or 128-bit tagged pointers. R-CODE can also encode actual types as type map numbers stored in objects, and use reduced pointers (page 81) to point at these type map numbers. Reduced pointers have only an object number and a displacement, and must be combined with the type map number they point at to make a full 128-bit tagged R-CODE pointer.

3.4.6 Component Descriptor Contents

A component descriptor might include some of the following information (see the above sections, particularly sections 3.3.1 and 3.3.3, for definitions of terms):

1. The displacement within the object of the memory unit containing the component and the displacement of the component within this memory unit, if the component is contained in a memory unit;

   or

   the displacement within the object of the component, if the component is <u>not</u> contained in a memory unit.

2. The size of the memory unit containing the component, if the component is contained in a memory unit;

or

the alignment of the component in memory, if the component is <u>not</u> contained in a memory unit.

3. The size of the component, if the component is contained in a memory unit;

   or

   if the component is <u>not</u> contained in a memory unit, the component will be a subobject, and its size will be stored in one of its components as determined by its type map (the size component may be a constant stored in the type map or a value stored in the subobject).

4. An identifier for the copy type of the component. Some standard copy types are integer, floating point, pointer, contiguous subobject, and discontiguous subobject (see page 81).

5. An identifier for the actual type of the component, for use in type matching;

   or

   an identifier for a match type map that is the result of matching the actual type with some formal type.

6. An access specification indicating whether the component is read-only, write-only, write-once, read-write, etc.

7. Identifiers of routines to be called to perform the load, store, and load-address operations on the component, if the component is not to be accessed by standard memory load, store, and load-address instructions.

8. The value of the component, if the component is a constant determined only by the type map. (Of course, in this case much of the above information is unnecessary).

Some of this information may be used by existing computers without too much inefficiency and without being compiled into instructions. Memory unit displacements fall in this category. Other information must be compiled into machine instructions on current computers to be efficient.

3.4.7 R-CODE Component Descriptors

R-CODE component descriptors are 128-bit tagged values. Type maps and component descriptors in R-CODE are like code, and are virtual in the same sense: their apparent value is not the same as their implementation.

R-CODE has two kinds of component descriptors: actual and formal. The actual component descriptors appear in actual and match type maps, while the formal component descriptors appear in formal type maps.

3.4.7.1 R-CODE Actual Component Descriptors

R-CODE has two kinds of actual component descriptors: constant and non-constant.

Constant component descriptors are just constants which are the "values" of the components described. The components are therefore constants dependent only on the type, or more specifically, on the type map. Function pointers are typically constant descriptors. A contiguous object size that is not stored in its object is a constant descriptor.

Any R-CODE 128-bit tagged value may be used as a constant component descriptor, except for non-constant component descriptor values.

Non-constant component descriptors have almost the same format as R-CODE 128-bit tagged pointers. Their format, given in Figure 3.10, is just like the 128-bit pointer except the object number in the pointer is replaced by a lock number, the displacement is signed in the component descriptor and unsigned in the pointer, and the tag is $FFFF14_{16}$ for the non-constant component descriptor and $FFFF0C_{16}$ for the pointer. Lock numbers are only used for components involved in atomic transactions: see section 5.6.10 of Chapter 5.

An R-CODE component load instruction specifies a pointer that points at an object or at a subobject inside an object, and also specifies a *component index*, that is the index of a type map element. The pointer must contain a mapped type (see Figure 3.4, page 80) that has a type map number identifying a type map. The load instruction component index is the index of an element in this type map vector, and this element is the component descriptor that will be used to access the component.

If this component descriptor is a constant component descriptor, the value of the component descriptor is the value of the component loaded by the load instruction.

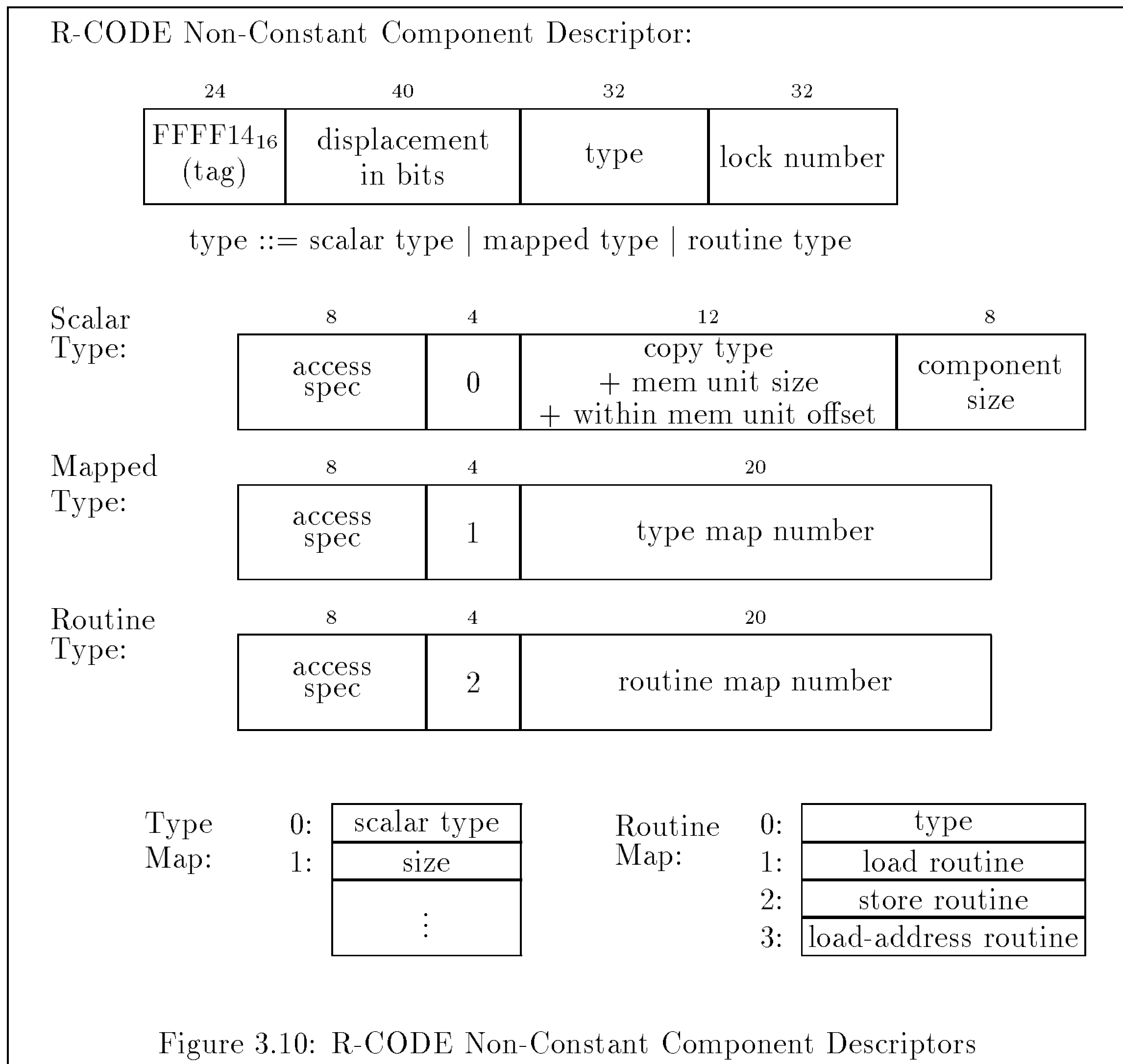


Figure 3.10: R-CODE Non-Constant Component Descriptors

If the component descriptor is not a constant component descriptor, a pointer to the component is formed by taking the type from the component descriptor, the object number from the object pointer, and the sum of the displacements from the component descriptor and the object pointer. This pointer is then used as input to the load instruction in order to load the component as described above in the section on "Loads and Stores" (page 82).

An R-CODE component store instruction computes a pointer in the same way as a component load instruction. Constant component descriptors cannot, of course, be used with store instructions.

3.4.7.2 R-CODE Formal Component Descriptors

R-CODE formal component descriptors have almost the same format as R-CODE actual component descriptors, but have some special values for the various fields, and have some extra fields. Any actual component descriptor can be used as a formal component descriptor, without any special field values or extra fields. When special values or extra fields are present, the tag in the descriptor is changed from $FFFF14_{16}$ to $FFFF18_{16}$ to indicate their presence. The extra fields are stored where the lock number is stored in an actual component descriptor; the lock number is not needed in a formal component descriptor.

Formal component descriptors are matched against actual descriptors to determine whether the actual descriptor is legal in some situation. If the formal component descriptor has no special values in any of its fields, and no extra fields, the actual descriptor must equal the formal descriptor exactly.

The displacement, component size, copy type, and access specification fields of a formal component descriptor can be given the special *UNKNOWN* value, which indicates the corresponding field of the actual component descriptor can be anything.

The displacement field of a formal descriptor can be given one of several special *ALIGNED* values, which constrain the displacement in the actual component descriptor to be a multiple of some power of two from 2 through 128, without otherwise constraining the actual displacement.

One of the special fields is the *match flag*. If this is set, any type map number supplied by the actual component descriptor must denote a match type map obtained by matching some actual type map number with the formal type map number supplied

by the formal component descriptor.

Another special field is the *load convert* field, which can be set to one of the values *NO-CONVERT*, *STATIC-CONVERT*, or *DYNAMIC-CONVERT*. If this field has the NO-CONVERT value, and reading the component is permitted by its access type, then any specified formal copy type and component size must match the actual component descriptor copy type and component size. If the value is STATIC-CONVERT, it must always be possible to convert a value with the actual component copy type and size to a value with the formal component copy type and size. No range error can occur during this conversion. But if the load convert value is DYNAMIC-CONVERT, then it must be possible to perform this conversion for many likely values of the actual type, but other values may lead to a range or overflow error during runtime.

So, for example, an 8-bit unsigned actual integer is statically convertible to a 16-bit signed formal integer, but only dynamically convertible to an 8-bit signed integer. In the latter case, values from 128 on will cause overflow errors.

Similarly there is another special *store convert* field with the same set of values as the load convert field that controls conversion of values of the formal scalar type to values of the actual scalar type in the same manner that the load convert field controls conversion in the opposite direction.

3.4.8 Subobjects

A subobject is part of an object. It can be the whole object, or just a subset.

A subobject has a *subobject address* that consists of the object number of the containing object and a displacement within that object.

A "*contiguous subobject*" is a contiguous set of disjoint memory units, the first of which is at the subobject address. A contiguous subobject has a size and an alignment. The subobject may be copied from one memory location to another by copying it as a bit string, respecting the require alignment. It may be copied into or from a sequence of registers by copying each subobject memory unit to or from a separate register.

All memory units in a subobject must be aligned relative to the beginning of the subobject. The alignment of a subobject is the size of its largest memory unit. A contiguous subobject can be copied to a computer with a different endianhood, and all that need be done on receipt is to reformat (e.g. byte reverse) each memory unit individually.

The size in bits of a contiguous subobject is stored as the second component of the

subobject, as described by the second element of a type map (the first component is the scalar type). The size in bytes or words may be stored in the subobject itself by giving this second component a type such as "unsigned8" or "unsigned32"(page 81).

A "*discontiguous subobject* is a set of parts of memory units, and has an alignment equal to the size of the largest of these memory units. All the memory units must be aligned properly relative to the address of the discontiguous subobject. A discontiguous subobject has no size.

A discontiguous subobject cannot be copied at all, and therefore cannot be passed as an argument or returned by copying. However, addresses pointing at the discontiguous subobject can be passed.

A 128-bit tagged pointer that points at a subobject contains the address of the subobject and a mapped type giving the type map to be used in accessing the subobject. The scalar type of the subobject has the "contiguous subobject" or "discontiguous subobject" copy type, and stores the alignment of the subobject as the component size field of the scalar type.

In addition to loading and storing components of subobjects, it is possible to load a contiguous subobject into registers or produce a copy of the subobject in the frame heap, and similarly a contiguous subobject may be stored from registers or from a frame heap copy of the subobject.

3.4.9 Examples of Deficiencies in C++

Note that C++ uses the word "*class*" where we use the word "type".

C++ has type maps called *virtual function tables* that hold virtual function addresses and displacements used to adjust the current object pointer when a virtual function is called [Str94, pp. 264–5].

A C++ type map also holds a pointer to a `typeinfo` object that holds information about the actual type of the largest object containing a given C++ object[Str94]. The type map also holds the displacement of the given C++ object in its largest containing object.

A C++ object may contain subobjects called *virtual bases* that can be at different displacements relative to the address of the object depending on the actual type of the largest object containing the addressed object. But C++ type maps do not contain

the displacements of virtual bases; rather, instead of having the object point at a type map and the type map have the displacement of the virtual base, C++ has the object point at the virtual base directly. This system is faster by one indirection, but has other overhead problems in terms of both amount of memory taken by an object and time required to adjust pointers when an object is copied. Otherwise the two systems are equivalent [Str94, p. 266].

Type maps permit any component to have a displacement determined by a type map for the containing object. C++ permits this only for virtual bases, which are like unnamed components. But a C++ object may not contain two different virtual bases with the same type. Because C++ derived its notion of type maps merely to support its notion of inheritance, C++ has an incomplete implementation of the notions of virtual object or type map.

C++ does not permit arbitrary constants in type maps: e.g. one cannot have what C++ might call a "virtual static datum", which would require a pointer to the datum in the type map. This makes it difficult to have variables associated with a class used as an abstract type (i.e. a formal type). For example, when coding a LISP interpreter in C++, one might like to associated a different free list with each derivative of the abstract type "LISP object", but this is not directly possible in C++.

Because C++ does not explicitly put addresses of type maps in other type maps, C++ cannot perform many base class to derived class strongly typed conversions as efficiently as it might.

Another limitation of C++ is that virtual function table addresses cannot be used as copyable data type tags for tagged data. If a class Y inherits from a class X, X has data and virtual functions, and Y has virtual function definitions but no new data, copying a Y value into an X variable gives a copied value that points at the virtual function table of X, not of Y. That is, the pointer to the virtual function table of Y that was stored in the Y value is not copied; it is overwritten after the copy by a pointer to the virtual function table of X.

C++ puts virtual function table addresses in objects, and does not permit virtual function table addresses to be made part of pointers or otherwise passed separately. As a consequence, many things are impossible with C++ that are possible with HASKELL. For example, in C++ one cannot have an array of 16 bit scalars which have operations determined by a virtual function table, without storing the address of the table in every array element.

C++ denotes type identifiers stored in objects by pointers instead of by integer codes. This makes it difficult to copy objects to and from external memory, or to place objects in memory shared by separately linked programs.

In summary, the notions of virtual object and type map are not really part of the C++ language, and the C++ implementation of type maps is very incomplete.

3.4.10 Type Maps in Other Languages

HASKELL uses type maps passed to functions as separate arguments. These type maps contain only function addresses and addresses of other type maps[Jon94, section 7.5.1]. They do not contain component displacements or addresses of static data. The use of type map addresses inside type maps is limited to static type matching, and is not used for dynamic type matching.

Conceptually EIFFEL uses a matrix whose columns are type maps, one for every actual type[MS94, page 7]. The actual type number is placed in the object. The matrix is very sparse, and is packed by a mechanism that requires an extra vector access. The component descriptors are either component displacements or function addresses. There is no type matching operation: every object can be viewed in only one way.

SATHER 0.5 uses hash tables. SATHER 0.6 uses type maps[MS94] that contain function addresses, component displacements, static data addresses, and simple constants. Objects contain type map numbers (or type map addresses) that designate the type maps needed to consider the objects as having different actual or formal types. An object address points to a vector of these type map numbers, and the type map is selected according to which view of the object is needed.

Neither EIFFEL or SATHER make provision for passing type maps as parameters or using type maps with numeric values, as in our lattice element example on page 85.

None of the languages we have discussed include type information in component descriptors.

None of the languages we have discussed make provision for placing type map addresses inside type maps for the purpose of dynamic type matching. Because doing this automatically might take memory proportional to the square of the number of types derived from a given type, a language should have features for specifying just which dynamic type matches are to be done this way.

3.5 Array Descriptors

An *array descriptor* describes a linear map from a list of subscripts to a displacement. This displacement is then added to a pointer to get a pointer to an element in an array. Or the displacement may be added to a component descriptor to get a descriptor for an array element within an object.

The following is an example use of array descriptors from my past.

> A system for running algorithms on medium resolution images is being built. Many image operators, such as convolution operators, shrink the image, so the output is smaller than the input. But the size of the image, at medium resolution, is not big enough to withstand this shrinkage. So some method of avoiding shrinkage is required.
>
> Algorithmically, the method is to expand the input by mirror imaging, so the output will be the same size as the original input. To effect this, an input matrix of the right size is created, and sliced up into parts corresponding to the original input and various mirror images of that original input.
>
> The software being used to handle arrays in this system treats array descriptors as first class objects. With the help of array descriptors, the parts of the expanded input matrix are presented as separate arrays to a standard array copy routine that simply copies one array to another. This array copy routine, by the way, is not so simple as to invite replicating its code: it optimally handles arrays of any of 9 element types.

An R-CODE *array descriptor* is a vector of R-CODE dimension descriptors followed by either a component descriptor or a 128-bit tagged pointer: see Figure 3.11. An R-CODE array descriptor can be part of a type map, can be passed as part of an argument list or a routine result list, and can be stored inside an object. R-CODE assumes that an array descriptor will not be changed during use.

An R-CODE *dimension descriptor* is a 128-bit tagged value with the format given in Figure 3.11. Given an integer subscript I and the dimension descriptor pictured in the figure, the check

$$base \le I < limit$$

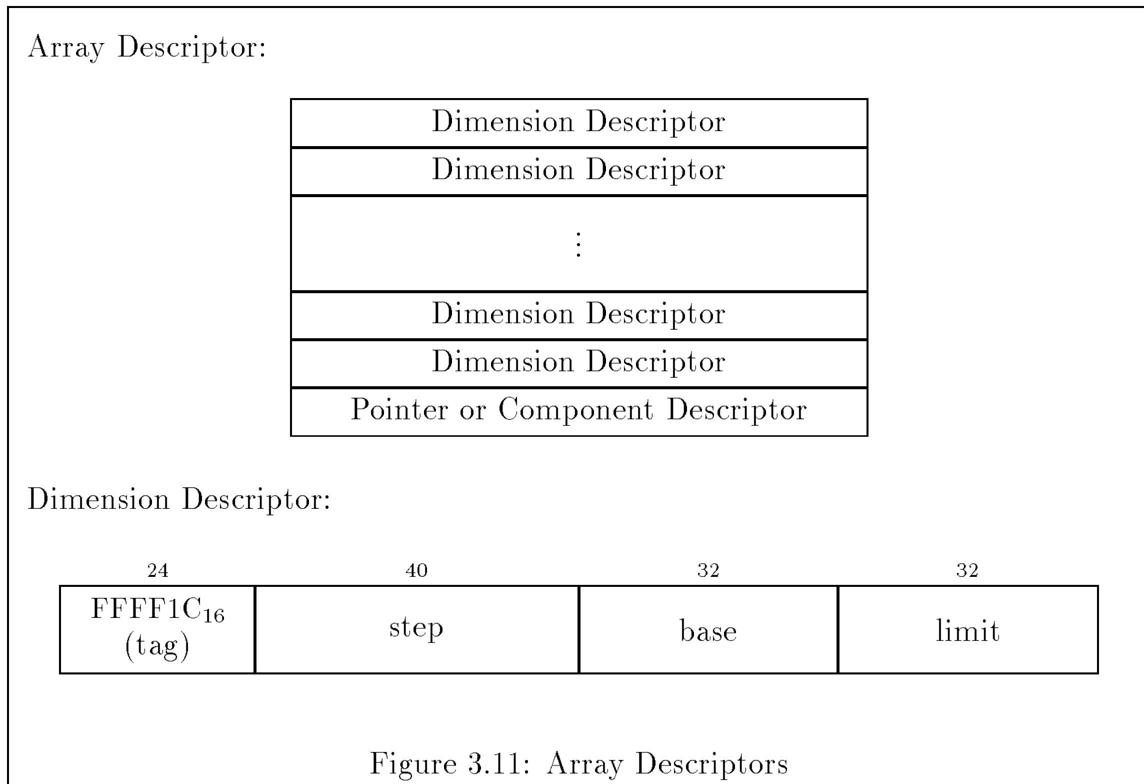


Figure 3.11: Array Descriptors

is made to see if the subscript is legal, and if yes, the displacement

$$I \times step$$

is generated and added to the displacement of the base element of the array. This latter displacement is provided by the component descriptor or pointer that ends the array descriptor.

Here I, *base*, and *limit* are all 32-bit signed integers, and *step* is a 40 bit signed integer displacement in units of one bit. Just as for component descriptors and 128-bit tagged pointers, R-CODE implementations do not store all 40 bits of the step: they store only 32 bits in units of one byte or one bit depending on the type field in the component descriptor or pointer that ends the array descriptor.

3.5.1 Summary

R-CODE enhances the data types of machine hardware with several types that promote interoperability of hardware and software. Tagged types are included for languages like LISP. Type maps are included to allow software and data written by different people and at different times to be mixed, and to allow one piece of code to run on data of more than one actual type. Array descriptors make it easier to address arrays in a strongly typed manner.

My hope is that this set of data types will advance the cause of languages that mix the features of LISP and C, and the causes of object oriented programming and array processing.

I also think it is clear that structures such as tagged data, type maps, and array descriptors need to be standardized if programming languages are to interoperate.

Chapter 4

Execution Flow

4.1 Goals

As an intermediate programming language, the R-CODE virtual machine language defines an instruction set and an execution flow semantics for that instruction set. Previous efforts to define intermediate representations for programs have generally used a sequential execution flow model. But R-CODE instead pursues a different goal:

> $\mathbf{G}_\rho$: Execution flow semantics should permit instructions to be reordered by implementations as much as practical.

The primary reason for this goal is efficiency. One consideration is the appearance of superscalar processors that execute several instructions at once (see $\mathbf{H}_1$, page 13). Another consideration is the fact that processors are becoming much faster than main memory, relatively speaking, so there is an increasing need to move memory load instructions as early in the instruction stream as possible (see the "memory wall", page 14).

A secondary reason this goal is acceptable is the desire to switch to more functional programming languages, because these languages are easier to understand and debug.

4.2 Requirements

Functional languages derive extra efficiency from the fact that they can be executed in dataflow order: each primitive operation (e.g. add) can execute whenever its inputs are ready. To maximize the parallelism this permits, special hardware is needed to detect when the inputs to an operation are ready (see, for example, the work of Burton Smith[Smi78, ACC+90] and Arvind[Nik91, Pap90]).

But R-CODE must run efficiently on existing sequential computers.

Therefore R-CODE uses a compromise. It is possible to compile a functional language efficiently for a sequential computer as long as all the variables are register-like, in that they are local to a routine, cannot be aliased at runtime using pointers, and therefore the compiler can figure out at compile time when they will receive their values. But variables that are global or are array elements addressed by general subscripting cannot be handled efficiently using dataflow execution order on existing computers. So the compromise is:

> $\mathbf{F}_7$: R-CODE will be a functional dataflow language with register-like variables that treats RAM memory as an I/O device.

$\mathbf{F}_7$ leads to languages that are functional with respect to register-like variables, but whose memory loads and stores have no certain order. To manipulate memory properly, a *barrier operation* is introduced, as in some modern functional languages (e.g. the ID "`-----`" barrier[Nik91]), to force completion of part of a block of code before the rest of the block is executed. We call such a language a "*register dataflow language.*"

The operations of a functional language that involve registers can be mapped directly to a register dataflow language. The operations that create values in memory require the following algorithm:

1. Allocate an object and return a write-only pointer to it.
2. Write the components of the write-only object. These component write operations may be done in any order: no component is written twice.
3. Execute a barrier that ensures all of the above has finished.

4. Convert the write-only pointer to a read-only pointer and discard the write-only pointer.

5. Return the read-only pointer for use by the rest of the program.

Functional languages can handle most programming tasks easily and efficiently. But, on the other hand, they usually cannot express everything in a single program. They cannot handle objects and arrays with changing state. They have trouble with algorithms that require state maintenance, such as histogram computation or graph tracing.

The *barrier operation* permits a register dataflow language to handle the necessary non-functional language operations. Memory operations between barriers can be reordered by the hardware, but operations on different sides of a barrier and dynamically within the same block cannot cross the barrier, and this introduces enough sequentiality to maintain state.

In addition to barriers, *atomic operations* are helpful in parallelizing algorithms that demand some sequentiality. For example, an atomic test-and-set bit operation can be used in a parallel graph traversal to ensure no node is visited more than once.

As a more specific example, in computing a histogram one may use barriers and atomic add operations as follows:

1. Allocate the histogram and return a write-only pointer to it.

2. Zero the components of the write-only histogram. These component zero operations may be done in any order: no component is written twice.

3. Execute a barrier that ensures all of the above has finished.

4. Convert the write-only pointer to an “accumulate-only” pointer and discard the write-only pointer.

5. Accumulate into the components of the accumulate-only histogram. Each accumulate operation is atomic, and the accumulate operations are pairwise commutative, so these operations may be reordered without affecting the result, even though many of them change the same component.

6. Execute a barrier that ensures all of the above has finished.

7. Convert the accumulate-only pointer to a read-only pointer, and discard the accumulate-only pointer.

8. Return the read-only pointer for use by the rest of the program.

The facts that pure functional languages can be translated easily into register dataflow languages and that pure functional languages can handle most coding tasks mean that users will <u>not</u> have to write explicit barriers very often. This means register dataflow languages may be practical.

A register dataflow language has several operations that have side-effects. Memory write operations are the most obvious of these. However, the subroutine return operation also has side effects when it is inside a case statement. This is because the case statement is selecting one of several possible returns from the routine, each with a different set of results. The side effect is the setting of the result list, which is analogous to the writing of a variable.

Because there are side effecting operations in the language, control flow operations must be sensitive to this. A case statement, for example, cannot evaluate memory write or subroutine return statements inside any case until it is certain that case has been selected.

Because of the presence of memory write operations, subroutine returns must be treated differently than they would be in a purely functional language. In a purely functional language, once all the values are returned from a routine, the routine can terminate. So a return statement can terminate its routine. But in a register dataflow language, the return statement may not generally terminate other operations in its routine because they may lead to memory writes.

The operation that throws an exception is roughly similar to a return operation, but it <u>does</u> terminate all operations in routines being returned from.

The *barrier operation* holds up all operations in its block that follow the barrier until all operations in the block before the barrier are done. Suppose a block executes a return operation before the barrier. It seems most natural in this case to <u>not</u> execute the part of the block after the barrier, on the grounds that the block has already returned. Code before the barrier should execute even after the return operation, as indicated above.

Memory read operations must not have side effects if we are to permit their speculative execution inside case statement cases that have not yet been selected.

Routine calls may be speculative. An inlined routine may appear to execute speculatively even on a serial computer, because its instructions may become interleaved with those of its caller, and some of these instructions may be executed speculatively.

Routine calls may execute in parallel. This will happen, even on a serial computer, when two routines are inlined and their instructions mixed.

So a register dataflow language is a dataflow language in which memory operations are input/output operations, there is a within block barrier operation, and other operations are modified accordingly.

The R-CODE computer is a virtual computer that executes a register dataflow machine language. The name "R-CODE" stands for "*Register dataflow CODE*".

The R-CODE machine language is built on certain principals that are introduced in the next section. Subsequent sections discuss related work, R-CODE machine language semantics, and implementation challenges.

4.3 Principals

The R-CODE virtual computer language register dataflow semantics are based on the following principals, which will be elaborated as necessary in subsequent sections.

R-CODE is a virtual computer machine language, so the most basic language operations are implemented by an R-CODE instruction. Thus I will use the word "*instruction*" below as a synonym for "*operation*".

In reading the following principals it may help to think of the R-CODE machine language as essentially a RISC language whose typical instruction has two input registers and an output register.

4.3.1 Virtual Computer Representation

R-CODE represents programs in a virtual computer machine language, instead of as a programming language.

Either a virtual computer machine language or a programming language can be used to represent programs. There are several advantages to using a virtual computer machine language.

1. The virtual computer can be simulated and used to explain to students how computers work.

2. The virtual machine language is closer to actual hardware, and this makes precise implementation easier.

3. The virtual machine language is similar enough to actual hardware that changes in actual hardware can be promoted directly by features of the virtual machine language.

For these reasons R-CODE takes the virtual computer approach, instead of the low level programming language approach.

A main danger in the virtual computer approach is that instruction and data *packing problems* will adversely influence the semantics of the result. To combat this, R-CODE uses very unpacked 64-bit virtual instructions and very unpacked 128-bit virtual tagged register values.

4.3.2 Dataflow for Register Values

Values that can be stored in registers are computed using *dataflow* execution order. Once a value is stored in a register, the register cannot be changed thereafter. Each register value may be computed as soon as sufficient inputs to its computing instruction are available. Any execution order which does this is permitted, unless constrained by special instructions such as barriers and cases.

This permits code to be reordered unless prohibited by barriers or cases.

4.3.3 RAM Memory is an Input/Output Device

RAM memory is treated as an input output device. Memory instructions can execute as soon as their register inputs are available. Memory read instructions do <u>not</u> wait until the memory location being read has been written.

This permits memory instructions to be reordered unless prohibited by barriers or cases. By not waiting for "empty" memory words to become "full", need for special hardware is avoided.

4.3.4 Based on Cases, Calls, Returns, Barriers, and Exception Throws

Normal execution flow is controlled by the register dataflow principal and just five other instructions: the case instruction that selects one of several cases to execute, the subroutine call, the subroutine return, the barrier instruction, and the exception throw instruction. Other constructs such as blocks and loops are equivalent to routines (see below). Still other constructs, such as coroutines and continuations, are defined by minor extensions of the basic model.

Limiting the number of ways execution flow may be controlled provides for an economical abstract description of execution flow rules.

Case instructions may select one of several return instructions to return from a subroutine. These return instructions each write different values into the result list, and therefore have a side effect.

4.3.5 Barriers are at Routine Top Level

A barrier instruction may appear at the top level of a routine to cause all instructions before the barrier to be done before any instructions after the barrier is done. The barrier instructions in a routine divide the routine into "partitions".

Barrier instructions may only be at the top level of a routine, and not inside cases or partitions. But they may also be at the top level of blocks or loops, since these are like routines: see below.

I do not know of any reasonable semantics for barriers inside cases, and so forbid such.

4.3.6 Return Terminates Subsequent Partitions

A return instruction in one partition of a routine terminates subsequent partitions of the routine. A return instruction does not terminate other instructions in its own partition, and these other instructions may complete after the return instruction.

Because the order of instructions within a partition may be rearranged, it may be necessary to execute some instructions within the partition after a return instruction execution. But terminating subsequent partitions is appropriate because I believe it is the semantics most users will expect.

4.3.7 Exception Catches are Cases Attached to Call Instructions

A call instruction can optionally have an exception catch case. With respect to this, the call instruction behaves like a case instruction, and selects the exception catch case if and only if the call instruction receives an exception value list from an exception throw.

Exception catch capabilities are normally attached to blocks. But in our scheme, call instructions and separate routines play the role of blocks, so I add catch capability to call instructions.

4.3.8 Exception Throws are like Returns with Termination

An exception throw instruction is like a return instruction, but with the following differences:

1. An exception throw instruction communicates a list of exception values, whereas a return instruction communicates a list of result values.

2. An exception throw instruction terminates all instructions in whichever routines it is completing, whereas a return instruction allows instructions in the same partition to complete after a return instruction executes.

4.3.9 Out-of-Line Equals Inline

Execution of out-of-line routines has the same semantics as putting the routine in-line.

This implies the implementation may start executing a routine before all of its arguments have been computed, because the routine may be inlined, and its instructions may be reordered and mixed with those of its caller. Also, code using a return value may start executing before the routine is completely done. But conversely, the programmer cannot depend on this behavior: see "Either Strict or Non-Strict" below.

The R-CODE debugging interface makes execution of inline and out-of-line routines look the same. However, inline routines often appear to execute in non-strict, or dataflow, order, even on a sequential machine, so this principal implies the ability to deal with non-strict execution order in debugging.

4.3.10 Blocks are Routines

A block provides a way of grouping instructions. Values can be returned from the block with return instructions.

Semantically a block is just like a routine, and it should be possible to convert blocks into routines and vice versa.

4.3.11 Loops are Tail-Recursive Routines

A loop is a block of instructions that repeats itself.

Semantically a loop is just like a tail-recursive routine that calls itself to iterate, and it should be possible to convert loops into tail-recursive routines and vice versa.

4.3.12 Traps are like Subroutine Calls

An instruction trap occurs when an instruction is confronted with operands it cannot handle. Instruction traps are semantically equivalent to subroutine calls, as if the instruction had been turned into a call instruction. The subroutine called is selected from a trap dispatch table, and can "emulate" the instruction by returning the proper results. The subroutine can also throw an exception.

Traps differ from explicit call instructions only in that implementations can optimize them away (see "Eager but for Traps" below).

Traps may be used in languages like LISP to perform arithmetic operations on special numbers, such as very long integers.

4.3.13 NaNs instead of Traps

Many conditionals are written using short-circuit AND instructions for which the first operand must be true in order for it to be legal to attempt to compute the second operand. For example, the first operand may test that a pointer is not NULL, and then the second operand may use the pointer.

A great deal of sequentiality would be introduced if one could not attempt to compute later operands to a sequence of short-circuit AND instructions before the first operand has been computed. To avoid such sequentiality, instructions are defined to produce non-signaling-NaN values when they are given illegal inputs, instead of trapping. Traps still occur if the inputs are such that a software trap routine might compute a

value, or if the instruction is a control flow instruction that involves side effects, such as a case instruction using a value that must not be a non-signaling-NaN to select a case.

If a short circuit AND instruction inputs a first operand that is false and a second operand that is a non-signaling-NaN, the AND instruction simply outputs false. The non-signaling-NaN is discarded, because it is an unneeded operand of a short circuit instruction.

When a non-signaling-NaN is output by an instruction, this value says the instruction could not execute. If a non-signaling-NaN is fed as input to an instruction that controls side effects, it will cause a trap to the debugger, and examination of the output of various previous instructions will reveal the original offending input.

Note that I assume a debugger interface that makes the virtual computer look real. In the virtual computer, each instruction outputs to a register which is not changed after it is initially set by the instruction, so these outputs will be available thenceforth for inspection by the debugger.

4.3.14 Either Strict or Non-Strict

An implementation should be free to optimize code by not computing inputs that are not required by an instruction before executing the instruction. For example, if the first input to a boolean AND instruction is false, the second input is not necessary, and need not be computed.

Inputs that are not required to execute an instruction are called "*non-strict*," so an implementation is free to treat inputs as non-strict.

But an implementation should also be free to insist that all inputs to an instruction be computed before executing the instruction. That is, code should not depend upon unrequired inputs not being computed.

Inputs that must be computed before executing an instruction are called "*strict*", so an implementation is free to treat inputs as strict.

The same holds for arguments to a subroutine, which an implementation is free to treat as either strict or non-strict. In the non-strict case, the subroutine may start to execute before all its arguments have been computed, and in the strict case, the subroutine will not start until all its arguments have been computed.

Note that the freedom to not compute inputs before executing an instruction or

routine is not the same thing as the freedom to never compute inputs at all. See "Eager but for Traps" below.

The freedom to view execution of routines as non-strict is used when the routines are inlined and their instructions are reordered among the other instructions of their caller.

4.3.15 Eager but for Traps

Dataflow languages can be "*eager*", meaning they compute everything as soon as necessary inputs are ready, or "*lazy*", meaning they compute something only when it is needed. Because memory write instructions have side effects, it is easiest to use eager semantics.

Thus everything is computed sometime, unless it is in a disabled case of a case instruction or it follows a barrier that is preceded by a return instruction execution.

As an optimization, an implementation can still suppress a computation if it can be shown to be unneeded by any side effect producing instruction. In order to make this useful, an implementation is permitted to suppress a computation whose only side effect might occur because of an instruction trap. However, an implementation is not required to suppress such computations.

In effect, the implementation is permitted to assume for the purposes of suppressing instruction execution, that if an instruction does not have side effects, then its trap routine will not have side effects.

4.3.16 Memory Reads have No Side Effects

A memory read instruction may not have a side effect. It may be done zero or more times without affecting other instructions.

Except, of course, that the time when a read instruction is done does affect the value read.

4.3.17 Case Statements Propagate Permissions

The computer schedules instructions for possible execution. The scheduled instructions form an *execution tree*, whose nodes include routine calls, case instructions, individual cases, and ordinary instructions (see Figure 4.3, page 134). In the following discussion I ignore blocks, loops, and barriers.

Each node of the tree may have one of at least four states:

no permission
side-effect-free permission
side-effect permission
withdraw permission

These states are ordered, and nodes can only change from an earlier state in the list to a later state in the list.

A case instruction selects one of several cases to execute, where each case is itself a sequence of instructions. The case instruction works by propagating permissions to its cases. It can initially propagate permission to execute side-effect-free instructions to any case it choses, as long is the case instruction itself has this permission. Later, when the case to be executed is known, "side-effect permission" is propagated to the selected case to permit the execution of all instructions, including those with side-effects, and "withdraw permission" is propagated to other cases to stop all instructions from executing.

Tree nodes other than case instruction nodes propagate states down the tree toward the leaves.

If a routine (or block or loop) is terminated by an exception, "withdraw permission" is propagated down the tree from the routine execution node.

Side-effect-free permission allows arithmetic and memory read instructions to execute. But memory writes, subroutine returns, and exception throws require side-effect permission.

Permissions are propagated from a caller to the called routine execution. In particular, if a trap routine executed with side-effect-free permission decides to execute an exception throw, that will be delayed until side-effect permission arrives (if it ever does).

Note that barriers are not accounted for in the execution flow model just given. They add extra tree nodes and extra states. Blocks and loops, on the other hand, can be treated by replacing them with equivalent routines.

By using permission propagation we can allow more instructions to be executed speculatively, allowing instructions in cases to be moved ahead of the computation of which case to select.

The above sounds like permissions are propagated dynamically at runtime. But

when implemented on sequential computers, this is not true. Instead the permissions are propagated statically by the compiler as it generates actual hardware code by compiling R-CODE virtual instructions. The instruction execution tree is determined by the program counter in such an implementation. The compiler uses the instruction execution tree at each point of execution to select the next instruction to be executed, and then computes a new instruction execution tree that will be valid just after that instruction executes.

4.3.18 Flexible Value Lists

A *value list* is a list of register values used to communicate between routine executions. Examples are *argument lists*, routine *result lists*, and *exception value lists*.

Value lists can be of arbitrary length and are received by special instructions. Value lists can be received incrementally so a pattern can be matched to the value list and code selected based on the case matched.

However, for efficiency, value lists can have a signature that can specify the length of the list and types of the values well enough to permit the value list to be stored in real machine registers. Thus there are more constrained value lists that are very efficient, and more flexible value lists that are less efficient.

Exception value lists often require the most flexibility. For example, it is desirable for these to contain diagnostic error messages that can be printed to tell what went wrong. When an exception is rethrown, it may be appropriate to make additions to these messages. The catcher of an exception may or may not want to print the messages.

Another use of flexible value lists is for argument lists for routines like those in an operating system shell language, where there may be very long variable length lists of files or similar arguments.

On the other hand, argument and result lists for many subroutines, such as math routines computing with a few real numbers (square root, sine, etc.), need to be very efficient.

4.3.19 Second Class Frame Pointers

Data can be "allocated in routine frames", but pointers to such data, which are called *frame pointers*, can only be stored in registers and value lists. Furthermore, it must

be possible to find all such pointers at R-CODE run time.

This permits values of arbitrary size to be passed as argument, result, and exception values; but permits the stack to be managed by a simple fast memory reclaimer.

4.3.20 Controllable Inlining

R-CODE routines should be inlined when they are compiled to machine code, and not before. There must be some easy to use scheme to control when a routine is inlined.

By delaying inlining until R-CODE is compiled, changes to inlined routines can be handled more efficiently.

4.4 Related Work

Several computers have been built with *full/empty bits* on memory words and load instructions that delay until the word loaded is "full". Examples of such computers are the Denelcor HEP[Smi78] and its modernization the Tera[ACC+90], and the MIT Monsoon[Pap90].

Our notion of barriers is taken directly from Arvind's ID language[Nik91] which was designed to run on the Monsoon. This notion arises because in a dataflow language it must be possible to determine when a routine has finished executing in order to free the memory taken by the routine's frame. So there is a well defined notion of when a routine execution is done. A barrier merely waits until some partition of a routine is done before permitting the next partition to execute.

Notice there is some question about whether routines called by a given routine execution must be done for the given routine execution to be done. In some ID implementations this would not be necessary just for the purpose of freeing the routine frame. This is because these implementations do not read data from routine frames: they only write data to frames, and each routine execution knows whether all writes to its frame are done yet. However, to make the ID barrier work properly, it is necessary to define a routine execution to be done only if all the routines it called are also done. R-CODE similarly requires called routines to be done before a routine is done.

R-CODE control flow is very similar to that of ID, with the following differences:

1. R-CODE is a virtual computer machine language, and ID is a programming

language.

2. R-CODE is optimized for sequential superscalar or very long word computers, with scheduling done at R-CODE compile time, while ID is designed for special hardware that does scheduling at run time.

3. R-CODE treats memory operations as I/O while ID provides flow control based on memory word full/empty bits.

4. R-CODE permits either non-strict or strict execution, while ID requires non-strict execution.

5. R-CODE supports exceptions and ID does not.

6. R-CODE supports traps and ID does not.

7. R-CODE supports speculative execution more explicitly than ID.

Another thread of research on control flow in functional languages is work on monads, continuations, and similar constructs, which is discussed in [JW93]. The idea is that there are "IO" functions that take as input a special kind of argument, a "world" value, that contains the state of the world, and these IO functions produce as output a normal value paired with a modified world value. First there are primitive "IO" functions, that call the operating system to perform actual input/output. Then there are schemes for creating composite IO functions which are effectively sequences of primitive IO functions, each with its output world connected to the input world of the next. Simply forcing computation of the final world value of a composite IO function will cause the sequence of primitive IO functions it represents to be executed in strict sequential order.

The world value behaves something like the done flag in implementations of ID. A fork operation, in which two parallel computations on the world are spawned, is proposed in [JW93, section 5.1]. A join operation, in which two parallel computations are both required to complete in order to produce a final world, does not seem to be discussed in the literature. It should be possible to introduce an operation to merge worlds to produce a new world, however.

Introduction of forks and joins loses the safety of the language, since parallel paths might conflict and induce non-deterministic results. It might be better to try to build world splitting and combining operations, that split a world into independent worlds,

and then combined modified versions of these independent worlds into a modified version of the original world.

In any case, I feel that the join operation, which in ID and our work is implemented by the barrier operation, is fundamental to parallelism, and languages need it to function properly.

4.5 R-CODE Machine Language Control Flow Semantics

The R-CODE virtual computer is a RISC style machine with a separate set of registers for each routine execution. In the following sections we will give the semantics of the R-CODE virtual computer that relate to control flow.

4.5.1 R-CODE Basic Register Dataflow

R-CODE endows each routine execution with a vector of registers. Each register is 128 bits, and stores tagged data. Registers are computed by instructions, each of which take some registers as input. The output registers of an instruction are distinct from the output registers of any other instruction. There are no "goto" instructions, so each instruction can execute at most once, and each register can be set at most once (loops are treated as tail-recursive routines). Each instruction contains a type code that restricts the permitted values of its output registers, and permits efficient execution on untagged architectures. Instructions may be executed in dataflow order, or they may be executed in sequential order, without difference except for side effects involving non-register memory or I/O.

We will go into these considerations in more detail in the following subsections.

4.5.1.1 Registers

An R-CODE *register* holds a 128-bit tagged value: see Figure 4.1. Registers are virtual; in an actual implementation most registers hold values of more restricted types that are stored in a more compact format (see "Type Codes" below, page 128). For example, an R-CODE register restricted to have a 32-bit integer value would be stored in a 32-bit machine register or a 32-bit memory word in a routine execution frame.

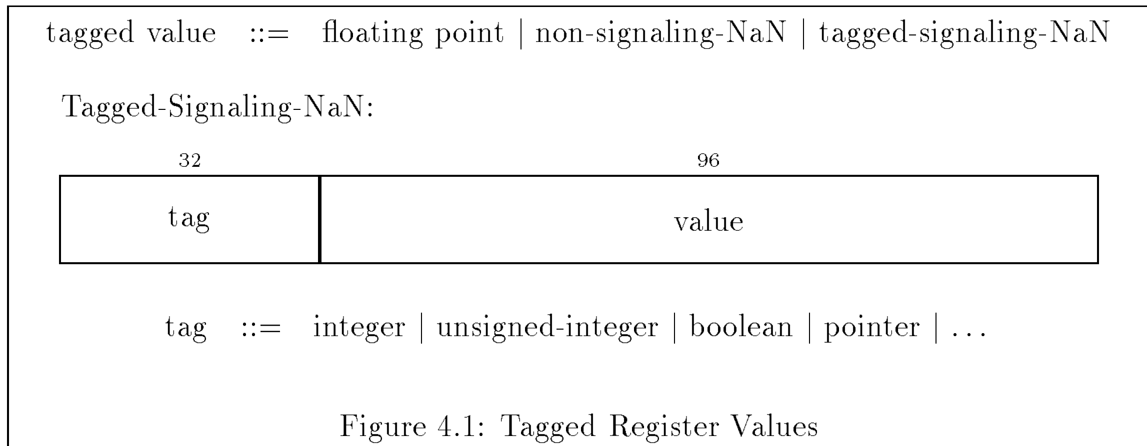


Figure 4.1: Tagged Register Values

The 128-bit tagged register value has the format of an IEEE 128-bit floating point number[Inc92, Table 3-5]. It may hold a standard floating point value, including the infinities; or a non-signaling NaN value, as is specially discussed below (see "Non-Signaling-NaNs", page 146); or a signaling-NaN value that represents a tagged value. In other words, the R-CODE tagged data scheme is built on top of the IEEE floating point number scheme, using the latter's signaling-NaNs for tagged values that are not floating point numbers or "missing values", and using non-signaling NaNs to represent missing values.

The types of tagged values represented by signaling-NaNs include:

boolean values (TRUE or FALSE)
64-bit signed integers
64-bit unsigned integers
pointers of various formats
other special values

See section 3.3.2 of Chapter 3 for a more complete description of R-CODE tagged values.

4.5.1.2 Instructions

Typical R-CODE *instructions* are 64-bits formatted as in Figure 4.2. Typical instructions have a single output register whose values are constrained by the type code field

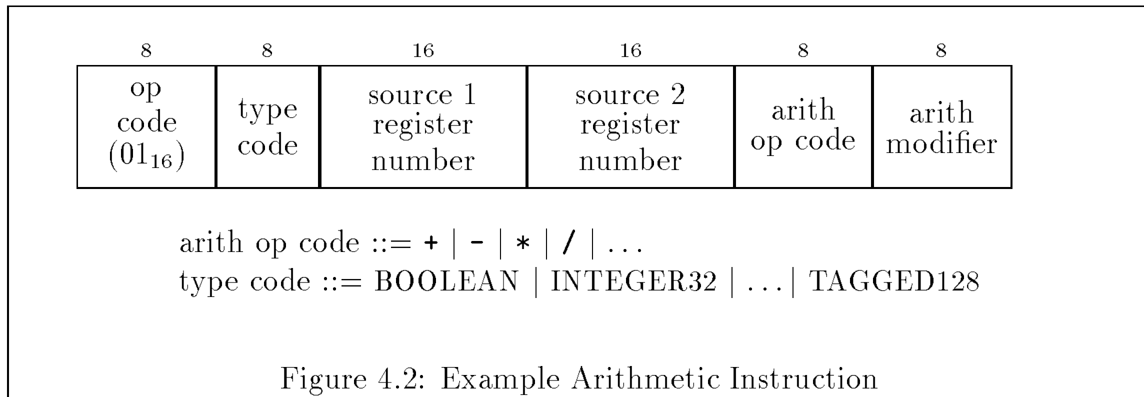


Figure 4.2: Example Arithmetic Instruction

associated with the register, as discussed below. The output register number is assigned according to the position of the instruction in its routine, and is not stored in the instruction.

Typical instructions have two input registers whose numbers are fields of the instructions. All register numbers are relative to the current routine execution frame. The input register numbers must be less than the instruction's output register numbers. This guarantees that strict execution order will work (see the "Either Strict or Non-Strict" principal above).

The instruction contains an operation code and operation modifier bits. A typical operation code is "addition" and a typical modifier is "round toward zero."

R-CODE instructions are virtual: they are not actually stored or interpreted by implementations. They define a language for representing programs that is to be compiled into actual machine code.

Some instructions define groups of other instructions. A case instruction is followed in memory by its cases, each of which begins with a special 64-bit instruction word followed by the instructions in the case. Routines are similarly organized into partitions, and instructions that send or receive value lists also include groups of 64-bit instruction words. We will not delve deeper into instruction formatting in this paper: see [Wal].

4.5.1.3 Type Codes

Each register has an associated 8-bit *type code* that is stored in the instruction that outputs the register. This type code constrains the possible values of the register. For example, the type code might specify that the register can only hold 32-bit signed integers. This would permit an implementation to use just a 32-bit hardware register to hold the value of a 128-bit virtual R-CODE register.

The following is a list of R-CODE type codes:

boolean
32-bit signed integer
32-bit unsigned integer
32-bit floating point number
64-bit signed integer
64-bit unsigned integer
64-bit floating point number
64-bit tagged value
128-bit floating point number
128-bit tagged value
various forms of pointer
other special data

The 128-bit tagged value type code means the value in the register is unrestricted. R-CODE also supports 64-bit tagged values that are based on IEEE 64-bit floating point numbers and are designed for implementing LISP.

No matter what the type code is, a register can store a non-signaling-NaN value. These values are used to indicate the instruction that output them had illegal inputs. See "Non-Signaling-NaNs" below (page 146) for more details.

4.5.2 Frame Memory

Frame memory holds all the data necessary to support register dataflow execution, but excluding the general heap memory. In the general heap memory objects may be freely allocated and freely point at each other. This is not so in frame memory, where objects may only be allocated with a stack or tree like discipline and may not be pointed at except by registers and value lists (see below for value lists).

In implementations that are not parallel, frame memory is stack like. In implementations that are parallel, frame memory is tree like.

Frame memory has three parts.

Register Frames. *Register frames* hold R-CODE registers. There is one register frame per routine execution.

Value Lists. *Value lists* are used for communication between routine executions. The kinds of value list are:

Argument lists created by call instructions and received by called routines.

Result lists created by return instructions and received by call instructions.

Exception value lists created by exception throw instructions and received by exception catches.

Frame Heap. The *frame heap* includes objects similar to those in the general heap, except all pointers to frame heap objects must be stored in registers or in value lists. The consequence of this last rule is that frame heap objects are allocated with a stack or tree like discipline which permits very fast efficient memory reclamation.

The frame heap works with register frames to permit routines to have local values that are full fledged memory objects, except for the restriction that pointers to frame heap objects may not be stored in any objects.

The following sections describe the three parts of frame memory in more detail.

4.5.2.1 Register Frames

A *register frame* is a vector of registers. The registers in a frame have *register numbers*, increasing from zero.

When an R-CODE routine is scheduled, a register frame is allocated for that routine execution. When the register frame is created, output registers are allocated for each instruction in the routine. These registers are allocated in the same sequence as the instructions appear in the routine, so the numbers of the output registers of two instructions are always in the same order as the instructions appear in the routine.

The number of registers in the frame is fixed by the number and kinds of instructions in the routine, and is determined when the frame is allocated, and not changed thereafter.

The types of values that can be stored in each register is determined by the type code of the instruction which outputs to the register. This is also determined when the frame is allocated, and not changed thereafter.

Each instruction may have input register numbers. These must be less than any output register numbers of the instruction. This enforces the rule that strict execution sequencing is permitted (see the "Either Strict or Non-Strict" principal above).

Register frames are organized as a tree: the parent of each frame is its caller's frame (except that the root frame has no parent). If completely strict execution is used, this *register frame tree* would be a stack. Here by strict execution we mean executing instructions in the order they appear within a routine, not executing any part of a routine until all its inputs are ready, and not using the results returned by a routine until the routine is done.

But, in the presence of inlined routines and instructions reordered to match hardware, routines will appear to execute non-strictly, even on existing sequential computers. The R-CODE virtual computer is required to make inlined routines appear to the debugger to have been executed out of line (by the "Out-of-Line Equals Inline" principal on page 117 above). But then these routines will appear to be non-strictly executing out of line routines, though they will in reality be inline routines whose instructions have been intermixed with instructions from other parts of their caller.

The register frame tree is sometimes called the *routine execution tree*, as its nodes can be thought of as routine executions, as well as register frames.

A register frame may be deallocated from memory when its routine is done, and when any return list or expression value list this routine is sending to another frame in the routine execution tree is no longer needed or has been re-attached to the receiving frame.

4.5.2.2 Frame Heap

Many languages permit arrays to be allocated in a routine execution frame on the "stack" with an array size not determined until execution of the routine. Some languages (e.g. Ada [ANS83]) permit such arrays to be returned as result values to

the caller of the routine.

The *frame heap* addresses these needs. It is a heap where ordinary objects may be allocated, as long as pointers to these objects are only stored in registers or value lists. This restriction on where pointers to the objects are stored permits fast stack-like memory reclamation to be implemented.

The reason such memory reclamation can be implemented is that when a routine execution is done, all frame heap objects allocated by the routine execution subtree rooted at the routine are either no longer in use, or are pointed to by the result list or exception value list returned to the call instruction that scheduled the routine. Thus a memory reclamation algorithm can merely compact the objects pointed at by the returned value list, and free the rest of memory allocated by the execution subtree. Since the values to be compacted are not visible until the value list is used, there is no problem moving them and adjusting their addresses in the value list.

4.5.2.3 Value Lists

An R-CODE *value list* is a vector of register values. There are three kinds of value list: *argument lists*, *result lists*, and *exception value lists*.

An R-CODE value list is constructed and sent by a call instruction (argument list), return instruction (result list), or exception throw instruction (exception value list). It is received by a routine execution (argument list), call instruction (result list), or call instruction exception catch case (exception value list).

Exception value lists may be received by a call exception catch case, partly read, and forwarded for further processing by an exception re-throw to another call exception catch case, or by a call to a subroutine. The receiver of an exception value list may known only enough about the list to decode the first part, and then pass the list on.

Argument lists may be received by a routine execution that similarly decodes and uses the first part, and then after replacing the first part passes the list on to another routine.

These capabilities suggest that value lists should be made fairly first class data in R-CODE. However, for efficiency reasons value lists may need to be restricted.

In R-CODE, flexible first class value lists are variable length vectors of tagged values stored in memory. Efficient restricted lists are fixed length vectors of register values, with each value being associated with an 8-bit type code, the same way as registers

are associated with a type code. R-CODE also supports hybrid lists, with a short part in real hardware registers, followed by a variable length part in memory.

Specifically, an R-CODE value list is a vector called a "*register value vector*". Each element of this vector is either a register value, or a pointer at or into a vector of register values called a "*memory value vector*". Normally a value list either has no pointers to memory value vectors, or only the last register value vector element is a pointer to a memory value vector. However, while writing subroutines to manipulate value lists, it is desirable to pass pointers into memory value vectors between routine executions, and to permit this, register value vectors are allowed to contain several pointers at memory value vectors.

A value list has a *signature*, which must be agreed upon by all the senders and receivers of the value list. The signature consists of a fixed length vector of 8-bit type codes, and an equal length vector of "*memory vector flags*". The length of these vectors is the length of the register value vector part of the value list.

Values in a value list are 128-bit tagged register values, virtually speaking. For each signature position whose memory vector flag is off, the signature 8-bit type code describes the permitted values of the register value vector element in that position, in the same way that the 8-bit type code in an instruction describes the permitted values of an instruction output register, but with one difference. The difference is that an implementation is free to permit or disallow non-signaling-NaN values, unless the value is a tagged value, in which case non-signaling-NaN values are allowed.

For each signature position whose memory vector flag is on, the signature 8-bit type code describes the permitted values for the elements of the memory value vector pointed at by that the register value vector element in that position. Again, the implementation is free to permit or disallow non-signaling-NaN values, unless a value is a tagged value, in which case non-signaling-NaN values are allowed.

The receiver of a value list must allocate as many registers as there are type codes in the value list signature. The associated values are copied into these registers if their memory vector flags are off. Pointers to memory value vectors are copied if the memory vector flags are on.

New memory value vectors may be allocated by a routine, and made by copying from other memory value vectors. Memory value vectors may be passed as the end of an argument list, result list, or exception value list to another routine execution. The length of a memory value vector must be determined when it is allocated, but the

vector may be written incrementally. Pointers into memory value vectors may be passed as arguments and returned as result values in register value vectors.

Memory value vectors are like frame heap objects, but differ in that they can hold pointers to frame heap objects (if their type code permits).

4.5.3 Execution State

4.5.3.1 The Execution Tree

All scheduled instructions may be viewed as part of a tree, the *instruction execution tree*. The nodes in this tree are as follows (see Figure 4.3 for an example):

Leaves. Instructions such as arithmetic operations are normally leaf nodes of the tree, and have no children. However, if they trap, they become like subroutine call instructions.

Calls. Call instructions have as children first the routine execution and second an optional exception catch case. A call instruction acts in part like a case instruction that may select the exception catch case if an exception value list is returned. If the exception catch case is selected, it may return a result list for the call, or may throw an exception itself.

Routine Executions. Routine executions have as children the partitions of the routine being executed, in the order the partitions appear in the routine.

Partition. Partitions have as children the individual instructions of the partition. Note that any instructions inside the cases of case instructions are excluded; only the case instruction is directly a child of the partition, and instructions inside cases are descendents of the case instruction.

Case Instructions. Case instructions have as children their cases.

Cases. Cases have as children the individual instructions of the case. Note that as for partitions, if a case instruction is such a child, then the instructions inside its cases are not such children.

We have excluded blocks and loops from this discussion. They are introduced below by giving equivalences between them and routines.

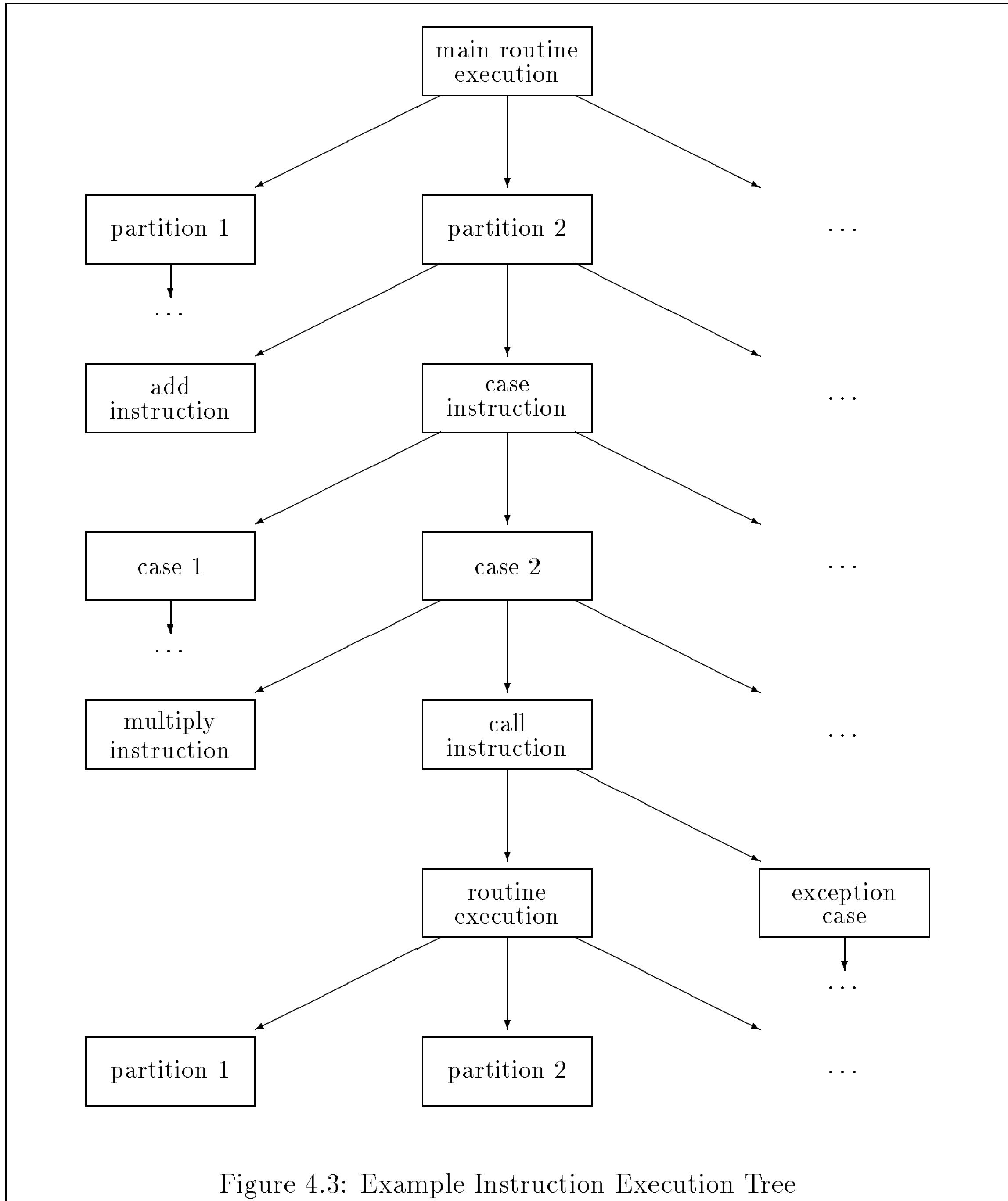


Figure 4.3: Example Instruction Execution Tree

4.5.3.2 Tree Node States

Each node in the execution tree has one of the following *states*.

no permission
read-free permission
side-effect-free permission
side-effect permission
withdraw permission
done

These states are ordered, increasing from "no permission" to "done".

A node is said to be "*withdrawn*" if and only if it has "withdraw permission."

Nodes can execute and schedule children. Scheduling children involves adding nodes to the execution tree. Execution involves setting output registers and performing memory operations or other input/output. A node can schedule children before or during its execution.

4.5.3.3 State Transition Rules

The following are general rules for assigning states to nodes.

1. When an instruction is scheduled, it is added to the execution tree and given the "no permission" state.
2. The state of a node never decreases, but always increases from "no permission" toward "done".
3. If a node has children, then it passes its state onto any child if that state is higher than the child's current state and the child is not a case or partition.
4. If all the children (if any) of a node achieve the "done" state, and the node has finished its own execution, then the node is given the "done" state.
5. New nodes may be added to the execution tree (when instructions are scheduled), but not as children of withdrawn or done nodes, or of nodes that have finished their execution. Nodes may not be deleted from the execution tree.

6. A node can begin to execute when it is in a state with sufficient permission and enough inputs are ready. For nodes other than those that involve side-effects or read from memory, "read-free" permission is adequate to begin execution. For memory read instructions, "side-effect-free" permission is required. For instructions that involve side-effects, "side-effect" permission is required. One of these three permissions with a higher state value is sufficient when only a lower valued permission is required.

7. A node that is not withdrawn cannot complete its execution until all required inputs are available to it. It need not require all its potential inputs.

8. During execution a node that is not withdrawn may decide to trap. If a node so decides, it spawns a new routine execution child and behaves thenceforth like a call instruction (with no call exception catch case).

9. If a node is withdrawn, it will quickly finish execution (and then pass to the done state as soon as its children are done). If the node has not begun execution, it "finishes execution" without having actually done anything. If the node has begun execution, it will finish execution without necessarily doing all of what it would normally do.

   An instruction is called an "atomic instruction" if, even when withdrawn, the instruction's execution either does nothing or does what the instruction would normally do. Unless specified otherwise, instructions are <u>not</u> atomic. In particular, non-atomic memory write operations may write only part of the memory they normally change.

The goal is to get all nodes in the execution tree labeled "done".

According to these rules, a node that is done or withdrawn cannot spawn new children. As a consequence, a subtree all of whose leaves are done or withdrawn cannot grow.

All states tend to propagate down toward the leaves of the execution tree, except the "done" state. Because a node may not become done until its children are done, this state propagates up from the leaves.

4.5.3.4 State Maintenance

The state transition rules are intended to be applied at R-CODE compile time by an R-CODE compiler, and <u>not</u> at run time, for superscalar and very long word com-

puters. This is to be done by assuming that out-of-line routine calls are strict, so all scheduling really concerns only the code within a routine and the routines it inlines.

An important consequence is that the state of the execution tree is a function of the program counter value on a superscalar or very long word computer. So the state does not have to be maintained at routine run time in order to be presented by the R-CODE debugging interface to debuggers. Instead, the program counter can be merely "looked up in a table" to find the instruction execution tree state relevant to a routine (but this table may be rather large, as noted in "Implementation Challenges" below).

4.5.4 Cases

A *case instruction* selects one of several *cases* to execute. Each case is a group of instructions.

A case instruction works by propagating permissions down a tree of scheduled instructions. A case instruction and its cases do the following:

1. When the case instruction is scheduled, it allocates registers for itself and its cases, and schedules its cases.

2. When the case is scheduled, it is allowed to schedule its children unless the case is withdrawn. However, a case is not required to schedule its children unless it has received side-effect permission.

3. When a case instruction receives read-free permission it is allowed to propagate that permission to any of its scheduled cases, but is not required to. Ditto with side-effect-free permission.

4. When a case instruction can compute which case it will select, it propagates withdraw permission to all other scheduled cases.

5. When a case instruction receives side-effect permission and can also compute which case it will select, it propagates side-effect permission to that case.

6. If a case instruction receives withdraw permission, it propagates that to all its scheduled cases that are not done.

7. If a case instruction receives an illegal input, such as a non-signaling-NaN that the case instruction is not prepared to accept, the case instruction propagates withdraw permission to all its scheduled cases that are not done, and schedules a new routine execution child to execute the case trap routine. The state of the case instruction is then propagated to this trap routine execution node, and the case instruction thereafter behaves like a call instruction.

   A case instruction may be configured to recognize a non-signaling-NaN as a legal input and use it to select a particular case.

8. An instruction in (i.e. descended from) a case cannot input registers that are the output registers of instructions in its sibling cases (i.e. cases with the same parent case instruction).

Cases propagate new states down to their children as per the default rules for execution tree nodes. And both cases and case instructions become done when all their scheduled children are done and the case instructions have executed, as per default rules.

4.5.5 Barriers

A routine is divided into *partitions*. Each partition is a sequence of instructions (just like a case of a case instruction). The partitions may be thought of as being separated by *barrier instructions*, though no such instructions are actually encoded in R-CODE.

Routines execute by propagating permissions down trees of scheduled instructions. A routine execution and its partitions do the following:

1. When the routine execution is scheduled it allocates registers for its partitions and schedules the partitions.

2. When the partition is scheduled, it is allowed to schedule its children unless the partition is withdrawn. However, a partition is not required to schedule its children unless it has received side-effect permission.

3. When a routine execution receives read-free permission it is allowed to propagate that permission to any of its scheduled partitions, but is not required to do so.

4. When a routine execution has side-effect permission, it must propagate that permission to its first scheduled partition that is not done, provided no partition is withdrawn. When that partition becomes done, this process repeats.

5. If a routine execution receives withdraw permission, it propagates that to all its scheduled partitions that are not done.

6. If a partition executes a return instruction, then withdraw permission is propagated to all subsequent partitions of the routine execution being returned from.

7. If a partition is withdrawn, all subsequent partitions of the same routine execution are withdrawn.

Partitions propagate new states down to their children as per the default rules for execution tree nodes. And both routine executions and partitions become done when all their scheduled children are done, as per the default rules.

4.5.6 Calls and Returns

A *call instruction* has an optional *exception catch case*. A call instruction that has no exception catch case cannot receive an exception value list. A call instruction with an exception catch case behaves like a case instruction that performs the call to determine whether to select the exception catch case, and selects that case only if an exception value list is returned.

The steps in executing the call and the routine it calls are as follows when the call does not receive an exception value list:

1. When the call instruction is scheduled, it allocates a set of registers for its argument list, for the address of the routine being called, for the result list to be returned by that routine, for exception values if an exception catch case is present, and for the exception catch case if that is present.

2. A call instruction with an exception catch case may chose to schedule the case and propagate read-free permission to it any time after the call instruction is scheduled and before the call instruction is done or withdrawn.

3. The address of the routine being called is computed.

4. The routine execution is scheduled, by allocating its register frame and scheduling all the partitions of the routine.

5. The routine and its caller execute according to permissions propagated from the call instruction to the routine execution. The routine may read values from the argument list after they have been computed. Not all values in the argument list need have been computed by this time.

6. A return instruction in the routine is given side-effect permission. This specifies the routine execution result list as the list of registers specified by this return instruction to hold results.

7. The routine and its caller execute. The routine may read more values from the argument list after they have been computed. The caller may read values from the result list after they have been computed. Not all values in the result or argument lists need have been computed by this time.

8. The routine execution becomes done, because each of its partitions becomes done. The routine's register frame is reclaimed, except for the result list, which may be still in use by the caller, and is detached from the called routine's register frame and re-attached to the caller's register frame.

9. If a call instruction has an exception catch case, then when the routine execution child of the call instruction becomes done, and no exception value list has been returned to the call instruction, the call instruction propagates "withdraw permission" to the exception catch case (if it has scheduled it).

It is quite possible for several result lists to be attached to the caller's frame at one time. The caller may make calls in order to process the first such list, and these calls may create more result lists.

If the call instruction receives an exception value list, what happens is as given above with the following modifications and additions:

1. If a call instruction receives an exception value list, it may or may not receive a result list. The routine execution may or may not execute a return instruction.

   Also, some result values may be received, but others not received. This is because a return instruction may have been executed, but only some result values may have been computed before the exception throw.

2. If a call instruction receives an exception value list, and the call instruction is not withdrawn, it waits till the routine execution is done, and then propagates side-effect permission to its exception catch case.

   Note that by propagating only read-free permission to the exception catch case, and by waiting for the routine execution to be done before propagating side-effect-permission, we have the effect of a barrier between the routine execution and the exception catch case execution (except that if the routine returns it does not stop the catch case).

3. A call exception catch case, once selected by an exception value list, executes like a normal case instruction case. However, if it does not rethrow the exception or throw a new exception past its parent call instruction, it must provide result values for its parent call instruction. It does this with a *special return instruction*. If the special return instruction attempts to return some result value that has already been returned by the routine execution, the special return instruction traps, and cannot be continued. It is possible for the exception catch case to discover which result values were returned by the routine execution.

A call instruction without an exception catch case cannot receive an exception value list.

The rules just given are based on an optimistic strategy that assumes exceptions will not be frequent. Thus the caller does not wait to see if a routine will have an exception before using results from the routine. So if a routine has an exception it may also have results. But if a routine has an exception and no results, results must be provided if execution is to continue from the call instruction. In general, exception catch code that tries to return its own results is likely to assume that no results were returned, and it will be an error if some are.

4.5.7 Exception Throws

An *exception throw instruction* does the following:

1. The exception throw instruction waits until it has side-effect permission and until all its inputs are available.

2. A search is made through the ancestors of the exception throw instruction for a call instruction with an exception catch case.

   All the nodes discovered during this search before the found call instruction are given withdraw permission (which then propagates down other branches of the tree).

   However, if during the search a node is found that already has withdraw permission, the exception throw instruction aborts and finishes execution without any other effect.

3. The inputs to the exception throw instruction are used to construct an exception value list which is sent to the found call instruction.

Note that if more than one exception throw happens in the same routine execution, all but one exception value list will be discarded.

4.5.8 Blocks

A *block instruction* introduces a block of instructions that are equivalent to a routine called by the block instruction. Thus blocks are inlined routines.

A block instruction is like a call instruction except for the following.

1. When a block is scheduled, it allocates registers for the partitions in the block, and after these allocates the result and exception value registers that a call instruction would allocate.

2. Block instructions do not have arguments. Instead, the instructions of a routine simply reference registers allocated before registers of the block. Since these are read-only, the block could be made into a routine with these passed as arguments.

Block instructions can have exception catch cases, like call instructions.

A common programming construct is to return to a block that is an ancestor of the current block. To permit this, R-CODE provides a special *block return instruction* that returns results to the N'th ancestor block up the execution tree. Note, however, that if a return to the N'th ancestor block is executed, returns to the $N-1$'st,

$N-2$'nd, ..., 1'st ancestor blocks must still be executed, to get all the intervening block instructions to complete. These N separate block return instructions may be executed in any order.

Thus, returning from each of a set of nested blocks is viewed as an independent activity.

4.5.9 Loops

A *loop instruction* introduces a block of instructions that are equivalent to a tail recursive routine called by the loop instruction.

A loop instruction is like a call instruction except for the following:

1. When a loop is scheduled, it allocates registers for the partitions of several executions of the the loop, and after these allocates the result and exception value registers that a call instruction would allocate.

2. The loop instruction has arguments like a call instruction that become the arguments of the first iteration of the loop.

3. The loop can execute a loop continue instruction that is like a call in that it starts the next execution of the loop, and passes arguments to that execution. However, the loop continue instruction arranges that the next execution of the loop will return directly to the loop instruction if it executes a normal return, and not to the previous execution of the loop.

   The new loop execution becomes a child of the loop instruction in the execution tree, and not of the loop continue instruction execution.

4. If a loop execution executes a return instruction, this returns results to the loop instruction. Similarly if the loop execution executes an exception throw, the throw behaves normally using the loop instruction execution as the parent of the loop execution.

5. Loop instructions can have exception catch cases. If an exception throw is caught by the loop instruction, all loop executions that are children of this loop instruction are withdrawn.

A loop instruction provides registers for some number of loop executions. When all of these have been scheduled, the first loop execution must be "done" so its registers can be reclaimed before the next loop execution can be scheduled. Thus the maximum number of simultaneous loop executions is decided when R-CODE machine language is generated, and not when R-CODE is compiled. Actually, this is not the real maximum number of simultaneous loop executions: it is only the apparent maximum number presented by the debugging interface, and restricts debugging more than actual execution.

4.5.10 Inlining

R-CODE routines are optionally inlined when they are compiled to machine code.

In R-CODE, each routine is given a *inlining priority*, which is higher if the routine is better off inlined. Each call is given an *inlining enable level*. A call is inlined if the called routine can be identified at compile time and the call enable level is less than or equal to this routine's priority.

Typically all calls in one routine are given the same enable level, but this is not required.

There needs to be some way to suppress inlining within inlined routines, i.e. recursive inlining. This is done by giving each routine and each case of a case instruction an *inlining enable increment*. When a routine is inlined, its increment is added to the enable level of all call instructions inside the routine. When a case is inlined, its increment is added the enable level of all call instructions inside the case.

This proposal is rather experimental, and will have to wait serious usage to see how well it works.

4.5.11 Localizing

To *localize* a set of routines within a parent routine means to compile special versions of the localized routines which can only be called by the parent or by one of the other routines in the set being localized. The directed call graph of the set of routines to be localized (excluding the parent) may not have any cycles.

Localized routines may be implemented to use part of their parent's register frame, instead of having their own register frame. This is hidden inside an R-CODE implementation, but makes for some increase in efficiency.

Localization is similar to inlining, but is useful when a routine is called several times by its parent.

In R-CODE, routines are given a *localization priority* that is separate from the routines' inlining priority. This is compared with call instructions' inlining enable level to determine whether the routine should be localized. A routine is localized if any one call by the parent or by another localized routine indicates it should be localized.

If the set of routines to be localized contains a call cycle, this is considered a compilation error. Implementations should break the cycle in some unspecified way and warn the user. Any recursive chain of routines should contain a routine that is not localizable.

Implementations should perform obvious optimizations such as inlining any localized routine that is called only once and removing arguments to localized routines that are the same for every call to the routine within its parent.

4.5.12 Coroutines

In R-CODE, *coroutines* are implemented by a "*coroutine channel*". A coroutine channel holds a pointer to a call instruction execution that is waiting for results.

Coroutines work as follows:

1. To start a coroutine, a special coroutine start instruction is used that calls the coroutine and places a pointer to the coroutine start instruction execution, which is a form of call instruction execution, in a coroutine channel.

   A pointer to the coroutine channel is passed as an argument to the coroutine.

2. Once started, the coroutine or one of its subroutines can then "call" the coroutine channel. This has the effect of taking the call argument list and using it as the result list for the call instruction execution pointed at by the channel. It then resets the channel to point at the execution of the instruction that called the channel.

3. Once a coroutine has called its channel, and the previous caller of the channel has received a return, then the previous caller can call the channel to call the coroutine back. This also resets the channel to point at the execution of the instruction that called the channel.

4. When the coroutine returns, its result values are sent to the call instruction execution pointed at by the coroutine channel.

5. When the coroutine is called by the start instruction, the coroutine execution is attached in the execution tree to some call instruction execution designated by the start instruction. This designated call instruction execution must be an ancestor of the start instruction, and is typically the parent of the routine execution containing the start instruction execution.

6. If the coroutine throws an exception, the execution subtree rooted at the coroutine execution is given withdraw permission, but withdraw permission is not propagated to the parent of the coroutine execution, and the exception value list is not sent to that parent or one of its ancestors.

   The exception value list is saved until the coroutine channel points at a call instruction execution that is not in the coroutine execution tree. Then the exception value list is returned to that call instruction execution.

A coroutine may be localized (see "Localizing" above) if it does not pass the pointer to the coroutine channel to any non-localized routine, and if the coroutine execution is to be a sibling in the execution tree of the routine executing the start instruction. In this case a call graph is made of calls that pass the coroutine channel pointer, and all routines connected to the coroutine in this call graph are allocated separate register space in the routine execution frame of the start instruction.

Non-localized coroutines cause the routine execution tree to cease to be stack-like. Localizing coroutines preserves any stack-like nature of this tree.

4.5.13 Non-Signaling-NaNs

R-CODE instructions that do not normally have side effects output IEEE *non-signaling-NaNs* when they cannot produce more reasonable results. They do this instead of trapping, if there is no possibility that a trap routine could produce a reasonable result.

The R-CODE short circuit boolean AND instruction will accept a non-signaling-NaN as a legal second input if the first input is false. Thus the second input can be computed in parallel with the first input, even if the first input must be true in order for the second input to be legal. If the first input computes to false, the second input

may compute to a non-signaling-NaN, but this will not affect the final result, which is false.

R-CODE case instructions can have an optional case that is selected by a non-signaling-NaN input. If they do not have such a case, they trap when given a non-signaling-NaN input, and this trap routine cannot return: it must throw an exception or invoke the debugger.

Whether or not the output of an instruction is a non-signaling-NaN can be thought of as an additional piece of state for the instruction execution node that is part of the instruction execution tree. However, unlike the rest of the instruction execution tree state, this extra piece may not be a function of the program counter value for a superscalar or very long word computer implementations. When it is not, the implementation must manipulate flags at run time to indicate non-signaling-NaN outputs, and this causes inefficiencies. Hopefully this will not be necessary often.

IEEE non-signaling-NaNs can contain many bits of information. R-CODE ignores this extra information, and may not preserve it when copying non-signaling-NaN values. This permits R-CODE implementations to use a single bit to represent the presence or absence of a non-signaling-NaN in a particular R-CODE register.

4.5.14 Traps

An instruction *traps* when it finds it has inputs that it cannot process and either it makes no sense to output a non-signaling-NaN or there is some possibility that a trap routine might be able to compute some other output for the instruction.

An instruction trap is just like a subroutine call. The routine to be called is selected from a dispatch table using the instruction operation code and input arguments, as appropriate. The routine may return values to emulate the instruction, if appropriate.

More specifically, when an instruction finds its inputs are such that it should trap, it may schedule the trap routine just as if the instruction were a call instruction. If the trap routine returns, the result values are placed in the output registers of the instruction, and the instruction is done. Instructions that trap behave like call instructions that do not have exception catch cases.

The implementation of traps is conceptually like turning instructions into case instructions which have a non-trapping and a trapping case. To permit all the desired traps and trap routine returns, extra code may be needed for some instructions on

some hardware. An implementation might have various optimizations for this, such as modes where certain trap routine returns are disallowed, and modes where the extra code is computed on the fly when first needed, and then cached for possible future use.

In most cases, traps are considered an unusual circumstance, but in some cases they are not. For example, arithmetic operation traps on signaling-NaN tagged values might become the standard way of implementing rational numbers in LISP.

4.5.15 Continuations

R-CODE has a special *continuation call instruction* that makes the new routine execution a child of the parent of the current routine execution, instead of a child of the continuation call instruction execution. A routine execution that executes a continuation call may not execute a routine return. The new routine execution created by the continuation call becomes responsible for returning results to the common parent.

One effect is that a call instruction may have more than one routine execution as a child, though only one is supposed to return.

Exception processing is extended so that if an execution sets the state of any routine execution of a call instruction execution to withdraw permission, then the states of any other routine execution children of the same call instruction execution are withdrawn.

If more than one routine execution attempts to return to the same call instruction execution, the result values are undefined, but at least one of the return instructions will trap so a debugger can intervene.

4.5.16 Recording State

The state of the execution tree must be recorded for the sake of the debugging interface. This is done in the R-CODE virtual registers.

When a typical R-CODE arithmetic instruction is scheduled, it is copied into its output register. The 64-bit instruction fits in this 128-bit register with a special tag indicating it is an instruction waiting to execute. When its inputs become available, it executes and replaces itself in its output register by its result.

More complex instructions like case instructions record their state in their output register or registers.

Although these state changes mean that R-CODE registers are not simply unchanging constants, the state changes are monotonic, leading from "less done" values to "done values."

All of this is, of course, virtual. It is only visible through the debugging interface, and is just the means this interface has of presenting the execution tree state to debuggers.

4.6 Implementation Challenges

To first order approximation, it is very easy to compile R-CODE for existing computers. Strict sequential execution semantics is used for out of line subroutine calls, and inlining is done in R-CODE before translation to real machine language. Instructions are issued in any order consistent with the above register dataflow semantics and strict execution of out of line calls. The type codes in instructions are used to generate efficient code that does not really produce 128-bit tagged values. Execution tree state is computed by the compiler as a function of the implementation program counter.

However, at second look there are some challenges.

4.6.1 Simulating the R-CODE Computer

The debugging interface needs to make the code execution look like it was done on a real R-CODE computer. Fortunately, there is no speed requirement for doing this.

The debugging interface may need remarkably little saved register frame state to reconstruct the virtual register frame contents. For a simple routine, all the debugging interface needs is the values of the arguments and any values read from memory or returned as result values from out-of-line routine executions. Everything else can be computed by simulation, including the execution tree state.

However, a lot of extra information is required about a routine to perform this reconstruction when the debugger interface is invoked during execution of the routine. Although this information is computed when each routine is compiled, it may be so voluminous that saving it all may not be a good idea. Instead, it can be dynamically recomputed as needed, and a cache maintained of the information for routines

recently examined using the debugger interface.

4.6.2 Non-Signaling-NaN Outputs

Instructions on existing computers do not simply output non-signaling-NaNs or some equivalent when confronted by illegal inputs, but instead are likely to trap, or set an overflow flag. Therefore it is important not to execute them if their production of a non-signaling-NaN would not cause a trap at some point in the routine execution. In some cases it may not be possible to arrange this, and the overflow flag will have to be saved for later use.

For example, if given a short-circuit AND instruction whose output is not affected by a non-signaling-NaN second input if the first input is false, the first input computation needs to be done before any part of the second input computation that might trap.

On the other hand, if the production of a non-signaling-NaN by an instruction, such as an add instruction, is guaranteed to cause an irrecoverable trap later, such as a case instruction trap because the case selection cannot be computed, then the add instruction can be executed and the trap made to look as if it happened later.

4.6.3 Trap Implementation

A number of issues occur when implementing traps that work as if the trapping instruction turned into a subroutine call.

First, some computers may not save enough state to permit this, and there may have to be code associated with the trapping instruction to assist in calling such trap routines.

Second, some traps may need extra code associated with the trapping instruction to permit traps to return properly.

For example, if a trap routine returns a non-signaling-NaN as the output value of an instruction, this may need to be translated into an immediate second trap of a downstream instruction that would trap if the first instruction outputs a non-signaling-NaN.

Faithful execution of trap semantics may be inappropriate in many cases, and implementations should provide modes to disable the generation of the code required in these cases.

4.7 Summary

R-CODE is to provide a new execution model for future programming languages which minimizes unnecessary sequentiality while running well on both existing and future computers. The future programming languages should be easy to program, but need not be strictly compatible with existing programming languages. The R-CODE model of non-memory dataflow plus barriers and I/O-like memory operations should work well for languages in which most code is functional, but a small amount of code is written using barriers and I/O-like memory operations.

Chapter 5

Shared Object Memory

5.1 Goals

Because there is a limit on the speed of a single processor, there is an ultimate need for parallel computation in any application in which latency is a factor.

Memory speeds have not been increasing as fast as basic processor speeds. In fact DRAM memory speeds have only been increasing at the rate of 7% per year, while processor speeds have been increasing at 80% per year. This phenomenon, called the "memory wall" (see page 14), will probably lead to computers in which a secondary cache miss will cause a pause of a few hundred instruction-execution-times.

There are only two methods I know to cover long cache miss times. The first is prefetching: predicting the need for a piece of memory far enough in advance, in this case a few hundred instruction-execution-times in advance. The second method is multi-threading: making a single processor look like several processors, so when one pauses, another runs. The latter is the more widely applicable solution, and leads to decomposing programs into a small number of threads that work together on a common shared memory.

Now consider the situation where many computers reside in the same building, as happens in schools. We may want these computers to work together on the same problem, where problems of interest include compilations, text processing, games, and simulations. The communication latency between computers is limited by the speed of light, and as fiber optic communication with throughputs on the order of

1 gigabit per second is becoming inexpensive, the speed of light latency is probably the only limit on inter-computer communications for these *within-building systems*. This latency will likely be a few thousand instruction-execution-times within our time frame (2005-2035 A.D., see $\mathbf{C}_1$, page 12).

This within-building situation leads to the idea of having many processors work on the same problem by sharing an object memory, where it takes a few thousand instructions to fetch an object. This memory is similar to disk, but this *shared object memory* has a response at least 1,000 times faster than disk, albeit 10 times slower than local memory.

Thus many computers in the same building should have a shared object memory, so they can work on the same problem, and a single computer on a desktop may want to run several threads simultaneously, with the threads sharing a memory that might as well be organized as a shared object memory too.

Thus the goal of this chapter:

> $\mathbf{G}_\sigma$: Find a design for a shared object memory that can be used for communications between separate execution threads. Assume latencies of several hundred to several thousand instruction-execution-times for fetching information from the shared object memory.

By chosing to use multiple heavy-weight threads to hide memory latency, I have excluded the 'micro dataflow' approach to handling long memory latency. This has been done because special hardware is needed to make micro dataflow work well (see the discussion in section 4.4), and because I feel the macro dataflow approach will be sufficient. Macro dataflow is just dataflow in which the basic data items are larger, typically at least a few hundred bytes each, and the basic operations are larger, at least a few thousand instruction-execution-times each. To the best of my knowledge, macro dataflow suffices for most (but not all) computer applications, while micro dataflow is necessary for only a small minority of applications. Micro dataflow may be easier to program for matrix applications, but macro dataflow may well be easier for text processing and compiling applications.

This chapter represents research into the possibilities of shared object memory, but it is in a less finished state than the research of previous chapters on other facilities in R-CODE. There are more loose ends, a few of which I will identify below, and the rest of which I do not know about yet.

5.2 Requirements

What kinds of operations can be performed on objects in shared object memory?

First, many objects will turn out to be read-only after they are created, just as many files are. The operations on these are creation, reading an object, and reading part of a large object.

For functional programming, it is also convenient to have objects that go through an initial write-only phase, during which many threads write them but cannot read them. Then the objects are switched to read-only, after which they cannot be written.

A histogram is similar but undergoes an accumulate-only phase, during which many threads may add to its elements. Then, when it is finished, it is switched to read-only.

However, it seems unreasonable to expect all components of an object to be in the same situation during one phase of the object. Some may be read-only while others are accumulate-only, for example.

This thinking leads to the following:

> $\mathbf{F}_8$: R-CODE will support a shared object memory in which individual object components can be be marked as read-only, write-only, write-once, or accumulate-only as part of the component type. Also, objects may change types dynamically, so their components can, for example, switch from write-only to read-only.

However, this approach is not always enough, so:

> $\mathbf{F}_9$: R-CODE shared object memory will support atomic multi-object transactions.

Special hardware will be necessary to make a real networked shared object memory, but this is not available, even approximately, today. However, shared memory *symmetric multi-processors* have been available for some time, and are working their way toward student computers (see $\mathbf{H}_2$, page 14). This leads to:

> $\mathbf{F}_{10}$: R-CODE shared object memory will be efficient on existing symmetric multi-processors, but may require specially designed hardware for efficiency on a within-building network.

Lastly there is an issue which I have not currently addressed very well, and do not list as a requirement, because I do not know whether or not it needs to be a separate requirement. This is the issue of how to do searches efficiently for the within-building shared object memory. I simply have not yet studied this problem sufficiently.

There may be other such issues that I have not yet identified.

5.3 Main Problems

There are several major problems that need to be solved by the shared object memory.

One of these is *cache coherency*: in a within-building system keeping caches coherent seems unreasonable.

A second problem is *thread synchronization*: when a thread runs out of things to do, it stops, and something must restart it and direct its attention to new things to do. In a within-building shared object memory, searching for things to do may be inefficient.

A third problem is *write delay*: in order to sequence writes to a shared memory, one must wait for earlier writes to complete before doing later writes, but this wait time can be unreasonably long for a within-building system.

A fourth problem is *atomic transactions*: threads need to perform operations that that lock some data, read and write the data, and then unlock the data.

A fifth problem is *hotspots*: if every thread in a building tries to read the same memory location or increment the same counter, the memory location to be read or incremented becomes a bottleneck, or 'hotspot'.

Shared object memory must offer solutions to these problems, both on existing symmetric multi-processors, and on future within-building systems. In the latter case, special hardware will be required, but it should be not be unnecessarily complex.

In the following subsections I introduce proposed solutions to each of these problems.

5.3.1 The Cache Coherency Problem

Symmetric multi-processors may have small incoherent primary caches, while within-building systems will have large incoherent caches. Both systems may have write buffers that delay writes, but I will assume these are small in both cases.

Most objects in shared object memory simply go through a write-only phase followed by a read-only phase. One problem for incoherent caches is how to handle this phase transition.

In a current symmetric multi-processor, the phase transition is handled by instructions that flush delayed writes to memory and invalidate incoherent cache entries used to read shared memory. These operations are fairly fast, and can reasonably be used to implement phase transitions. One of the reasons these operations are fast is that the larger caches in such a system are coherently maintained, although the cost of doing this is beginning to become great enough that coherency is becoming an “option”.

On future within-building computer systems, the local caches for a shared object memory may be many megabytes in size, and will not be coherently maintained. The caches are so large that invalidating an entire cache is unreasonable. One approach is to have instructions that will invalidate only particular regions of the cache, but even this might be time consuming.

A different approach, which I propose for actual use, is a cache using virtual object addresses that consist of an object number and a displacement within the object. This has the advantage that objects can be moved in memory without modifying the caches, and provides a necessary alternative to the address registers of section 2.4.2 (page 34) for permitting such movement. Keeping address registers coherent in such a large distributed system would be difficult, but with virtual address caching this is not necessary.

The phase transition of an object is handled by giving the object a new object number and invalidating the old object number. Because the new object number will not be in the virtual address caches, there is no need to invalidate any cache entries.

Invalidating the old object number is only needed to detect serious programming errors. Normally an optimistic system can be used that invalidates an object number in a few thousand instruction-execution-times while the application program keeps on running. This assumes programming errors will not occur just during this invalidation delay; either they will not occur at all, or they will also occur outside the delay and be found and fixed. Sometimes a pessimistic system that delays use of the new object number until the old object is withdrawn may help debugging.

Some objects have some components that can change any time, and are therefore *volatile*. We will discuss these in conjunction with thread synchronization in the next section.

5.3.2 The Thread Synchronization Problem

There is a standard system for synchronizing threads that is commonly used for symmetric multi-processors. This system, which I will call *standard synchronization*, uses two operations, *WAIT* and *NOTIFY*. WAIT delays a thread until the thread has been notified by a NOTIFY. More precisely, when a thread executes a WAIT, it stops until some thread has executed a NOTIFY targeted to the waiting thread at some time since the waiting thread executed its previous WAIT (i.e., the WAIT instruction execution previous to the current WAIT instruction execution).

Generally any thread can wait for some value to appear in memory by executing a loop in which it first checks whether the value is in memory, and then, if it does not find the value, does a WAIT followed by a repeat of the loop. We will call this the *standard synchronization loop*. Any thread that writes the value into memory must NOTIFY all the threads that might be waiting for that value after the value is written.

It is up to the programmer to maintain lists of threads that might be waiting for a value. As this can be a major efficiency issue, there is no end to the ways of doing this in different situations.

Often considerable data is transmitted between threads by writing the data and then writing a flag that indicates the data has been written. The writer must flush delayed writes after writing the data and before writing the flag, and then must flush delayed writes after writing the flag to be sure the flag is promptly visible. The reader must invalidate incoherent caches before looking for the flag, to be sure the flag is promptly visible. And the reader must invalidate incoherent caches after reading the flag and before reading the data, to be sure the data read is the data that existed when the flag was written.

The reason I call this "standard synchronization" is that it is widely used both with symmetric multi-processors and to synchronize any processor with its I/O devices (the device registers are shared memory, and traps are notifications).

In a within-building shared object memory, most of the data goes through a write-only to read-only phase transition as described above. Assuming virtual address caches are used, it will not be necessary to invalidate caches in order to read current data. However, invalidating the cache entry containing the flag needs to be done in order to see the current flag value, as the caches in a within-building system will not be coherent.

Invalidating the cache entries of flags is a problem since each cache miss required to check a flag to see if there is work to be done will cause a multi-thousand instruction-execution-time delay. To this problem I propose the following solution.

Certain object components, namely the flags, are marked *volatile* in R-CODE so they can be processed specially by the R-CODE compiler. For a within-building system there is a separate special per-thread *volatile cache* for volatile components. This cache is invalidated at the beginning of the standard synchronization loop.

The problem with standard synchronization loop using this cache is mostly that the loop will want to look at many volatile values to decide what to do. Waiting for one or two cache misses is not a problem, but waiting for 10 or 100 is inefficient. So we will try to process all these cache misses in parallel without changing the basic structure of the standard synchronization loop.

To do this, instructions that read volatile values behave specially on a cache miss. Instead of waiting for the cache to be loaded, they immediately return a special value to indicate that the value is currently missing. We will call this special value the *missing value* (it might actually be a non-signaling-NaN as in section 4.5.13).

The code that processes the volatile value should do the appropriate thing when confronted with a missing value: namely nothing. The loop will go on to read more volatile values. The first time through the loop will execute many read instructions that cause cache misses and initiate within-building shared object memory reads. The volatile component read instructions will return missing values, and the loop will find nothing to do.

The volatile cache will remember whether it has delivered any missing values since the beginning of each synchronization loop. If it has, then at the end of the loop, the loop will repeat without doing a WAIT. It will continue doing this until the loop sees no more missing values, at which time, if the loop has found no work, it will execute a WAIT.

The volatile cache must be large enough to hold all the volatile values required by one loop. Furthermore, cache entries must not be purged onced loaded during a loop, so the cache should be fully associative. However, a large fully associative hardware cache is not necessary; a smaller non-associative hardware cache could be supported by a software, since the R-CODE compiler knows which read instructions are reading volatile values, and can maintain a software cache of successes.

The advantage of all this is that most within-building network reads for volatile

values will be done in parallel, being scheduled by the cache misses occurring during the first iteration of the loop. Thus the entire standard synchronization loop will run in approximately a single within-building network latency delay: typically a few thousand instruction-execution-times.

As an optimization, the loop can delay somewhat between successive iterations, if the processor can switch threads fast enough to use the time efficiently.

To improve the semantics, the volatile cache should deliver nothing but missing values after it has delivered the first such value in a loop, even if it has actual values to deliver. This ensures that work which was of lower priority and whose flags are therefore checked later in the loop is not scheduled ahead of work of higher priority checked earlier in the loop, just because the higher priority flag values arrived latter through the building-wide network.

Among the many possible ways of handling the thread synchronization problem, I have chosen this one because it is compatible with standard synchronization, and because it requires only simple special hardware.

5.3.3 The Write Delay Problem

After an object created in the functional programming style has all its components written, the object is converted to read-only, and a read-only pointer to the object is returned to some potential user of the object. However, if the read-only pointer is written to shared memory and then read and used by another processor, it is possible the other processor will read some part of the object before the writes to that part of the object are finished. If this happens, the second processor will read wrong data.

On the other hand, if the original object creating processor were to wait until all component writes completed before returning the read-only pointer, it would have to wait a few thousand instruction-execution-times for a within-building system. This is an unreasonable overhead for creating an arbitrary object.

When data is being written to be received by other processors, generally the data is written first, and then some flag is written to indicate the presence of the data. In the above case, the read-only pointer might be the flag. More commonly, a cluster of new objects is created, and a flag is written indicating the presence of the entire cluster.

The solution I propose to the inefficiency of waiting for writes to finish is the following.

Code is separated into a partition that writes data followed by a partition that writes flags. At the boundary between these particular types of partition, the processor waits for all writes outstanding in the network to complete. Thus a single network latency time delay will occur only when flags are about to be written.

5.3.4 The Atomic Transaction Problem

An atomic transaction locks some data, reads and writes it, and then unlocks the data. One problem with such transactions is that they are inefficient if the lock/unlock operations take very long, or if read-only atomic transactions have to wait on each other's locks. A second problem is that all of the data involved is volatile, and accessing it could take a long time in a within-building system.

One example of an atomic transaction is adding an item to a queue. Another example is adding an item to a directory. A problem with the latter example is that searching the directory to find out whether an item to be added is already there may involve reading long lists and take a long time.

For symmetric multi-processors I propose an implementation of atomic transactions based on the *two counter locking* scheme of Lamport[Lam77]. Sets of data are protected by *count locks* that consist of a pair of aligned 32-bit counters that can be atomically read or written. The counters are called the *pre-counter* and *post-counter*. The data sets may be as small as a queue header or as large as a big directory.

The pre-counter is incremented by a data set writing routine before it does the write, the post-counter is incremented after the write is done, and the counters are equal when no write is in progress. A reader reads and remembers the post-counter first, copies information from the data set, and then reads the pre-counter and checks for equality with the saved value of the post-counter. If there is inequality, the data read may be garbage, and the reader retries.

The data read by a reader may not be consistent if reading was overlapped by a write. A reader may check at any time to see if the data is consistent by reading the pre-counter and checking for equality with the saved post-counter. If the two are unequal, the data read may be inconsistent, and the the reader will fail.

An atomic transaction that updates one or more data sets goes through a read phase followed by a write phase. During the read phase, the transaction reads and remembers the post-counters of data sets, reads data from each data set after reading the

set's post-counter, and makes a *deferred write list* of the locations in the data sets to be written and the values to be written.

During the write phase, the transaction raises to high priority on its local processor to lock out competing transactions on that processor, acquires a test-and-set lock for each data set to lock out competing transactions on other processors, tests the data set pre-counters for equality with the saved post-counter values, and if there is equality, performs the writes indicated by the deferred write list before clearing the test-and-set locks and lowering to the normal priority level. If the counter equality test fails, the transaction must retry.

Read-only transactions, and the read phase of updating transactions, do not lock out each other. The write phase of a transaction is the only part that conflicts with other transactions, and it is very short, since the writes are pre-planned by the deferred write list. However, this is an optimistic locking scheme, and it can thrash in the presence of a large number of writes, so it may be desirable to detect thrashing and have a transaction do something special when thrashing occurs, such as reserving data sets for a time interval normally long enough to include the read phase of the transaction.

The next problem is how to do all this in a within-building shared object memory. The following solution is quite simple.

The lock counters and components of the data sets being locked are treated as volatile components, and cached in the volatile cache. This cache is completely invalidated just before each read of a post-count. This guarantees that all components read after the post-count will be at least as recent as the post-count. Here all volatile reads wait for their data to be delivered before completing: they do not return missing values as reads did in the standard synchronization loop above.

In the write phase, a test-and-set lock is set for each data set, and then the volatile cache is invalidated again, the pre-counts are all read and compared with the saved post-counts, and if the pre-counts are equal, the deferred write list data is written, before clearing the test-and-set locks. Also, the write phase must occur at an appropriate priority level to lock out transactions by different threads on the same processor.

As an example, for a queue with a count lock in the same object as the queue data set components, there would be (1) a network read to obtain the object with the post-count and data set elements, (2) a second network read to obtain the pre-count

in order to verify that the data set elements read are valid, (3) a network test-and-set, (4) a third network read for the pre-count, (5) overlapping write operations to process the deferred write list, and (6) a clear of the test-and-set lock. Thus a queue operation would take about six network latency times. Some of the six steps just given could be combined by optimizations: e.g. acquiring the test-and-set lock and reading the pre-count.

For a simple operation like maintaining a small queue, it might be better to simply do the test-and-set at the beginning, returning the object contents with the result that tells whether the test-and-set was successful. It might also be possible to batch clearing the test-and-set flag with component writes in a single operation. But in the more general case, as for a large directory with many objects involved in the locked data set, the more general approach just described would have to be used.

5.3.5 The Hotspot Problem

A hotspot occurs when many processors attempt to read the same location, or perform some accumulate operation such as addition on the same location.

To solve the hotspot problem for a building-wide system, I assume that network nodes can be built that will combine requests from several processors into a single request on the rest of the system, and when the response comes back from the single request, will redistribute the response, appropriately modified, to the original requesters.

For example, if a network node receives three requests to read the same block of memory from three processors, it can make one request on the rest of the system to read the memory, and then distribute the results to the requesting processors.

As another example, if a network node receives requests from two processors to add to a memory location, it can sum the requests and send the sum to be added to the location. If the result returned from such an operation is supposed to be the location value before any addition, then when the node gets the result, it can send it to one processor, and send to the second processor the sum of this result and the value added by the first processor.

It is important in this latter scheme that the value returned by a request to accumulate to memory be the value before the accumulate. Trying to return the value after accumulation makes it impossible for a network node combining processor requests to compute values to be send to processors unless the operation is invertible. Not all

operations are invertible: taking the maximum is not.

I assume that at some future time it will be reasonable to equip network nodes with RISC processors with small amounts of memory. For example, a network node might be a single integrated circuit, and the RISC processor might be part of that circuit. Accumulate operations can be implemented by functions whose only variable input is the values they are combining. These functions may have small amounts of constant input, such as small tables, that may be loaded into various network nodes to permit these nodes to perform the accumulate operations.

The only thing that R-CODE does to compensate for hotspots is provide a design that permits combining network nodes to be built easily.

5.4 Principals

R-CODE shared object memory is based on several principals that are described in this section. These mesh somewhat with the discussion in the previous sections on shared object memory problems.

5.4.1 The Ordered Partition Model

It is assumed that programs are divided into partitions that are executed in some well defined order, but that memory operations within a partition are unordered. This model conforms to the partition model in Chapter 4 (see section 4.5.5, page 138). It also conforms to barrier models that are widely used in parallel programming (e.g. the BSP model[Val90, GV94]).

5.4.2 Commutative/Associative Operations

The memory operations within a partition must be reorderable without changing the semantics of the program. This means these operations must be commutative and associative.

Operations on different memory components have these properties. Thus if a partition has as set of component write operations, all on different components, these operations commutate and associate.

Strictly read operations have this property. If all operations on one component in a partition are read operations, all these read operations commute and associate. They will all return the same value.

Typical accumulate operations, such as addition or taking the maximum, commutate and associate, even on a single component.

5.4.3 Type Change

Individual objects are expected to go though phases whose boundaries are marked by object type changes. Each phase must complete before the next phase starts. The phases may be implemented by the partitions mentioned above.

For example, a typical functional programming object will have a write-only phase, followed by a read-only phase. In the write only phase, one write operation is executed for each component. In the read-only phase, each component may be read as often or as little as desired.

A histogram may go through three phases: write-only, accumulate-only, and, after it is complete, read-only.

5.4.4 Per Component Access Disciplines

There are different disciplines for accessing a component within an object phase. Example disciplines are read-only, write-only, and accumulate.

Different components of an object may have different access disciplines within the same phase of the object. Thus some components may be read-only, while others are write-only, and still others are accumulate.

5.4.5 Volatile Component Caching

A volatile object component is a component that may change value independently of the thread looking at it.

Volatile components are handled by a special volatile cache. This cache is not automatically coherent with shared memory, and has a special cache invalidate instruction to clear the cache.

Special attention must be paid to organizing the program and the volatile cache to avoid unreasonable delays due to volatile cache misses.

5.4.6 Probabilistic Error Detection

Fatal errors, such as accessing an object that has been deleted, or accessing an object whose type has changed using an obsolete virtual address, are assumed not to occur in perverse patterns. Therefore, methods of detecting these that have a high but not perfect probability of detection should usually suffice.

5.5 Related Work

The use of small incoherent primary caches has been pioneered by several computer companies, for example Digital Equipment Corporation[Sit92].

Alternate methods of thread synchronization that involve read instructions which wait for empty words to become full may be found in the work of Burton Smith[Smi78, ACC+90] and Arvind[Nik91, Pap90].

The notion of a shared object memory with caches that use the virtual address of the objects has been explored in the MUSHROOM project[WW93].

5.6 R-CODE Shared Object Memory Semantics

In this section we will give some details of the R-CODE implementation of shared object memory.

5.6.1 Access Disciplines

In R-CODE, each component load or store treats the component as having one of the following *access disciplines*:

read-only
write-only
write-once
accumulate
transaction
volatile

Below we will describe how R-CODE specifies which access discipline to use with a particular load or store, and how each of the access disciplines behave.

5.6.2 Access Specifications

In R-CODE, the access discipline used to load or store a component is determined by the *access specification* that is part of the type field in the R-CODE pointer to the component.

R-CODE component load and store instructions define the component to be loaded or stored by means of an R-CODE pointer, formatted as in Figure 5.1. This pointer is a virtual value: in contains the relevant information, but the information is not actually store in this form during program execution. The access specification is part of the type field of the pointer.

The rest of the type field includes the numeric type and size of the component being accessed, and this information must be compiled into instructions on typical modern computers. Similarly the access specification must be compiled into instructions. When either the type or access specification information is not available at compile time, load and store instructions turn into dynamic case statements that switch on the type field value and compile new cases of themselves when they see new type field values. This permits the type field information to be compiled into machine instructions even when it is not known at compile time (see page 82 for more discussion of dynamic case statements).

Some of the access specifications correspond directly to the different access disciplines, and some have a different use. The READ-ONLY, WRITE-ONLY, VOLATILE, WRITE-ONCE, and ACCUMULATE access specifications correspond directly to access disciplines. The NEUTRAL and TRANSACTION access specifications are inputs to an operation which will be described below that combines a pointer to an object or subobject with a component descriptor. The OPEN, OPEN-READ-ONLY,

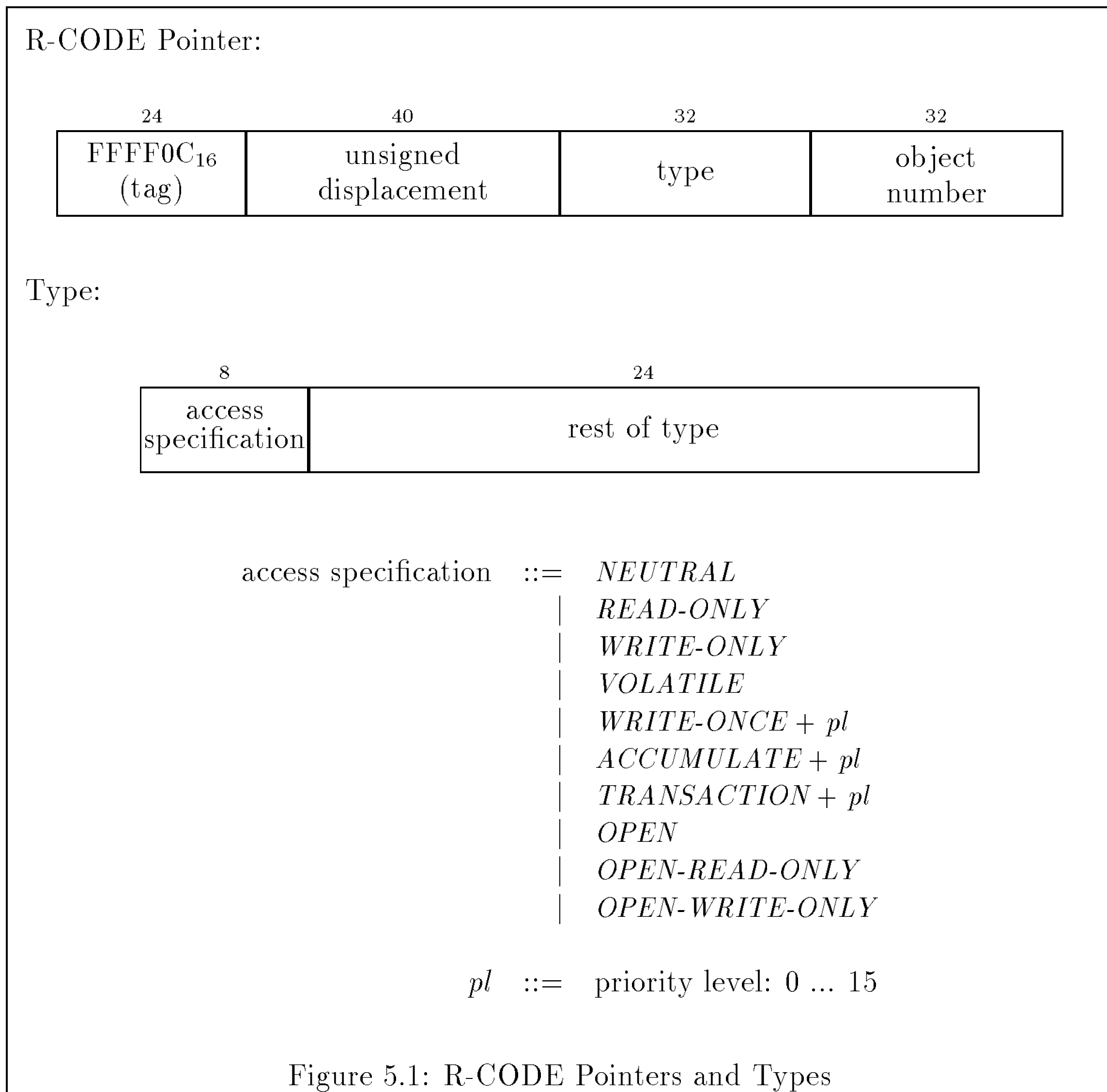


Figure 5.1: R-CODE Pointers and Types

and OPEN-WRITE-ONLY are outputs from this combination operation, and are used as component access specifications for the transaction access discipline. The word "OPEN" signifies that the transaction has obtained the required count locks (by merely reading the post-count), so that the locks are "open".

Several access specifications include a priority level. These access specifications cannot be used by any thread running at a higher priority level. Implementations are therefore free to lock out other threads on the same processor by raising to the priority level indicated by the access specification. Usually the operation governed by the access specification is short enough that the access specification priority level can be so high that any thread can use the specification.

5.6.3 Access Specification Combination

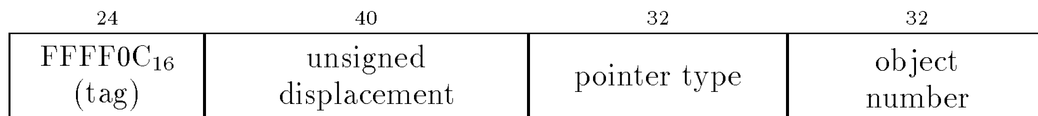
The component pointer used by a load or store instruction is usually created by combining a subobject pointer with a component descriptor that describes how to access a component of the subobject: see Figure 5.2. The component pointer output by this combining operation takes its object number from the subobject pointer, and its displacement from the sum of the subobject pointer and descriptor displacements. The type of the output component pointer is taken from the component descriptor, except for the access specification, which is derived by combining the access specifications of the subobject pointer and component descriptor according to the table in Figure 5.3.

The blank entries in this table are errors, as is any combination of access specifications not listed in this table.

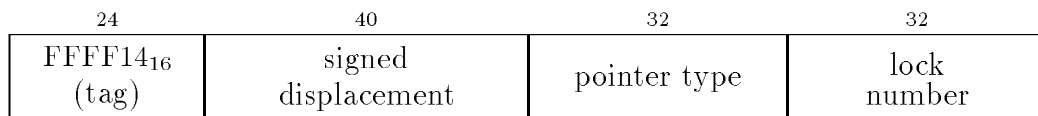
If the subobject pointer access specification is NEUTRAL, the component pointer access specification is taken from the component descriptor. The exception to this rule is that a TRANSACTION component descriptor produces an OPEN component pointer, and also obtains a lock (see "The Atomic Transaction Access Discipline" below, page 173).

READ-ONLY and WRITE-ONLY subobject pointers always force the access spec of the component pointer to be READ-ONLY or WRITE-ONLY, respectively. This feature is commonly used to initialize a subobject by using a WRITE-ONLY pointer to make all its components WRITE-ONLY. This saves creating a separate type map (see section 3.4, page 84) for initialization. The companion feature of forcing all components to be READ-ONLY can be similarly used to implement an object type

R-CODE Pointer:



R-CODE Component Descriptor:



Combination of Pointer and Component Descriptor:

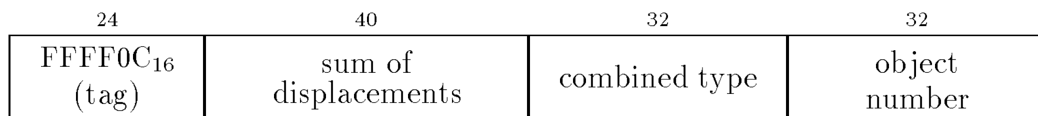


Figure 5.2: Combining R-CODE Pointers and Component Descriptors

.

	Subobject Pointer Access Specification					
Component Descriptor Access Specification	NEUTRAL	WRITE ONLY	READ ONLY	OPEN	OPEN WRITE ONLY	OPEN READ ONLY
NEUTRAL	NEUTRAL	WRITE ONLY	READ ONLY	OPEN	OPEN WRITE ONLY	OPEN READ ONLY
WRITE ONLY	WRITE ONLY	WRITE ONLY	READ ONLY	OPEN WRITE ONLY	OPEN WRITE ONLY	
READ ONLY	READ ONLY	WRITE ONLY	READ ONLY	OPEN READ ONLY		OPEN READ ONLY
VOLATILE	VOLATILE	WRITE ONLY	READ ONLY			
WRITE ONCE	WRITE ONCE	WRITE ONLY	READ ONLY			
ACCUM- ULATE	ACCUM- ULATE	WRITE ONLY	READ ONLY			
TRANS- ACTION	OPEN	WRITE ONLY	READ ONLY			

Figure 5.3: Access Specification Combination

change that makes an entire object read-only without requiring an additional type map.

Other details on the access specifications are given below with the discussion of particular access disciplines.

5.6.4 The Volatile Cache

R-CODE has a special per-thread cache for volatile components, called the *volatile cache*. For a within-building system, this would be special hardware. For a symmetric multi-processor with an incoherent primary cache and a cache invalidate instruction, the primary cache effectively implements the R-CODE volatile cache.

The volatile cache is not coherent with shared memory. There is a special *INVALIDATE-CACHE instruction* that clears the volatile cache. Writes may or may not update the volatile cache.

Read instructions use the volatile cache when the access specification for the component being read is VOLATILE, WRITE-ONCE, ACCUMULATE, OPEN, or OPEN-READ-ONLY (these last two are for transactions).

The cache has the following missing value capabilities that permit implementation of standard synchronization loops as described in section 5.3.2 above.

Read instructions have an optional *cache miss modifier* that indicates the instructions should return a special *missing value* immediately instead of waiting if the volatile cache suffers a cache miss during a read. Even in this case the cache will issue shared memory operations to fill itself. Also, in this case, the cache will be big enough and associative enough to hold all volatile data in a synchronization loop without losing any due to cache entry conflicts. Special code may be compiled for read instructions with a cache miss modifier to provide software support to a hardware cache in order to implement this size capability.

The cache sets a *missing value returned flag* if it returns any missing values, and this flag can be read and set. The cache has a *missing value control flag* that if set causes the cache to return only missing values if the missing value returned flag is set, even if the cache has actual non-missing values to return.

5.6.5 The Read-Only Access Discipline

A component accessed using the *read-only access discipline* can only be read. Furthermore, it is assumed that such a component cannot be changed, and therefore maintenance of cache coherence for such components is unnecessary if the cache uses virtual addresses as described in section 5.3.1.

Read-only components can only be the result of an object type change in which a previously writable component was made read-only.

5.6.6 The Write-Only Access Discipline

A component accessed using the *write-only access discipline* can only be written. Furthermore, it is assumed that in general there will be at most one write of the component within a program partition where memory operations are unordered.

Note there may be implementation difficulties in permitting two components to be written independently if these are both part of the same byte, for example. To overcome these, writes of unaligned parts of memory may be much slower than one would expect.

There may be partitions of a routine that are specially marked so that they do not start until all the write operations in the previous partition are done. This is the sole method of sequencing writes.

5.6.7 The Volatile Access Discipline

A component accessed using the *volatile access discipline* can only be read. Only caches that are coherent immediately after an INVALIDATE-CACHE instruction can be used for such reads.

See sections 5.3.2 and 5.6.4 above for more details on reads using the volatile cache.

5.6.8 The Write-Once Access Discipline

A component accessed using the *write-once access discipline* can be read or written, but must be a tagged datum that is initialized to the special *OMITTED* value.

A write to a write-once component will succeed if the previous value was the OMITTED value, but will not write anything otherwise. In any case, the previous value is returned, and can be checked to see if it was the OMITTED value or if it is equal in the appropriate sense to the value that was to be written.

The access specification used to write a write-once component contains a priority level. Some implementations raise to this priority level or higher to write the component. Threads of higher priority cannot write the component.

Reads of the component may attempt to read the component from the normal non-volatile cache. If the value so read is the OMITTED value, then the read must read the component from using the volatile cache. After a non-OMITTED value is read, it will not change, and may be recorded in a non-volatile cache.

5.6.9 The Accumulate Access Discipline

A component accessed using the *accumulate access discipline* is written by using a special accumulate routine. The component may also be read in a fashion identical to that of the volatile access discipline.

An *accumulate routine* combines two component values and produces a new output component value. The component values may be register values or contiguous subobject values (i.e., aligned bit strings stored in memory). The accumulate routine can take additional arguments that are constants, again either register values or contiguous subobject values. It must be possible to copy the accumulate routine and these additional arguments throughout a network to combine values, in order to apply the method of section 5.3.5 above to control hotspots.

A write instruction to an accumulate component returns the previous value of the component.

The access specification used to write an accumulate component contains a priority level. Some implementations raise to this priority level or higher to write the component. Threads of higher priority cannot write the component.

5.6.10 The Atomic Transaction Access Discipline

A component may be read or written inside an atomic transaction by using the *transaction access discipline*.

Atomic transactions are initiated and terminated by special instructions. During an atomic transaction, a *list of acquired locks* is kept. Every time a NEUTRAL subobject pointer is combined with a TRANSACTION component descriptor to produce an OPEN pointer, a lock designated by the descriptor and pointer is located, acquired, and added to the lock list. It is also possible that the lock is already on the list, in which case it need not be acquired a second time or added to the list a second time. Thus an OPEN pointer is a pointer to a component whose lock has been acquired.

The lock acquired is identified by the *lock number* in the component descriptor. Typically this lock is a component of the same object as the component being accessed.

An OPEN component may be read or written. Writes to an OPEN component do not actually change the component, but instead make an entry in a *deferred write list* which causes the write to be done when the atomic transaction terminates.

Locks in this system are acquired optimistically, meaning that conflicting atomic transactions may acquire locks simultaneously. As a consequence, the values read by an atomic transaction for OPEN components may be incorrect, because a different atomic transaction may have written those components since the current atomic transaction acquired their locks. There is a validate operation that may be executed at any time to see if the current atomic transaction is still valid, i.e. no OPEN component has been written since its lock was acquired. At the end of the atomic transaction, if the transaction is still valid, all delayed writes are done atomically; whereas if the transaction has become invalid, it fails and no writes are done.

The OPEN-WRITE-ONLY access specification is just like OPEN but excludes reads. OPEN-READ-ONLY is just like OPEN but excludes writes.

When a TRANSACTION component descriptor is used to acquire a lock, a check is made to be sure the current atomic operation was initiated at a priority level below that specified by the component descriptor.

Often an entire subobject inside some object is presented as a TRANSACTION component of the containing object. Then an OPEN pointer to this subobject is computed and the proper lock acquired, and this OPEN pointer is used to make OPEN-READ-ONLY or OPEN-WRITE-ONLY pointers to components of the subobject without acquiring additional locks. For example, a queue header might be such a subobject.

The actual locking scheme used it that described in section 5.3.4. The volatile cache is implicitly used and cleared during an atomic transaction.

5.7 Summary

The goal of this chapter has been to define a shared object memory that might become a standard for writing parallel text processors, compilers, games, and simulations. Unlike the other chapters in this thesis, the results presented here are more preliminary, and it is more likely that there are either undiscovered flaws or a better way.

Bibliography

[ACC+90] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings 1990 International Conference on Supercomputing*, pages 1–6, September 1990.

[ANS83] ANSI, Washington, DC. *Reference Manual for the Ada Programming Language*, 1983.

[App94] Apple Computer. *Dylan: Interim Reference Manual*, 1994. Available by http://www.apple.com and see 'Technology and Research' and then Dylan.

[Bro84] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *1984 ACM Symposium on LISP and Functional Programming*, pages 256–262, August 1984.

[CN83] Jacques Cohen and Alexandru Nicolau. Comparison of compacting algorithms for garbage collection. *ACM Transactions on Programming Languages and Systems*, 5(4):532–553, October 1983.

[Coh81] J. Cohen. Garbage collection of linked data structures. *Computing Surveys*, 13(3):341–367, September 1981.

[CPW74] S. S. Coleman, P. C. Poole, and W. M. Waite. The mobile programming system, Janus. *Software Practice and Experience*, 4:5–23, 1974.

[Def94] Defense Research Agency, Attn. Dr. N. E. Peeling, St. Andrews Road, Malvern, Worcestershire, UK WR14 3PS. *TDF Specification*, 1994.

[Dig94] Digital Equipment Corp. *Migrating to an OpenVMS AXP System: Planning for Migration*, March 1994. Order No.: AA-PV62A-TE.

[E+95] John H. Edmondson et al. Internal organization of the Alpha 21164, a 300-Mhz 64-bit quad-issue CMOS RISC microprocessor. *Digital Technical Journal*, 7(1):119–135, 1995.

[F+95] David M. Fenwick et al. The AlphaServer 800 series: High-end server platform development. *Digital Technical Journal*, 7(1):43–65, 1995.

[GV94] Alexandros V. Gerbessiotis and Leslie G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22:251–267, 1994.

[H+92] P. Hudak et al. Report on the programming language Haskell. *SIGPLAN Notices*, 27(5):Section R, 1992.

[Hay94] Barry Hayes. Key objects in garbage collection. Technical Report CS-TR-94-1510, Stanford, August 1994.

[HC84] Tim Hickey and Jacques Cohen. Performance analysis of on-the-fly garbage collection. *Communications of the ACM*, 27(11):1143–1154, November 1984.

[HF92] P. Hudak and J. Fasel. A gentle introduction to Haskell. *SIGPLAN Notices*, 27(5):Section T, 1992.

[HMS92] Anthony L. Hosking, J. Eliot B. Moss, and Darko Stefanovic. A comparative performance evaluation of write barrier implementations. In *OOPSLA 92*, pages 92–109, 1992.

[HW78] B. K. Haddon and W. M. Waite. Experience with the universal intermediate language Janus. *Software Practice and Experience*, 8:601–616, 1978.

[Inc92] Sparc International Inc. *The SPARC Architecture Manual, Version 8*. Prentice Hall, 1992.

[Int94] International Computer Science Institute, Berkeley, California. *The Sather 1.0 Specification*, 1994. Available by ftp to icsi.berkeley.edu.

[Jon94] Mark P. Jones. The implementation of the Gofer functional programming system. Technical Report YALEU/DCS/RR-1030, Yale University, Department of Computer Science, May 1994.

[JW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Symposium on Principles of Programming Languages*, pages 71–84. ACM, January 1993.

[Lam77] Leslie Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.

[M+91] Gerald Masini et al. *Object Oriented Languages*. Academic Press, 1991.

[Maca] Stavros Macrakis. From UNCOL to ANDF: Progress in standard intermediate languages. http://www.osf.org and see Grenoble Office Papers.

[Macb] Stavros Macrakis. The structure of ANDF: Principles and examples. http://www.osf.org and see Grenoble Office Papers.

[Mey92] Bertrand Meyer. *EIFFEL: The Language*. Prentice Hall, 1992.

[MS94] Scott Milton and Heinz W. Schmidt. Dynamic dispatch in object-oriented languages. Technical Report TR-CS-94-02, Australian National University, Canberra, January 1994.

[Nik91] Rishiyur S. Nikhil. ID language reference manual. Technical Report Computation Structures Group Memo 284-2, MIT, July 1991.

[NOPH92] Scott Nettles, James O'Toole, David Pierce, and Nicholas Haines. Replication-based incremental copying collection. In Y. Bekkers and J. Cohen, editors, *Memory Management: IWMM 92*, pages 357–364. Springer-Verlag LNCS 637, 1992.

[NR87] S. C. North and J. H. Reppy. Concurrent garbage collection on stock hardware. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 113–133. Springer-Verlag LNCS 274, 1987.

[Org83] E. I. Organick. *A Programmer's View of the Intel 432 System*. McGraw-Hill, 1983.

[Pap90] Gregory Michael Papadopoulos. *Implementation of a general purpose dataflow multiprocessor*. MIT Press, Cambridge MA, 1990.

[Sit92] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, 1992.

[SKW93] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: An efficient, portable persistent store. In Anotonio Albano and Ron Morrison, editors, *Persistent Object Systems, San Miniato 1992*, pages 11–33. Springer-Verlag, 1993.

[Smi78] Burton J. Smith. A pipelined, shared resource MIMD computer. In *Proceedings of the International Conference on Parallel Processing*, pages 6–8. IEEE, August 1978.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison Wesley, 1994.

[U+93] David Ungar et al. *How to Use SELF 3.0*. Sun Microsystems and Stanford University, 1993. Available by ftp to self.stanford.edu or self.smli.com.

[Ume91] Kyoji Umemura. Floating-point number LISP. *Software Practice and Experience*, 21(10):1015–1026, October 1991.

[UU94] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Conference on Programming Language Design and Implementation*, pages 326–335. ACM, June 1994.

[Val90] Leslie G. Valiant. A bridging model for parallel computations. *Communications of the ACM*, 33(8):103–111, August 1990.

[VD93] Francis Vaughan and Alan Dearle. Supporting large persistent stores using conventional hardware. In Anotonio Albano and Ron Morrison, editors, *Persistent Object Systems, San Miniato 1992*, pages 34–53. Springer-Verlag, 1993.

[Wal] Robert L. Walton. R-CODE documentation and papers. ftp://das-ftp.harvard.edu/pub/walton/rcode/rcode.html and in Harvard University Tech Reports to appear.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In Y. Bekkers and J. Cohen, editors, *Memory Management: IWMM 92*, pages 1–42. Springer-Verlag LNCS 637, 1992.

[WK95] Wm. A. Wulf and Sally A. KcKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, March 1995.

[WW93] Mario Wolczko and Ifor Williams. Multi-level garbage collection in a high-performance persistent object system. In Anotonio Albano and Ron Morrison, editors, *Persistent Object Systems, San Miniato 1992*, pages 396–418. Springer-Verlag, 1993.

[YHS] Taiichi Yuasa, Masami Hagiya, and William Schelter. GNU Commonlisp. ftp://prep.ai.mit.edu.

[Zor93] Benjamin Zorn. The measured cost of conservative garbage collection. *Software Practice and Experience*, 23(7):733–756, July 1993.

Index

abstract type, 86
access discipline, 165
access specification, 80, 99, 166
ACCUMULATE, 167
accumulate, 166
accumulate access discipline, 173
accumulate routine, 173
actual object, 86
actual type, 17, 85, 89
    identifier, 99
    specification, 95
    specification example, 96
actual type map, 89
address, 25
address register, 34
    loading, 36
    overhead, 39
    storing, 37
ALIGNED, 102
aligned, 74
alignment, 99
    of object, 75
ANY
    the formal type, 93
argument list, 122, 129, 131
array descriptor, 18, 107
Arvind, 123, 165
ASCII Text Code, 4, 10
AT-CODE, 4, 10
atomic operation, 112
atomic transaction
    problem, 155, 160

barrier instruction, 116, 138
barrier operation, 111–113
base
    of array descriptor, 108
big endian, 74
bit field, 75
bitstring AND write barrier test, 62
block, 118
block instruction, 142
block return instruction, 142
Brooks, 42

$\mathbf{C}_1$, 12
$\mathbf{C}_2$, 13
$\mathbf{C}_3$, 13
C++, 87
    deficiencies, 104
cache coherency
    problem, 155
cache miss
    overhead, 39
cache miss modifier, 171
call, 133
call cycle, 145
call instruction, 117, 139
case, 133, 137

case instruction, 133, 137
case statement, 120
class
    C++, 104
code generator, 8
Cohen, 24
compact, 27
component descriptor, 84
    constant, 99, 100
    contents, 98
    non-constant, 100
component index, 100
component size, 81
component-ID, 90
cons cell, 14
    overhead, 41
conservative garbage collector, 30
conservative pointer component, 30
constant component descriptor, 99
context, 12
contiguous subobject, 82, 103
continuation, 124
continuation call instruction, 148
conversion
    dynamic, 103
    endian, 75
    static, 103
copy, 27
copy type, 81, 99
copy-on-mark, 55
copy-on-scavenge, 55
copying collector, 28, 55
    the, 58
coroutine, 145
coroutine channel, 145
count lock, 160
create, 26
data types, 16
dataflow, 115
    register, 125
deferred action buffer, 63, 64
deferred write list, 161, 174
deletion
    manual, 30, 69
demand object swizzling, 29
demand pointer swizzling, 29
detection
    garbage, 43
dimension descriptor, 107
discontiguous subobject, 82, 104
displacement, 76, 98
    field in pointer, 82
done state, 135
dynamic case statement, 17, 82, 94
DYNAMIC-CONVERT, 103

eager, 120
eager forwarding, 28, 57
EIFFEL, 87, 106
enable level
    inlining, 144
endian
    transmission between different, 75
ephemeral garbage collection, 44, 69
ephemeral object, 44
ephemeral root, 44
ephemeral root pointer, 45
ERROR, 77
exception catch, 117
exception catch case, 139
exception throw, 117
exception throw instruction, 141
exception value, 117

exception value list, 117, 122, 129, 131, 139, 140
execution flow, 18, 116
execution tree, 120, 122, 133
  node states, 135
  transition rules, 135

$\mathbf{F}_1$, 16, 24
$\mathbf{F}_2$, 16, 24
$\mathbf{F}_3$, 16, 24
$\mathbf{F}_4$, 18, 73
$\mathbf{F}_5$, 18, 73
$\mathbf{F}_6$, 18, 72
$\mathbf{F}_7$, 19, 111
$\mathbf{F}_8$, 20, 154
$\mathbf{F}_9$, 20, 154
$\mathbf{F}_{10}$, 20, 154
floating point number, 73
fork, 124
formal component descriptor, 90
formal type, 17, 85, 89
formal type map, 89
format
  memory unit, 76
forwarded, 28
forwarding, 28, 57
forwarding scenario, 28
frame, 26
frame heap, 129, 131, 133
frame memory, 128
frame pointer, 122
free block, 25
from-space, 28, 55
full garbage collection, 44
full/empty bit, 123
functional language, 18

$\mathbf{G}_\alpha$, 1, 22, 72
$\mathbf{G}_\beta$, 1
$\mathbf{G}_\iota$, 71, 86
$\mathbf{G}_\mu$, 22
$\mathbf{G}_\rho$, 110
$\mathbf{G}_\sigma$, 153
gap, 25
garbage collection, 16
garbage detection, 43
gc cycle, 68
gc time overhead, 68
generational garbage collection, 45
global memory
  address registers in, 38

$\mathbf{H}_1$, 13
$\mathbf{H}_2$, 14
$\mathbf{H}_3$, 14
$\mathbf{H}_4$, 14
$\mathbf{H}_5$, 15
hardware trends, 13
HASKELL, 87, 88, 106
Hayes, 24
heap memory, 128
HEP, 123
high performance operation, 2
histogram
  example, 112
Hölzle, 84
Hosking et al., 67
hotspot
  problem, 155, 162

$\mathbf{I}_e$, 46, 47, 54
$\mathbf{I}_l$, 63
$\mathbf{I}_s$, 46, 47, 54
ID, 123
increment

inlining enable, 144
inline, 117, 144
inlining
controlling, 123
inlining enable increment, 144
inlining enable level, 144
inlining priority, 144
instruction, 114, 126
instruction execution tree, 133
integer, 73
interoperate, 7
interrupt check, 41
INVALIDATE-CACHE
instruction, 171
invariant
ephemeral marking, 45
marking, 44, 46

join, 124

lattice element, 85
lazy, 120
lazy forwarding, 28, 58
length, 25
limit
of array descriptor, 108
LISP
floating point, 77
load convert, 103
load operation, 99
load-address operation, 99
load-root, 26
local heap, 63
localization priority, 145
localize, 144
lock list, 174
lock number, 174
loop, 118
loop instruction, 143

M bit, 46
$\mathbf{M}_1$, 11
$\mathbf{M}_2$, 11
$\mathbf{M}_3$, 11
manual deletion, 16, 30, 69
invalidating object number, 156
strongly typed, 30, 31
mapped type, 79
mark, 25
mark-copy-scavenge, 55
marked, 46
marked bit, 25
marking, 25
invariants, 46
marking algorithm
ephemeral conditions, 46
non-ephemeral conditions, 46
standard, 43
marking algorithms, 43
match flag, 102
match type map, 86, 90, 99
memory, 25
memory management, 15
memory manager, 7
memory unit, 74
format, 76
memory unit size, 81, 82
memory value vector, 132
memory vector flag, 132
memory wall, 14
methodologies, 11
Milton, 87
missing value, 158, 171
missing value control flag, 171

missing value returned flag, 171
monad, 124
MONSOON, 123
Monsoon, 123
Moss et al., 67
move
    dynamic, 16, 31
multi-process single-processor, 36
multi-processor
    shared memory, 14, 20, 38, 154
MUSHROOM, 165
mutator, 26
mutator action time, 67
mutator test time, 67

Nettles et al., 34, 60
networks, 15
NEUTRAL, 167
no permission, 121, 135
NO-CONVERT, 103
non-signaling-NaN, 118, 146, 150
non-snapshot, 47
    ephemeral detector, 45
non-strict, 119
North and Reppy, 34, 58
not-ephemeral-root, 44, 46
not-ephemeral-root bit, 44
NOTIFY, 157
NREVERSE, 51
null, 25

object, 25
object map, 31
    hardware, 33
object number, 31, 77
offset
    within memory unit, 81, 82
OMITTED, 77, 172
OPEN, 167
OPEN-READ-ONLY, 167
OPEN-WRITE-ONLY, 167
operation, 114
out-of-line, 117
overhead
    address register, 39
    non-mutator, 67
    subroutine call, 14
    write barrier, 64

packing problem, 115
paging
    an object, 19
partition, 133, 138
permanent, 46
permanent bit, 44
permanent object, 44
permission
    propagate, 122
pointer, 25
    128-bit tagged, 78
    64-bit tagged, 77
pointer component, 25
polymorphic, 17
pop, 27
post-counter, 160
pre-counter, 160
priority
    inlining, 144
    localization, 145
priority level
    for copying, 38
priority scheduling
    when not allowed, 38
propagate

permissions, 122
push, 27

R-CODE, 4, 7
feature selection, 15
RAM
as I/O device, 115
reach, 25
reachable, 25
read, 26
memory, 120
read barrier, 49
read barrier detector, 49, 50, 53
read-barrier-forwarding, 57, 60
read-free permission, 135
READ-ONLY, 167
read-only, 166
read-only access discipline, 172
real-time, 16
reduced pointer, 81
reference, 26
register, 125
register code, 4, 19
register dataflow, 114, 125
register dataflow code, 114
register dataflow language, 111
register frame, 129
register frame tree, 130
register number, 129
register value vector, 132
replication-forwarding, 59, 60
result list, 122, 129, 131
return instruction, 116
block, 142
special, 141
return operation, 113
root set, 25
routine execution, 133
routine execution tree, 130
routine map number, 79
routine type, 79, 83

S bit, 46
S-CODE, 3, 4
SATHER 0.5, 106
SATHER 0.6, 87, 106
scalar type, 79, 81
scanning, 25
scavenge, 25, 26
scavenged, 46
scavenged bit, 26
scavenger, 17, 64
Schmidt, 87
SELF, 84
shared heap, 63
shared memory
multi-processor, 14, 20, 154
shared object memory, 20, 153
side effect, 120
side-effect permission, 121, 135
side-effect-free permission, 121, 135
signature, 132
size, 98
slot, 29
Smith, 123, 165
snapshot, 47, 48
ephemeral detector, 45
snapshot detector, 53
source analyzer, 8
space, 27, 55
special return instruction, 141
stack, 26
standard synchronization, 157
standard synchronization loop, 157

state
  maintenance, 136
  of execution tree node, 135
  recording, 148
STATIC-CONVERT, 103
step
  of array descriptor, 108
stopping
  during object copying, 38
store a pointer, 47
store convert, 103
store operation, 99
strict, 119
subobject address, 103
subroutine
  call overhead, 14
superscalar, 13
surface code, 3
sweep, 26
swizzled, 29
swizzling, 16, 29
  demand object, 33
  demand pointer, 33
swizzling scenario, 29
symmetric multi-processor, 154
synchronize, 64

$\mathbf{T}_e$, 49
$\mathbf{T}_{ns}$, 49
$\mathbf{T}_{ss}$, 49
tag, 77
tagged data, 18
tagged value
  128-bit, 78
  64-bit, 76
  in register, 125
TERA, 123
termination, 117
thread synchronization
  problem, 155, 157
to-space, 28, 55
top frame, 26
TRANSACTION, 167
transaction, 20, 154, 166
transaction access discipline, 173
transition rules
  execution tree state, 135
trap, 118, 147, 150
trap flag, 31
  set for move, 31
two counter locking, 160
two level addressing, 31, 60
type
  field of pointer, 79
type code, 128, 132
type map, 17, 18, 84
  constancy, 94
  issues, 86
type map number, 77, 79, 96
type matching
  dynamic, 88, 89, 91
  static, 88
type matching operation, 86, 88, 89
type variable, 95
type-change, 30
  strongly typed, 30, 32
`typeinfo` in C++, 104

Umemura, 77
UNCOL, 8
Ungar, 84
unifying clausal grammar, 6
universal glue, 2
UNKNOWN, 102

unsigned8, 81
used object, 69

V-CODE, 4, 9
value list, 122, 129, 131
virtual base, 104
virtual computer, 114
virtual function table
    in C++, 104
virtual instruction, 78
virtual object, 86
virtual register, 78
visualization code, 4
visualization program, 9
vocalization code, 4
vocalization program, 9
VOLATILE, 167
volatile, 156, 158, 166
volatile access discipline, 172
volatile cache, 158, 171

WAIT, 157
Wilson, 24
withdraw permission, 121, 135
withdrawn, 135
within-building system, 153
write, 26
write barrier, 49
    bitstring AND test, 62
    R-CODE, 61
write barrier detector, 49, 50, 53
    end thrashing, 51
    ending, 50
write delay
    problem, 155, 159
write-barrier-forwarding, 58, 60
WRITE-ONCE, 167
write-once, 166
write-once access discipline, 172
WRITE-ONLY, 167
write-only, 166
write-only access discipline, 172