(12)

Technical Report 1321

Concurrent Smalltalk on the Message-Driven Processor

Waldemar Horwat

MIT Artificial Intelligence Laboratory

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE September 1991

3. REPORT TYPE AND DATES COVERED technical report

4. TITLE AND SUBTITLE

Concurrent Smalltalk on the Message-Driven Processor

5. FUNDING NUMBERS

N00014-87-K-0825
N00014-87-K-0738
MIP-8657531

6. AUTHOR(S)

Waldemar Horwat

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Artificial Intelligence Laboratory
545 Technology Square
Cambridge, Massachusetts 02139

8. PERFORMING ORGANIZATION REPORT NUMBER

AI-TR 1321

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Office of Naval Research
Information Systems
Arlington, Virginia 22217

10. SPONSORING MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

None

12a DISTRIBUTION/AVAILABILITY STATEMENT

Distribution of this document is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Million-transistor processors are being manufactured today, and soon it will be possible to put several million transistors on one integrated circuit. While memory applications of this technology are clear, it is not obvious how best to use it for computation purposes. One possibility is the architecture of the Message-Driven Processor (MDP), which consists of a 32+4-bit CPU, memory, and a network interface together on one chip. MDPs can be connected directly to each other to form a 65536-processor, message-passing, MIMD, parallel computer, the J-Machine. The MDP's architecture is unusual in that it provides a very high processing power to memory ratio.

(continued on back)

14. SUBJECT TERMS (key words)

compiler message passing
parallel processing object oriented

15. NUMBER OF PAGES 240

16 PRICE CODE

17 SECURITY CLASSIFICATION OF REPORT	18 SECURITY CLASSIFICATION OF THIS PAGE	19 SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED

Block 13 continued:

Concurrent Smalltalk is the primary language used for programming the J-Machine. Concurrent Smalltalk is the the language of choice because it fits the J-Machine's fine-grain, message-passing model well. This thesis describes Concurrent Smalltalk and its implementation on the J-Machine, including the Optimist II compiler and Cosmos operating system. Optimist II can perform global optimization of programs, including inline function expansion, type inference, and global evaluation of constant expressions. Next, Cosmos and the Concurrent Smalltalk runtime environment are described. Finally, some quantitative and qualitative results are presented. The grain size (the average amount of time a method executes before suspending) was found to be about 60 instructions, and the MDP was found to execute one instruction every two or four cycles, depending on whether external DRAM is used. A number of qualitative issues are described, along with a few preliminary results for addressing difficult problems such as controlling parallelism.

Concurrent Smalltalk on the Message-Driven Processor

Waldemar Horwat

May 12, 1989
Updated September 26, 1991

This report describes research done at the Artificial Intelligence Laboratory at the Massachusetts Institute of Technology. Funding for this project was provided in part by the Defense Advanced Research Projects Agency and monitored by the Office of Naval Research under contracts N00014-88K-0738 and N00014-87K-0825. Funding was also provided by a National Science Foundation Presidential Young Investigator Award, grant MIP-8657531, with matching funds from General Electric and IBM Corporation. The author was supported by an ONR Fellowship.

Keywords: Compiler, Concurrent Smalltalk, CST, Fine-grained, Message-Driven Processor, Message-passing, Massively Parallel, Object-oriented, Parallel Processing, Programming Language

Concurrent Smalltalk on the Message-Driven Processor

by
Waldemar Horwat

Submitted to the Department of Electrical Engineering and Computer Science on
May 12, 1989 in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Updated September 26, 1991

Abstract

Million-transistor processors are being manufactured today, and soon it will be possible to put several million transistors on one integrated circuit. While memory applications of this technology are clear, it is not obvious how best to use it for computation purposes. One possibility is the architecture of the Message-Driven Processor (MDP), which consists of a 32+4-bit CPU, memory, and a network interface together on one chip. MDPs can be connected directly to each other to form a 65536-processor, message-passing, MIMD, parallel computer, the J-Machine. The MDP's architecture is unusual in that it provides a very high processing power to memory ratio.

Concurrent Smalltalk is the primary language used for programming the J-Machine. Concurrent Smalltalk is the the language of choice because it fits the J-Machine's fine-grain, message-passing model well. This thesis describes Concurrent Smalltalk and its implementation on the J-Machine, including the Optimist II compiler and Cosmos operating system. Optimist II can perform global optimization of programs, including inline function expansion, type inference, and global evaluation of constant expressions. Next, Cosmos and the Concurrent Smalltalk runtime environment are described. Finally, some quantitative and qualitative results are presented. The grain size (the average amount of time a method executes before suspending) was found to be about 60 instructions, and the MDP was found to execute one instruction every two or four cycles, depending on whether external DRAM is used. A number of qualitative issues are described, along with a few preliminary results for addressing difficult problems such as controlling parallelism.

Thesis Supervisor:	William J. Dally, Ph.D.
Title:	Associate Professor of Computer Science and Engineering

Acknowledgements

I would like to thank:

Professor Bill Dally for his support during the difficult times as well as the easy ones and for advice without giving orders.

Andrew Chien for providing many insightful comments while this work was being done, reviewing drafts of this thesis, and sharing his ideas about the development of abstractions in concurrent programming.

Scott Wills for providing suggestions and taking his time to show me around the MIT purchasing bureaucracy.

Carl Manning for providing feedback on Concurrent Smalltalk and other ideas.

The co-authors of Concurrent Smalltalk, Andrew Chien and Scott Wills, and the original creator of the language, Bill Dally, for making a wonderful language for programming the J-Machine.

Peter Nuth for his well-developed, Canadian sense of humor and a healthy attitude towards MIT, as well as for his numerous contributions to the MDP project.

The other hardbeaners, Stuart Fiske, John Keen, Rich Lethin, Mike Noakes, and Debbie Wallach for working long hours striving to make the MDP chip a success.

Brian Totty for writing the operating system on which Cosmos is based.

Lucien Van Elsen for providing relevant ideas about micro-optimization of floating point calculations on the MDP.

Ricardo Jenez and John Wolfe for knowing everything about MIT's computers, sharing that knowledge, and for connecting UNIX computers to the Macintosh.

Ellen Spertus for getting the first dataflow compiler to work on the MDP and for rescuing this thesis by getting Word 4.0 directly from Microsoft in less than twenty hours.

Kathy Knobe for listening well and suggesting ideas for further research.

Scott Furman, Todd Dampier, and Shaun Kaneshiro for debugging the J-Machine, bringing up Cosmos on it, and adding extensions to it.

Steve Keckler and Larry Dennison for being friends.

Anant Agarwal, my advisor, for advice.

The Office of Naval Research for providing a fellowship with no strings attached and amazingly little bureaucracy.

Also, I would like to thank Bernard and Maria Horwat, my parents, for their love and support which made my education possible.

Most of all, I would like to thank God for the abilities and opportunities He has granted me.

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

Goals

This thesis describes the Concurrent Smalltalk language and its implementation on the Message-Driven Processor. Concurrent Smalltalk, also known as CST, is a concurrent version of the object-oriented programming language Smalltalk [20]. The implementation consists of a global, optimizing compiler and a streamlined operating system for the J-Machine.

This thesis covers quite a broad scope of the implementation of Concurrent Smalltalk, including subjects ranging from issues in parallel programming in general and the design of Concurrent Smalltalk itself to some of the fine points of the design and optimization of the MDP architecture. The goal of the thesis is to demonstrate a working implementation of Concurrent Smalltalk on the Message-Driven processor. Although the implementation is not yet complete, it does provide hooks for all of the advertised functionality of Concurrent Smalltalk and is based on solid ground. Versions of the implementation are running on recently manufactured MDP chips, and I hope that the programs described herein will survive and evolve for the next five years.

Another goal of this thesis was to discover and, whenever possible, fix design flaws in the MDP architecture and language specification so as to make an implementation of Concurrent Smalltalk practical. Several errors in the MDP architecture and Concurrent Smalltalk were found, as well as numerous bugs in the simulation tools used to verify the hardware.

The next section gives a brief overview of the J-Machine hardware and the Concurrent Smalltalk language. It is followed by an outline of the software bridging the gap between Concurrent Smalltalk and the MDP hardware—the Optimist II compiler and the Cosmos operating system. The relationship of this work to others' in fine grain concurrent computation is then described.

Second Edition

This work was originally a Master's thesis completed in May 1989. It has been updated for the state of Optimist II compiler, Cosmos operating system, and MDPSim 7.0 simulator as of the end of May 1991. The Optimist II compiler now produces better code, and several Cosmos routines, especially the CFUT fault handler, have been sped up. Furthermore, Cosmos has been updated for a few minor architectural revisions.

The compiler and operating system have been evolving rapidly in the past few months due to the recent availability of MDP chips. This document does not include these newest changes, which include support for hardware I/O, debugging aids, and workarounds for first-silicon chip bugs, as they have little effect on the ideas in this work. Other members of the Concurrent VLSI Architecture group, including Scott Furman, Rich Lethin, Todd Dampier, Shaun Kaneshiro, John Keen, and Mike Noakes, are now working on CST applications and Cosmos enhancements such as floating-point arithmetic, queue overflow handling, and garbage collection. These will be published in separate documents as they are completed.

1.1. Hardware and Software Architecture

The J-Machine

Million-transistor processors being manufactured today, and soon it will be possible to put several million transistors on one integrated circuit. While memory applications of this technology are clear, it is not obvious how best to use it for computation purposes. One possibility is the architecture of the Message-Driven Processor (MDP), which consists of a 32+4-bit[1] CPU, memory, and a network interface together on one chip. MDPs can be connected directly to each other to form a 65536-processor, message-passing, MIMD, parallel computer, the J-Machine [14]. The network is a three-dimensional mesh fast enough to provide communication between the farthest pair of processors on a 65536-processor J-Machine in a few microseconds—on an unloaded network an 8-word message can be transmitted from one corner of the J-Machine to the other in just 4 microseconds. The processors are optimized for sending and receiving messages; a processor can be working on a message even before the entire message has arrived. The MDP's architecture is unusual in that it provides a very high processing power to memory ratio.

The Message-Driven Processor

The MDP has a register-based architecture and operates on 32-bit data words with 4-bit tags. Tags are essential in efficiently supporting late binding for object-oriented languages such as Concurrent Smalltalk. In addition, tags are necessary for garbage collection and valuable for debugging programs.

The MDP is message-based. In its normal mode of operation, the MDP listens on the network for messages. When it receives a message from the network, it stores the message in a FIFO input message queue and dispatches on the address given in the first word of the message. Messages are used for all communication tasks, including function and method calls, replies, object transfers, and other synchronization facilities.

A detailed but slightly obsolete description of the MDP architecture is in [16]; a updated summary is presented in Appendix D. MDPSim [24] [25] is an instruction level simulator, assembler, and debugger used to run MDP assembly language programs and test the operating system.

Concurrent Smalltalk

Concurrent Smalltalk is the primary language used to program the J-Machine. One of the main goals of designing Concurrent Smalltalk was to take advantage of the J-Machine's unique features. A new software architecture was needed that would efficiently support fine-grain, message-passing computation. Whereas some existing parallel computers have message routing times measured in milliseconds, the routing time for a message sent from one end of even a large J-Machine to another is on the order of several microseconds. Operating system overhead on processing and dispatching that message of more than a few microseconds is not acceptable.

Concurrent Smalltalk introduces concurrency to standard Smalltalk by evaluating arguments to method calls in parallel as well as allowing the computation of the value of a variable to proceed in parallel with the other computations of a method until the variable's value is actually needed. Furthermore, Concurrent Smalltalk adds *distributed objects* to Smalltalk. A distributed object is an object that can process many methods at the same time without any serialization bottlenecks other than those required by the algorithm in use. Although

---

[1] Each word consists of 32 bits of data and a 4-bit tag.

standard objects can also process several methods simultaneously, they can only dispatch on one method at a time[1].

Concurrent Smalltalk is an ideal language for programming the J-Machine because it is easy to parallelize and yields small, fine-grain methods as well as a considerable amount of flexibility in the system software implementation. The methods dealing with a particular class can travel to the data object as opposed to the data traveling to the code. Concurrent Smalltalk also provides excellent facilities for creating data abstractions; the Optimist II compiler amplifies this power by providing global optimizations so performance does not suffer because abstractions are used.

Another advantage of Concurrent Smalltalk is that it is low-level enough to be useful in implementing parts of the J-Machine runtime system, while being at a level high enough that the programmer does not have to worry about the infamous problems of parallel process synchronization and deadlocks. In fact, once the data structures are defined properly, programming in Concurrent Smalltalk feels much like programming in a standard sequential language.

---

[1] This restriction is relaxed for immutable standard objects because they may be copied at the operating system's discretion. Nevertheless, a distributed object can be mutable and still have no synchronization bottlenecks.

1.2. Overview

Foundations

Some of the pieces comprising the Concurrent Smalltalk environment were available before this thesis was done. A primitive compiler was available [21], as were a description of the operating system kernel [38], several descriptions of the language [13] [21] [17], and an MDP assembly language simulator (MDPSim 5.2) [24]. Unfortunately, none of the pieces really fit together—the various versions of the language were inconsistent, the output of the compiler was incompatible with the untested operating system kernel, which itself was written for an obsolete version of the MDP architecture [23].

It became clear that it would be easier to design the language, the compiler, and the operating system from scratch than to try to fit the existing pieces together. Nevertheless, the existing code and ideas were useful as guides to which approaches would likely yield good results and which techniques should be abandoned. I took advantage of this opportunity to extend Concurrent Smalltalk to support several programming styles and add functions, closures, continuations, arrays, nested local variables, and inline classes to produce a language with a compact implementation yet powerful libraries. The new features did not complicate implementation; in fact, by providing a small set of fundamental primitives, the new features often simplified the implementation of existing functionality, a phenomenon noticed in the design of the Scheme language [31] [1].

The contributions of this thesis include:

- A redesign of the Concurrent Smalltalk language.
- *Optimist II*, a new Concurrent Smalltalk compiler and interpreter.
- *Cosmos* (*Co*ncurrent *Sm*alltalk *O*perating *S*ystem), an operating system that supports Concurrent Smalltalk on the MDP.
- Runtime libraries for Concurrent Smalltalk.
- Modifications to MDPSim, the MDP assembler/simulator, to facilitate downloading programs, simplify debugging, and collect performance measurements.
- Modifications to the MDP architecture that make it more suitable for Concurrent Smalltalk.

I am indebted to Scott Wills and Andrew Chien for helping with the redesign of the Concurrent Smalltalk language, and Richard Lethin, John Keen, and Stuart Fiske for helping with the MDP architecture changes. Professor William Dally supervised the project.

System Overview

The Optimist II Compiler

The Optimist II compiler continues in the tradition of the Optimist compiler by compiling Concurrent Smalltalk to assembly code that is as small as possible without sacrificing speed. In addition, Optimist II contains an interactive Concurrent Smalltalk interpreter that is useful for prototyping and debugging Concurrent Smalltalk programs at the source level. Optimist II is also a platform for experimenting with compiler optimizations. Global optimizations such as function inlining and the reduction of method calls to function calls were added and found to be highly successful.

The compiler itself is divided into several phases, which are described in more detail in Chapter 3. It produces an MDPSim command file which can be downloaded into MDPSim and run on a simulated J-Machine.

Cosmos

Cosmos is the operating system used on the Message-Driven Processor to support code output by Optimist II. Many of the ideas in Cosmos are borrowed from JOSS [38] written by Brian Totty—JOSS introduced the concept of a Birth/Residence Address Table (BRAT) and the protocol for migrating object between processors. Nevertheless, Cosmos's code bears little resemblance to JOSS.

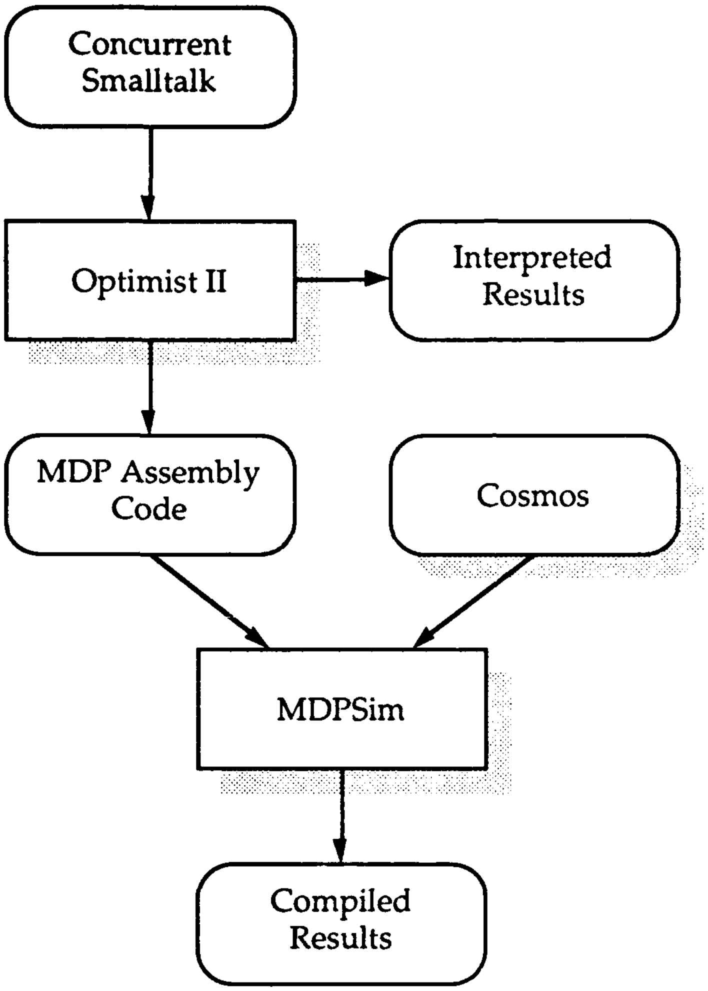


Figure 1-1. Software Environment Organization

A Concurrent Smalltalk program can be either compiled or interpreted by the Optimist II compiler. Interpretation is useful to debug Concurrent Smalltalk programs and interactively experiment with language features. When a Concurrent Smalltalk program is compiled, it is loaded into MDPSim, a J-Machine simulator, together with the Cosmos operating system. MDPSim will then run the program to obtain its results as well as program performance statistics.

The main goals of Cosmos were to make a working operating system, make it as efficient as possible, and make it simple, all subject to the time constraints of a Master's thesis. Those three goals have been achieved to a large extent, in that the operating system does work, and simple programs have been run on it. Unfortunately, controlling a large parallel computer is a difficult task, and Cosmos still falls short in many ways which are described in Chapter 8. In particular, higher-level resource management and load balancing issues are yet to be adequately addressed. Nevertheless, Cosmos is a good start and a platform for experimenting with the more difficult problems.

Example

A very simple example of the use of the system to compile and run a factorial program is listed below. Please refer to chapter 5 for a more detailed example of the transformations in the compiler and Appendices B and C for information about using the compiler and the operating system.

```
CST:(defun fact (n)
     (if (<= n 1)
       1
       (* n (factorial (- n 1)))))
#<Cst-Lambda 5090060 FACT>
CST:(fact 3)

When interpreting: (FACT 3)
Error: Unbound global FACTORIAL
> Break:
> Type Command-/ to continue, Command-. to abort.
1 > Continuing...Fatal error: Can't apply #<Nil>
> Break:
> Type Command-/ to continue, Command-. to abort.
1 > Continuing...
CST:(defun fact (n)
     (if (<= n 1)
       1
       (* n (fact (- n 1)))))
#<Cst-Lambda 4920924 FACT>
CST:(fact 4)
#<Integer 24>
CST:(compile fact "NewFact.mdp")
```

Figure 1-2. Compiling Fact

The user entered a factorial function, corrected an error in it, tested it on a sample input, and then compiled it into MDP assembly code in the NewFact.mdp file. The user's input is shown in bold.

First the user starts the compiler and enters the compiler's interactive mode (see Appendix B) as shown in Figure 1-2. He enters the fact function and runs it only to find an error—fact's recursive call should be to fact, not factorial. The user corrects the error and then uses the compiler's interpreter to successfully compute the factorial of 4.

Afterwards the user compiles fact to MDP assembly code, quits Optimist II, and launches MDPSim, where he loads the object file, and calls fact on 4 to get the correct answer—24 (Figure 1-3). The `stats` command can then be used to determine some running statistics, such as the frequencies of instructions executed, the amount of parallelism used, and the total time taken to run the program. Starting from a cold start, fact takes 725 steps on a 2×2×1 J-Machine to compute its answer.

Implementation

The Optimist II compiler is written in CLOS [27], the Common Lisp Object System. Except for the use of the LOOP iteration macro [7], Optimist II adheres to standard Common Lisp as specified in [35] and amended in [6] and in the amendments specified by the Common Lisp Cleanup Committee that were available at the time of this writing. The LOOP macro is itself written in standard Common Lisp, so Optimist should run on any machine with a faithful implementation of Common Lisp. A slightly modified version of the 12/7/88 version of Xerox's PCL was used to implement a subset of CLOS before Apple Common Lisp 2.0 became available.

```
MDPSim -x 2 -y 2 -msize 0x1000 ::Cosmos:Cosmos.m NewFact.mdp

Message-Driven Processor Simulator
Version 7.0 Rev B
Accompanies MDP Architecture Document 11B
Written by Waldemar Horwat
Architecture Updates by Brian Totty and Jerry Larivee
UROPs for Bill Dally

4 MDPs present.

@0..3}MESSAGE fact4
Message}MSG:msgApplyFunction|5
Message}{fFact}
Message}4
Message}IONODE
Message}0
Message}END
@0..3}inject fact4@3
@0..3}resetstats
@0..3}run
Tick   724   Received priority 0 message:
    OBJ:$801BF004 u=1 f=0 offset=$006EC=Reply length=$0004
    INT:$0000FC00 = 64512
    INT:$00000000 = 0
    INT:$00000018 = 24

@0..3}stats
725 ticks executed.
... More statistics ...
```

Figure 1-3. Running Fact

The user loaded the fact object code and typed a few magic incantations that invoked the fact function on the input 4 (the third word in the injected message). The result 24 (the fourth word in the ejected message) was returned after 725 steps on a 4-node J-Machine. Most of the time was spent distributing the fact code throughout the J-Machine; the second time it only takes 498 steps to compute the answer (some code is still being distributed), the third time takes 289 steps, and afterwards the execution time is about 265 steps.

Optimist II was developed on a Macintosh using Apple Common Lisp 1.2.2 and 2.0 written by Coral Software Corp (now merged with Apple Computer, Inc.). It runs on a 5-megabyte Macintosh II, although 8 megabytes are recommended and at least 16 are needed to run Optimist II and MDPSim simultaneously.

Cosmos is written in MDP assembly language [16]. MDPSim [24] [25] was used as an assembler and simulator for a small J-Machine.

All of the software needed to compile and run Concurrent Smalltalk programs exists on both a Macintosh II platform and on Sun workstations.

Results

The primary result of this work is a demonstration of a working implementation of Concurrent Smalltalk on a J-Machine. In addition, a number of secondary results were obtained. These include the qualitative and quantitative benefits of optimizations in the Optimist II compiler, data on the expected grain size (the number of instructions executed in response to a message), and a number of qualitative observations about the shortcomings of the current system. The results did not always come out as expected. For example, the finding that the grain size is about 60 instructions was surprising; it was expected to be much lower. Code statistics indicate that the MDP will take about 1.9 cycles per instruction, although most instructions execute in 1 cycle; if slow external DRAM is used to hold user programs and data, the MDP could take as many as 3.5 cycles per instruction. Network loading calculations indicate that network congestion will become a concern when the size of the J-Machine exceeds

343 nodes; either a faster network or some means of exploiting locality will be needed for larger J-Machines.

The quantitative results are listed in Chapter 7, while the qualitative ones are in Chapter 8. Chapter 8 may seem a little pessimistic, but many of the current shortcomings listed there would not have been found had this work not been done; furthermore, the current implementation of Cosmos provides a great, highly accurate platform for research into the issues presented there.

Caveats

Due to the availability of only a finite amount of time for writing this thesis, which could potentially involve an infinite amount of work, some features could not be included in the current implementation of Concurrent Smalltalk. The biggest omission is the lack of garbage collection—if enough storage isn't reclaimed, the machine will fail. Garbage collection, although interesting, was omitted to keep this project to a reasonable size—a good garbage collector and load manager would require more effort than is desirable for a Master's thesis.

Full futures were also not implemented. They were omitted from the interpreter in the compiler because simulating them is difficult on a sequential machine in a sequential language (Common Lisp). Futures were omitted from the run-time system because of the considerable amount of work needed to implement all the fault handlers and special cases involved. Nevertheless, almost all Concurrent Smalltalk programs still attain reasonable parallelism through the use of cfutures[1], which are fully operational.

Other features that were not implemented are I/O facilities at both the Optimist II and Cosmos levels and runtime support for local (non-distributed) arrays and floating point numbers. I/O facilities, while useful, do not contribute much to the project and are easy to add later. Local arrays and floating point numbers are supported by the Optimist II compiler but not the runtime system; supporting them at the runtime level will require writing MDP assembly language; no major surprises are expected there.

Some of the optional features of Concurrent Smalltalk were not included due to a lack of time. All class inline declarations are currently ignored; I anticipate that it will be possible to inline objects inside other objects sometime in the future, but that is not a high priority at this time. The omission of class inlining does not change the semantics of Concurrent Smalltalk programs. Function inlining is more useful, and it does work now.

Reading Guide

The remainder of this chapter describes related work in fine-grain concurrent computation. The succeeding chapters delve into various aspects of the system, starting from the top—Chapter 2, **Concurrent Smalltalk**, provides an introduction to the Concurrent Smalltalk language in general. Chapter 3, **The Optimist II Compiler**, describes the Concurrent Smalltalk compiler and interpreter. Chapter 4, **The Cosmos Operating System**, describes the operating system. To avoid overlap, the compiler features documented in [21] are not documented here; thus, it might be helpful to consult [21] when reading Chapter 3.

Chapter 5, **Sample Program**, traces the progress of a sample program from the Concurrent Smalltalk source level down to object code. Chapter 6, **Debugging**, provides some debugging techniques for Concurrent Smalltalk and MDP programs. Chapters 7, **Performance Measurements**, and 8, **Future Evolution**, present the results of this work. Chapter 7 contains quantitative measurements of the performance of Cosmos and the compiled code, while Chapter 8 describes some of the less tangible, qualitative shortcomings of the current system and ideas for correcting them. Chapter 9, **Conclusion**, concludes the main body of the thesis.

---

[1] A cfuture, also called a context future, is a local future which cannot be passed outside the function without being touched (i.e. replaced by its value).

The appendices parallel the main chapters with more detailed information. Appendix A, **Concurrent Smalltalk Reference**, is the most important, for it contains the specification of Concurrent Smalltalk. Appendix B, **Using Optimist II**, provides a detailed description of the Optimist II features not listed in Appendix A. Similarly, Appendix C, **Using Cosmos**, is a guide to running Cosmos on MDPSim; the latest MDPSim reference manual [25] should also be consulted when running Cosmos. Appendix D, **MDP Architecture Summary**, summarizes the current version of the MDP architecture. Finally, Appendix F, **Cosmos Listing**, contains a listing of the entire operating system.

Since this thesis also serves as a reference manual for Concurrent Smalltalk, Chapter 2 and Appendices A and B have been indexed. The index appears at the end of the thesis.

1.3. Related Work

The ideas of optimizing Smalltalk and running object-oriented software on concurrent, fine-grain systems are not new, but they have not been integrated previously to the extent found on the J-Machine. While most of the efforts concentrated on either optimizing Smalltalk for conventional computers or developing radically new programming methodologies, Concurrent Smalltalk presents a somewhat conventional Smalltalk environment to the programmer (with a few new features such as futures and distributed objects), which is at the same time efficiently implemented on a fine-grain parallel computer.

A major contribution of this work is the actual optimized implementation of Concurrent Smalltalk on an assembly language architecture. While theoretical studies and simulations in higher-level languages can yield asymptotic and qualitative results, an implementation yields the constant factors determining a system's performance. These performance measurements are an important part of this work, as they indicate the relative costs of the primitive operations and can be used to gauge the true performance of a concurrent computer.

Smalltalk Systems

Smalltalk-80

Early Smalltalk-80 optimization efforts such as [18] concentrated on optimizing Smalltalk within the constraints of the byte code interpreter. In addition, the work was limited by the Smalltalk-80 constraints of making contexts and methods program-visible data structures, which required some effort to convert between the optimized and standardized versions of the structures. Several context optimizations are also presented in [18], including determining which contexts which can be referred to as first-class data objects and which contexts can be pointed by blocks. Most contexts do not fall into either category, and they can be placed on the stack. Such optimizations are now also commonly done in Lisp compilers [36].

Whereas early Smalltalk-80 implementations were constrained to compatibility with byte codes and were run on stack machines, Concurrent Smalltalk is bound by neither constraint. The formats of contexts and method code are not defined in the language, and there are no portable means to store a pointer to a context in a programmer-visible variable. Thus, Optimist II and Cosmos can use the most efficient format for a context or even several different formats if they so desire. Furthermore, the MDP is not a stack-based machine, so there are no clear advantages to determining which contexts will be live for a long time. Also, contexts are fully self-contained, so a closure cannot refer to a context. Finally, several techniques are used to optimize closures. As will be seen in Chapter 3, when a closure is created, either the lexical variables are copied into the closure, or a common object is made to which both the context and the closure refer.

Optimized Sequential Smalltalk

A few years later it became clear that global analysis and optimization were necessary to optimize Smalltalk programs further. Optimizing Smalltalk well required an ability to convert method dispatches into more efficient function calls, which led rise to several type systems for Smalltalk [5] [26]. When a type system could be applied to a Smalltalk program, the compiler could optimize it by a factor of 5 to 10 over interpreted Smalltalk. The main compiler optimizations of TS [26] are similar to those of Optimist II: Both TS and Optimist II can convert a message send into a case statement of procedure calls, substitute functions in-line, and optimize tail recursion. In addition, TS can beta-reduce blocks, which Optimist II currently cannot do. On the other hand, Optimist II contains a number of other powerful dataflow optimizations (see Chapter 3 and [21]) commonly found in C compilers, which make its assembly language output close to optimal. Moreover, Optimist II can evaluate large constant expressions at compile time, and it can infer types of variables, allowing it to produce good code even though type declarations in Concurrent Smalltalk are completely optional. TS, on the other hand, has difficulties combining typed code with untyped code.

The MDP hardware also plays an important role in making Optimist II efficient. By providing tags and checking them on primitive operations, the MDP architecture frees Optimist II from the difficult and often unrewarding process of analyzing programs trying to determine information such as whether an integer variable could contain a `large-integer` (an integer which does not fit into a single 32-bit word) or whether the arguments to + are known to be numbers. Although this information is generally difficult to determine, in most cases integers are small and the arguments to arithmetic primitives are usually numbers, so hardware tag-checking is the right approach to this problem. Thanks to the MDP hardware, even if Optimist II cannot determine the type of some expression, performance does not suffer too much.

CONCURRENTSMALLTALK

A recent language close to Concurrent Smalltalk and having an almost identical name is CONCURRENTSMALLTALK [39] [40] independently developed by Yasuhiko Yokote and Mario Tokoro. CONCURRENTSMALLTALK shares with Concurrent Smalltalk the cfuture facility (called a CBox in CONCURRENTSMALLTALK) and the ability to process messages asynchronously. In addition, CONCURRENTSMALLTALK defines atomic objects, which Concurrent Smalltalk does not have but can easily emulate using locks. On the other hand, Concurrent Smalltalk includes distributed objects, which CONCURRENTSMALLTALK does not provide. Furthermore, the implementation of Concurrent Smalltalk is more optimized. Whereas CONCURRENTSMALLTALK is implemented as a byte code interpreter, Concurrent Smalltalk compiles to assembly language.

The two languages have somewhat different flavors. CONCURRENTSMALLTALK is very close to Smalltalk-80, and most of the concurrent features are add-ons that have to be explicitly requested by the programmer. Concurrent Smalltalk makes concurrency the default, and the programmer has to explicitly request sequential processing if he wants it. At the same time, the MDP hardware assists Concurrent Smalltalk by making the use of concurrency very cheap. For example, a hardware tag is provided that implements cfutures in Concurrent Smalltalk using much less overhead than cboxes in CONCURRENTSMALLTALK.

In [40] several changes to the original CONCURRENTSMALLTALK are discussed. Blocks are treated differently depending on whether they were created by atomic objects' contexts or not. Concurrent Smalltalk's model of only having one kind of object and using locks where necessary to make atomic transactions does not lead to these difficulties. Finally, secretary objects were introduced to CONCURRENTSMALLTALK to keep track of which threads are waiting for a resource. An equivalent facility is used internally in locks in Concurrent Smalltalk.

Actor Systems

Another recent development in object oriented programming was the rise in actor systems [2]. An actor system is a programming paradigm in which simple self-contained entities called actors communicate with each other to run a program. Much of the program's content is held in the interconnections among the actors. From the implementation standpoint, Concurrent Smalltalk shares many of the ideas with actor systems, but the language itself is not designed exclusively as an actor language. Instead, Concurrent Smalltalk is as a language closer to Smalltalk and Lisp, but it is possible to write actor-like programs in Concurrent Smalltalk without too much trouble.

Cantor

Cantor [4] is both a programming language and a formalism for reasoning about the problems that arise in fine-grain, message-passing parallel computers. In Cantor each object (the Cantor equivalent of a Concurrent Smalltalk context) can only perform a bounded amount of computation on receiving a message, and that computation is atomic. Also, messages sent from one object to another are guaranteed to arrive in the original order. Concurrent Smalltalk is similar to Cantor at the implementation level—when a message is sent to a context, it performs a bounded amount of computation[1], perhaps sends a few more messages, and then either suspends or waits for the next message. The state of a computation is composed mostly of idle objects and messages traveling between objects, with only a few objects executing. Hence, at a superficial level, a Concurrent Smalltalk object code program is a Cantor program. Nevertheless, the Concurrent Smalltalk object code program is more complicated because it might fault while performing the computation of the next state. One can view this possibility as either computation being non-atomic or treating faults as if they were message sends and suspends, preserving the Cantor model. Another distinction is that Concurrent Smalltalk does not guarantee that messages between a pair of objects will arrive in the order in which they were sent.

Probably the best relationship between Concurrent Smalltalk and Cantor is that Concurrent Smalltalk is a high-level language that compiles to Cantor-like object code. At the source level, Concurrent Smalltalk frees the programmer from the myriad of error-prone synchronization details found in Cantor. Concurrent Smalltalk encapsulates the Cantor concept of future flow into a few easy-to-use primitives such as `touch` and `nconcurrently`. At the same time, Concurrent Smalltalk presents the appearance of global and nested data structures (such as lexical scoping of local variables) which are compiled into interacting objects.

Nevertheless, Cantor is a good theoretical model for computation on the J-Machine. For example, the load balancing and management results in [4] are expected to also apply to the J-Machine. However, the J-Machine can also suffer from problems not discusses in [4], such as having too much parallelism. Some of the load balancing issues are presented in Chapter 8.

Acore

Acore [30], an "actor core language," is another recent actor language. Like Cantor, it provides an environment in which a computation is done by interacting actors with limited abilities; however, actors in Acore can compute arbitrary functions to determine state, and Acore has a notion of a transaction (a message send and a reply), which greatly simplifies programming.

Acore and Concurrent Smalltalk are similar in many ways. Both languages implement message sends, replies, concurrent evaluation of subexpressions, local variables, static scoping, and instance objects (called actors in Acore). However, there are also a few differences. Due to its Smalltalk-80 heritage, Concurrent Smalltalk permits local variables to be altered, while Acore does not; both languages allow mutation of instance variables. In addition, Acore implements a sponsorship mechanism for higher-order control of the course of a computation and a complaint mechanism for handling exceptions. It remains to be seen whether these mechanisms will be necessary in Concurrent Smalltalk[2].

Acore is compiled into Pract, which is a form of an actor assembly language, whereas Concurrent Smalltalk is compiled into MDP assembly language. As a result of this difference, some actions which are cheap in one language are expensive in the other, which affects the language design. Actor creation is very cheap in Acore, while instance object creation, moderately expensive in Concurrent Smalltalk, is avoided whenever possible. On the other hand, futures are fairly expensive in Acore, while they are very cheap in Concurrent Smalltalk; thus, Concurrent Smalltalk creates a future (or a cheaper cfuture) as a result of every non-primitive function call, achieving maximum concurrency within a method in most cases. Acore, on the other hand, often has to do a relatively expensive join operation. For the same

---

[1] As will be discussed in Chapters 5 and 10, the amount of computation done by a Concurrent Smalltalk process on receiving a message truly is bounded, but it is done for a more prosaic reason than keeping a clean model—user Concurrent Smalltalk methods are not allowed to loop without a message send somewhere to break the loop to prevent the incoming message queues on an MDP from overflowing if the loop lasts for a long time. Also, long, indivisible loops would degrate latency for other messages that are waiting in an MDP's incoming message queue.

reason, futures are transparent in Concurrent Smalltalk, while they are programmer-visible in Acore[1].

The two languages use the same mechanism for calling messages. When a Concurrent Smalltalk process or an Acore actor makes a function or method call, it passes a continuation to which results should be sent. The continuation includes both a process and a slot within that process in which the result should be stored.

J-Machine References

[13] and [14] are good descriptions of the philosophy of the J-Machine project and the early Concurrent Smalltalk language; [15] is a recent status report on the MDP from the hardware perspective. [22] describes some of the experiences gained from designing the previous version of Concurrent Smalltalk and implementing the first-generation Optimist compiler. [10] contains a nontrivial program written in an older dialect of Concurrent Smalltalk. [8] and [9] describe *Concurrent Aggregates*, a higher-level language than Concurrent Smalltalk for programming the J-Machine. [33] and [34] describe a parallel project to implement dataflow on the J-Machine. Finally, [41] and [42] analyze the desirability of supporting the more common existing parallel programming paradigms on the J-Machine.

---

[2] A complaint mechanism could be built on top of Concurrent Smalltalk by using the multiple-value return feature—one of the values could denote a continuation to which exceptions should be routed. Acore uses a similar implementation to handle exceptions.

[1] Nevertheless, a language that hides futures could be built on top of Acore.

Chapter 2. Concurrent Smalltalk

Introduction

A Concurrent Smalltalk program is a sequence of top-level definitions. Figure 2-1 shows a sample program that calculates Fibonacci numbers using double recursion.

```
(Defmethod fib Integer ()
  (if (<= self 2)
   1
   (+ (fib (- self 1)) (fib (- self 2)))))
```

Figure 2-1. A simple Fibonacci program

This program calculates Fibonacci numbers using double recursion. Although it does not use the most efficient algorithm to calculate Fibonacci numbers, it does illustrate Concurrent Smalltalk's implicit concurrency.

The program is a single *method* associated with the *selector* `fib` and *class* `integer`. The fact that the method takes no arguments other than the integer receiver is indicated by the empty list, `()`, on the first line. The following three lines contain the body of the method. `Self` represents the *receiver object*, which is the number to which `fib` was applied. The `if` statement checks whether that number is less than or equal to 2. If so, `fib` returns 1. Otherwise, `fib` returns the sum of `(fib (- self 1))` and `(fib (- self 2))`, which are computed concurrently. This concurrent evaluation of arguments is one of the important differences between Concurrent Smalltalk and sequential Smalltalk.

`Fib` can be invoked by calling it on an integer (the receiver object):

```
(fib 30)
```

`Fib` would then calculate and return the answer `832040`. If `fib` had any more arguments, they would be included after the receiver object, as in:

```
(fib 30 x y z)
```

Functions

The Fibonacci program was defined as a method. It is also possible to define it as a function, as in Figure 2-2. A function is a method not associated with any class or selector. Although in this example methods and functions are equivalent, in other cases, such as in iterators, functions may be more useful than methods.

```
(Defun ffib (n)
  (if (<= n 2)
   1
   (+ (ffib (- n 1)) (ffib (- n 2)))))
```

Figure 2-2. A simple Fibonacci program as a function

Functions have no receiver object, so the parameter n has to be specified explicitly.

The syntax for a method and a function call is the same, so `ffib` would also be called by:

```
(ffib 30)
```

The meaning of applying `ffib` to arguments (`30` in this case) depends on whether `ffib` is a selector or a function. If `ffib` were a selector, a method lookup would be done to determine the class of the first argument and then call the method corresponding to the selector and that class, while if `ffib` is a function, it is called directly.

Extracting Methods

A manual method lookup can be done using the `method` primitive. `Method` takes two parameters, a selector and a class, and returns a function which performs the same action as the method. For example, the method shown in Figure 2-1 can be extracted using

```
(method fib integer)
```

The result behaves just like the `ffib` function in Figure 2-2. It can be called using

```
((method fib integer) 30)
```

A method extracted in this way does not have to be a direct method of the class; it can be an inherited method.

Classes

A Concurrent Smalltalk class is a type; the two words are used interchangeably in the language definition[1]. A few built-in classes are predefined; these include symbols, booleans, integers, floating point numbers, characters, functions, and other classes. A complete list is given in table A-2. All classes are subclasses of the class `object`.

The `defclass` primitive can be used to add user-defined classes. A class definition consists of a list of superclasses and zero or more new instance variables. Each instance object of that class contains those instance variables. The user may also define a number of methods for that class. A simple class that implements Lisp-like lists is shown in Figure 2-3.

```
(Defclass pair (object) car cdr)

;(Defmethod car pair () car)
;(Defmethod cdr pair () cdr)
;(Defmethod get-car pair () car)
;(Defmethod get-cdr pair () cdr)
;(Defmethod put-car pair (value):pair (set car value) self)
;(Defmethod put-cdr pair (value):pair (set cdr value) self)

(Defun cons (first second):pair
  (put-car-cdr (new pair) first second))

(Defmethod put-car-cdr pair (first second):pair
  (cset car first)
  (cset cdr second)
  self)
```

Figure 2-3. The `pair` class

The six methods that are commented out by semicolons are defined automatically by `defclass` (in addition to a few others described in Section A.4). `Car` and `get-car` do the same thing; both are defined because `car` is more convenient, but it cannot be used in the body of a method of class `pair` because static scoping shadows the method `car` by the instance variable `car`.

The `:pair` constructs define the result types of the methods. They are unnecessary, but they do improve efficiency and allow rudimentary type checking.

The class `pair` is defined on the first line of Figure 2-3. The `defclass` primitive specifies the class name (`pair`), the superclasses (`(object)`), and the *instance variables* (`car` and `cdr`).

Whenever a class c is defined, a *class predicate* and *reader* and *writer* methods are defined automatically, as well other, less-used methods described in Section A.4. The *class predicate* is a function named c? that accepts one argument a and returns `true` if a is a member of class c (or one of its subclasses) and `false` otherwise. Also, for each instance variable x of c,

---

[1] Nonetheless, the words *type* and *class* have slightly different meanings in the discussion of the compiler in Chapter 3.

the methods x, `get-x`, and `put-x` are defined. The first two methods take an instance object o as an argument and return the value of x in o, while `put-x` takes two arguments, an instance object o and a new value v of x, and assigns v to x in o. The methods x and `get-x` are known as *reader methods*, while `put-x` is called a *writer method*. The writer methods return o, the object to which the value is written.

After a class is defined, additional methods may be defined for it. In the above example, a method `put-car-cdr` is defined for the class `pair`. `Put-car-cdr` sets the value of a pair's `car` and `cdr` variables and returns the pair. Inside a method, the receiver's instance variables can be accessed by their names.

Overriding Methods

Consider a class c2 which is a subclass of c1. When a class c2 defines a method m2 with the same selector s as a method m1 of c1, the class c2 is said to be *overriding* the method m1. When selector s is applied to an object of class c2 or one of its descendants, method m2 will be used instead of m1.

Nevertheless, sometimes it is desirable to call m1 on an object of class c2. For example, method m2 might want to call the method it is overriding. An overridden method m1 can be called by performing a manual method lookup using the form `(method s c1)`. The resulting method can be called normally.

Type Restriction

The type of an overriding method must be a subtype of the type of the overridden method. For instance, in the above example the type of m2 must be a subtype of the type of m1. This means that both methods must have the same number of arguments, the types of the arguments of the overriding method must be supertypes (superclasses) of the types of the arguments of the overridden method, and the result type of the overriding method must be a subtype (subclass) of the result type of the overridden method. If any argument of the overridden method is declared inline or using any other declaration, either explicitly or by default, the corresponding argument of the overriding method must have the same type and declarations. The results of violating the above rules are undefined. The compiler may issue errors if the above rule is violated, but it is not guaranteed to do so.

The above restrictions apply only to methods being overridden. There are no restrictions on methods with the same name declared for disjoint classes (i.e. classes which are not subclasses of each other).

The Class `Object`

Methods of class `object` are very similar to functions. There are two main differences between functions and methods of class `object`:

- A method of class `object` can be overridden by a method of a more specific class. For example, if `cons` in Figure 2-3 is defined as a function, no other function or method may be called `cons`. On the other hand, if it is defined as a method of class `object`, it may be overridden by a method `cons` defined for integers. However, a method may not be overridden by a function.

- A function that takes no parameters can be defined, while a method must always take at least one parameter—the instance object.

In the interest of code maintenance and readability, it is recommended that functions be used in cases when overriding makes no sense; parameter functions to iterators fall into this category. On the other hand, if overriding a function might be desirable, that function should be defined as a method of type `object`. It is not clear whether overriding `cons` (Figure 2-3) would be useful, so it might be defined either as a function or a method, depending on one's taste.

Local Variables

A method or a function can declare local variables using the `clet` or `let` statements or their derivatives. For example, the function `fib` from Figure 2-1 could be rewritten using two local variables as in Figure 2-4.

```
(Defmethod Integer lfib ()
  (if (<= self 2)
    1
    (clet
      ((a (lfib (- self 1)))
       (b (lfib (- self 2))))
      (+ a b))))
```

Figure 2-4. Fibonacci program with local variables
The above program is equivalent to the one in Figure 2-1 and actually compiles into the same code.

Local variables declared with a `clet` or a `let` statement have a scope which is the body of the `clet` or `let` statement (except for the bindings themselves). `CLet` and `let` statements can be nested. Local variables can be altered using a `cset` or a `set` statement; the difference between the two will be explained in the **Concurrency** section below.

Types

The types (i.e. classes) of various values can be declared explicitly. Such declarations serve three purposes:

- Types allow the compiler to generate faster code by allowing it to perform operations such as method lookup at compile time.
- The compiler can perform type checking to find simple errors such as passing a value of one type to a function that is expecting a value of a different type.
- Declaring types of function parameters and results serves to document the code.

For the purposes of type inclusion, a type is its own supertype and subtype.

Due to the common use of generic types, the compiler's type checking is necessarily limited. In particular, when an expression of type t1 is assigned to a variable of type t2 or passed as a parameter to a function that expects type t2, the compiler usually will give an error or a warning if t1 is not t2, t1 is not a superclass of t2, and t2 is not a superclass of t1. This does not mean, however, that the semantics of function parameter and return type declarations are any different from their standard interpretations—when a function parameter is declared type t, every value passed as that parameter must be a member of type t, and when a function result is declared type t, the function must return a value that is a member of type t as that result—the only difficulty is that the compiler is not able to do full type checking, so it usually follows the rules outlined above.

For example, `integer` and `boolean` are both subclasses of the `object` and `magnitude` classes (see Figure A-2), but they are otherwise unrelated to each other. Thus an `integer` can be passed to a function that expects an `object`, an `object` can be passed to a function that expects an `integer`, but a `boolean` cannot be passed to a function that expects an `integer`. The second possibility, passing a more general type to a function that expects a less general one, is included to handle the common case of extracting values from general storage class. One could, for example, keep a `pair` of `integers` and desire to add the pair's `car` and `cdr` together. Since a `pair` is a generic data structure, it can contain values of type `object`;

a compiler has no simple way of knowing at compile time that the `pair` will contain `integers`, so the best it can deduce is that the `pair`'s `car` and `cdr` are `objects`.

Types can be declared as follows:

- To specify the type of a local or an instance variable, follow the variable name with a colon and its type. Several locals can be declared using the same type by separating their names with commas.
- To specify the type of a function or method formal, follow the formal name with a colon and its type. Several formals can be declared using the same type by separating their names with commas.
- To specify the result type of a function or method, follow the list of formals with a colon and the result type[1].
- A type of an intermediate result can be specified using a type-assertion statement[2].

The three kinds of declarations are illustrated in Figure 2-5, yet another copy of the Fibonacci program. All untyped variables, parameters, and functions and methods are typed `object` by default.

```
(Defun tfib (n:integer):integer
  (if (<= n 2)
   1
   (clet
     ((a:integer (tfib (- self 1)))
      (b:integer (tfib (- self 2))))
     (+ a b))))
```

Figure 2-5. Fibonacci program with types

There are three type declarations here. In order, they are a declaration of the parameter type of `n`, a declaration of `tfib`'s result type, and declarations of the types of the local variables `a` and `b`.

Concurrency

Concurrency is expressed in Concurrent Smalltalk in several ways:

- Concurrent argument evaluation. In
  ```
  (+ (big-computation 3) (time-sink 738))
  ```
  the expressions `big-computation` and `time-sink` can be evaluated in parallel.
- Expressions in `concurrently` statements may be evaluated concurrently. The expressions in `parallel` statements are always evaluated concurrently.
- The variable bindings in `clet` and `let` statements can also be evaluated concurrently. For example, the expressions `big-computation` and `time-sink` can be evaluated concurrently in
  ```
  (cset a (big-computation 3))
  (cset b (time-sink 738))
  (+ a b)
  ```
  as well as in
  ```
  (let ((a (big-computation 3))
        (b (time-sink 738))
      (+ a b))
  ```
- The computations in assignments using `cset` and in function calls whose result values are unused can be done concurrently with neighboring statements.

---

[1] See also **return values** in section A.5 for a description of specifying types of multiple results.

- The computations done for *futures* are always evaluated in parallel.

The action of a `cset` can be thought of as storing a promise (known as a *cfuture*) to calculate the value of a variable. For example, after

```
(cset a (big-computation 3))
```

is executed, a will contain either the value of `(big-computation 3)` or a cfuture promising to deliver that value when it is needed. If a contains a cfuture, `(big-computation 3)` is evaluated in parallel by a different task. At the same time, execution of the method can proceed and the method can perform another time-consuming task. It will not have to wait for `(big-computation 3)` to complete until the value of a is needed.

Sometimes it is desirable to explicitly wait until the value of an expression is available before continuing. This is called either *touching* or *forcing* the expression. Touching or forcing an expression that evaluates to a normal value does nothing. Touching or forcing an expression that evaluates to a cfuture causes evaluation to wait until the value of the cfuture is available. Finally, touching an expression that evaluates to a future does nothing, while forcing it causes evaluation to wait until the value of the future is available. The resulting value is then touched or forced again until the touch or force operation does not change it.

An expression can be touched using the `touch` statement and forced using the `force` statement. Since built-in methods and functions usually touch or force their arguments, touching and forcing are rarely done explicitly.

The reference manual in Appendix A defines more precise semantics for what expressions may or may not be evaluated in parallel.

Locks

```
(defclass resource (object)
  l:lock
  ... other fields)

(defmethod init resource ()
  (cset l (new-simple-lock)) ;Creates an initially available lock
  ... other initialization code)

(defun new-resource ()
  (init (new resource)))

(defmethod access resource (parameters
  (acquire l)
  ... code to perform the access using parameters ...
  (release l))

(defmethod access2 resource (parameters
  (with-locks (l)
    ... code to perform the access using parameters ...))
```

Figure 2-6. Lock Example

This example defines a class `resource` that contains a lock. Every call to `access` acquires the lock when it starts and releases it when done, so the code in the middle of the `access` method cannot be interrupted by another `access` method. The `with-locks` macro is a convenient shorthand for acquiring and releasing locks; the `access` method could have been rewritten as `access2`.

Locks are used to synchronize computation by Concurrent Smalltalk programs. Locks are especially useful around critical sections of code where only one process may access a resource; a process that wants the resource acquires a lock before accessing the resource and releases it when it is done. Two variants of locks are provided. `Simple-locks` are fast locks which, however, perform poorly when many processes are waiting for a resource; `simple-`

---

[2] See section A.6.

`locks` should be used in situations in which the probability of contention for a resource is small. `Queueing-locks` are slower locks designed to handle a large amount of contention.

As an example of the use of locks, suppose one wants to restrict the use of a resource so that only one process can access it at a time. To accomplish this exclusion, a lock can be associated with the resource, in which case every process should acquire the lock before using the resource and release it when done. Figure 2-6 shows sample code used to access the resource.

Distributed Objects

```
(defclass distarray (distobj)
  value)

(defun new-distarray (size:integer)
  (new distarray size)

(defmethod get distarray (index:integer)
  (get-value (co group index)))

(defmethod put distarray (index:integer new-value)
  (set (get-value (co group index)) new-value))

(defmethod size distarray ()
  (logical-limit self))
```

Figure 2-7. Distributed Object Example

This example defines a class `distarray` used for distributed arrays. The `get` method returns the element at position index in the array; since each constituent contains only one element of the array, the `get` method returns the value in the constituent specified by the given index. Similarly, the `put` method routes the message to the constituent specified by index, where it stores new-value. The `size` method simply returns the array's size.

Whereas standard objects serialize messages sent to them[1], distributed objects can accept and process many messages at a time. A distributed object is comprised of an array of constituent objects and a common, group name. When a message is sent to the group name, the operating system routes it to a constituent of its choosing. The constituent can then process the message or send it to another constituent; constituents know how to address each other. The `co` primitive is used to find a particular constituent of a distributed object, while the `group` instance variable can be read to determine the group name of a distributed object given one of its constituents.

For example, a large array might be implemented as a distributed object. When a `get` message is sent to the array to read a value of a particular element, the message is routed to one of the constituents. That constituent examines the given index and forwards the message to the constituent containing the element, which reads and returns the value.

Figure 2-7 shows a simple example of the use of distributed objects to create a distributed array. Each constituent contains only one element of the array to keep this example short; a better implementation would use a `simple-array` at each constituent to reduce the number of constituents needed.

The advantages of using a `distarray` class like the one in Figure 2-7 is that many accesses can be made to the array simultaneously; they do not have to pass through a common bottleneck to access the array. In addition, as will be clarified in Section 3.3, the `get` and `put` methods do not access any instance variables of `distarray` themselves, so they could be inlined wherever they are called[2]; thus, reading or writing the `distarray` in Figure 2-7 could

---

[1] Except for a few special cases such as immutable objects and messages which do not need to access an object's data to execute, only one message may be processing on a standard object at a time.

[2] The compiler's handling of `group` would have to change a little to permit this optimization; the compiler currently treats `group` solely as an instance variable, but there is no intrinsic reason why the compiler could not provide a bypass path that checks whether a method was called on a group ID (as opposed to a constituent ID) and just uses the

involve only two message sends, which is no less efficient than reading or writing a `simple-array`.

Macros

Concurrent Smalltalk provides a macro facility which can be used to extend the language. A macro consists of a pattern and a replacement. The pattern can contain variables or keywords. If it matches with an expression, that expression is replaced by the replacement, which can be either another pattern or a Common Lisp function[1]. Much of the language itself has been implemented in terms of macros. Figure 2-8 contains a sample macro which defines a `when` form that is the equivalent of a Common Lisp `when`.

```
(defmacro (when ?test . ?body)
  (if ?test
    (begin . ?body))
```

Figure 2-8. `When` macro

The `when` form defined by this macro takes a test and a number of statements comprising the body. If the test is true, the statements are executed one after another, as in `begin`. If the test is false, `when` returns `nil`. This macro takes advantage of the fact that `if` returns `nil` if there is no else-clause and the condition is false. The Lisp dot notation is used to indicate that the body forms the rest of the given list.

---

group ID if it was provided instead of always using the `group` instance variable. When this optimization is implemented, distributed arrays such as the one above will be as efficient as simple arrays.

[1] Concurrent Smalltalk functions may be added as replacements later, when the entire compiler and development system is rewritten in Concurrent Smalltalk.

Chapter 3. The Optimist II Compiler

Optimist II is an optimizing compiler for the Concurrent Smalltalk language described in Appendix A. The compiler generates assembly language code for the Message-Driven Processor.

Optimist II is based on the Optimist compiler described in [21]. Optimist included many standard optimizations such as register variable assignment, dataflow analysis, copy propagation, and dead code elimination [3] [43] that are used in compilers for conventional processors. In addition, Optimist included fork and join mergers that try to merge similar (not necessarily identical) statements on both sides of conditionals, a powerful move eliminator, and numerous code generator optimizations to accommodate various idiosyncrasies of the MDP.

Optimist II is a substantial improvement over the Optimist compiler. While Optimist supported only a small subset of an early Concurrent Smalltalk language, Optimist II implements almost the entire new Concurrent Smalltalk language. Some language features supported by Optimist II that were not present in the original Optimist include:

- Method lookup (Optimist could compile method code but could not associate a method with a selector)
- Global variables
- Class and variable declarations
- Macros
- Lambdas and closures
- Multiple inheritance of classes
- Distributed objects
- Multiple return values
- Nonlocal exits
- Functions
- Methods referencing more than one object at a time
- Synchronization primitives
- Arrays
- Methods overriding primitive selectors such as +
- Compile-time evaluation of expressions

Furthermore, Optimist II contains an interactive language environment, including a Concurrent Smalltalk interpreter and facilities to view code in various stages of compilation. Optimist II gives helpful warnings and errors when it encounters questionable language constructs. It also includes entire new categories of optimization, including type inference and global program optimizations. Finally, Optimist II's code generator has been updated to conform to and optimize for MDP Architecture version 11B [16][1] instead of Optimist's Architecture 10 [23].

---

[1] This reference is to MDP Architecture version 11. Version 11B has not been published yet.

The only language features listed in Appendix A missing from Optimist II are full futures and I/O facilities. It is expected that they will be added later, when the operating system is updated to support them. In addition, some optional features of the language such as inline objects and first-class continuations have not been implemented, although facilities have been provided that will simplify their implementation in the future.

Structure

Figure 3-1 shows the overall structure of the compiler. Concurrent Smalltalk code is read and parsed by the reader and parser, transformed by the preoptimizer, and saved in the global environment. It can be either interpreted using the global environment or optimized further by the optimizer and then compiled into MDP assembly code by the compiler and assembler. The treewalker controls the compilation process and prevents unused modules and objects from being compiled and assembled.

Reading Guide

The **Data Structures** section introduces the common data structures used in the Optimist II compiler. A few data structures such as digraphs and hcode appear throughout the compiler, and familiarity with them is assumed in the later sections.

The next three sections discuss the three main components of the compiler environment: The **Initial Phase** includes facilities to read Concurrent Smalltalk expressions and compile them into hcode (an intermediate code format), interpret that hcode, and maintain the global Concurrent Smalltalk environment. This phase executes until the user requests a compilation of the program to MDP assembly code, at which time the other two phases are invoked. Most of the optimizations in Optimist II are done in the **Optimization** phase, although a few appropriate optimizations are scattered in the other phases. The **Code Generation** phase compiles the optimized hcode into MDP assembly language and outputs that assembly language, together with immediate objects, class descriptors, and method tables, after performing a few final optimizations. The output of the Code Generation phase can be read directly into MDPSim. The code generator and MDPSim share the task of linking programs. Finally, the **Summary** section summarizes the important ideas in the compiler.

Chapter 5, **Sample Program**, shows the progress of a sample program through various phases of the compiler, and it may be helpful to illustrate some of the optimizations.

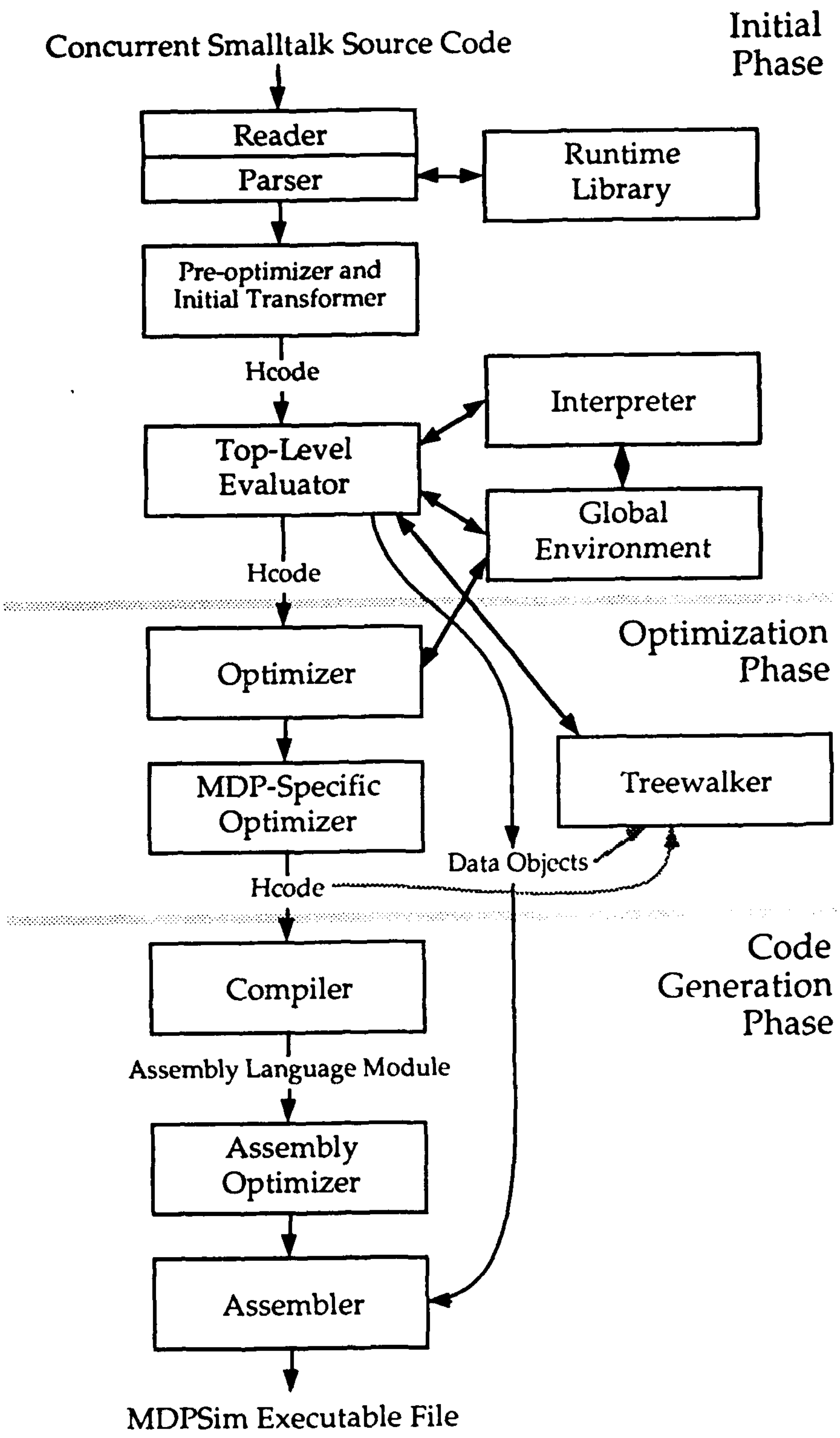


Figure 3-1. Optimist II Organization

3.1. Data Structures

Utilities

Optimist II uses a number of supporting data structures throughout the compilation process. These include abstractions such as environments, queues, ordered sets, bit sets, and extensions to CLOS. The supporting data structures are defined in the System Utilities, Utilities, and Digraph files[1].

An environment associates keys with values. Environments can be atomic, linked to each other, and either simple or based on hash tables. Atomic environments allow a series of changes to be cancelled, which is a useful operation if a syntax error is found in the input Concurrent Smalltalk expression. See the System Utilities file for more information about the internal Optimist II environment formats.

The implementation of digraphs (directed graphs) is discussed in [21]. The digraphs in Optimist II extend that implementation by taking advantage of CLOS's class inheritance mechanism and by automatically marking a digraph altered when any change to it is made, eliminating some hard-to-find consistency errors. For example, using the dfs function to ask for a listing of the nodes of a digraph will always yield an up-to-date list. Furthermore, an Optimist II digraph has a root dinode that is attached to both the digraph's starting nodes and the ending nodes, allowing easy identification of the digraph's exit points. Including the root node generalizes some algorithms; for example, the join merger can now join statements at the end of the digraph.

The traversal returned by dfs is not quite a depth-first search—the search order is depth-first modified to avoid listing a node ahead of its predecessors whenever possible. If the graph is acyclic, no node (except the root) is listed before its predecessors. The digraph dataflow problem solver [21] [3] has been updated to detect this condition and solve a dataflow problem on a digraph in one pass if the digraph is acyclic; otherwise, the dataflow solver makes two or more passes until no node changes. Moreover, dfs automatically detects and removes dead dinodes from a digraph; dead dinodes are dinodes which cannot be reached by following the edges in the digraph in the forward direction starting at the root, but which can be reached from the root by following edges in the undirected digraph.

The other structures based on digraphs such as modules are similar to those in Optimist. See the Digraph file for more details, including the dataflow problem solver and a directed graph mapper utility.

Hcodes

Hcode is the primary intermediate code format of the Optimist II compiler. It is loosely based on I-code found in Optimist. Hcodes[2] are represented by instance objects of CLOS classes, and there is no uniform syntax for reading and writing programs in hcode form, although the show utility prints hcodes fairly well. In addition, the usage of hcodes is not uniform throughout the compiler. The sets of hcodes allowed in different stages of the compiler differ—some hcodes are used early and then banned, while others are introduced just before assembly code is generated. The number of hcodes used in the compiler is small and fixed—there are only thirteen hcodes, and nine of them are limited to certain phases the compiler. Since there are few hcodes, most operations can be expressed in only one way in hcode, and the optimization algorithms have to handle only a few cases instead of many synonymous I-codes, as used in Optimist.

---

[1] See Appendix E for information on getting copies of the files.

Table 3-1. Hcodes

HCode	Arguments	Usage*	Action
Directive	Directive Directive-args	I, Pre	Evaluate top-level directive such as `add-method` on the directive-args.
Application	Targets Funct Args	I, Pre, O, Post, C	Apply funct to args and put the result values in the targets.
Assert-Type	Argument Type	I, Pre, O, Post	Assert that the argument value's type is a subtype of type.
Move	Target Source	I, Pre, O, Post, C	Move value from source to target.
Make-Closure	The-Lambda Sources	I, Pre, O, Post	Make a closure out of the-lambda using sources as the values of the display arguments.
Nconcurrently	Threads	I, Pre	Execute threads concurrently.
If	Condition Argument	I, Pre, O, Post, C	Branch if argument satisfies condition. Table 3-2 lists the allowed conditions.
Touch	Argument	I, Pre, O, Post, C	Touch the argument
Force	Argument	I, Pre, O, Post	Force the argument
Make-Future	Target Argument Lazy	I, Pre, O, Post	Make a future which will evaluate the lambda passed as an argument. Store the future in target. The future is lazy if lazy is true.
Enter		I, C	Commence function or method execution.
Exit		C	Terminate function or method execution.
Grab	Argument	C	Temporarily dereference an instance object.

*The Usage column specifies the stages of the compiler in which the hcode is valid. The stages are:

I	Hcode before initial transformations.
Pre	Pre-optimized hcode. This hcode is stored in the global environment.
O	Hcode during most of the main optimization phase.
Post	Hcode during the MDP-specific post-optimization phase.
C	Hcode just before it is compiled into MDP assembly language.

Table 3-1 lists the hcodes. Most hcodes contain fields such as arguments and targets. An argument field can contain any *rvalue*[1], while a target field can contain any *lvalue*. Also, a type field can contain any *type*, while a class field requires a *class*. The formats of those fields are listed in Table 3-3.

There is no hcode that returns a value from a function or a method. Instead, a special lvalue is used to represent a continuation to the caller. A value is returned by storing it using a reference to the continuation as a target. Thus, a move hcode with a reference to a continuation as a target is really a return statement, while an application hcode with a reference to a continuation as a target is a tail-forwarded application. More complicated combinations are also permitted—an application hcode that returns two values can forward one to a continuation and store the other in a local variable, or continuations to several different callers within whose static scopes a function resides could be used. The benefits of not including a return hcode are a more orthogonal set of hcodes and a simplification in the tail forwarder, which now becomes a somewhat specialized move eliminator.

Every hcode has exactly one successor in the digraph except the if hcode, which has two, corresponding to evaluating the conditional as true or false. The nconcurrently hcode has only one successor, but it also contains a set of nested digraphs, which may be evaluated concur-

---

[2] Sometimes the word *statement* will also be used to refer to an hcode.

[1] Rvalues are defined below.

Table 3-2. Conditions

Condition	Expression
:true	Branch if the argument is true. The argument must be a boolean.
:false	Branch if the argument is false. The argument must be a boolean.
:nil	Branch if the argument is `eq` to `nil`. The argument can be any object.
:non-nil	Branch if the argument is not `eq` to `nil`. The argument can be any object.
:zero	Branch if the argument is `equal` to `0` using the = predicate.
:non-zero	Branch if the argument is not `equal` to `0` using the = predicate.

rently, sequentially, or interleaved in any fashion. There is no restriction on the number of predecessors an hcode can have.

Hcodes are rarely processed alone; usually hcodes are embedded in a code-lambda or cst-lambda, which represent digraphs of hcodes with header information. A code-lambda contains a digraph of hcode statements together with a database of local variables used by those statements. Each local variable has an optional name, a type, and some declarations such as whether it can hold inline objects. Furthermore, the locals in a code-lambda are consecutively numbered to allow the efficient use of bitmaps to keep track of variable data while solving dataflow problems. In addition, a code-lambda shares with cst-functions (another internal Optimist II class that describes all functions, including primitives) the interface fields which consist of a list of parameters, return values, and display variables used by closures.

Hcodes are documented in the HCode file.

Values

An Optimist II *value* is a representation of a Concurrent Smalltalk object—it can be, say, an integer, a character, a distributed object, a function, a class, or any other valid Concurrent Smalltalk object. On the other hand, a variable or a parameter is not a value, but it may *contain* a value. In addition, values of a few hidden types such as continuations and continuation displacements are also used. Many different representations are used for values, and these representations will not be described further here; please refer to the Types file for more details on this subject.

An *rvalue* can be either a value or a location that can be read to obtain a value. Thus, a local or a global variable is an rvalue, and so is the Concurrent Smalltalk integer 7. An instance variable in general is not an rvalue, but a reference to an instance variable in a particular instance object is. The common rvalue kinds are listed in Table 3-3.

Table 3-3. Rvalues

Rvalue	Specializers	Notes
Value		Any value is also an rvalue.
Local	Name, scope, etc.	A local variable.
Global	Name	A global variable.
Option	Name	A Concurrent Smalltalk option.
Ivar-ref	Instance variable, Instance object	An instance variable of an instance object. The instance object must also be an rvalue.

An *lvalue* is a location into which a value can be written. Examples of rvalues include local variables, references to instance variables in instance objects, and references to continuations. A continuation by itself is not an lvalue, but a reference to one is. The common lvalue kinds are listed in Table 3-4.

Table 3-4. Lvalues

Lvalue	Specializers	Notes
Local	Name, scope, etc.	A local variable.
Global	Name	A global variable.
Continuation-ref	Continuation or Context and Displacement	A reference to a continuation specified either as a continuation rvalue or as a pair of context and displacement rvalues (See Section 3.3).

All rvalues are instances of the rvalue CLOS class, all lvalues are instances of the lvalue CLOS class, and all values are instances of the value and rvalue CLOS classes. CLOS's multiple inheritance is used to define objects that are both rvalues and lvalues or other combinations of the above.

Types and Classes

Table 3-5. Types

Type	Specializers	Notes
Class		Any class is also a type.
Continuation-type	Continuation-type	A type based on the `continuation` class that represents a continuation that will return a value of the continuation-type type.
Displacement-type	Continuation-type	A type based on the `displacement` class that represents a displacement field of a continuation that will return a value of the continuation-type type.

A Concurrent Smalltalk class is a Concurrent Smalltalk value that is an instance of the `class` class. Classes are implemented in Optimist II as instances of the cst-class CLOS class. In addition to itself being a value, a class also represents a set of values. For example, the class `integer` represents the set of all integers, which includes, among others, the values `4` and `-17`. The class `null` represents the singleton set {`nil`}. The class `class` represents the set of all Concurrent Smalltalk classes, including itself.

In addition to classes, Optimist II includes types which provide finer discrimination than classes for describing sets of values. Types are listed in Table 3-5. Currently a type is either a class or a continuation that returns an object of some type. A type can be always projected to a class; the base-class Lisp generic function performs this conversion. A type that is also a class projects to itself, while a continuation type projects to the class `continuation`. Although a class is always a value, a type is not necessarily a value.

Multitypes

When describing the possible contents of variables, Optimist II uses the concept of a *multitype*. A multitype is a list of zero or more types; a value is a member of a multitype (satisfies that multitype) if it is a member of one of its types. No value satisfies a null multitype, while every value satisfies a multitype that has `object` as one of its types. Routines are provided to calculate unions (least upper bound) and intersections (greatest lower bound) of multitypes and simplify representations of multitypes. Since multitypes are not necessarily closed under those operations, the lub and glb routines may conservatively enlarge their multitype results.

Global Data Structures

Two atomic environments, the global environment and the class environment, contain most of the state of the Concurrent Smalltalk interpreter. The global environment contains all Concurrent Smalltalk globals, parameters, and constants, while the class environment contains all known Concurrent Smalltalk classes. The global environment is linked to the class environment, so the latter is searched if an identifier is not found in the global environment.

The classes are themselves heavily linked together. Each class object has lists of its immediate superclasses and subclasses and all of its superclasses and subclasses, as well as a metaclass, a description of its instance variables, and sundry options such as whether the class is immutable. To allow typed recursive data structures, an "undefined" class structure is created when a class name is encountered in a program without being defined. An "undefined" class can turn into a normal class when the class is defined; CLOS's change-class construct is very valuable here. A substantial number of classes have to be updated whenever a new Concurrent Smalltalk class is defined, but compilation speed does not seem to suffer because of this. The heavy linking of classes made defining a bootstrapping subset of Concurrent Smalltalk classes challenging; some CLOS objects had to be created with the wrong classes and then transformed to the right classes. Once the bootstrapping subset of Concurrent Classes was defined, defining the remaining classes on top of it was easy.

A method is associated with both a class and a selector. There is no single method table in Optimist II; instead, whenever a method is added, it is added to the selector's list of methods hashed by class and the class's list of methods hashed by selector. Thus, a selector knows all of the methods defined for it, as does a class. Methods are not replicated in these hash tables unless a method is added more than once; instead, the lookup-method function, which returns a method associated with a class and a selector, searches the superclasses when a method is not defined for a selector and a class; an ambiguous selector error is signalled if there is more than one superclass and they are associated with differing methods.

Current settings of the options are also kept in a global data structure. Each option is declared as a dynamic Lisp variable, and a list of all options and their default values is kept in an object. The `#&name` reader macro expands into a reference of the option named name.

Concurrent Smalltalk symbols are not accumulated in any data structure; however, when a Lisp symbol is used as a Concurrent Smalltalk symbol, its cst-symbol property is set to the Concurrent Smalltalk symbol object to ensure that that object is reused if the symbol is referenced again; otherwise, `(eq 'sym 'sym)` would be false according to the interpreter. Number objects are not reused, so `(eq 13 13)` is false according to the interpreter[1], but `(clet ((x 13)) (eq x x))` is true.

---

[1] Nevertheless, compiled code will currently return `true` if `eq` is used to compare two equal integers. The action of `eq` on numbers is purposely not defined in Concurrent Smalltalk to allow an implementation of a bignum package.

3.2. Initial Phase

The initial phase of the compiler reads the Concurrent Smalltalk input and converts it into a rough hcode form. Several early transformations have to be done on the resulting hcode before it becomes suitable for optimizations.

The most complicated early transformations create statically scoped functions. The initial phase determines parameter interfaces for lexical variable displays [3] used by closures, and it does a considerable amount of work to pick those interfaces well. Delaying this decision would have made manipulation of functions in that stage very difficult; the advantages of splitting nested functions into components early are that every function is self-contained and completely owns its local variables—no other function can alter or examine the local variables.

Reader

A customized Common Lisp reader is used to read the Concurrent Smalltalk programs. The customizations consist of using a special readtable and reading all Concurrent Smalltalk names into the CST package. The readtable is used to implement the special characters in the Concurrent Smalltalk syntax. Most special characters expand into lists; for example, `!a` expands into `(! a)`. Some character tokens such as `:`, `::`, and `,` (comma) expand into symbols with the same names.

The CST package is used to prevent conflicts between Concurrent Smalltalk symbols and any symbols the compiler or the Common Lisp environment might be using. For instance, `nil` is just the name of a constant (which happens to have the value `'nil`) in Concurrent Smalltalk; `nil` is not confused with the Lisp nil, which also represents an empty Lisp list. Since the colon has a special readtable meaning in Concurrent Smalltalk mode, Concurrent Smalltalk symbols are restricted to the CST package.

Read macros have been inserted into both the Common Lisp readtable and the Concurrent Smalltalk one to facilitate easy switching between the two tables. The `#$` macro in standard Lisp input reads the next token in Concurrent Smalltalk mode, while `#^` can be used inside a `#$`-expression to switch back to Lisp mode. In addition, the `#L` macro in Concurrent Smalltalk mode reads a list expression and returns a two-element list with the symbol lisp as its first element and the expression read as the second.

Parser

The parser parses the input expressions into a prototypical hcode form. The parser is a recursive descent macro evaluator. Each primitive in Concurrent Smalltalk is implemented as a macro. There are three main kinds of macros: normal macros substitute Concurrent Smalltalk text with other literal Concurrent Smalltalk text as described in Section A.14, nonterminal macros substitute Concurrent Smalltalk text with Concurrent Smalltalk text produced by a Lisp function, and terminal macros read Concurrent Smalltalk text and perform an action such as emitting hcodes. Furthermore, macros can be restricted to evaluate at the top level only.

The parser, when asked to parse an expression, compares it against macros in its macro list in reverse chronological order until it finds a match; when a match occurs, the macro is expanded as above. If the macro was not a terminal one, the resulting text is expanded again until either no macro matches the text or a terminal macro is expanded. If no macro applies, the text must be a symbol, which is looked up in the current lexical environment. If the symbol is not found in the current environment, it is assumed to be an undefined global unless it happens to be one of the Concurrent Smalltalk primitive names or the `warn-free-references`[1] declaration is in effect, in which case an error or a warning is given.

Macro Implementation

Since the parser is an intensive user of macros, a fast implementation of macros is used to make the parser in the compiler fast. Macros are stored in linked lists hashed by the first non-variable symbol in the macro pattern; macros with no such symbols are stored in a separate list. Thus, relatively few macros have to be examined for a given piece of Concurrent Smalltalk text. Furthermore, the macros themselves are compiled Lisp functions that check that their patterns are satisfied and, if so, compute the text replacement or perform their terminal actions. Compiling macros avoids the costly interpreted unification step during pattern matching. The make-macro-text function in the Environment file compiles a macro into a Lisp function.

If a macro contains an `@` directive in its pattern, the macro expander calls itself recursively on the text matching the `@` directive. In this case it does not allow terminal macro expansion on that text.

Environments

While the parser is generating code, it frequently needs to determine the meanings of identifiers. It uses linked environments to keep track of statically scoped identifiers such as the names of local variables and continuations. The last local environment is linked to the global environment to cause a search of the global and class environments when an identifier is not defined locally. Optimist II distinguishes local variables according to whether they are eq to each other or not. Thus, no alpha-renaming is necessary anywhere in the parser. Also, a lambda may reference local variables it captured from an enclosing lambda. Since most of the optimizations cannot handle externally visible local variables, such local variables are "unshared" before the optimization pass is invoked.

Concurrent Smalltalk Runtime

Most of the Concurrent Smalltalk directives described in Appendix A are macros which expand into either other Concurrent Smalltalk primitives or hidden primitives. The Runtime file contains a listing of all macros used by Concurrent Smalltalk.

Top-Level Primitives

Most Concurrent Smalltalk top-level primitives listed in Appendix A expand into the directive hcodes and are evaluated at expression interpretation time. Directive hcodes may be interpreted but not compiled; to ensure that no directive will be compiled, directives are prohibited inside `lambda`s (and, of course, any constructs which expand into `lambda`s). A few directives such as `include`, top-level `set`, and `defclass`[2] are evaluated by the reader; those directives must be placed at the top level—they may not be nested in any expression except a top-level `begin`, which evaluates its arguments sequentially at the top level.

Method-Lambdas

A method-lambda of a class c expands into a lambda with a formal self of type c prepended to the method-lambda's formals and a (_with-object (self:c) ...) form surrounding the body of the lambda. The _with-object form establishes bindings in the parser's environment that associ-

---

[1] See Appendix B.

[2] `Defclass` isn't really evaluated by the reader; nevertheless, it must be a top-level form because it expands into a top-level `begin` containing the internal class definition followed by definitions of accessor and predicate methods. The internal class definition has to have been *interpreted* before the accessor method definitions are *read*; otherwise, the reader will complain about an undefined class. Grouped forms not at the top level and not in a top-level `begin` are read as a group and then interpreted as a group.

ate names of c's instance variables to ivar-refs of the corresponding instance variables pointed by the self object. The action of _with-object is analogous to that of the symbol-macrolet construct in CLOS [6].

Optimist II does not restrict a lambda to referencing only one instance object; in fact, through inlining of method-lambdas or accessor methods, a lambda can reference many objects at the same time. Objects may also be referenced through the use of _with-object directly in Concurrent Smalltalk code, but this practice is discouraged, as it uses a nonstandard feature of the language and gains no real functionality.

Loops

Although Optimist II can optimize and output code with loops in it, loops are currently not implemented this way. The problem is that a Concurrent Smalltalk function with a loop in it might execute for a long time and not allow any other messages to be processed at its node. To prevent this problem, loops are implemented as closures which pass themselves as arguments—`(while (< i 10) (set i (+ i 1)))` expands into:

```
(clet ((_loop
          (lambda ((_loop-arg:function &no-leak))::(_while)
            (if (< i 10)
              (set i (+ i 1))
              (return _while 'nil))
            (_loop-arg))))
  (_loop _loop))
```

The `_loop` function is called and passed itself as an argument. If `i` is less than 10, `_loop` increments `i` and calls its argument tail-recursively; otherwise, it returns `nil` to the caller. The tail-recursive call breaks the long invocation of the function.

The compiler is not yet sophisticated enough to detect that the value of the `_loop` variable never changes, so the `_loop-arg` argument to the internal function can be eliminated and the function could call itself recursively directly.

Initial Transformations

Immediately after the hcode is created by the parser, a transformation and an optimization are done on it. The first transformation flattens all exit hcodes out of every newly created lambda. Exit hcodes are generated by the `exit` Concurrent Smalltalk primitive, which may also be a result of the expansion of a `return` statement. Each exit hcode in the lambda is removed and the preceding statement linked to the digraph's root dinode to indicate that the execution of the lambda should terminate at that point. Sometimes exit hcodes can be found nested inside nconcurrently hcodes; if that is the case, the exit flattener moves as many of the nconcurrently's threads outside as it needs to remove all exit hcodes from the nconcurrently. Then it flattens the exits as usual. An example is shown in Figure 3-2.

Simple structural optimizations are done immediately after the exits are flattened. These optimizations do not depend on dataflow analysis and can, therefore, be done before lexical variables are untangled. The optimizations consist of the following transformations:

- If statements with identical consequents and alternatives are deleted.
- If statements conditioned on constants are deleted, and resulting dead code, if any, eliminated.
- Move statements with identical sources and destinations are deleted.
- Assert-type statements on constants are checked and deleted. The compiler generates an error if an assertion fails.

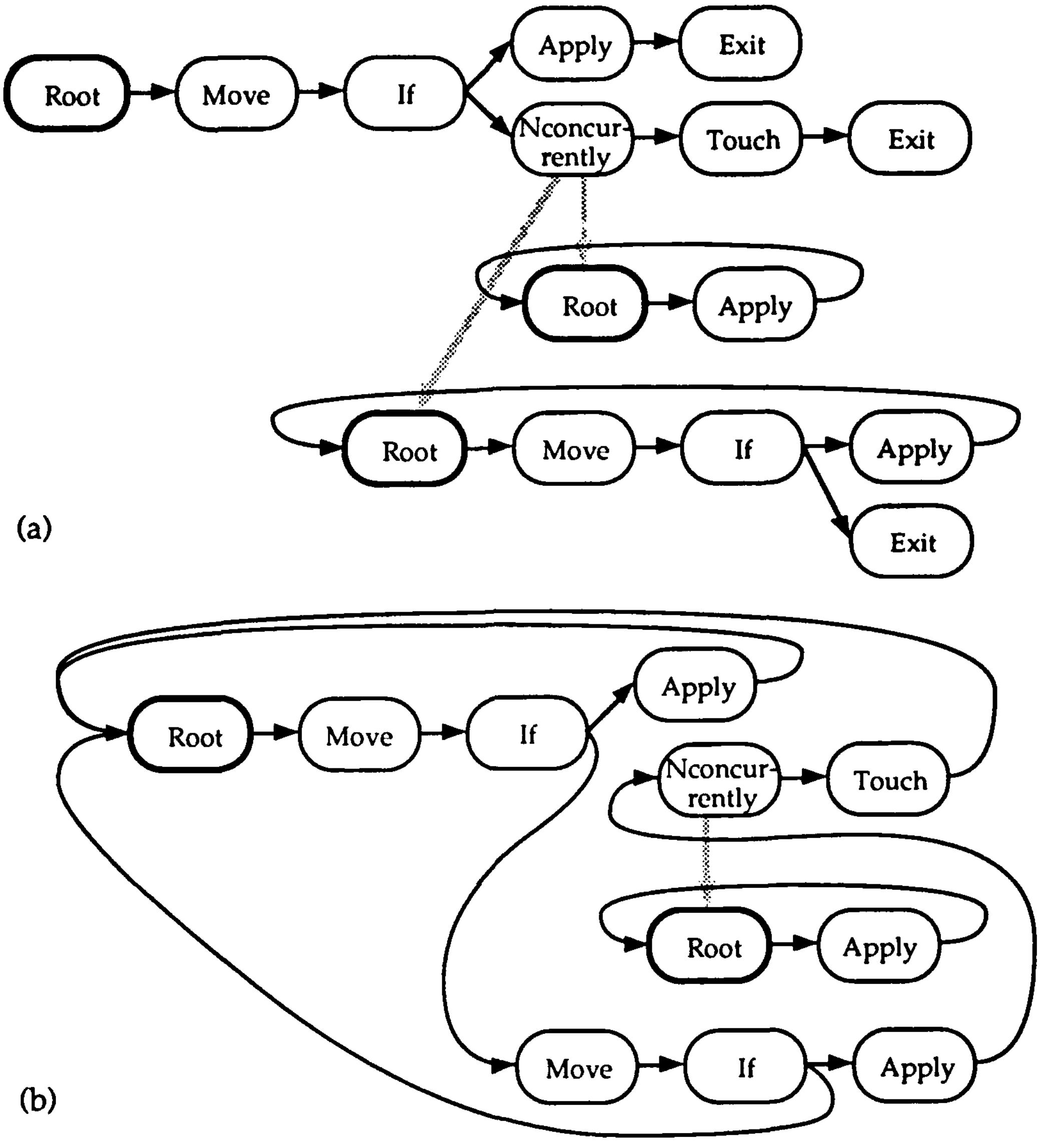


Figure 3-2. Exit Flattening Example
Exit statements are inserted by the parser in all places in which the execution of a lambda should terminate. As the first transformation, those exit statements are removed and replaced with links back to the root of a digraph. For example, part (a) shows the main body of a lambda with two sub-digraphs that are the threads of a nconcurrently. After exit removal (b), all exit paths are linked back to the root of the main body of the lambda, which also required the inlining of one of the nconcurrently's threads.

- Touch and force statements on constants are deleted.
- Empty nconcurrently statements are deleted.
- One-thread nconcurrently statements are replaced by their threads.

The structural optimizations are done for two reasons: First, structural optimizations shorten the hcode, using less memory in the later compiler stages and making them run faster. Second, structural optimizations may remove some variable references, improving the quality of the code produced by lambda-collapsing and the nconcurrently flattener in the optimization phase.

Lambda-Collapsing

Lambda-collapsing is the process of unnesting nested lambdas. After lambda-collapsing, each lambda has exclusive access to its local variables. Lambda-collapsing becomes difficult when the inner lambdas reference the outer lambdas' local variables and continuations. Since continuations are restricted local variables, they will not be discussed here further. Lambda-collapsing occupies most of the Preoptimizer file. Since lambda-collapsing is a complex process, an illustrative example is provided at the end of this section.

The lambda-collapser (the assign-lexicals Lisp function) examines each outermost lambda in the hcode produced by the initial transformations. For each outermost lambda L it looks at the lambdas $N_1, N_2, ..., N_k$ nested in L and their free variables. Each nested lambda N_i is considered to also include any lambdas nested in it. Thus, if, say, N_2 contains a lambda $N_{2,1}$ that references a variable x that is not defined in $N_{2,1}$ or N_2, then x is a free variable of both $N_{2,1}$ and N_2. If a nested lambda N_i does not reference any free variables, it is a self-contained lambda and a first-class data object and does not present any difficulties here. Otherwise, N_i is the code portion of a closure.

The lambda-collapser first calculates the sets of free variables read and written by N_i. Next, the lambda-collapser considers each local variable x_j of L. A local x_j is called a *mutable lexical* if it is either (1) written by any N_i or (2) read by any closure N_i and written by L after the closure N_i has been created by L and before the closure was called for the last time. Mutable lexicals of the first kind are easy to determine by scanning every N_i and checking which free variables are written in any hcode in it. To determine mutable lexicals of the second kind, the lambda-collapser solves a few dataflow problems on L. In effect, to each variable x_j in L, it assigns a state machine S_j (Figure 3-3) and uses the dataflow problem solver to run S_j through all possible control paths in L. If S_j ever enters state 4, x_j is a mutable lexical of the second kind. The state machine assumes that any local variable x_j that is modified after the creation of a live[1] closure which reads x_j is a mutable lexical. Since the compiler cannot currently determine when a lambda finishes executing, it cannot optimize local variables that are modified by L only after the closures have completed execution.

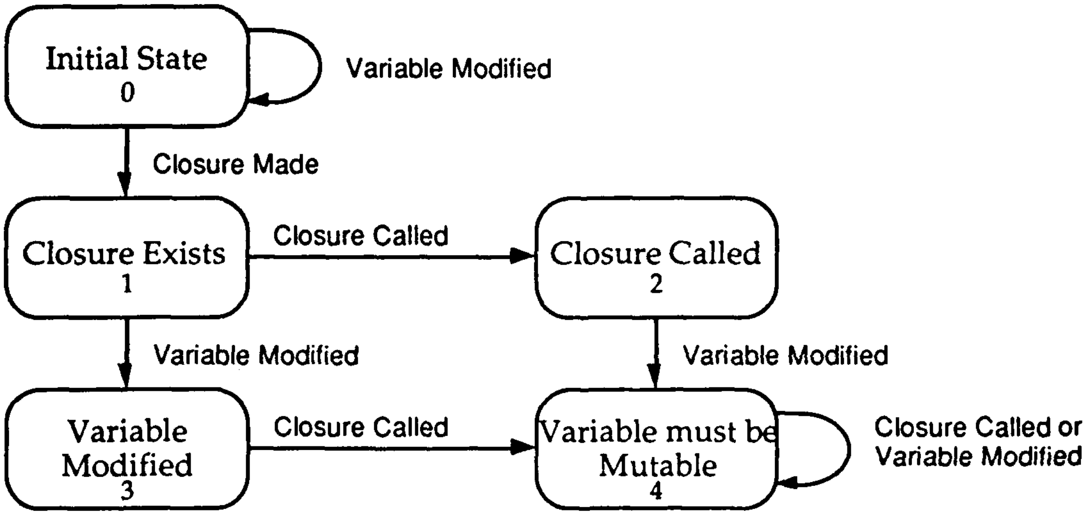


Figure 3-3. Lexical Variable State Machine
Each local variable starts in state 0 at the beginning of the lambda. For each local variable every possible path of control flow is traversed and a state updated as above. If the variable ever enters state 4, it must be a mutable lexical of the second kind—the variable's value cannot be saved with the closure when the closure is made.

---

[1] If the closure is not called, it is not a live closure, and the variable is not necessarily a mutable lexical.

Any variable that is free in one of the lambdas N_i and is not a mutable lexical is an *immutable lexical*. Once all of the free variables in the sublambdas of L have been classified, the lambdas are separated.

Each sublambda N_i of L that has free variables is assigned a number of *display* parameters in addition to the normal parameters it has. The values of the display parameters are determined at the time a closure of N_i is created. Immutable lexicals are stored directly in the display, while mutable lexicals are stored in an object whose pointer is passed in the display. More than one such object may be present if N_i uses mutable lexicals from several levels of enclosing lambdas.

Once the display parameters are assigned to the sublambdas, the code of L is modified to store the display parameters into a closure whenever one is created, and the N_i's are modified to use the display parameters instead of referencing L's locals directly. If L has any mutable lexicals, it creates an object containing them upon entry and treats mutable lexicals as if they were instance variables of that object; any mutable lexicals that are also parameters of L are copied into that object as soon as it is created. The object containing mutable lexicals is itself mutable, so only one copy of it per invocation of L can be present on the J-Machine. The object is not disposed because Optimist II cannot determine the temporal lifetime of a closure; the object and the closures have to be garbage-collected.

After the above transformation, L has exclusive access to its locals. Since some of the N_i's could themselves have locals used by their sublambdas, the lambda-collapser calls itself recursively on every lambda and closure contained in L, even if that lambda did not have any external free variables.

Efficiency Considerations

There are several advantages for using immutable lexicals instead of mutable lexicals:

- Immutable lexicals are stored directly in a closure's display, so the closure has immediate access to their values.
- Closures are immutable objects. If many closures are executing simultaneously, many copies of the closures and their immutable lexicals can be made. On the other hand, if many copies of a closure with a mutable lexical are executing, the copies will be contending for the single object containing that lexical's current value.
- The outer lambda can store immutable lexicals in its context or in registers, while it has to allocate an object for mutable lexicals and keep their values there.

In order to ensure that lexically scoped variables are immutable lexicals, the programmer should check that their values are not altered after any closures which might reference them are created.

Example

Consider the following code:

```
(defun outer (x)
  (clet ((y 3)
         (z 4)
         (t 1))
     (clet ((inner1
              (lambda ()
                ((lambda () (cset x z))) ;inner11
                (write x y)))
             (inner2
               (lambda ()
                 (write y)))
             (inner3
               (lambda (a)
                 (write a))))
        (if (zero? x)
          (inner1)
          (cset x 5))
        (cset z 3)
        (inner 2)
        (write x y z t))))
```

The lambda-collapser first determines that the outer lambda has no free variables, so it is made into a normal function instead of a closure. Next it examines the three sublambdas within outer: inner1, inner2, and inner3. Inner1 will become a closure because it has three free variables, x, y and z. It writes to x, so x becomes a mutable lexical; although inner1 does not write to y and z, another lambda might, so y's and z's statuses are unknown. Inner2 will also become a closure because it has one free variable, y, whose status is still unknown. Since inner3 has no free variables, it becomes a normal function.

Next the lambda-collapser runs the state machines on the x, y, and z locals in outer; outer also has other locals such as t, inner1, inner2, and inner3, but those are not referenced by any inner lambdas. X is already known to be a mutable lexical of the first kind. Y is not written anywhere after inner1 and inner2 are created, so it is an immutable lexical. Z is written after the inner1 closure is created, and the compiler makes it a mutable lexical of the second kind. Unfortunately, the compiler does not realize that z is altered only after inner1 finishes executing; if it were smarter, it could have made z an immutable lexical. Finally, the lambda-collapser creates the displays and alters the code of the lambdas to produce a parameter-passing pattern shown in Table 3-6.

Table 3-6. Lambda-Collapser Example Results

Name	Outer	Inner1	Inner11	Inner2	Inner3
Parameters	x (copied into lexical-object)				a
Returns	continuation-0	continuation-1	continuation-2	continuation-3	continuation-4
Display		lexical-object y	lexical-object	y	
Locals	y t inner1 inner2 inner3 lexical-object				a

	lexical-object
Instance Variables	x z

Top-Level Evaluator

Lambda-collapsing was the last preliminary hcode transformation. At this point the hcode is in a format understood by the interpreter. If it found no syntax errors, Optimist II now evaluates the Concurrent Smalltalk expression it just read by running the expression's hcodes through the hcode interpreter. If the expression contained any directives, the interpreter executes them at this time.

Interpreter

The interpreter is a simple hcode interpreter for executing Concurrent Smalltalk programs. The interpreter is completely sequential. Except for full futures and some unimplemented input/output facilities, the interpreter is a valid Concurrent Smalltalk implementation—the Concurrent Smalltalk definition allows cfutures to be touched at the implementation's discretion, so a completely sequential Concurrent Smalltalk interpreter trivially "touches" each cfuture as soon as it is created. While the interpreter never achieves any parallelism, it couldn't use parallelism if it had any because it is running on a sequential computer.

The interpreter in Optimist II was provided for three reasons:

- It is a powerful constant expression evaluator for expressions encountered while compiling Concurrent Smalltalk programs.
- It is the most interactive Concurrent Smalltalk environment, allowing methods and functions to be changed almost instantly.
- It permits debugging of Concurrent Smalltalk programs before they are compiled into MDP assembly language.
- It maintains the Concurrent Smalltalk global environment and permits interactive examination of that environment.

Currently the interpreter can only interpret unoptimized hcode; however, a bypass hcode path could be added to transfer optimized hcode back to the interpreter. This bypass is not quite as simple as it sounds because the format of continuations changes during optimization.

3.3. Optimization

As long as no MDP code output is desired, Optimist II does not leave its first phase. Only when a compile command is issued does Optimist II enter its second phase, its first goal being to determine just what it should compile. Every compile command requires a *root set* of objects that should be compiled. The compiler uses the treewalker to automatically determine the minimum amount of code that has to be compiled and loaded in order to permit running the functions in the root set on the J-Machine.

Treewalker

The root set specified in the compile command is passed to the treewalker, which appends it to its own permanent root set of objects which must always be compiled (Table 3-7). The treewalker then calls the optimizer on each code object in its set and scans the optimized hcode (if the object is not code, the treewalker scans it directly). If, while scanning, it encounters an object not in its current set of objects, it adds that object to its set, optimizes it if necessary, and scans it. The process continues until every object referenced by any object in the treewalker's set is also in that set. At that point the second phase of the compiler has completed and the treewalker calls the compiler's third stage to compile and assemble each object in the set and print the resulting MDPSim code into a text file.

Table 3-7. Permanent Root Objects

`_closure`	`boolean`	`character`	`#:class`	`context`
`#:continuation`	`displacement`	`distobj`	`distributed-class`	`#:false`
`float`	`funct`	`function`	`global`	`integer`
`magnitude`	`null`	`number`	`object`	`primitive-class`
`real`	`selector`	`standard-class`	`symbol`	`#:true`

These objects are emitted in the output assembly file regardless of which objects were compiled. `_closure`, `context`, `displacement`, `#:continuation`, and `global` are internal Optimist II classes.

Calling the Optimizer

The optimizer is called simply by requesting the value of the hcode or mdp-hcode CLOS slot in a Concurrent Smalltalk lambda (cst-lambda). If the lambda has already been optimized, these slots contain the optimized hcode and hcode optimized for the MDP, respectively. If not, those slots are unbound, and CLOS calls the optimizer to calculate their values. Thus, a lambda's optimized hcode can be requested repeatedly by the treewalker or the optimizer without a performance penalty. To prevent infinite loops, a semaphore keeps a function optimizing a lambda from requesting that lambda's optimized hcode. One of the consequences of this rule is that a function may not be inlined inside itself.

Guide to Optimizations

The transformations done by the optimizer are summarized in Figure 3-4. The transformations can be divided roughly into two classes: general hcode optimizations and MDP-specific optimizations and transformations. The general optimizations occupying the first half of the optimizer produce optimized hcode. If MDP assembly code output is desired, the second half of the optimizer is invoked to convert a number of hcode constructs into simpler, MDP-specific ones. For example, the second half of the optimizer converts globals into references to global objects, CAS built-ins into code that explicitly compares and sets values, and three-argument sums into two two-argument sums. The order of optimization is critical; expansion of CASes into compare-and-set code could not have been done in the first half of the optimizer because there was no way to assure its atomicity.

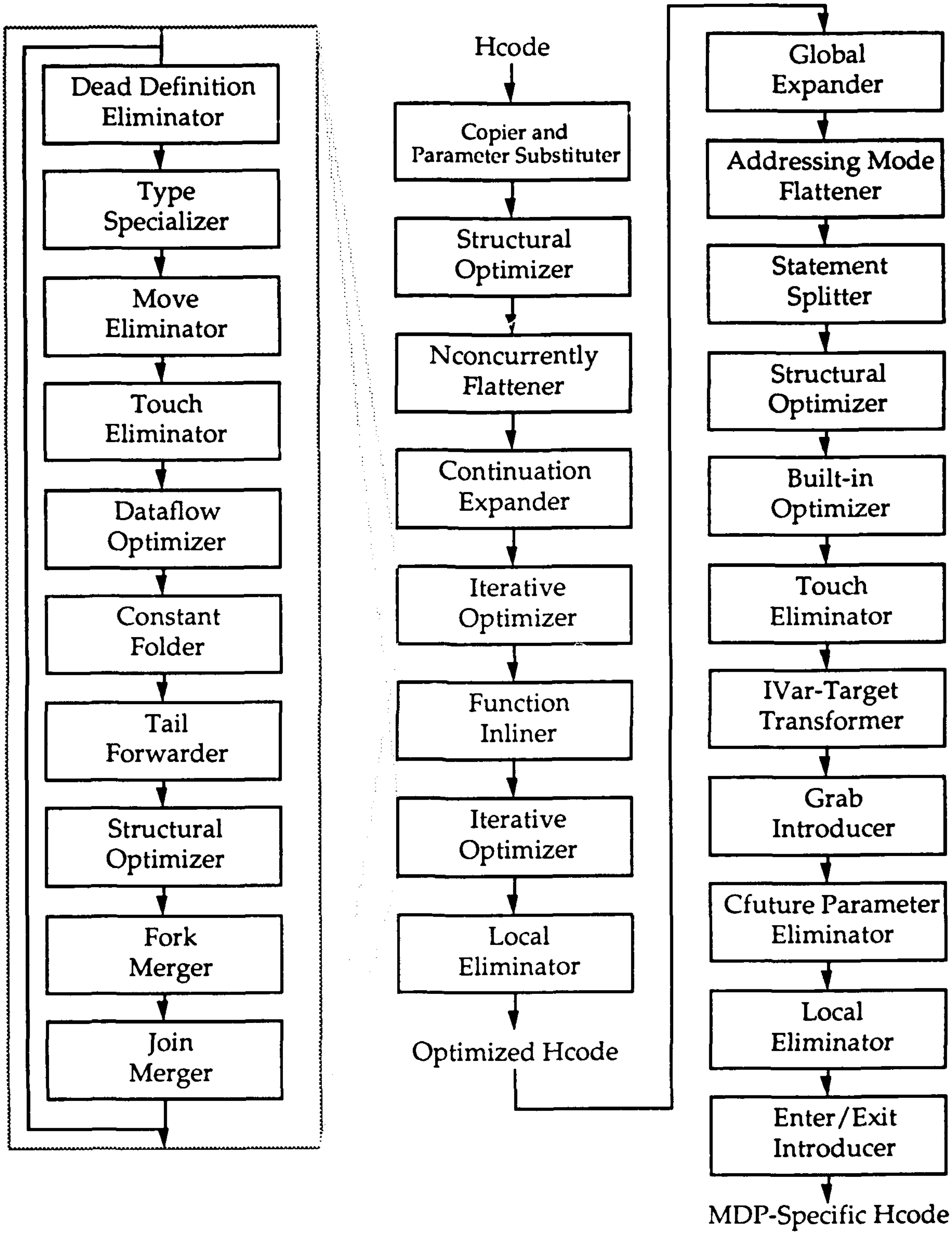


Figure 3-4. Optimizer Organization
The first few filters convert the hcode produced by the initial phase into a format usable by the optimizer. The iterative optimizer and function inliner perform the major optimizations. The remaining filters implement some Concurrent Smalltalk features out of more basic ones and fix a few quirks in the Cosmos and MDP architectures.

The transformations new in Optimist II will be described in the order in which they are performed. The following optimizations will not be described here because they were present in Optimist and their concepts have not changed significantly:

- Dead Definition Eliminator
- Move eliminator
- *Touch eliminator*
- Tail forwarder
- Fork merger
- Join merger

Also, the structural optimizer was used in the first phase of the compiler and is described there.

Preparatory Transformations

Lambda Copier and Structural Optimizer

Before optimizing a lambda, the optimizer first makes a copy of it to avoid destroying the copy used by the global environment and the interpreter. While copying the lambda, the optimizer assigns consecutive indices to the lambda's local variables. These indices will allow the use of fast bitmaps to represent local variable data during later dataflow analysis routines.

At this stage the optimizer replaces all references to parameters[1] with their values. Once the compilation process has begun, values of parameters cannot change, and replacing parameters with constants as early as possible enables early constant folding and dead code elimination. Parameters are usually used to hold global functions and compiler conditionals such as a debugging flag. Debugging code can be compiled conditionally by enclosing it within an `if` statement conditioned on a debug parameter. If debug is false, the code and the `if` statement are removed by the structural optimizer immediately following the copier; the remaining optimizations don't even see that code. Dead code is best removed early because removing it enlarges basic blocks, permits additional function inlining, and improves the performance of the dataflow optimizer and tail forwarder. It is unfortunate that conditional debugging code cannot be removed before lambda collapsing, but doing that would prevent changes in the debug parameter from having any effect on existing code.

The structural optimizer cleans the code to *give the nconcurrently flattener maximum latitude* in scheduling nconcurrently hcodes.

Nconcurrently Flattener

The nconcurrently flattener removes nconcurrently hcodes from the lambda being optimized. Later optimizations run many dataflow calculations on the lambda, and the presence of nconcurrentlys would complicate dataflow analysis and make some optimizations less effective. In the interest of compiler simplicity I decided to remove nconcurrentlys at this stage.

The nconcurrently flattener uses a heuristic to interleave the nconcurrentlys it is flattening. If it finds a nconcurrently statement with more than one thread, it first calls itself recursively on each thread and then separates each thread into a leading and a trailing set of statements. A thread's trailing set of statements contains the longest string of consecutive hcodes at the end of the thread which are not considered worth advancing relative to other hcodes in the lambda. The trailing set cannot contain any forks or joins of flow-of-control paths. All

---

[1] In this paragraph *parameters* means parameter globals defined in Section A.3, *not* function parameters.

other statements in the thread are placed in the thread's leading set. Once the nconcurrently flattener separates each thread into the two sets, it replaces the nconcurrently hcode with all leading sets concatenated together followed by all trailing sets concatenated together.

Hcodes worth advancing are non-built-in function and method calls and any hcodes which return values through continuations; all other hcodes are not considered worth advancing. Hcodes not worth advancing are pushed as far back as possible by the nconcurrently flattener, which displaces hcodes worth advancing forward.

The nconcurrently flattener could use more complicated heuristics to increase parallelism. For example, it could realize that no matter how it orders function calls in statements such as `(f (a (b 1)) (c (d 2)))`, there would remain a possibility of a loss of concurrency caused by touching the intermediate results `(b 1)` and `(d 2)` in the wrong order. Hence, it could split the calculation of, say, `(a (b 1))` into a separate function call to avoid a potential loss of concurrency. Nevertheless, the nconcurrently flattener's current heuristic seems adequate.

Continuation Expander

The continuation expander is the one MDP-specific transformation that is done early. So far in the compiler, continuations have been represented as single words, while on an MDP a continuation is two words—the context to which the continuation is pointing and an offset of a slot within the context where the return value should be stored. I originally planned to implement continuations as a special case of inline objects, but writing a general implementation of inlined objects would have been too time-consuming and inappropriate for an initial version of the compiler. Hence, I included a partial implementation of inline objects that only inlines continuations.

The continuation expander expands each local variable of type `continuation` into two variables, one of type `context` and the other of type `displacement`. Similarly, each formal and display parameter typed `continuation` is made to correspond to two local variables. A move hcode moving a continuation is changed into two moves, while an application hcode calls its function with both new locals as arguments.

Changing structures of instance objects and global variables containing continuations is hard at this stage of compilation, so to avoid this problem continuations have not been made first-class objects—there is no way to store a continuation in an instance variable of an object; disallowing programmer-visible continuation local variables ensures that no continuation becomes a mutable lexical which would get stored in an instance object.

Iterative Optimizations

The iterative optimizations perform general dataflow and constant propagation optimizations. They are called in a loop until none of them changes the lambda. All of the optimizations were altered in some way since Optimist; most had to be updated to handle multiple return values and typed variables, and some were changed because reply is no longer an explicit hcode. However, only the new features will be described below.

Type Specializer

Local variables in Optimist II are associated with types in two ways:

1. The variable itself has a type supplied by the programmer when the variable is declared. This type applies throughout the variable's lifetime.

2. The programmer can declare types through the use of the type assertion primitive (Section A.6), or the compiler can infer from its knowledge about the types of function and method arguments and results that a variable has a particular type at a given point in the lambda. These type assertions apply only to a particular point in the variable's lifetime.

Each type asserted in this manner must have a non-null intersection with the variable's type; otherwise, no legal value could be stored in the variable and Optimist II generates an error.

The type specializer examines each variable and calculates the lub of the types it can assume throughout its lifetime, combining all the knowledge it has from assertions of the second kind. It then intersects the variable's type with the lub and makes that the variable's new, more restricted type.

Type specialization is done to improve the quality of the move elimination optimization and to permit inlining of values in the future. When the move eliminator merges two variables, it sets the new variable's type to the lub of the variables' types. The temporaries created by other optimizations often have type `object` even though they can contain only more restricted values, and if one of them were merged into an existing variable, that variable's type would also become `object` unless the temporary's type were specialized first by the type specializer. When Optimist II supports inline classes in the future, type specialization of a variable to an inlineable class will permit some objects such as double-precision floating point numbers and locks to be inlined in local variables.

Dataflow Optimizer

The dataflow optimizer has an extra optimization in addition to those mentioned in [21]. The dataflow optimizer always checked whether an if statement would always branch one way and eliminated the if statement and the dead branch if that is the case. In addition to that check, if the if statement has several predecessors, the dataflow optimizer now checks each one separately whether it would cause the if statement to always branch one way; if so, that predecessor is connected directly to one of the if statement's branches. This situation arises often when `sc-and` and `sc-or` are used. A code fragment like the one below is generated for `(if (sc-and a b) (f))`:

```
(IF :FALSE (LOCAL CST::A) 2246)
(MOVE (LOCAL 387) (LOCAL CST::B))
(JUMP 2248)
(LABEL 2246)
(MOVE (LOCAL 387) #<False>)
(LABEL 2248)
(IF :FALSE (LOCAL 387) 2252)
(APPLY NIL (#<Lambda CST::F>))
(LABEL 2252)
```

It is optimized to:

```
(IF :FALSE (LOCAL CST::A) 2252)
(IF :FALSE (LOCAL CST::B) 2252)
(APPLY NIL (#<Lambda CST::F>))
(LABEL 2252)
```

Constant Folder

The constant folder performs two duties: it evaluates constant expressions and replaces method calls with function calls. The constant folder examines each application statement in the lambda. If the arguments are all values, the function or method to be invoked is side-effect-free, and the precise mode is off[1], the constant folder calls the interpreter to evaluate the function or method call and replace it with move statements of the results to the application's targets. If the interpreter generates an error, the compiler aborts the compilation; the error is not hidden until runtime. The call could potentially invoke many functions and methods.

One has to be a little careful with this optimization—if all inputs to a program are specified, Optimist II is perfectly willing to precalculate the program's results and compile the entire program to a single function that returns the answer. This will happen often on benchmarks,

---

[1] See Appendix B.

especially when Optimist II learns how to automatically determine which functions are side-effect-free; currently it assumes that a function is not side-effect-free unless explicitly declared so by the programmer.

In addition to evaluating applications with all arguments specified, the constant folder also simplifies built-in operations such as arithmetic and logical primitives according to the identity rules listed in Table A-4.

When the constant folder encounters a method call, it looks in the selector's table of methods and selects all methods which match the number and types of arguments provided. The arguments' types are determined by dataflow analysis of the same information as is used by the type specializer. If no methods match, the constant folder signals an error. If exactly one method matches, the constant folder replaces the method dispatch with a direct call of the method's code, which may even be inlined later. If two methods match, the constant folder uses a heuristic to determine whether it is better to do a standard method dispatch or to get the type of the first argument and call one of the two methods depending on that type.

The heuristic is as follows: The classes of the first arguments accepted by the two methods are determined. If the two classes are disjoint, the constant folder picks the class that is easier to check[1]. If one is a subclass of the other, the constant folder picks the subclass; otherwise, the constant folder gives up and does not optimize. If the picked class is easier to check than doing a method dispatch, the constant folder replaces the application with a call to the class's predicate followed by an `if` statement with direct calls to the two methods on the two sides of the conditional.

Function Inlining

Functions are inlined after all iterative optimizations have been performed and can yield no more improvements. To inline functions, the function inliner considers each function call[2] in the lambda. If the function is not a built-in and is declared inlineable, the function inliner attempts to inline it; however, there is no *a priori* guarantee that it will succeed. A function is considered inlineable if it is either declared `inline` by the user or heuristically inlineable and not declared `not-inline` by the user. To be heuristically inlineable, a function has to be small—its optimized hcode can contain no more than two full-fledged function or method calls and no more than twelve built-in calls. A point system is used to determine a function's "size;" the threshold can be varied by adjusting the `inline-size-cutoff` option.

Furthermore, to prevent object thrashing on the J-Machine, a function is heuristically uninlineable if it references an instance variable of an object passed as its first argument if the caller of that function does not pass its first argument through as the first argument of the function. To see an example of this rule, consider the function `sum4` in:

```
(defclass pair () car cdr)
(defun sum4 (p:pair q:pair) (+ (car p) (car q) (cdr p) (cdr q)))
```

The `car` and `cdr` accessor methods are well under the size threshold. However, only the `(car p)` and `(cdr p)` calls are inlined—`(car q)` and `(cdr q)` are not because the calling function `sum4` does not pass its first argument `p` as `car`'s or `cdr`'s first argument. There is no problem with inlining `(car p)` and `(cdr p)` into direct accesses of `p`'s instance variables because `sum4` is executed on the same node on which `p` resides. However, if `sum4` were to reference `q`'s instance variables directly, it would force `q` to travel to the same node on which `p` resides, thrashing `q`. Instead, `sum4` calls `(car q)` and `(cdr q)` in the usual manner, and

---

[1] Each class has an integer that specifies how easy it is to test an arbitrary object for membership in that class. If that integer is zero, doing this check is no easier than doing a method dispatch; if that integer is a high positive value such as six or seven, this test can be done in one or two assembly language instructions. Built-in classes such as `boolean` or `null` allow easy membership checking, while user-defined classes do not.

[2] Method calls cannot be inlined unless they were previously converted into function calls by the Constant Folder.

the `car` and `cdr` methods are executed on q's node and return their results to `sum4` running on p's node.

The function inliner tries to avoid forcing objects to migrate whenever possible. This is not necessarily the optimal strategy—in some cases it might be better to migrate an object to a method that accesses it frequently—but the desirability of migrating the object is difficult to determine by the compiler because it depends on the frequency of the object's use by other processes in the system. Thus, the simple solution of minimizing object migration was taken; in the cases outlined above, a method that makes numerous distant object accesses can usually be rewritten as several communicating methods which only access local objects.

Once the function inliner decides whether it would like to inline a function, it attempts to inline the function's optimized hcode. Nevertheless, it might still encounter difficulties if the inlined function performs nontrivial processing after it returns its result. For example, consider the functions `silly-add`, `shell1`, and `shell2`:

```
(defun silly-add (x y)
  (reply (+ x y))
  (prove-fermats-last-theorem)
  (exit))

(defun shell1 (x y)
  (cset ((z (silly-add x y)))
    (+ z 5)))

(defun shell2 (x y)
  (silly-add (+ x 5) y))
```

If the function inliner were to inline `silly-add` in `shell1`, it might convert a terminating program into a nonterminating one (assuming `prove-fermats-last-theorem` does not terminate in any reasonable amount of time in this example) because `shell1` would try to execute all of `silly-add` before continuing with the addition of 5 to `z`. Thus, the function inliner should not inline any function that performs nontrivial processing after it replies to its caller. On the other hand, there is nothing wrong with inlining `silly-add` in `shell2` as long as `shell2` is tail-forwarded because `shell2` would still return the sum to its caller before trying to prove Fermat's last theorem. Other interesting scenarios with callers and callees accessing the same lock are also possible.

The general rule for determining whether it is safe to inline a function is as follows: inlining is safe unless the inlined function performs nontrivial processing after replying to the caller all return values that the caller is not tail-forwarding. It does not matter if or when the inlined function replies to any other functions in whose lexical environment it might be; i.e. non-local lexical returns by the inlined function are fine as long as they don't transfer control to the caller[1].

After copying the inlined function, the function inliner implements the above rule. It runs a dataflow analysis on the continuation local variables in the inlined function to determine where each continuation reference can return its value; if it has any problems with performing this analysis, it does not inline the function. Next, the function inliner uses the dataflow problem solver again to verify that no statement that returns a value to the caller is followed by any statement that might not terminate.

Once all of these conditions are satisfied, the function inliner splices the inlined function's code and local variables into the caller. Then it introduces move statements to move the caller's arguments to the appropriate locals in the callee. If the callee wasn't non-strict, each argument is touched as it is moved. Also, the statements returning values from the callee to the caller are modified to store the values in more temporaries, which are moved to their proper destinations after the spliced callee's code. Needless to say, the move eliminator will

---

[1] However, for simplicity Optimist II does not inline closure calls.

have a lot of work cleaning up the extra moves just introduced, but they are necessary to make sure that functions are inlined correctly in all cases.

To make sure that the compiler terminates, it does only one pass of function inlining for each lambda; otherwise, it could peel invocations of recursive functions forever. However, the single pass of inlining does not mean that functions are only inlined one level deep; on the contrary, the callees are themselves fully optimized before being considered for inlining, and in the process of being optimized they may let other functions be inlined into them. It is true, though, that the treewalker's antirecursion rules prevent a function from being inlined into itself.

Once all potential functions are inlined, Optimist II performs another pass of iterative optimizations to clean up and optimize the code introduced by the inlining process.

Cleanup Transformations

Just one final cleanup transformation is done on hcode. The preceding optimizations generated a number of local variables in the lambda, many of which are no longer used. The local eliminator removes all unused locals and renumbers the remaining locals to fill the gaps. This simple transformation has no effect on the code generated by the compiler because Optimist II's third phase will compact the locals anyway. The local eliminator is present solely for aesthetic and compilation speed reasons—hcode is less readable if it has many unused local variables. Also, since variable bitmaps are represented as integers, the dataflow code runs much faster if no more than about thirty variables are present so Lisp can use fixnums instead of bignums.

MDP-Specific Transformations

A number of MDP-specific transformations have to be done on hcode before MDP assembly code can be generated. These transformations and optimizations are sketched below and are listed in the Postoptimizer file.

Global Expander

The global expander implements global variables as instances of the `global` class. Each reference to a global variable is replaced with a reference to a `global` instance object's `global-value` slot[1]. The global instance object itself is a mutable immediate object; its ID and initial value are known to the compiler, so the instance object can be referenced by any lambda without having to access another global.

Addressing Mode Flattener

The addressing mode flattener flattens nested hcode lvalue and rvalue expressions[2] because the assembly language compiler can only compile one-level expressions. Whenever the addressing mode flattener finds a nested lvalue or rvalue expression, it unnests it and precedes the hcode containing it with other hcodes that calculate the expression's components and store them in local variables.

Statement Splitter

The statement splitter is the first of two MDP built-in optimization filters. This filer converts associative built-ins such as + and `and` with more than two arguments into chains of two-argument built-in calls, removes type-assertion statements which are no longer needed,

---

[1] The `global` class name and accessors to its `global-value` slot are all `undef`'d just after they are created, so they cannot be referenced by user Concurrent Smalltalk programs, and no name conflicts can result.

[2] See Tables 5-3 and 5-4.

and expands many primitives and hcodes such as `cas`, `make-closure` and `force` into their components.

Built-in Optimizer

The second MDP built-in optimization filter is the built-in optimizer. This optimizer reduces the strength of some built-in operations such as multiplication and division by converting them into logical shifts using the identities in Table A-4.

The built-in optimizer is followed by another call to the touch eliminator, which is able to eliminate more touches than it could previously. At this point the touch eliminator can depend on built-ins not being optimized out, so it can remove touches of values which are subsequently used by built-ins. For example, if a touch of a is immediately followed by an application of + to a and b, the touch can be eliminated; it could not have been eliminated before because the + might have been eliminated or another statement inserted between the touch and +.

Instance Variable Target Transformer

This transformation and the following two correct quirks in the MDP and Cosmos architectures. One restriction of the Cosmos design is that the targets of full-fledged applications can only be local variables in the context; applications other than built-ins cannot store their results directly into instance variables or into locals in places other than the context. The instance variable target transformer scans for application statements that store their results in instance variables and modifies them to store the results in local variables and then move them into the instance variables.

Grab Introducer

The grab introducer generalizes the instance object access mechanism in Optimist. While Optimist could access at most one instance object in a lambda, Optimist II can access many. Unfortunately, there is only one MDP address register, `ID2`, assigned to holding pointers to instance objects. Hence, before every statement s that might access an instance object, the grab introducer checks the value of `ID2` left from the previous statement; if that value is incorrect, the grab introducer inserts a grab statement just before s to put the right object into `ID2`. If s accesses many instance objects, the grab introducer inserts moves and uses other statement-specific techniques to make s access only one instance object; doing this well can become quite an involved process for some hcodes.

The grab introducer also generalizes the instance object part of the Context Optimization transformation found in Optimist—if an instance object is not referenced, there is no need to point `ID2` to it and possibly force it to migrate.

Cfuture Parameter Eliminator

The cfuture parameter eliminator complements the instance variable target transformer by eliminating application statements that store their results back in a lambda's parameters. Unlike Optimist, Optimist II allows function and methods to use their parameters just like any other local variables, and, in particular, write into them. However, the operating system does not support cfutures in a function's parameter area. Hence, if the cfuture parameter eliminator finds a parameter p used as a target of a full-fledged application, it creates a new local variable l, emits a move to copy p into l upon entry to the function, and substitutes l for every use of p in the function.

Enter/Exit Introducer

The last two filters are another call to the local eliminator and the introduction of enter and exit hcodes at the beginning and end of the lambda, respectively. The compiler will compile these hcodes to the entry and cleanup code for the lambda.

3.4. Code Generation

The third phase of Optimist II contains the hcode compiler, assembly optimizer, and assembler. The hcode compiler compiles hcode into an assembly language *module*, which is a digraph of assembly language statements. The assembler and the assembly optimizer then insert branches into the module and perform peep-hole optimizations on it. Since the hcode compiler, assembly optimizer, and assembler were all present in Optimist, only the differences will be described here.

New Hcode Compiler Features

The hcode compiler has been updated for CLOS, the new Concurrent Smalltalk, and the new Architecture version 11B. Major Concurrent Smalltalk changes affecting the compiler include introduction of multiple values to application statements and the introduction of many built-ins which compile into MDP system calls or sequences of MDP instructions. Built-ins for even such low-level facilities such as reading or checking tags were provided, and are accessed by the Concurrent Smalltalk runtime system.

The context and variable allocation schemes have changed somewhat. Optimist's graph-colorer for allocating context local variables worked well and has been extended to also allow slots in the message to be reused as local variables; thus the slots in the incoming message and the slots in the context form a pool of slots to which the compiler can allocate local variables at will. The only restriction imposed by Cosmos is that local variables which might contain cfutures cannot be assigned to incoming message slots.

Unlike JOSS, Cosmos fixes the locations of the saved registers in the context. If a function would need more slots than the fourteen provided in a standard context, Optimist II assigns the extra locals to slots after the saved register area in the context[1], up to a limit of 53 total slots; the MDP cannot readily address more than 64 words in an object, and 11 are used for overhead. If a large context is needed, Optimist II emits code to create it when the function starts execution and dispose it when it is done.

One architectural change had considerable impact on all stages in the third phase. Architecture 11B allows long immediate constants and long displacement into objects on most two-operand instructions but not three-operand ones. Optimist II takes advantage of these operations whenever possible, but handling the worst case possibilities is now more complicated. For example, it is no longer true that an ADD instruction allows the same addressing modes as a NEG.

New Assembler Features

The assembler has been upgraded to output many kinds of objects instead of just code. When it encounters the use of a pointer to an object inside another object, it outputs an MDPSim reference to the pointed object. MDPSim resolves all of these references when it downloads the objects to its simulated J-Machine.

Global Compilation

Unlike Optimist, which compiled isolated modules, Optimist II compiles entire programs. Hence, it has the additional duty of emitting the "glue" that holds programs together. In particular, it emits class definitions, method tables, data objects, and code objects. It emits all class definitions first because they are needed to load other objects. The order of the other

---

[1] See Figure 4-9.

objects does not matter because MDPSim can resolve references in any order. After emitting objects, Optimist II emits code that automatically downloads the objects into the J-Machine.

Identifiers

Since MDPSim currently allows only alphanumeric characters and underscores in its identifiers, Optimist II converts any identifier characters outside that set into strings of characters in that set. Next, Optimist II prepends the kind of identifier to each identifier it emits. The kinds are listed in Table 3-8. Finally, Optimist II checks whether another identifier with the same name has been emitted. If so, and if the other identifier is not eq to the current one, Optimist II disambiguates the current identifier by appending two underscores and a number to it. This transformation is necessary because sometimes many anonymous functions are generated.

Table 3-8. Identifier Prefixes

Kind	Prefix
Class	c
Selector	sel
Symbol	sym
Function	f
Other Object	o

IDs

To allow downloading of circular data structures, Optimist II assigns IDs to all objects it emits. In order to do this assignment, it has to know how many nodes there are in the J-Machine for which it is compiling. This number is provided in the n-nodes Optimist II option.

Optimist II uses increasing positive integers to generate serial numbers for classes, selectors, and symbols. Functions and other objects are assigned IDs starting with the serial number $7FFF and decreasing to avoid conflicts with serial numbers generated by Cosmos, which start at $0000 and increase. Optimist II tries to distribute the objects it creates evenly throughout the MDPs in the J-Machine.

Method Tables

The Optimist II's assembler generates a method table which associate methods with classes and selectors. The method table is distributed among the class and selector objects loaded into MDPSim. Each selector (Figure 4-17) contains a list of class/method pairs that describe all methods defined for that selector. In addition, each class object (Figure 4-13) contains an ordered list of the class's ancestors.

Together, the two objects contain enough information to deduce the method associated with a class and selector: first the class is looked up in the selector's list of class/method pairs; if it is not found, the ancestor classes are looked up, one by one, in that list. Either a binding is found, in which case the binding contains the desired method, or no binding is found, in which case there is no method associated with the given class and selector.

Data Formats

The formats of built-in objects emitted by Optimist II are described in more detail in the next chapter. Primitive objects are listed in Figure 4-2, instance objects of user-defined classes are shown in Figure 4-4, and functions are shown in Figure 4-20 and closures in Figure 4-22. Optimist II cannot emit immediate distributed objects, but they can be created at runtime.

3.5. Conclusion

The main goal of writing Optimist II was to bridge the gap between Concurrent Smalltalk and the J-Machine. Optimist II is the first compiler that can compile a Concurrent Smalltalk program into code that can be run on a J-Machine without any changes. Unlike Optimist, which compiled only modules, Optimist II compiles entire programs, including the class hierarchy, method tables, functions, and immediate objects. Furthermore, Optimist II supports a much larger subset of Concurrent Smalltalk than Optimist. Optimist II supports the entire language except for full futures and I/O facilities.

Observations

The global optimizations included in Optimist II are very useful, as they free programmers from having to break abstraction barriers in order to achieve reasonable performance. This consideration alone was a great help in writing the runtime system. Many of the built-in functions such as `zero?` and instance variable accessors are candidates for inlining, and, in fact, they are often inlined into user programs. Without global optimizations writing an efficient runtime system would have been difficult and error-prone. `zero?` could perhaps have been implemented as a macro, but then it would not be possible for a user program to override it for its own classes. Moreover, `zero?` would then suffer from the classic Lisp problem of a macro not being a first-class data object and interchangeable with functions. Also, inlining of functions may be controlled by fairly sophisticated heuristics, while macros would always be expanded.

The substitution of function calls for method calls is also a useful optimization. In simple programs almost all method calls are replaced with function calls and then often inlined. In fact, in all the simple and non-contrived examples I have compiled, Optimist II was able to remove all method dispatches and replace them with function calls. Even in applications using Lisp-style lists, there are usually at most two methods defined on an object—one method handles the `nil` case, while the other handles the nontrivial case—and Optimist II turns the method call into a comparison of the argument against `nil` followed by one of two function calls, often inlined.

Generality or Simplicity?

One recurring issue was whether Optimist II should be a compiler for a general target or a compiler specifically tailored to the J-Machine. Ideally, Optimist II should have a back end that could be replaced to compile code for a different architecture. Unfortunately, this ideal was not achieved. Although many MDP-specific transformations are collected near the end of the Optimizer, some, such as the continuation expander had to be placed earlier in the compilation process. Worse, much of the runtime system at the very front of the compiler is heavily dependent on the MDP architecture.

The two issues at odds here are generality and simplicity. Due to the limited scope and experimental nature of this project, I resolved conflicts in favor of simplicity. For example, Concurrent Smalltalk is a useful systems programming language, and it was desirable to implement some features of Concurrent Smalltalk in Concurrent Smalltalk. While this approach would make the Optimist II front end nonportable, I decided to use this approach anyway because it made the runtime system simple to write, understand, and modify.

Future Plans

Optimist II is still an evolving compiler, and it will surely change in the future. In addition to implementing the remaining language features and fixing bugs, Optimist II could be extended to implement inline objects and the load balancing ideas discussed in Chapter 8. In addition, a number of minor tweaks mentioned in [21] are still possible. Now that branches have a longer range, Optimist II could be more liberal with the use of MDP register R0 to hold values between statements[1]. A smarter register allocator could assign a variable to a register for part of its lifetime. The peephole optimizer could replace branches to SUSPEND instructions with SUSPEND instructions themselves. The implementation of closures could be made faster. The compiler could automatically detect side-effect-free and no-leak functions; this information might permit it to explicitly deallocate some objects such as closures if it could prove that they could not be referenced again. Overall, though, it seems that, except for loops which are deliberately broken to avoid hogging processors, no more than a few percent more performance can be squeezed out of the code generated by Optimist II; however, since the operating system overhead time overwhelms the execution time in Concurrent Smalltalk methods, there might be room for improvement through coordinated compiler and operating system changes.

---

[1] In Architecture 10, all but the shortest branches required the value of R0 to be altered, rendering that register practically useless for holding values between statements.

Chapter 4. The Cosmos Operating System

Design Goals

The Cosmos operating system was designed primarily as a support kernel for running Concurrent Smalltalk programs on the J-Machine. Nevertheless, Cosmos is not specialized to Concurrent Smalltalk, and many of the operating system's components could be used to support a general message-passing environment.

The goals in designing the operating system were, in order:

1. To make a working operating system.
2. To make the operating system as efficient as possible.
3. To make the operating system as simple and flexible as possible.

The design of the operating system also had to be small enough to allow both it and most of the Optimist II compiler to be written in one semester; for this reason garbage collection and load management facilities were not included in the operating system. Several steps were taken to achieve goal (1), including the criticality system and the debugging techniques described later. The criticality system is an organized accounting method used to ensure that no re-entrancy problems occur when operating system routines call each other. Features were added to MDPSim to detect and signal race conditions known as hazards. To achieve goal (2), the entire operating system kernel was written in hand-optimized assembly language. Poor J-Machine performance can no longer be blamed solely on the operating system. Goal (3) was achieved by providing general data structures that are reused in many components of the system.

Functionality

The operating system assists Concurrent Smalltalk programs by providing the following services:

- Initialization and setup of the J-Machine.
- Providing fault handlers for faults needed to keep the J-Machine running.
- Global function calls and returns.
- Looking up methods corresponding to class/selector or object/selector pairs.
- Context allocation and deallocation facilities and conventions.
- Local and global object allocation, deallocation, lookup, and migration facilities. Mutable objects exist on only one node at a time, while immutable objects can exist on many nodes at a time; all but the primary copy can be purged when extra memory is needed.
- Support for distributed objects as defined in Concurrent Smalltalk.
- Support for Concurrent Smalltalk primitives such as determining the type of an object.
- Calls assisting in the creation and evaluation of closures.
- An integer division routine.
- Debugging and consistency-checking facilities.

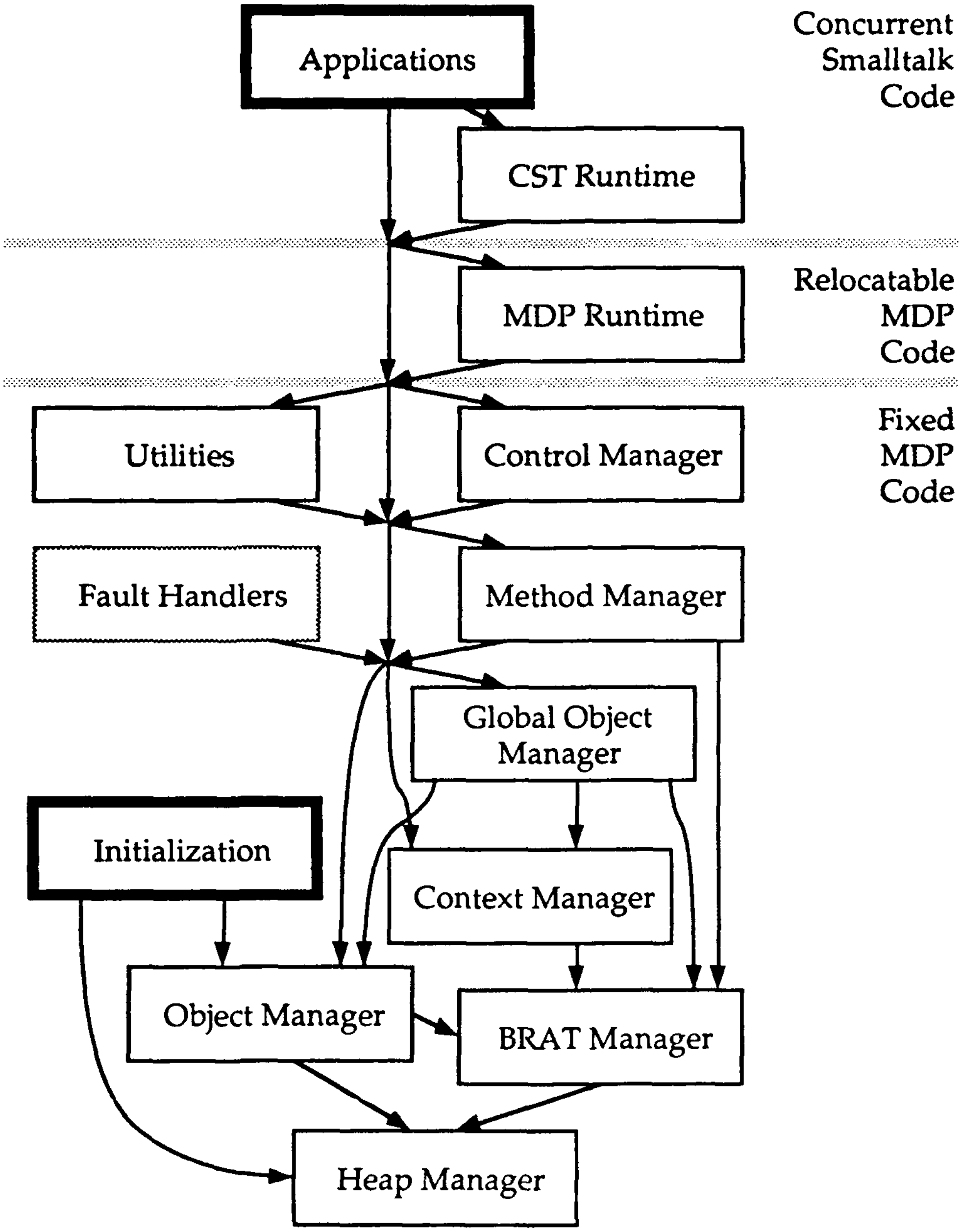


Figure 4-1. Operating System Organization

The arrows represent calling patterns in the Cosmos operating system. Every module uses the fault handlers; those dependencies were omitted for clarity. The modules in bold boxes are roots—they are invoked by the user.

The modules in the top section of the figure are written in Concurrent Smalltalk; however, the CST Runtime module may not necessarily be portable to other Concurrent Smalltalk implementations because it references some MDP data structures. The modules in the middle section are written in MDP assembly code because they implement functionality that cannot be easily expressed in Concurrent Smalltalk. From the point of view of the rest of the operating system, though, these modules are indistinguishable from compiled Concurrent Smalltalk code. The modules in the bottom section are fixed in the memory of every MDP either because they are critical to the MDP's operation or because calling them as functions would be inefficient.

After Cosmos initializes the J-Machine, a Concurrent Smalltalk program can be loaded using Cosmos's downloading facilities. Once the program is loaded, a single call to Cosmos's `Apply` handler can start the execution of one function in the program. Whenever a function needs to invoke another function or method, it first calls the Cosmos `ObjectNode` routine[1] to determine a good node for that invocation and then sends an `Apply` message or one of its variants to that node. The target node, upon receiving that message, executes the Cosmos `Apply` handler that fetches the function or method code and calls it.

Many functions need to store local state in memory, either because they need more variables than will fit in the MDP's registers or because they make function or method calls and need a place in which to save state for the duration of the call. Cosmos uses contexts to save state and provides routines to allocate and deallocate them.

In addition, Cosmos manages objects globally, migrating objects and code to the nodes that need them. Cosmos keeps only one instance of immutable objects, but it can make copies of immutable objects and code. Also, Cosmos provides routines to determine the type of an object and to create and address distributed objects. Finally, Cosmos provides primitives such as division that would be hard to implement in Concurrent Smalltalk.

Structure

The operating system is composed of interacting modules shown in Figure 4-1. The high-level modules are built in layers out of lower-level ones; however, the low-level modules are deeply interrelated because of the hardware restrictions of the MDP. Furthermore, due to efficiency considerations and hardware restrictions on faulting, much of the code in some of the managers is inlined inside other managers. This is especially common at the lowest levels such as the heap and context managers.

Reading Guide

This chapter describes the handlers in the two lower sections of Figure 4-1; the Concurrent Smalltalk code is described in Chapter 3. After a brief overview, the handlers will be described in this chapter from the bottom level up.

Heap Manager

The heap manager manages the heap on each MDP. The heap allows allocation, deallocation, and purging of arbitrary objects in the local memory on the MDP. All object references are bounds-checked, and primitive compaction facilities are provided.

BRAT Manager

The BRAT manager keeps track of the *BRAT*—Birth/Residence Address Table [38]. The BRAT is an associative table used mainly for translating virtual addresses to physical addresses, although it is also used for some housekeeping tasks in object migration.

Object Manager

The object manager combines the facilities of the heap manager and the BRAT manager to provide a virtual name space for the objects allocated by the heap manager. The object manager is capable of allocating objects on the local node and giving them unique names. It can also determine that an object does not reside on the local node, but it cannot access nonlocal objects.

Context Manager

The context manager keeps track of *contexts*. A context is the MDP equivalent of an invocation descriptor on a conventional computer. The context contains values of the local variables

---

[1] Sometimes that call is optimized by Optimist II to a single `MOVE` from `NNR` instruction.

of a process, saved data and ID register values, and the instruction pointer (IP) when a process suspends.

Global Object Manager

The global object manager is an extension of the object manager to the global virtual address space of the J-Machine. The global object manager can access nonlocal objects, and it can migrate objects between nodes. It can distinguish mutable objects from immutable ones and maintain copies of the latter on many nodes.

The global object manager also can determine the class of an arbitrary object. and it is the lowest level in the operating system that implements distributed objects.

Method Manager

The method manager implements an association between classes, selectors, and methods on top of the global object manager. The method manager can, given a class and a selector, quickly determine the appropriate method that represents applying the selector to an object of that class.

Control Manager

Function and method calls and replies are dispatched by the control manager. Every function or method call is actually a message send to an entry point in the control manager, which interprets the incoming message, makes sure it is valid, fetches the called code, and runs it. The control manager also handles suspending after cfuture faults and resuming when a called function or method returns a value.

Utilities

The operating system kernel includes commonly-used utilities that would suffer too much overhead if they had to be called via the standard function call mechanism. The current utilities include a divide system call and calls that create and evaluate closures.

MDP Runtime

The MDP runtime system contains other utilities that have to be coded in MDP assembly language. Currently MDP runtime utilities include a method table lookup routine and functions that create distributed objects. When arrays are implemented, they will also be implemented as MDP runtime utilities.

CST Runtime

The CST runtime system contains utilities which could be coded in Concurrent Smalltalk. These utilities implement most of the functions and macros listed in the Concurrent Smalltalk reference manual (Appendix A), including locks, some array code, and object-handling functions such as copiers and destructors, as well as lower-level functionality such as global variables.

Data Representation

Figure 4-2 shows an overview of the representations of various Concurrent Smalltalk objects. The representations of the complex Concurrent Smalltalk object such as functions, selectors, and classes will be explained in more detail in the following sections.

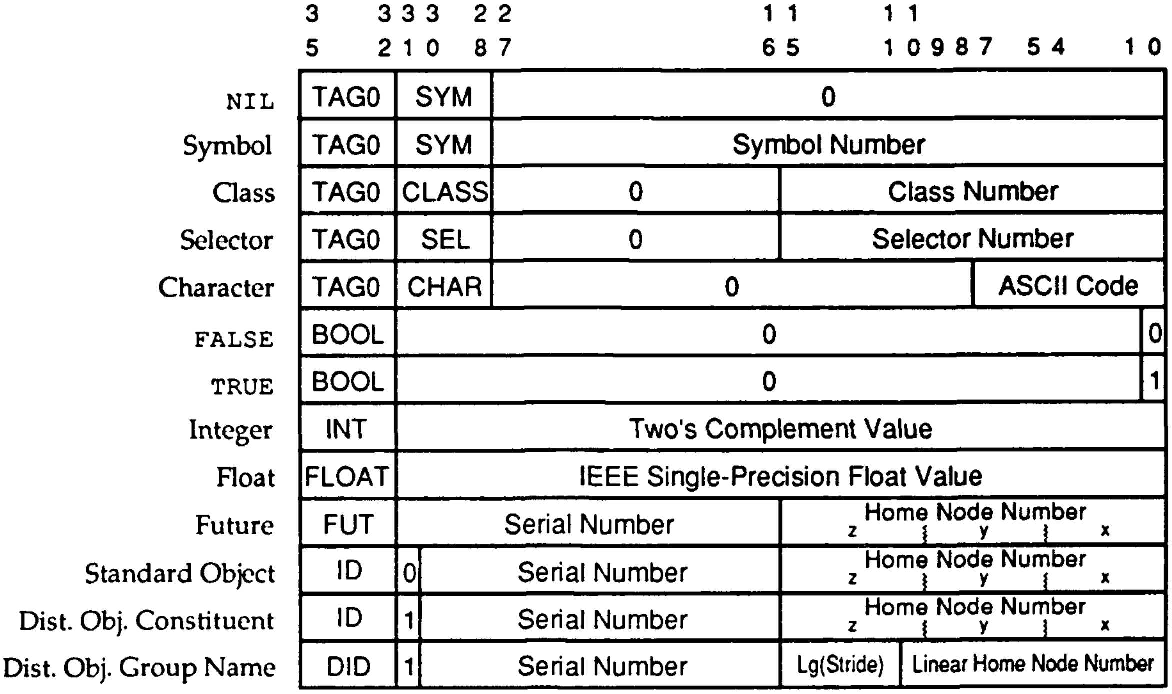


Figure 4-2. Concurrent Smalltalk Object Representations

Primitive objects are represented as above using the MDP's 32-bit words with 4-bit tags. Objects not shown above are represented as standard objects using the ID tag. Due to a shortage of tags, `NIL`, symbols, classes, selectors, and characters share the same MDP tag, TAG0 (also known as SYM), and are distinguished by the upper four bits of the data word. One MDP tag, TAGA, has been retained for future expansion.

With the current bit layouts, Cosmos is limited to representing 268435456 symbols, 65536 classes, 65536 selectors, 65536 futures, 32678 objects per node, and 32768 distributed objects in the entire system. The last three limitations are especially severe and will be considered in Chapter 8.

4.1. Hardware Building Blocks

Memory Organization

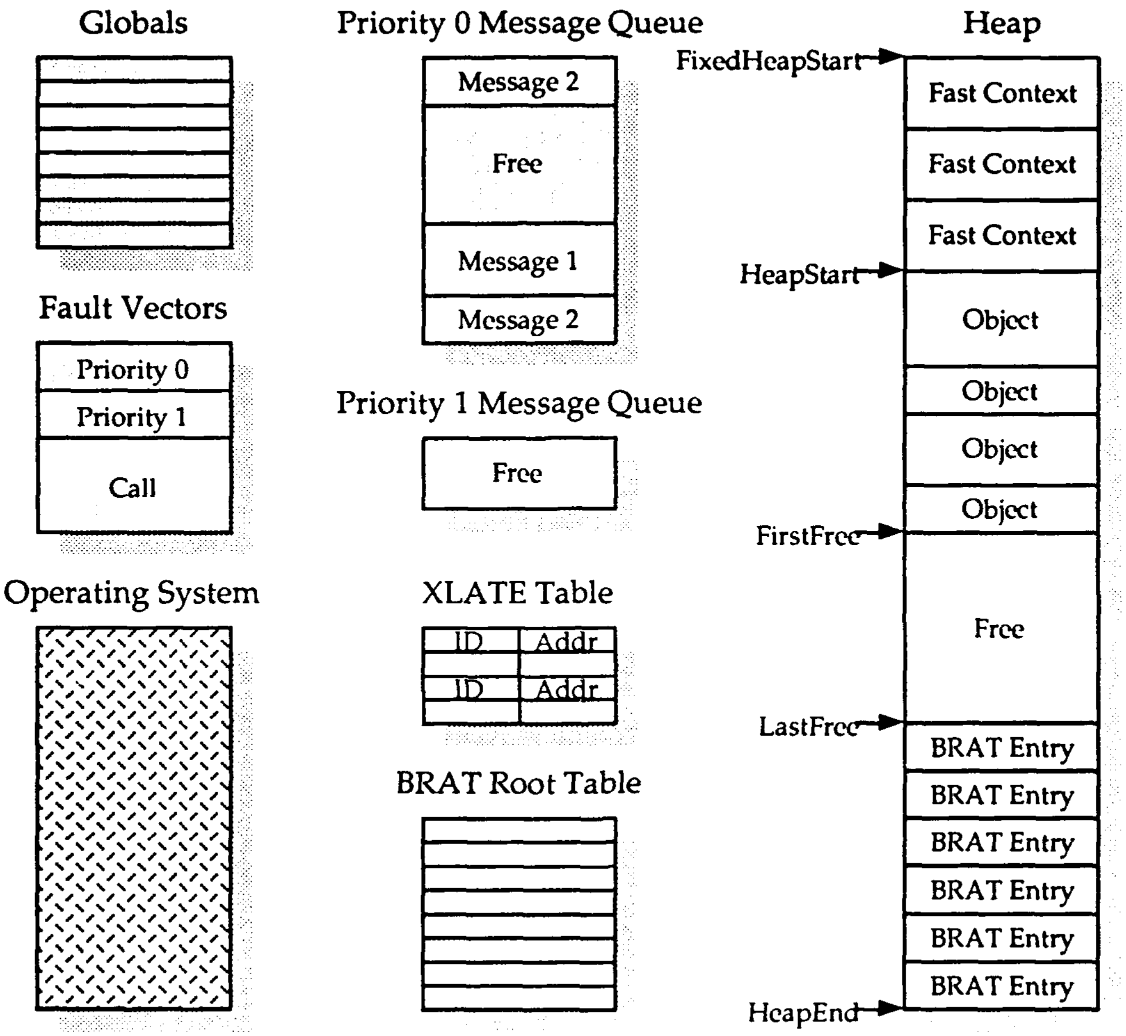


Figure 4-3. MDP Memory Organization

The data structures above are replicated on every MDP in the J-Machine. All of the data structures except the heap reside in fast RAM. The top of the heap resides in fast RAM, but most of it is in slow RAM.

Figure 4-3 maps the structures addressable in the physical address space of every MDP. The heap occupies most of memory and is used for storing and keeping track of Concurrent Smalltalk objects and contexts. The BRAT root table is a separate hash table that points to the BRAT entries in the heap. The XLATE table is a table used for hardware-assisted associative lookups. In addition, every MDP contains a copy of the Cosmos code and fault vector assignments and a small set of globals used by Cosmos and some of the runtime routines. Finally, every MDP contains two hardware-managed incoming message queues.

Priorities

Each MDP provides three levels of execution priority—background, priority 0, and priority 1. The network allows messages to be sent at priority 0 or 1; when a message of a given priority arrives at a destination node, it is queued in the appropriate priority's queue. The queues are constantly monitored by the CPU, and if a queue contains a higher-priority message than the task currently running, the current task is pre-empted to handle the message.

Cosmos currently only uses the background and priority 0 levels. It is anticipated that priority 1 will be used in the future for garbage collection and resolving emergencies such as queue or memory overflow. In addition, on a real J-Machine (as opposed to MDPSim), priority 1 will make a good debugging channel. Cosmos's use of the background priority is currently limited to initialization; it would be nice if background mode could be used for incremental heap compaction, but that may be difficult—because of flaws in the MDP architecture, the background priority and priority 0 share the same sets of globals, ID and fault registers, and fault vectors, meaning that execution of a priority 0 message is likely to clobber the state of a background process.

4.2. The Cosmos Kernel

Criticalities

Cosmos was fairly difficult to write because almost all of its routines are non-reentrant; thus, locations of faults inside Cosmos code have to be carefully controlled. The MDP does not include any stacks, which means that each routine and fault handler must save its state in a different set of global variables. Furthermore, the low-level routines have to be very careful not to alter the same global or register through some combination of system calls and faults. Another class of problems consists of critical sections of code in which physical addresses are manipulated in data registers or objects are referenced assuming they are present in the local memory. No heap compaction or object migration is allowed in those sections. If a heap compaction or object migration were to occur in such a section, the physical address or object reference would become invalid.

To make these problems tractable (but, nevertheless, still difficult), the concept of a *criticality* was introduced. The criticality of a system call is a number which reflects what actions that system call is allowed to perform. The criticalities are listed in Table 4-1.

A routine with a given criticality may not call another routine with a lower one. For example, if a routine is sending a message, it may not make a system call or allow a fault of criticality less than 4 while it is sending the message. Thus, the routine has to force any potential cfutures before sending the message, because a cfuture fault has criticality 1. If a routine stores a physical address of a heap block in a data register, it must have criticality at least 5 as long as the address can be read out of the data register. If a routine runs with the MDP's fault bit set, it must have criticality at least 6 to prevent a catastrophic double fault. There will be no re-entrancy problems as long as each routine's criticality is correct, the criticality rules are obeyed, and all possible faults are anticipated.

Heap Manager

The heap manager manages the heap on each MDP, allowing allocation, deallocation, and purging of arbitrary objects in the local memory on the MDP. The heap manager does not use the network, so most of its routines run at criticality 5.

Heap Structure

The heap, shown in Figure 4-3, is organized as a contiguous block of memory. Objects are allocated from the bottom (lower addresses) up, while BRAT entries are allocated from the

Table 4-1. Criticalities

Value	Actions Allowed
0	All actions are allowed. Caller's registers do not have to be preserved.
1	Caller's registers must be preserved. May suspend, so MDP's globals are not preserved.
2	No suspending faults, no modification of context state.
3	No suspending faults, no modification of context state, no object migration.
4	No message sends, no object migration.
5	No heap compaction, no message sends.
6	No faults or system calls, no heap compaction, no message sends.
7	No priority 1 interrupts, no faults or system calls, no heap compaction, no message sends.

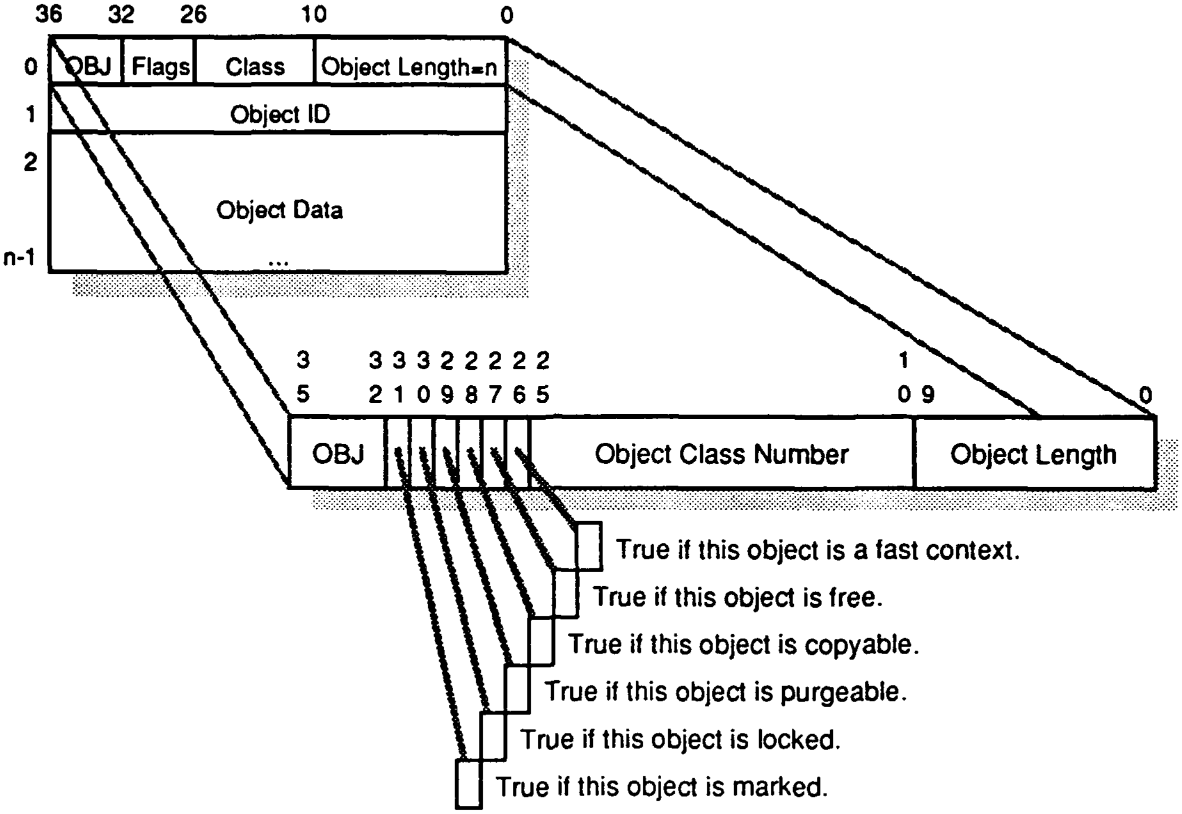


Figure 4-4. A Heap Block
Each MDP heap block consists of a header and ID words followed by user-defined data.

top down by the BRAT manager. The objects in the heap between `FixedHeapStart` and `HeapStart` are nonrelocatable—once allocated, they are never moved. Currently that area is used for storing a few fast contexts. The rest of the heap is dynamically divided between relocatable objects and BRAT entries. The `FirstFree` pointer points to the first unused word of heap memory, while the `LastFree` pointer points to the first word used for BRAT entries.

Heap Blocks

Each heap block has the structure shown in Figure 4-4. The presence of the length of the block in the first word and its virtual ID in the second word allows the heap to be scanned and compacted quickly.

The heap manager uses only the free, purgeable, and marked flags, which have the following meanings:

- Free. The heap manager will reclaim storage from those blocks when it needs extra memory.
- Purgeable. The heap manager can purge those blocks when it needs extra memory.
- Marked. A purgeable block is marked if it has not been accessed for a while. It will be purged at the next opportunity.

The copyable and locked flags are managed by the global object manager, while the context manager uses the fast context flag to distinguish fast contexts from standard ones.

Object Allocation

Allocating an object on the heap is usually quite fast, taking about twenty instructions. Given the object ID and header word, the `AllocObject` heap manager routine checks whether there is enough room in the heap for the object[1]. If so, it creates and returns a relocatable ADDR-tagged word pointing to the physical memory that will be occupied by the object, after initializing the object's first two words and advancing the `FirstFree` pointer. If there is not enough free memory, `AllocObject` calls the heap compactor to try to free enough memory for the object.

Heap Compaction

The heap compactor is called whenever a memory request cannot be satisfied. First it invalidates all relocatable addresses cached in the address registers and the XLATE table[2]. Then it scans through the heap starting from `HeapStart`, moving each block as far to the front of the heap as possible. As each block is moved, its physical address is updated in the BRAT, but not the XLATE table[3]. Deleted blocks are not copied, nor are marked purgeable blocks[4]. If a purgeable block was unmarked, it is copied and then marked. The next time the block is referenced, that block's marked bit will be cleared by the XLATE fault handler.

A heap compaction increases the amount of contiguous available memory between `FirstFree` and `LastFree`. However, if the compaction did not free enough memory to satisfy the allocation request, another compaction is immediately done. The second compaction purges the remaining purgeable blocks from the heap. If the second compaction does not free enough memory, the system halts.

Utility Routines

The heap manager contains a couple of general-purpose utility routines which illustrate creative use of the MDP's fault mechanism. One, `BlockMove`, quickly moves a block of memory from one address to another. The routine uses straightline code followed by an infinite loop to copy data. The loop is terminated by a LIMIT fault when a copy is attempted of the first word out of bounds of the source block. Similarly, `BlockSend` quickly sends words of an object until terminated by a LIMIT fault. Without using LIMIT faults these routines would be two to four times slower.

BRAT Manager

The BRAT manager maintains the *BRAT*—Birth/Residence Address Table [38] and the XLATE table. The BRAT is a general-purpose associative table used mainly for translating virtual addresses to physical addresses. The XLATE table is used mostly as a cache for the BRAT table. Table 4-2 lists the associations currently maintained by the BRAT manager. Like the heap manager, the BRAT manager does not use the network and runs mostly at criticality 5.

The format of the XLATE table is dictated by the MDP hardware. The table is a two-way set-associative cache whose location and position are specified by the MDP TBM register. Each

---

[1] Actually, `AllocObject` makes sure that there are three more free words in the heap than necessary to hold the object in case a BRAT entry will also be allocated for the object. This avoids the difficult situation of being able to allocate a heap object but not its BRAT entry; a heap compaction in the BRAT manager would violate criticality rules.

[2] Just re-entering each association between a virtual ID and the new physical address would not work because several virtual IDs may alias to the same physical object; the copying code would find only one such association in the XLATE table.

[3] Physical addresses are not updated in the XLATE table because if they were, there would be no easy way of determining which blocks were referenced between heap compactions. The XLATE fault handler clears the marked bit of every block it encounters without a binding in the XLATE table.

[4] Nevertheless, if an object's locked flag is set, the object is preserved, even if it is also indicated as deleted or purgeable and marked. This action is required to maintain consistency in the global object manager.

binding in the XLATE table consists of a key word and a data word. Invalid bindings have a NIL data word. The XLATE and PROBE instructions hash the key they receive into the XLATE table and check the two possible bindings whether they contain the right key; if so, the corresponding data word is returned. The ENTER instruction enters a new binding into the XLATE table; that binding might overwrite an existing binding of a different key, so the XLATE table is only a cache—bindings are not guaranteed to remain in the table. The hash function used is the exclusive-or of the four bytes that constitute the data portion of the key word; the tag of the key word does not participate in the hashing. Thus, the XLATE table is limited by hardware to 512 bindings, which may not be enough if there are many small objects on a node.

Table 4-2. XLATE and BRAT Associations

"Virtual" Tag	"Physical" Tag	Tables	Association
ID	ADDR	XLATE, BRAT	Physical object location
ID	INT	BRAT	Node number of node containing object
ID	context ID	BRAT	Context waiting for object
DID	ADDR	XLATE	Physical location of nearest constituent
TAG0:SEL	ADDR	XLATE, BRAT	Physical location of selector object
TAG0:CLASS	ADDR	XLATE, BRAT	Physical location of class object
TAG0:CLASS	INT	BRAT	Node number of node containing class object
TAG0:CLASS	context ID	BRAT	Context waiting for class object
TAG0:SYM	none	XLATE	Symbols are primitive objects
TAG0:CHAR	none	XLATE	Characters are primitive objects
INT	none	XLATE	Integers are primitive objects
BOOL	none	XLATE	Booleans are primitive objects
FLOAT	none	XLATE	Floating point numbers are primitive objects
CS (INST1)	ID	XLATE	Class/selector lookup

The above table contains the current associations kept in the virtual tables. A general object (tagged ID or TAG0:CLASS) can associate either to a physical address, the node number of the node thought to contain the object, or a context waiting for the object. In the last case, if the object is being accessed, the current process suspends and puts itself onto the list of contexts waiting for the object. Selector objects are just like general objects except that they do not migrate. The DID→ADDR association is used for quickly getting to constituents of distributed objects from the group ID. The results of the DID→ADDR must be consistent through time—looking up a DID on the same node must always yield the same constituent. Looking up a primitive object other than the ones just mentioned in the XLATE table must always miss. Finally, due to a shortage of virtual tags, words tagged INST1 are used as class/selector keys to the method manager's method cache.

XLATE and BRAT Table Formats

Unlike the XLATE cache, entries in the BRAT table are guaranteed to remain in the table until they are deleted. As shown in Figure 4-7, the BRAT table is rooted by a small root hash table. Each entry in the root table points to a linked list of BRAT bindings with keys that hash to the same value. In addition, there is a linked list of free BRAT entries. There are several advantages to keeping the BRAT table organized this way instead of the flat hash table in [38]:

- Deleting entries from the BRAT is easy, while at the same time searching the BRAT for a missing key is fast. Such searches are common because they occur almost every time an object not present in local memory is referenced.

- The boundary between BRAT memory and the memory used for objects in the heap is adjusted dynamically. Thus, accurate predictions of the average size of an object needed in [38] become unnecessary.

- No memory is wasted keeping the flat hash table no more than 70% full. On the other hand, linked lists require one additional word per BRAT entry for the links; however, it is conceivable that BRAT entries could be stored contiguously with their objects, eliminating this waste.

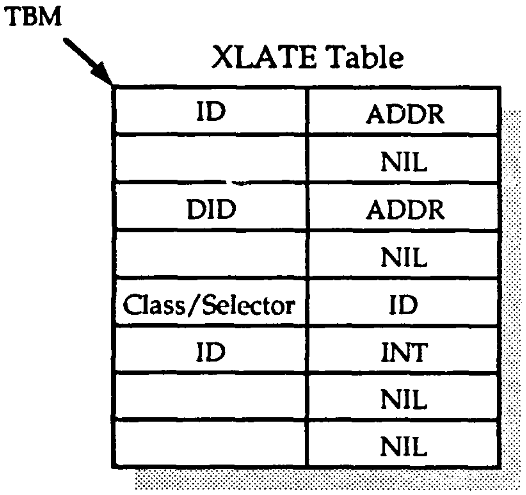


Figure 4-5. XLATE Table Format
The XLATE table's position and length are specified by the MDP TBM register. The XLATE table is a two-way set-associative cache composed of key/data pairs of words. A NIL data value specifies an invalid entry. The XLATE and PROBE instructions provide hardware support for quickly looking up keys in the cache.

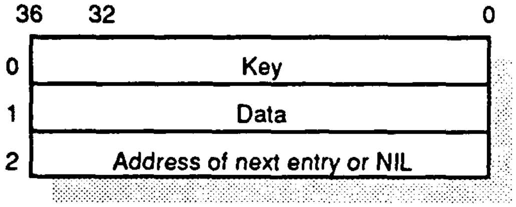


Figure 4-6. BRAT Entry Format
Each BRAT entry is a linked list entry associating a key word to a data word.

BRAT Routines

There are three main routines for managing the BRAT table. They are:

- `EnterBinding`, which enters a new binding of a key to a data word. This routine uses a binding from the `BRATFree` linked list whenever possible. However, if that list is empty, memory is allocated from the back of the heap, moving `LastFree` forward by three words, which might force a heap compaction.

- `LookupBinding`, which returns the data word associated with a key or `NIL` if there is none.

- `DeleteBinding` and `PurgeBinding`, which remove a binding from the BRAT. The binding must have been present in the BRAT. In addition, `PurgeBinding` removes the binding from the XLATE table.

Heap Compaction

The current heap compactor in the Heap Manager does not attempt to compact free BRAT entries linked on the `BRATFree` list. Thus, once memory is used for a BRAT entry, it can only be used for another BRAT entry. Nevertheless, performing such compaction by moving BRAT entries up in memory would not present any special difficulties.

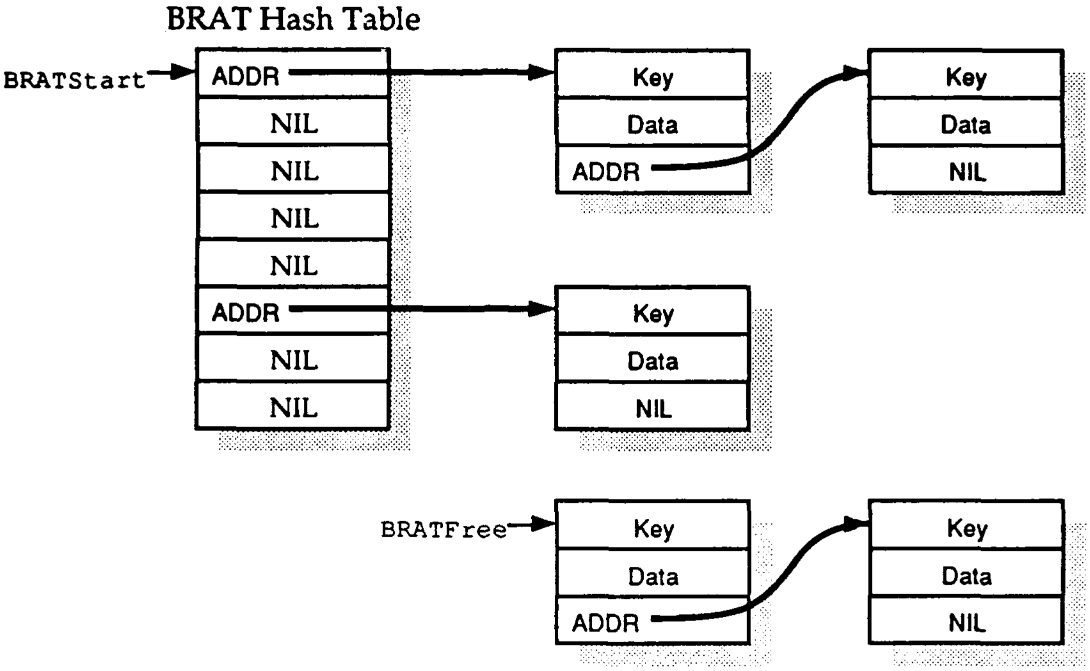


Figure 4-7. BRAT Table Format
The BRAT entries are kept in linked lists rooted in the BRAT hash table. Free BRAT entries are linked in a separate linked list.

Object Manager

The local object manager combines the facility of the heap manager and the BRAT manager to provide a virtual name space for the objects allocated by the heap manager. The local object manager can allocate objects on the local node and give them unique names. The local object manager is tightly interwoven with the global object manager, so the distinction between the two managers is only conceptual—their code is inlined together in common routines.

	35 32	31	30 28	27 16	15 0
Class	TAG0	CLASS		0	Class Number
Selector	TAG0	SEL		0	Selector Number
Future	FUT	Serial Number			Home Node Number z y x
Standard Object	ID	0	Serial Number		Home Node Number z y x
Dist. Obj. Constituent	ID	1	Serial Number		Home Node Number z y x

Figure 4-8. Object ID Formats
Words with the above formats are virtual addresses of objects on the heap. Special care must be taken when handling virtual addresses which are also futures to avoid forcing them prematurely.

Object IDs

The Object Manager recognizes several formats of object IDs and virtual addresses, as shown in Figure 4-8. In addition, the Object Manager can generate unique new standard object IDs by incrementing a local serial number counter and adding it to the local node number. Since no mechanism exists currently for reclaiming IDs, the system will fail after 32768 local objects have been allocated at one node. See Chapter 8 for a discussion of what could be done about this problem.

Each of the IDs in Figure 4-8 contains a home node number in the lowest 16 bits. For futures, standard objects, and distributed object constituents, the home node number is merely the network number of the MDP that serves as the object's home; any unused bits must be zero. However, for classes and selectors, any of the lowest 16 bits not used for storing the network number are used to distinguish among several class or selector objects sharing the same home. For example, on a 1024-node J-Machine arranged as 16×16×4, bits 0-3, 5-8, and 10-11 hold the home's x, y, and z coordinates, respectively, while bits 4, 9, and 12-15 disambiguate among classes or selectors living on the same node. For this configuration, the class object's home node number can be obtained by logically ANDing the class number with %0000110111101111. See Figure 5-12 for more on this.

Why not use bits 16-27 to disambiguate classes and selectors living on the same node, as is done for objects? The reason is that several parts of Cosmos require class and selector numbers to be no greater than 16 bits. For instance, a class number is stored in every heap object's header, and the class and selector numbers are concatenated to make a 32-bit word during method lookup.

Routines

The local object manager provides routines to allocate and deallocate objects. The object-allocating routine has two variants—`AllocNewObject` allocates an object given its ID and header word, while `AllocNextObject` takes a header word and generates a new ID for the object. Both variants then allocate local memory for the object and enter the binding of the ID to the physical address in the BRAT and the XLATE tables. `AllocNextObject` is used for most of the general object-allocating needs, while `AllocNewObject` is used in special cases—downloading of objects or allocation of distributed object constituents—where an object's ID is predetermined.

`DeallocateObject`, the local object deallocator, deletes an object's bindings from the BRAT and the XLATE tables and sets the object's deleted flag. Thus, the object will be compacted during the next heap compaction. If the object was a distributed object constituent, it might have had more than one binding in the XLATE table; only one such binding is deleted, so it might still be possible to access a deleted constituent object through the other bindings until the second heap compaction. This is not an error because the consequences of accessing a deleted object in Concurrent Smalltalk are undefined.

The object manager also provides a handler for XLATE faults. When an XLATE instruction that searches for a local object misses, the object manager searches the BRAT for the binding. If it finds such a binding, it returns the object's physical address and enters the object's binding back in the XLATE table. This is also the point at which the heap manager unmarks the object if it was previously marked. If the object's binding was not found in the BRAT, further action depends on the value of the XLATE action code[1]—the XLATE fault handler might use the global object manager to bring the object onto this node, return `NIL`, or fail.

Context Manager

The context manager maintains contexts which contain local variables and saved register values and messages of processes. The structure of a context is shown in Figure 4-9. MDP's register `ID1` contains a virtual address of the current context at all times when a context switch is possible, while `A1` contains the physical address and length of the context. Contexts are used for the following purposes:

---

[1] The XLATE action code tells the XLATE handler what the user of the XLATE instruction wanted to accomplish. The action code conveys information such as whether the caller really needed to reference an object (and the object should be brought locally if it isn't present) or the caller only wanted to tell if the object exists.

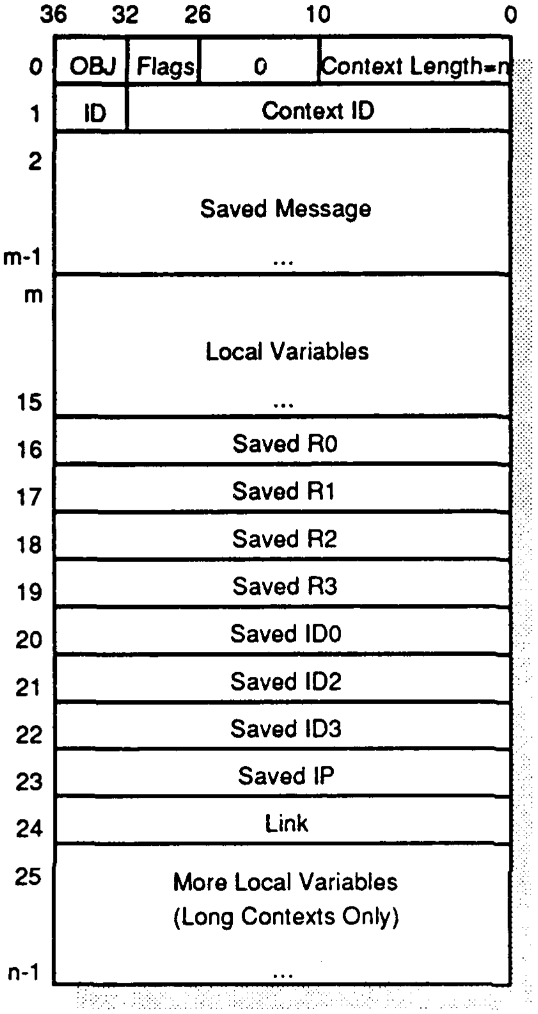


Figure 4-9. Context Format

Standard and fast contexts have the above format except that they are only 25 words long, while long contexts can be up to 64 words long (the MDP only allows convenient addressing of the first 64 words of an object).

There is no saved ID1 field because ID1 points to the context itself, so it has to be known by whatever routine is resuming the context.

The link field is used for several purposes. Contexts on the `FastContextQueue` are linked together by their link fields. When a process suspends execution, the resumption condition is stored in the link field: if the process suspended because it read a cfuture from a local variable in the context, the offset (tagged CFUT) of that local variable is stored in the link field. If the process suspended because it referenced a non-local object, the context is put on a linked list of contexts waiting for the object rooted at the object's BRAT binding. The old data value of the BRAT binding is placed in the link field of the last context waiting for the object. Since the data value of a BRAT entry can be an integer, the INT tag cannot be used to represent contexts waiting for cfutures.

- When a function calls another function, it stores a cfuture in a local variable in its context and then proceeds to fault on that cfuture. The reply from the called function will store its value into the designated local variable, overwriting the cfuture.

- When evaluation of a function needs to be suspended for any reason, including a cfuture fault, the function's registers are saved in a context.

- When evaluation of a function is suspended, the message that invoked that function is copied into the beginning of the context (except for the first two words of the message, which are then lost). When the function resumes, `A3`, the register which originally pointed to the message, is aliased to point to the context to allow the function to use `A3` to refer to the incoming message regardless of whether the message has been copied into the context yet or not.

Context Availability

There are four fundamental approaches to allocating contexts:

1. Always allocate a context at the beginning of every function and deallocate it at the end.
2. Allocate a context at the beginning of a function that needs a context and deallocate it at the end.
3. Lazily allocate contexts only when necessary.
4. Always keep a context allocated, even when no message is being processed.

Approaches 1 and 2 are commonly used for stack frames on stack-based computers. Initially I chose approach 3 for the context allocation strategy. Approaches 1 and 2 are simpler but have the disadvantage of often allocating unnecessary contexts—most of the leaf nodes of computations do not require contexts, and allocating contexts unnecessarily is a considerable overhead. Approach 3 worked by storing an invalid address in `A1`, the MDP's context address register. When a context was needed, the access through `A1` would fault, and a context would be allocated. However, I ran into two difficulties with approach 3: allocating contexts through faulting on `A1` was slow because determining the cause of an INVADR fault on the MDP is quite involved, and there were some difficult code sections in the object manager where a fault might allocate a context, violating criticality rules.

Due to the above difficulties, I switched to approach 4, which combines the advantages of lazy context allocation with the advantages of always allocating a context. In approach 4, when a function finishes executing, it does not deallocate its context[1]; thus, the next message that arrives does not have to allocate a context. There are two places where approach 4 involves a little extra work than approach 3: when a function suspends on a cfuture or object migration wait, it must allocate a new context to avoid having its own context overwritten; and when the value of a cfuture is returned or an object arrives, the currently allocated context must be deallocated and replaced with the suspended function's context. The additional context allocation on a cfuture or object migration wait is not a significant penalty because it occurs on the tail end of message processing—it does not affect the latency of message processing until the J-Machine is fully loaded. The context deallocation on the reception of a cfuture value or an object does add to the latency, but context deallocation is always fast—it takes only four instructions.

To avoid reentrancy and criticality problems, the value in register `A1` is required to be always valid; therefore, any routine, such as the heap compactor, which might invalidate `A1` must recalculate the value of `A1` when it is done.

Kinds of Contexts

There are three kinds of contexts: fast contexts, standard contexts, and long contexts. A fixed number of fast contexts is preallocated when an MDP is initialized. Each fast context is 25 words long. The fast contexts are nonrelocatable heap objects between `FixedHeapStart` and `HeapStart`. The physical addresses of these contexts never need to be invalidated, so these contexts are especially fast. Fast contexts are never deallocated. Enough of these contexts should be allocated to serve a normal computation load on an MDP; the current operating system allocates eight per MDP, which is probably too few.

Standard contexts are like fast contexts in that they are 25 words long, but they are relocatable objects allocated from the main heap area; thus a heap compaction invalidates their

---

[1] This only applies to functions which use 25-word contexts; functions which use long contexts must deallocate their contexts and allocate 25-word contexts upon exiting.

physical addresses. Unlike fast contexts, the storage occupied by standard contexts can be reclaimed.

Fast contexts and standard contexts are eligible to be queued on a linked list of free contexts rooted by the global variable `FastContextQueue`. Whenever a 25-word context is desired, `FastContextQueue` is checked first; if it contains a context, that context is unlinked from the queue and used. Otherwise, a standard context is allocated. When a fast context is disposed, it is linked back on the queue. When a standard context is disposed, it is either liked back on the queue or deallocated, at the caller's discretion. These queue operations are fast—allocating a context from the queue takes five instructions, while deallocating one onto the queue takes four.

Long contexts are contexts for functions which require extra space for local variables. Long contexts are identical to standard contexts except that they are longer and ineligible for queueing on the `FastContextQueue`. When a function that might need a long context starts executing, it calls the `NewContext` routine, which replaces the present context with a newly created long context. `NewContext` also copies any relevant state such as the message from the fast context to the new, long context. A function which allocates a long context must terminate with a call to `Suspend`, which disposes the long context and allocates a new fast or standard context. `DisposeContext` can be used to dispose a context without allocating a new one.

Allocation and Deallocation Calls

The routines to allocate and deallocate 25-word contexts are short enough that they are inlined whenever they are needed. The following calls are available for handling long contexts and the case in which the `FastContextQueue` is empty:

- `AllocFastContext` creates a new fast context when the queue is empty.
- `Suspend` checks whether a fast context was used by the routine. If so, it links it into the fast context queue; otherwise, the context is disposed by the heap manager, and a new 25-word context allocated.
- `NewContext` allocates a new long context. If a context is currently in use, it is deallocated after the message has been copied from it to the new context.
- `DisposeContext` is like `Suspend` except that it does not allocate a new 25-word context.

Suspending and Resuming Processes

When a process must be suspended because it tried to read a cfuture, perform an operation on a future, add two user-defined objects together, or reference a nonlocal object, the process's state must be saved in its context. In particular, the values of registers that need to be preserved must be stored in the context along with the IP at which execution should resume. Furthermore, the reason for suspending must be stored in the link field of the context; otherwise, the context might be restarted prematurely, which would lead to a disaster if the context was waiting for an object[1]. Finally, a new 25-word context is allocated in `A1` and `ID1` to prevent the suspended context from being reused.

When a process is to be resumed, the resuming event is checked against the context's link field to make sure that the context should, in fact, be resumed. If it should, the existing context in `A1` and `ID1` is deallocated, and the values of the registers and `IP` read from the con-

---

[1] The reason why restarting a context early would crash the computer is not obvious. The problem is *not* that the process would access a bad object or value—the process would fault and suspend again because it still cannot reference the nonlocal object. Instead, the system crash would occur because if a context that had been waiting for migration of a nonlocal object were restarted early, the context would not be unlinked from the list of contexts waiting for the object. The `Reply` handler would not even be aware that the context had been present on the linked list. Then, when the context's process faulted again on the missing object, it would be added to the list of contexts waiting for the object a second time, corrupting that list.

text. If a resource for which several processes were waiting arrives, one of these processes is resumed immediately, while the other ones are resumed later by `RestartContext` messages (Figure 4-10) which the node sends it itself. A `RestartContext` message deallocates the existing context in `A1` and `ID1` and then restarts the specified context.

0	MSG	RestartContext	2
1	ID	Context ID	

Figure 4-10. `RestartContext` Message

The `RestartContext` message restarts the context specified by the ID. The context must be present on the target node.

Reclaiming Contexts

The current strategy for reclaiming free contexts by the heap compactor is somewhat haphazard. Fast contexts are never reclaimed. Long contexts are always reclaimed because they are required to be deallocated before their processes can exit. On the other hand, standard contexts are reclaimed only if enough processes call `Suspend` when they are done; otherwise, once a standard context is allocated, it is never deallocated. This may be an advantage because once a working set of fast and standard contexts is allocated on an MDP, allocation of 25-word contexts will always be fast. If the lack of regular deallocation of standard contexts turns out to be a problem, it would only be a simple modification to the heap compactor to have it scan the `FastContextQueue` and deallocate any standard contexts it finds there.

Global Object Manager

The primary means of invoking the global object manager is through the local object manager when the latter cannot find a local object. The global object manager extends the local object manager to the global virtual address space of the J-Machine. Together, the two managers provide an integrated facility for efficiently managing objects globally on the J-Machine. The managers can distinguish mutable objects from immutable ones and cache copies of the immutable objects on many nodes.

Data Structures

Every object on the J-Machine has a *home* node. The home node most likely created the object, and that node has the responsibility of keeping track of the object's location throughout the object's life. Objects may migrate from node to node, but the object must inform the home node of every such move. If a node needs an object and does not know where it is, it asks the home node. Certain objects such as contexts, selectors, and immutable objects do not migrate, so such objects can always be found at their home nodes. The address of the home node is usually encoded in the lowest 16 bits of an object's ID (see Figure 4-8). This is a convenient format because the network ignores the upper 16 bits of a routing address, so messages may be sent to an object's home node simply by transmitting the object's ID as the routing word.

In addition to the flags used by the local object manager, each object has three additional flags: copyable, purgeable, and locked. An object is copyable if it is immutable. Many primitive objects are immutable, as are objects belonging to classes declared immutable by the Concurrent Smalltalk programmer. Furthermore, the compiler might be able to determine that objects of a particular class cannot be mutated and mark them copyable, although the compiler does not perform this optimization at this time. When a copy of a copyable object is made, the copy is marked purgeable. Thus, many copies of immutable objects can be made, and the heap compactor can reclaim storage used by copies that are no longer needed. Setting the locked flag prevents an object from migrating or being deleted during critical protocol sections.

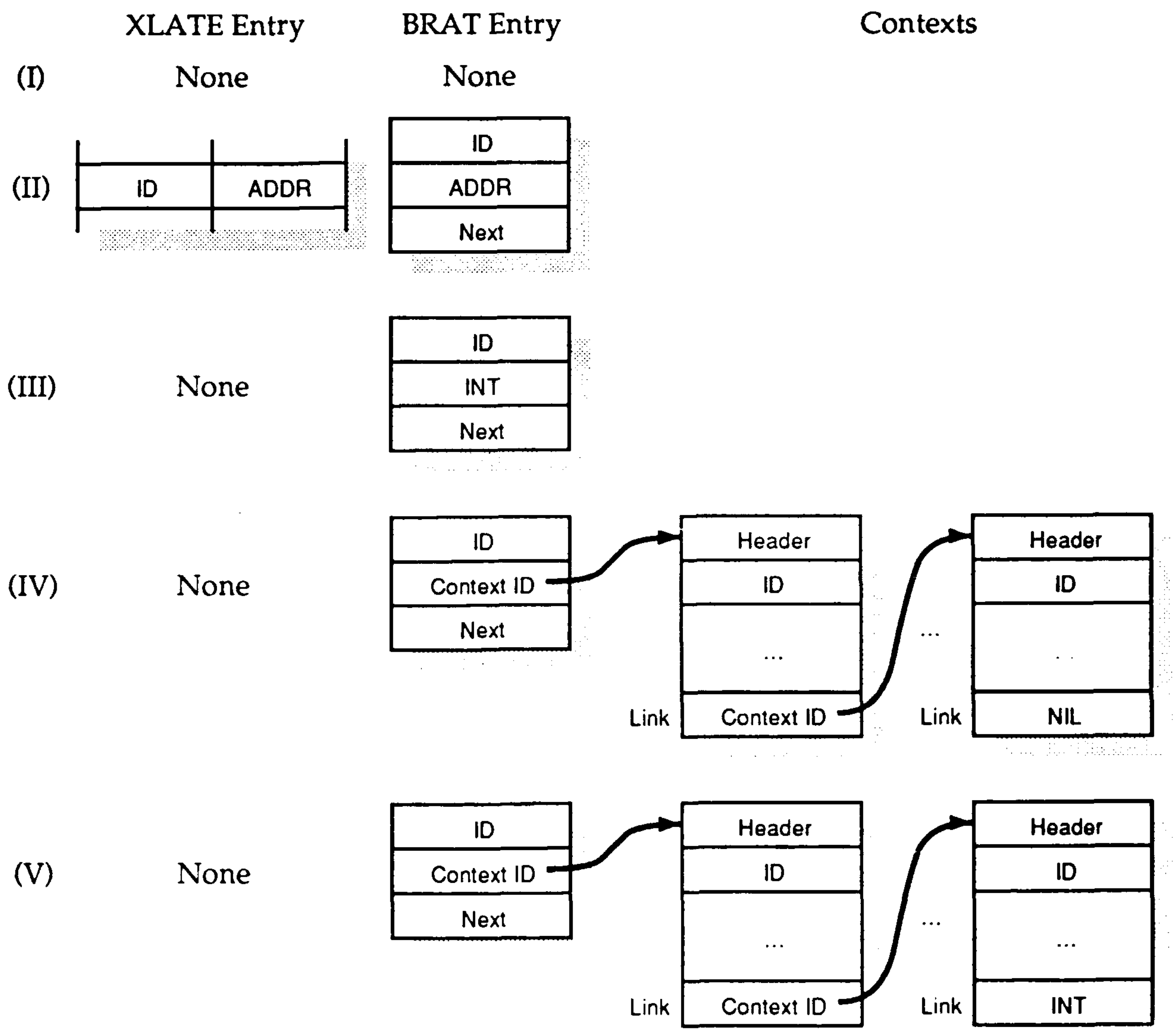


Figure 4-11. Object XLATE Table and BRAT Entries

There are five possible BRAT table states for a particular object. Each object must have a BRAT entry on its home node. The XLATE table entry, where specified, is optional. The states are as follows:

I. The object does not exist on this node, and its whereabouts are unknown.

II. The object exists on this node. Its physical address is given.

III. The object does not exist on this node, but it is believed to reside on the node specified by the integer.

IV. The object does not exist on this node, but the contexts linked to its BRAT entry are waiting for its arrival.

V. The object does not exist on this node, but the contexts linked to its BRAT entry are waiting for its arrival, and the object is believed to reside on the node specified by the integer.

Only states II, III, and V are allowed on an object's home node, while only states I, II, and IV are allowed on the other nodes.

Every object must always have a BRAT entry on its home node. The BRAT entry can be in one of the states shown in Figure 4-11. When an object is initially allocated, its BRAT entry is in state II. If an object is in state I on its home node, that object does not exist, and any attempt to access it halts the system.

Object Migration

The object migration protocol is a slightly simplified version of the protocol in [38]. When a node requests an object because it does not have the object in local memory, it sends a `RequestObject` message (Figure 4-12c) to the object's home node. If the home node does not currently have the object (its BRAT table entry is in states III or V), it forwards the `RequestObject` message to the node thought to contain the object. If the home node does not know about the object (BRAT state I), it halts the system. This halt is deliberate, for it detects accesses to deleted objects. If the `RequestObject` message was forwarded to a node that has the object, the message is processed there; otherwise, that node forwards the `RequestObject` message back to the home node, and the two nodes keep forwarding the message to each other. Nevertheless, since the home node is required to know an object's whereabouts most of the time, the home node will eventually learn of the object's true location and forward the `RequestObject` message to the right place.

0	MSG	AcceptObject		Msg. Length=n+2
1	INT	Reply Node Number		
2	OBJ	Flags	Class	Object Length=n
3	Object ID			
4	Object Data			
n+1	...			

(a) `AcceptObject` Message

0	MSG	AcknowledgeObject	2
1	Object ID		

(b) `AcknowledgeObject` Message

0	MSG	RequestObject	3
1	Object ID		
2	INT	Reply Node Number	

(c) `RequestObject` Message

0	MSG	MigrateObject		Msg. Length=n+1
1	OBJ	Flags	Class	Object Length=n
2	Object ID			
3	Object Data			
n	...			

(d) `MigrateObject` Message

0	MSG	UpdateHome	3
1	Object ID		
2	INT	Node Now Containing Object	

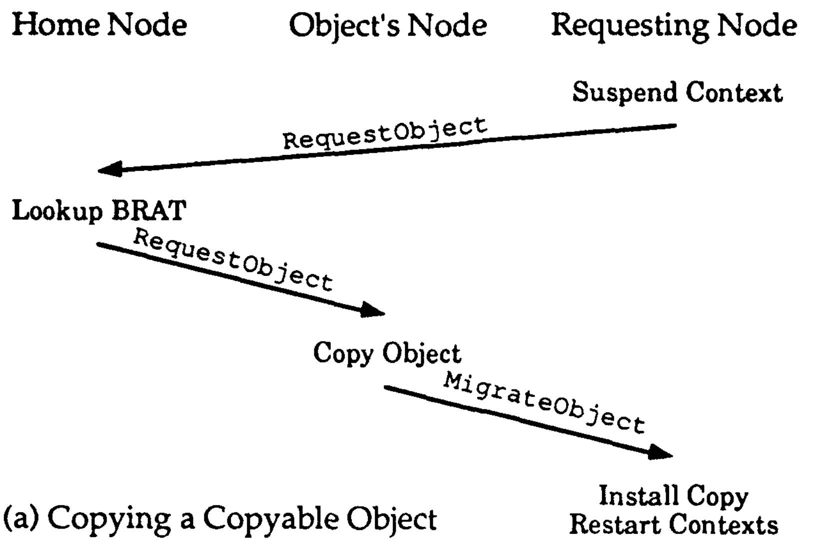
(e) `UpdateHome` Message

0	MSG	Unlock	2
1	Object ID		

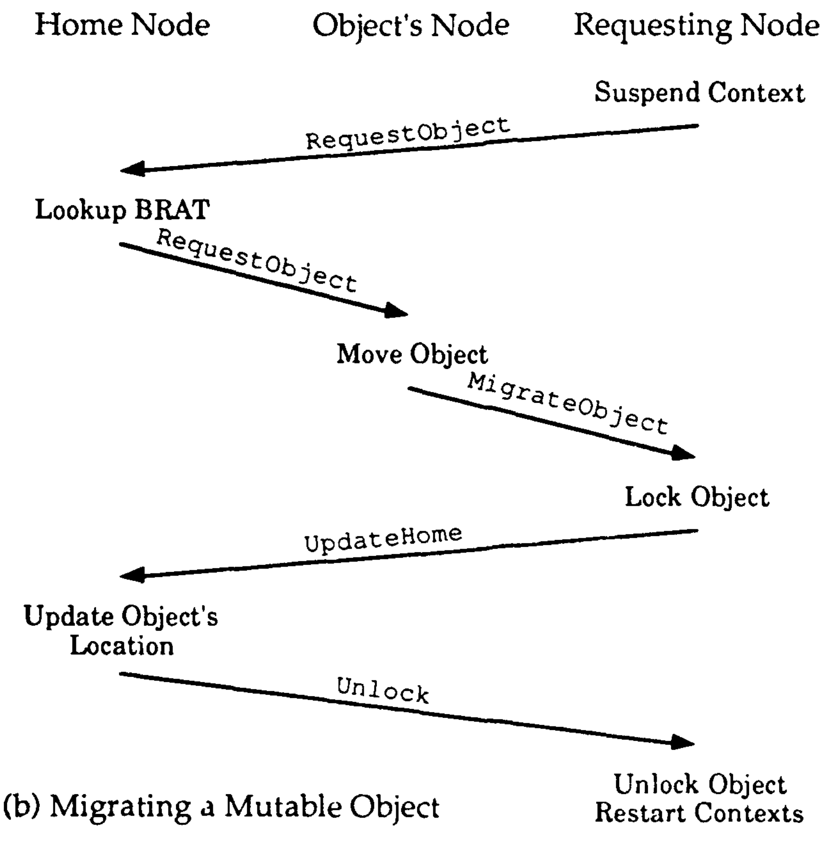
(f) `Unlock` Message

Figure 4-12. Object Migration Messages

The `AcceptObject` and `AcknowledgeObject` messages are used only for downloading objects into the J-Machine and for debugging The other four messages are used for successive steps of object migration



(a) Copying a Copyable Object



(b) Migrating a Mutable Object

Figure 4-13. Object Migration Protocol
When a copy of an immutable object is made, the copy is simply sent to the requester as in part (a). If a mutable object has to be moved, the protocol is more complicated because the object's home node has to be kept informed about the object's location.

What happens when the `RequestObject` message finds the object depends on whether the object is copyable or locked. If the object is locked, the node forwards the message back to itself; the message will be handled once the object is unlocked. If the object is copyable, the node simply mails a purgeable, copyable copy of the object in a `MigrateObject` message to the requesting node, which then installs the copy in its memory (Figure 4-13a). If not, the protocol becomes more complicated (Figure 4-13b). The node on which the object is residing deletes the object from its memory and BRAT and sends the object to the requesting node in a `MigrateObject` message. The requesting node installs the object in its memory, locks it, and sends an `UpdateHome` message to the birthnode, telling it about the object's new whereabouts. Finally, the birthnode sends an `Unlock` message to acknowledge receipt of the `UpdateHome` message and allow the object to be moved again. Since a locked object might have been deleted, the `Unlock` message checks the object's deleted flag and deletes it and its BRAT entry if it was set. The last two messages are optimized out if the requesting node happens to be the object's home node.

The object is locked in the last phase of the protocol to prevent the home node from receiving the `UpdateHome` messages from two successive migrations out of order; if that were to happen, the home node would lose track of the object's location. Alternatively, counters could be used to achieve the same synchronization, but that solution would require an extra word in the BRAT and in the object.

Object Allocation and Deletion

An object can be allocated either at the local node or on a remote node. The `NewLocalObject` system call allocates an object locally. Unlike the `AllocNextObject` call, `NewLocalObject` takes a class as a parameter and extracts the appropriate header word from the class object (Figure 4-14) to use for the object. Reading the class object may involve another call to the global object manager if a copy of the class object is not present in local memory.

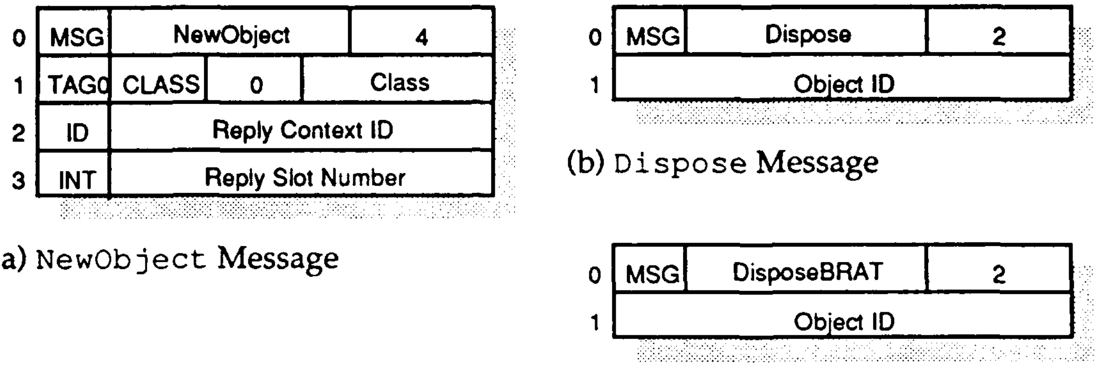
0	OBJ	Flags	Metaclass	4+n
1	TAG0	CLASS	0	Class
2	OBJ	Instance Object Header Word		
3	INT	n=Number of Ancestors		
4	TAG0	CLASS	0	Class
5	TAG0	CLASS	0	Ancestor
			...	
3+n	TAG0	CLASS	0	Object Class

Figure 4-14. Class Object Format

The instance object header word is the word that is stored as the header of every object of this class. That word is `nil` if the metaclass is `primitive-class`.

In addition, each class object contains an ordered list of the class's ancestors from the most specific to the least specific. The class's ancestors consist of the class itself, its superclasses, its superclasses' superclasses, and so on; each class is listed at most once. The ancestors are ordered according to a partial order which always places a class before any of its superclasses; thus the class itself is always the first ancestor and `Object` is always the last ancestor.

The `DisposeObject` system call is used to dispose objects, both locally and globally. `DisposeObject` first tries to dispose the object locally; if the object is locked, it is marked as deleted but not disposed; it will be disposed when it is unlocked. If the object does not reside on this node, a `Dispose` message is sent to the object's home node, which follows a route analogous to the `RequestObject` message above and will not be discussed further. If the object is present on this node but this is not the object's home node, a `DisposeBRAT` message is sent to the home node to dispose the object's BRAT entry there. If the `DisposeBRAT` message happens to find another instance of the object on its home node, it deletes that instance too.



(a) `NewObject` Message

(b) `Dispose` Message

(c) `DisposeBRAT` Message

Figure 4-15. Object Creation and Disposal Messages
The `NewObject` message creates an object of the given class on the remote node and returns its ID in a `Reply` message. The `Dispose` message disposes an object on the remote node, while the `DisposeBRAT` message disposes an object's home BRAT entry.

This protocol successfully deletes the single instance of a mutable object and the unpurgeable original of an immutable object along with, perhaps, one copy. Other copies, if any exist, of an immutable object are not disposed; however, they will simply be purged out if they are not referenced for a while.

Other Services

The global object manager provides two routines, `ClassOf` and `TypeOf`, that can determine the class of any of the objects listed in Figure 4-2. If the object is a primitive object, the global object manager returns it class directly. Otherwise, the global object manager extracts the class from the object's header and returns. In addition, the global object manager provides the `ObjectNode` routine which returns the node number of a node likely to contain the object. If the object is primitive, `ObjectNode` returns a random node number. This system call is frequently used in Concurrent Smalltalk to determine the node to which an application message should be sent.

The global object manager actively participates in the process of downloading a Concurrent Smalltalk program to the J-Machine. It provides support for installing objects on nodes without migrating them from anywhere. If a node receives an `AcceptObject` message (Figure 4-12a), it installs the object and its ID in its memory and the BRAT and responds with an `AcknowledgeObject` message (Figure 4-12b) containing the object's ID.

To avoid difficulties with downloading objects recursively referencing each other, object IDs are assigned by MDPSim (see the section about late-binding references in [25]) before the objects are downloaded into the J-Machine; hence, an MDP accepting an object must also accept the object's ID instead of generating a new one. The IDs assigned by MDPSim use serial numbers in the upper range of the allowed numbers, thus preventing ID conflicts with objects generated at runtime.

Finally, the global object manager provides support for distributed objects. This support is documented in the distributed object section later.

Initialization

Upon powerup each MDP performs the following actions:

- Clear the address and ID registers at all priorities.

- Clear the globals to CFUT-tagged words. If an uninitialized global is accidentally referenced, the MDP will halt because the cfuture handler can distinguish a valid cfuture from a CFUT-tagged word that just indicates an uninitialized value.
- Clear the XLATE table and the BRAT root table to `NIL`.
- Initialize and enable the network queues, but block network message dispatching until initialization is done.
- Clear the heap to CFUT-tagged words.
- Initialize the global variables that need initializing.
- Create eight nonrelocatable fast contexts, link them onto `FastContextQueue`, and initialize `HeapStart` to the first word after those contexts.
- Unlink one fast context and point priority 0's A1 and ID1 to it.
- Enable message dispatching and fall into an infinite loop in background mode.

The version of Cosmos for running on a real J-Machine instead of MDPSim has a startup sequence that also includes a self-test of the CPU, a memory test, a network test, debugging utilities, and a protocol to let each MDP determine its location on the network.

Downloading Programs

In the MDPSim emulation of the J-Machine, a special non-MDP network node called the I/O Node acts as the bridge between the compiler and the J-Machine. The compiler outputs an MDPSim script which queues a series of objects in the I/O Node. The I/O Node then sends `AcceptObject` messages to the appropriate nodes, waits for the `AcknowledgeObject` replies, and sends more objects until all objects have been downloaded.

On the real J-Machine, Concurrent Smalltalk programs are also downloaded through a MDP that includes special software to communicate with the outside world. Each MDP contains a diagnostic port that lets the user halt the MDP and directly examine and change its memory and state. The Cosmos kernel is loaded onto the MDPs through these diagnostic ports.

4.3. The Cosmos Higher-Level Facilities

Method Manager

The method manager associates class/selector pairs with methods, although it could also be used for keeping general immutable associations. It provides only one routine, `LookupMethod`, with a variant, `LookupMethodU`, which performs less processing of its arguments to make it more efficient. `LookupMethod` takes a class word and a selector word and attempts to find the method associated with them; it is the equivalent of the Concurrent Smalltalk `method` primitive.

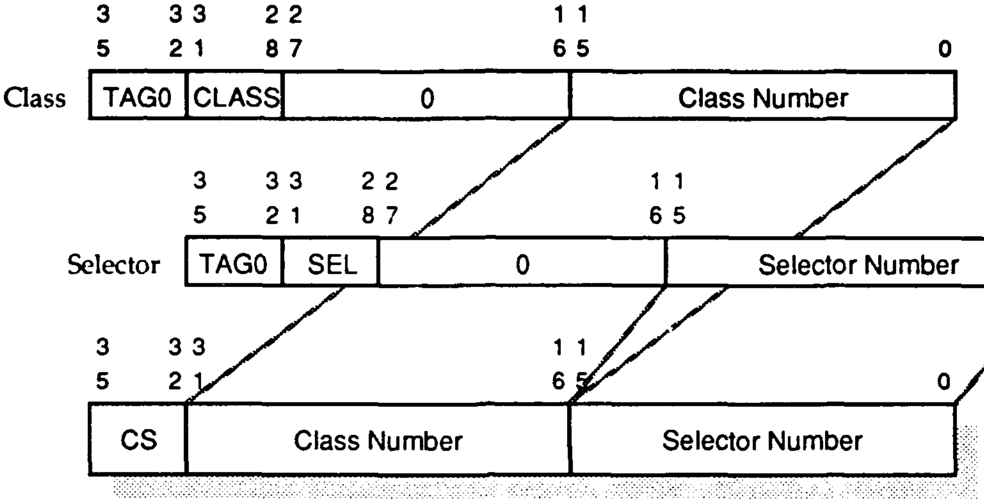


Figure 4-16. Class/Selector Word Format

The Class/Selector word is formed by combining a 16-bit class number with a 16-bit selector number. The word is tagged CS (which is also the INST1 tag) to avoid conflicts with other kinds of bindings stored in the XLATE table.

`Lookupmethod` first attempts to look up the association in the local XLATE cache. It combines the 16-bit class and selector numbers into a single word, tags that word `CS` (Figure 4-16), and looks for a binding in the XLATE table. If it finds a binding, the binding's data word is immediately returned as the desired method. If no such binding exists, `Lookupmethod` sends a `Lookupmethod` message (Figure 4-18a) to the selector's home node. The message will invoke the `LookupMethod` runtime function on the selector and the class.

The `LookupMethod` runtime function executes on the same node as the selector object (Figure 4-17). Each selector has a list of methods defined for it together with their classes. `LookupMethod` first tries to find the given class in the selector object; if it finds it, it returns the corresponding method. If `LookupMethod` cannot find a method for the given class, it gets the class object (Figure 4-14) and searches the selector's method list for the class's ancestors until it either finds a method or runs out of ancestors. In the latter case the method lookup fails and `LookupMethod` returns `nil`. In either case `LookupMethod` returns the result in a `MethodReply` message (Figure 4-18b). The requesting node then associates the class/selector pair with the result in its XLATE table.

The method lookup strategy is conservative in the use of space, taking space roughly proportional to the number of methods defined in the program. However, the method lookup time suffers somewhat, especially when a method is requested corresponding to a deeply nested class and a selector with many methods defined; in the worst case the method lookup time is the product of the number of ancestors of a class and the number of methods defined for the selector. A binary search could have been used for searching the method table, but it would have much worse constant factors, resulting in slower lookup for most methods, because the MDP does not have enough registers to support the inner loop of a binary search.

The methods are stored in selector objects indexed by the class instead of storing them in class objects indexed by the selector because the number of selectors is usually much larger than the number of classes, and selectors tend to be accessed more uniformly than classes; thus, the method lookup table can be distributed more evenly on the J-Machine.

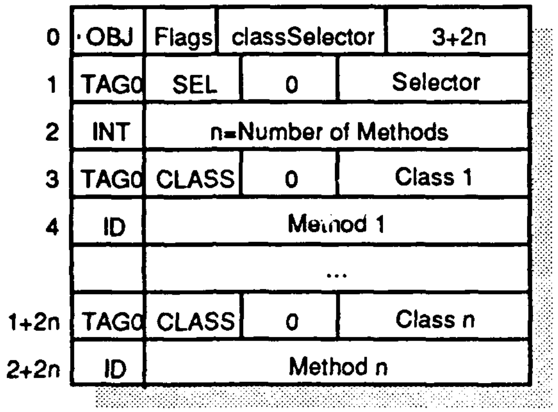
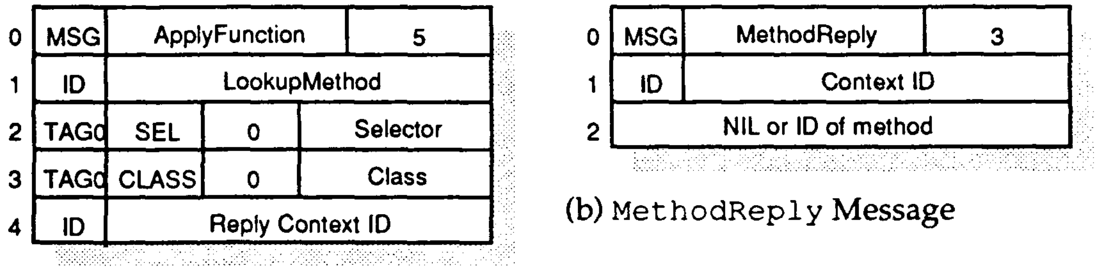


Figure 4-17. Selector Object Format
Each selector object contains a table associating classes to methods.



(b) `MethodReply` Message

(a) `LookupMethod` Message

Figure 4-18. Method Manager Messages
The `LookupMethod` message requests a lookup of the class and selector to get NIL or a method ID; the `MethodReply` message replies to the lookup.

Control Manager

The control manager dispatches function and method calls and handles replying from functions, a task shared with the context and global object managers. The control manager's code is relatively short because so much groundwork has been laid by the previous managers.

Function and Method Dispatch

The control manager handles three types of messages for calling functions and methods: `Apply`, `ApplyFunction`, and `ApplySelector` (Figure 4-19). The first message can be used for applying an arbitrary object—a function or a selector, while the other two messages can only be used for applying functions or selectors, respectively. The `Apply` handler checks the type of its argument and jumps into either the `ApplyFunction` or `ApplySelector` handler, as appropriate; the check takes three to five instructions.

0	MSG	Apply...	Message Length
1	Function or Selector		
2	Argument 0		
3	Argument 1		
	...		
1+n	Argument n-1		
2+n	Context ID for first reply value or `NIL`		
3+n	Context slot for first reply value or `NIL`		
	...		
2m+n	Context ID for last reply value or `NIL`		
1+2m+n	Context slot for last reply value or `NIL`		

Figure 4-19. Application Messages

The `Apply`, `ApplyFunction`, and `ApplySelector` messages have identical formats except for the address stored in the header word. Each value returned by the called function corresponds to one two-word continuation passed to the function. The continuation specifies either the context ID and slot to which that value should be replied or two `NIL`s if that value is ignored by the caller. The context IDs passed to the function are not necessarily the same due to tail forwarding.

`ApplyFunction` reads the ID of the function from the message, stores it in MDP's registers `ID0` and `A0` (the code segment registers), and jumps into the fourth word of the function object (Figure 4-20). The entire process takes only 4 instructions.

`ApplySelector` reads the selector and the first argument (the receiver object) from the message, uses inline code to quickly determine the class of the receiver, and calls `LookupMethodU` to determine the ID of the method that should be called. If the ID is `NIL`, `ApplySelector` halts; otherwise, `ApplySelector` initializes `ID0` and `A0` and jumps into the fourth word of the function object. `ApplySelector` takes 23 instructions in the best case, and considerably more if the class of the receiver is hard to determine or if `LookupMethodU` misses in the XLATE cache.

Either of the above handlers can suspend even before the first instruction of the function is executed if the function code or, in the case of `ApplySelector`, the receiver object is not present locally. Hence, it is important that a valid context be always present in `ID1` and `A1`. In fact, a valid context is present in those registers as explained in the context manager section.

Function Calls and Replies

The control manager's other task is handling CFUT faults. There are two primary causes for a CFUT fault: a function accesses the result of a computation that has not finished yet, or any routine accesses some uninitialized variable. The control manager distinguishes these two cases by the data in the CFUT-tagged word that caused the fault, which is conveniently stored in MDP's `FOP0` register.

If the data is positive, the fault was a cfuture fault, and the control manager stores that CFUT word in the current context's link field and suspends the context. The Optimist II compiler arranges for the data portion of the CFUT word to contain the offset of the context variable that was accessed; this way the cfuture handler does not have to disassemble the faulted instruction to determine the offset. The offset is needed later by the `Reply` handler to determine whether the context should be restarted.

If the data in the CFUT-tagged word was zero or negative, the control manager halts the computer because an uninitialized variable was accessed. On startup, all memory in the MDP's heap is cleared to CFUT:-1.

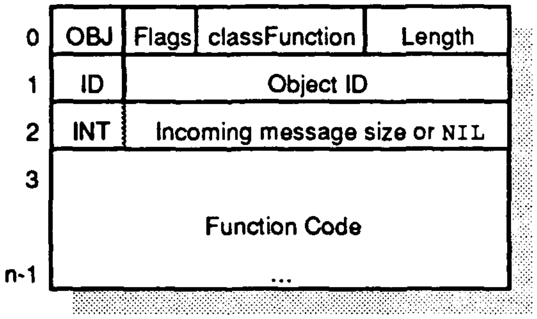


Figure 4-20. Function Object Format
The function object contains the code for a function. Registers A0 and ID0 point to the function while it is executing. *The third word contains the size of the message expected by the funcion or NIL if the size is not known or the* function expects a variable number of arguments. The compiler initializes that word, but the operating system *does not check it against the size of tne message that invoked the function; that check would add at least five in*structions to the function dispatch time.

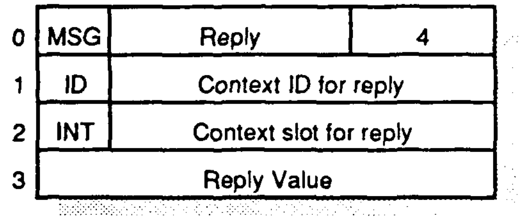


Figure 4-21. Reply Message Format
The `Reply` message carries the reply value to the specified slot in the specified context. The context ID and reply slot may not be `NIL`—if they were `NIL` in the `Apply` message, no `Reply` message is sent.

Functions return results to their callers via `Reply` messages (Figure 4-21). If a function returns multiple values, it sends one `Reply` message for each value returned. The `Reply` handler on the caller's node performs the following processing when it receives the message:

1. The value from the message is stored over the cfuture in the caller's context. However, if the slot indicated in the `Reply` message did not originally contain a cfuture, the `Reply` handler halts because some function replied twice to the same slot or the compiler generated incorrect code.

2. The CFUT-tagged link field in the caller's context is checked against the slot number of the newly updated slot. If the numbers match, the context is resumed; otherwise, the `Reply` handler exits because the context is waiting for some other event.

Actually, for reasons of efficiency the check in (1) is done only if the slot number in (2) doesn't match.

Utilities

The operating system kernel currently contains three utilities: a divide routine, a closure maker, and a closure evaluator. The `Divide` system call divides one integer by another and returns the quotient and remainder using the sign conventions described in Appendix A. The divide routine includes considerable overhead to evaluate all signed 32-bit results correctly, including special cases such as dividing -$80000000 by 1 or -1 because a `large-integer` package might be implemented on top of the normal integer arithmetic routines sometime in the future.

`NewClosure`, the closure maker, allocates and returns a new closure object (Figure 4-22) on the local heap. The caller should then initialize the closure's display arguments before using the closure.

`CallClosure` is the function called by a closure when it is invoked as a function. `CallClosure` calls the function specified in the closure with the additional display arguments in the closure.

It is true that `Divide` and `NewClosure` could have been implemented as functions instead of system calls; however, these routines are used frequently enough and are short enough that it was decided that it would be best to make them readily available whenever they are needed. The additional overhead that would be required in making a function call is comparable to the time it takes to divide two numbers or allocate a new closure object.

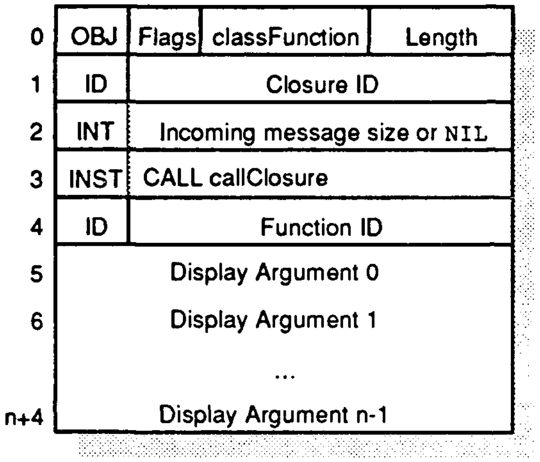


Figure 4-22. Closure Format
Closures are treated just like functions by Concurrent Smalltalk and the control manager. When the control manager calls a closure, it executes the instruction at offset 3, which is a `CallClosure` system call. That system call forwards the message appended with the display arguments included in the closure to the function with the ID specified in the word with offset 4 in the closure.

MDP Runtime

The MDP runtime system contains utilities for which it is not important that they reside on every node. Currently the MDP runtime system includes a method lookup routine and two routines that allocate distributed objects and are described below.

Distributed Objects

A distributed object is an object composed of many constituents. A message sent to the group name of a distributed object arrives at a constituent chosen by the operating system; the hope is that the operating system chooses the constituents evenly enough so as not to overload some constituents and underutilize others. In addition, each constituent of a distributed object is itself a Concurrent Smalltalk object.

Distributed objects are supported by the global object manager and the MDP runtime system. The MDP runtime system handles allocation of distributed objects, while the global object manager handles accessing constituents of distributed objects.

Implementation

Each distributed object is implemented solely as a set of constituent objects; there is no "group" data for a distributed object anywhere in the system. The group name of a distributed object contains enough information to permit quickly finding the ID of any of its constituents as well as a convenient way to find a nearby constituent. The structure of the group name is shown in Figure 4-23.

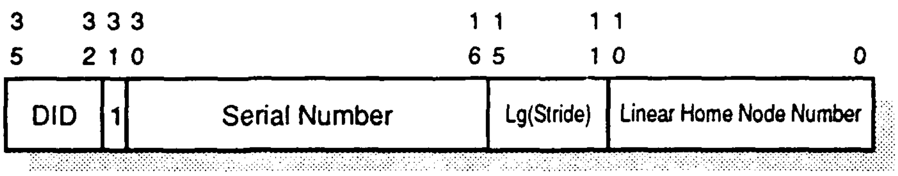


Figure 4-23. Distributed Object Group ID

The group ID (DID) contains the distributed object's serial number, linear "home" node number (explained in Figure 4-24), and a signed base-2 logarithm of the distributed object's stride, which is the ratio S of the number of nodes in the J-Machine to the physical number of constituents. Both the physical number of constituents and the number of nodes in the J-Machine must be powers of two. The Lg(S) field is signed and 5 bits long, ranging from -16 (S=1/65536; 65536N constituents on an N-node J-Machine) to 15 (S=32768; 1 constituent for every 32768 nodes) by powers of two. The linear home node number H must be less than S. The kth constituent, counting from k=0, is located on the node with the linear number $H+\lfloor kS \rfloor$.

If the stride S is 1 or greater, each constituent object has the same serial number as the group object. If S is less than 1, several constituents reside on every node in the J-Machine, and more than one serial number is required to distinguish them. Hence, the distributed object reserves 1/S consecutive constituent serial numbers, and the kth constituent has serial number N+(k mod 1/S) and resides on the node with the linear number $\lfloor kS \rfloor$, where N is the group name's serial number. H should be zero in this case.

The linear home node number is used to distributed sparse distributed objects evenly throughout the J-Machine. The linear home node number is always zero for dense distributed objects (ones with stride 1 or less).

The physical size of a distributed object has been constrained to be a power of two for two reasons. First, it is desirable to be able to find any constituent from just the information contained in the DID, and encoding an arbitrary distributed object size in the DID would require too many bits; recording the logarithm of the size requires only five bits for any potential size. Second, unless some radically different addressing scheme were used, distributing the constituent objects evenly throughout the J-Machine would require a division operation either in the `Co` routine or in the `PreferredConstituent`[1] routine.

A variant of the current scheme has been considered in which the constituents above the logical size of the distributed object are not created. The `Co` system call would work fine in such a scheme (except that its range checking would no longer be valid), but the `PreferredConstituent` routine might return a nonexistent constituent of the distributed object, and since it does not know the logical size of the distributed object, it would not know that the constituent does not exist. It could, however, inquire at the constituent's home node, at the expense of complicating and slowing down the implementation of distributed objects in Cosmos. This variant may be adopted if the loss of memory caused by rounding the sizes of distributed objects up to powers of two becomes too large.

Another consequence of rounding the sizes of distributed objects up to powers of two is that the MDPs with high node numbers contain mostly unused constituents. This difficulty could be alleviated by always allocating a 11-bit random "home" node number, and adding that number to the node number of the constituent modulo the size of the J-Machine, at the expense of complicating the `PreferredConstituent` routine somewhat. If a J-Machine has more than 2048 nodes, bits could be stolen from the serial number field and added to the home node number field. To avoid placing too severe a restriction on the number of dis-

---

[1] `PreferredConstituent` returns the ID of a constituent near to the current node.

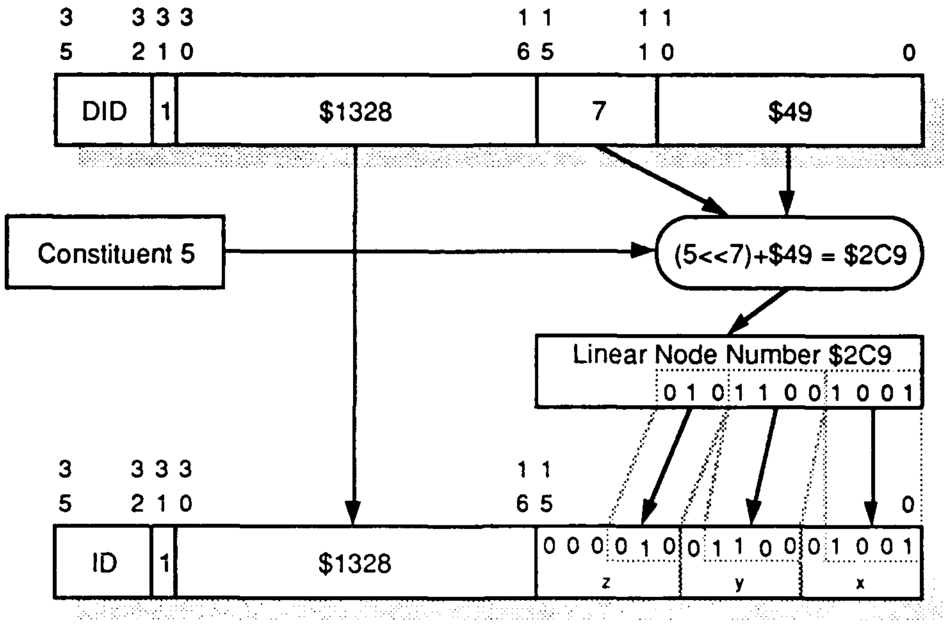


Figure 4-24. Looking up a Constituent in a Sparse Distributed Object
This figure illustrates the `co` system call looking up constituent 5 in a 16-constituent distributed object on a 2048-node J-Machine organized as 16×16×8. The stride is 2048/16=128, so lg(stride) is 7. Constituent 0 is located on the node with the linear number $49. The distributed object's serial number is $1328.

Since the stride is greater than 1, the constituent number 5 is multiplied by the stride 128 and added to $49 to get constituent 5's linear node number, $2C9. The dimensions in the linear node number are packed together to simplify arithmetic operations; the `co` system call unpacks them to get the constituent's ID.

tributed objects in the system, `NewDistobj` could use both the home node number and the serial node number fields to distinguish distributed objects.

Locating Constituents

The `Co` system call implements Concurrent Smalltalk's `co` primitive. To find the *k*th constituent ID of a distributed object, the global object manager shifts *k* by *lg*(*stride*) bits to the left and adds the linear home node number to obtain the constituent's linear node number and ANDs *k* with a right-justified mask of *max*(*-lg*(*Stride*),*0*) ones and adds it to the serial number from the group object to obtain the constituent's serial number (see Figures 4-24 and 4-25).

When a message is sent to the group name, the translation from the group name to a constituent object happens transparently in the global object manager. The `PreferredConstituent` system call also performs this translation. Just like any ID-to-physical-address translation, the object manager first checks the XLATE table. If it finds a match for the DID there, it immediately returns the physical address from the XLATE table. If not, it constructs the ID of a nearby constituent by appending the group serial number to the local linear node number with the lowest *max*(*lg*(*Stride*),*0*) bits replaced with the lowest bits from the group linear home node number. Then the resulting constituent ID is looked up in the usual object manager manner. If a physical address of the constituent is found, it is entered into the XLATE table bound with the DID to accelerate the lookup next time.

The above algorithm deterministically maps every node in the J-Machine to exactly one constituent of the distributed object. Having such a deterministic mapping is important because a method running on a distributed object may reference the distributed object several times during its execution, and it is very important that it get the same constituent every time. For example, the method might be suspended while accessing fields of a constituent. When the method restarts and references the constituent again, it is important that it refer to the

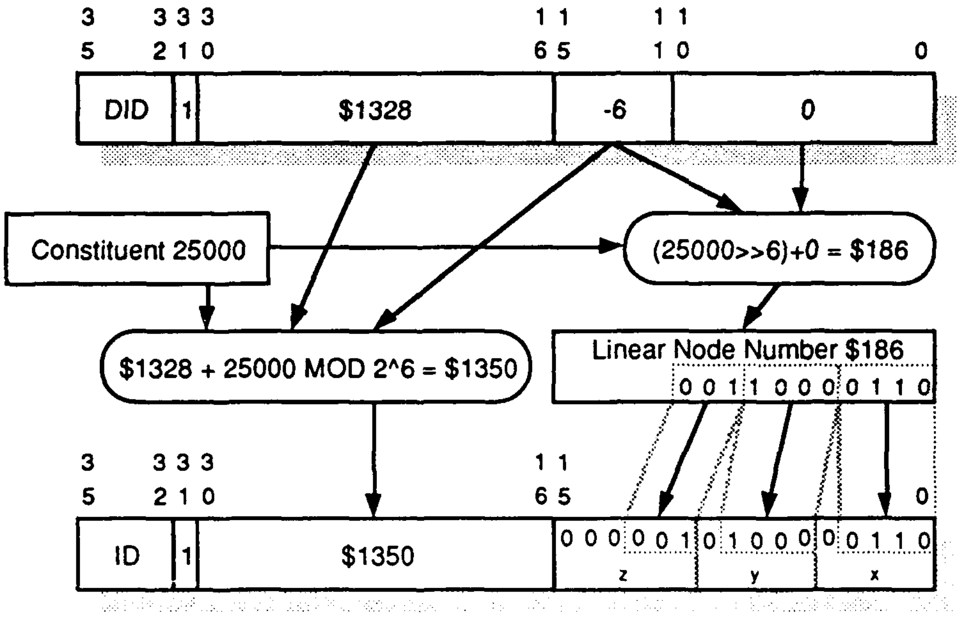


Figure 4-25. Looking up a Constituent in a Dense Distributed Object

This figure illustrates the co system call looking up constituent 25000 in a 131072-constituent distributed object on a 2048-node J-Machine organized as 16×16×8. The stride is 2048/131072=1/64, so lg(stride) is -6. The home node number should be zero in a dense object. The distributed object has a block of 64 reserved serial numbers starting with $1328.

The constituent number 25000 is multiplied by the stride 1/64 and added to 0 to get constituent 25000's linear node number, $186. The constituent's serial number is determined by calculating 25000 MOD 64 and adding it to the base serial number. As before, the dimensions in the linear node number are unpacked to get the constituent's ID.

same one. Since processes can't migrate across nodes, the function will, in fact, refer to the same constituent every time it translates the DID to a physical address.

The above mapping will utilize the distributed object's constituents uniformly if calls to the distributed object come from a uniform distribution of nodes, unless the stride is less than one, in which case only one distributed object representative is chosen per node. If the MDPs were arranged in a linear array, the above mapping would always yield either the closest or the second-closest constituent to a given node. Since the MDPs are actually arranged in a two or three-dimensional mesh, the mapping will tend to cluster the constituents in lines or planes of the mesh, which may or may not produce favorable communication patterns. Overall, though, the current mapping approach does have the advantage of simplicity, and it is useful for small-scale J-Machines.

Allocating Distributed Objects

(NewDistobj class:class **size**:integer) :distobj **Function**

Distributed objects are allocated by calling the `NewDistobj` function in the MDP runtime system. That function first checks whether it was called on node 0; if not, it forwards its message to node 0, and the function is invoked there. If invoked on node 0, the function calculates the physical size of the distributed object by rounding the given logical size size to the nearest higher power of two. Then the stride is computed by dividing the number of MDPs in the J-Machine by the physical size; since the relevant numbers are all powers of two, the computations are done using base-2 logarithms. *Max*(*1*/*stride*,*1*) consecutive distributed object serial numbers are allocated for this distributed object, and a random home node is chosen between 0 and $\lceil stride \rceil - 1$, inclusive. A global variable is used to maintain the next free DID number. Finally, a DID is constructed from the above information, and a `NewDistobj`-

`Tree` message is sent to the zeroth constituent of the distributed object (which does not exist yet, but the `Co` function can calculate its ID anyway). When that message returns, the DID is returned to the caller.

(`NewDistobjTree` `class`:class `size`:integer `ID`:distobj `start`,`logDelta`:integer) :null
Function

`NewDistobjTree` creates constituents numbered start through $(\text{start}+2^{\text{logDelta}}-1)$ of the distributed object with the DID ID and then returns. Each constituent has group, index, and logical size instance variables, which are initialized to the appropriate values; size is the logical size. `NewDistobjTree` works by creating the constituent start if logDelta is zero or by recursing itself on the two halves of its range if logDelta is positive.

The current implementation will have to be extended on a larger system so as not to bottleneck node 0, but it is adequate for small and medium-range systems.

4.4. Summary

The Cosmos operating system provides the software extension to the MDP architecture needed to run Concurrent Smalltalk programs. The operating system is comprised of a kernel resident on each MDP and a set of Concurrent Smalltalk functions written in either MDP assembly language or Concurrent Smalltalk.

The operating system is built in layers which include the heap manager, BRAT manager, object manager, context manager, global object manager, method manager, control manager, utilities, and MDP and CST runtime systems. Efficiency and re-entrancy problems were recurring issues in the design of the operating system kernel. The criticality system was developed to deal with the re-entrancy and double faulting problems. In addition, many routines are inlined in other routines to make the efficiency reasonable and avoid double faults and re-entrancy problems (in some cases a system call cannot call another system call but can use it inlined because there are no more free data registers on the MDP; global variables cannot be used as temporaries in routines running at criticality less than 2).

The operating system facilities were streamlined and simplified compared with those proposed in [38]. The emphasis was on making resource allocation decisions as late as possible. Thus, the size of the BRAT is varied dynamically at run time instead of being fixed at operating system compile time as in [38]. The object migration protocol has been streamlined compared with the one in [38]. The resource wait table in [38] has been eliminated entirely; the BRAT manager is a general-purpose mechanism that can perform the same task better.

Finally, a scheme for quickly addressing constituents of distributed objects was designed. The scheme is very fast and requires only knowledge of a group ID to find either some nearby constituent or any given constituent. Disadvantages of the scheme include the necessity of rounding the size of a distributed object up to the nearest power of two and a resulting decreased load on the higher-numbered MDPs in the J-Machine. Means of circumventing these disadvantages were explored.

Chapter 5. Sample Program

This chapter presents the progress of a simple program through the various stages of compilation. Unfortunately, it is difficult to write a simple sample program that exercises all of the features of a compiler. Instead of trying to write a contrived sample program that exercised as many features as possible, I decided that a simpler program that exercised the major optimizations would make a better example. If an illustration of a more esoteric optimization is desired, one can write an appropriate Concurrent Smalltalk program, compile it with Optimist II, and watch the intermediate output.

The source program, listed in Figure 5-1, returns the sum of the integers from 0 to n. Figure 5-2 shows a transcript of the interactive Optimist II session in which the program was entered, tested on a few inputs, and then compiled.

```
(defmethod average integer (b:integer)
  (// (+ self b) 2))

(defmethod average boolean (b:boolean)
  false)

(defmethod rangesum integer (high)
  (if (= self high)
    self
    (let ((middle (average self high)))
      (+ (rangesum self middle)
         (rangesum (+ middle 1) high)))))

(defun sum (n)
  (rangesum 0 n))
```

Figure 5-1. The Rangesum Program

The sum function adds the integers from 0 to n, inclusive. The rangesum method adds the integers from self to high, inclusive. The average method returns the average of two integers; the definition of average for booleans was included just to confuse the compiler a bit.

```
CST:(+ 2 2)
#<Integer 4>
CST:(include)
#<Cst-Lambda 5024988 SUM>
CST:(sum 0)
#<Integer 0>
CST:(sum 1)
#<Integer 1>
CST:(sum 2)
#<Integer 3>
CST:(sum 10)
#<Integer 55>
CST:(average 3 5)
#<Integer 4>
CST:(average true false)
#<False>
CST:(sum 100)
#<Integer 5050>
CST:(rangesum 10 13)
#<Integer 46>
CST:(compile sum "::fact:Rangesum.mdp")

Optimizing #<Cst-Lambda 4713968 CST::SUM>
Expanded continuations
Folded constants
Forwarded replies

Optimizing #<Cst-Lambda 4711636 CST::RANGESUM>
Collapsed nconcurrentlys
Expanded continuations
Specialized local types
Deleted moves
Deleted touches
Folded constants

Optimizing #<Cst-Lambda 4709940 CST::AVERAGE>
Expanded continuations
```

```
Specialized local types
Deleted locals

Back to #<Cst-Lambda 4711636 CST::RANGESUM>
Substituted inlines
Specialized local types
Deleted moves
Deleted touches
Propagated values
Deleted dead definitions
Deleted locals

Back to #<Cst-Lambda 4713968 CST::SUM>
Deleted locals
Inserted ENTER and EXIT
Split statements
Optimized built-ins
Inserted ENTER and EXIT

Generating code

Assembling
Initialized vlocs

Printing
Assigned labels

Generating code

Assembling
Inserted branches
Initialized vlocs
Compacted SENDs

Printing
Assigned labels
#<Cst-Lambda 4713968 SUM>
```

Figure 5-2. Rangesum Interactive Session

The Rangesum file was read in the (include) directive, at which time the user interactively chose the file name using a Macintosh dialog. A few functions were then tested, after which point the file was compiled.

The following sections will illustrate the actions of some of the compiler's optimizations on the program in Figure 5-1. Please refer to Chapter 3 and [21] for explanations of the transformations.

Initial Phase

The initial phase of the compiler first performs a few macro expansions on the input program, compiles the program into hcode, and then performs some transformations on that hcode to get it into a form that the rest of the compiler can use. Figure 5-3 shows the macroexpansions which are done by the Optimist II parser, and Figure 5-4 shows the hcode produced by the parser. To save space, only the transformations on the rangesum method will be shown from this point on.

Optimization Phase

The Optimist II optimization phase performs local and global optimizations on the program. The order of the optimizations can be seen in the transcript in Figure 5-2; the compiler often interrupts the optimization of one function to optimize another because it wants to inline the second function in the first.

The first transformation done by the optimization phase is the collapsing of nconcurrentlys and the expansion of continuations to the two-variable format, yielding the hcode in Figure 5-5. The threads of the nconcurrently are inlined in the function's main body, and the nconcurrently statement is removed. Then, since an MDP continuation is actually two words (a context ID and an offset within that context where the return value should be stored), each continuation variable is replaced by two variables.

```
(defmethod rangesum integer (high)
  (if (= self high)
    self
    (let ((middle (average self high)))
      (+ (rangesum self middle)
         (rangesum (+ middle 1) high)))))

(DEFMETHOD RANGESUM INTEGER (HIGH):#:OBJECT
  (IF (= SELF HIGH)
    SELF
    (LET ((MIDDLE (AVERAGE SELF HIGH)))
      (+ (RANGESUM SELF MIDDLE) (RANGESUM (+ MIDDLE 1) HIGH)))))

(DEFMETHOD RANGESUM INTEGER (HIGH)::(CONTINUATION:#:OBJECT)
  (IF (= SELF HIGH)
    SELF
    (LET ((MIDDLE (AVERAGE SELF HIGH)))
      (+ (RANGESUM SELF MIDDLE) (RANGESUM (+ MIDDLE 1) HIGH)))))

(BEGIN
 (DEFSELECTOR RANGESUM)
 (ADD-METHOD RANGESUM INTEGER
   (METHOD-LAMBDA INTEGER (HIGH)::(CONTINUATION:#:OBJECT) &NAME RANGESUM
     (IF (= SELF HIGH)
       SELF
       (LET ((MIDDLE (AVERAGE SELF HIGH)))
         (+ (RANGESUM SELF MIDDLE) (RANGESUM (+ MIDDLE 1) HIGH)))))))

... (LAMBDA (SELF:INTEGER HIGH)::(CONTINUATION:#:OBJECT) &NAME RANGESUM
     (_WITH-OBJECT (SELF:INTEGER)
       (IF (= SELF HIGH)
         SELF
         (LET ((MIDDLE (AVERAGE SELF HIGH)))
           (+ (RANGESUM SELF MIDDLE) (RANGESUM (+ MIDDLE 1) HIGH)))))) ...
```

Figure 5-3. Rangesum Macroexpansion

The rangesum function is first macroexpanded through two macros that add the class of the continuation to the `defmethod` syntax (see Section A.5). *Then the* `defmethod` *itself is expanded into a combination of a* `defselector` and an `add-method` of a `method-lambda`. Later the `method-lambda` is expanded into a `lambda`.

```
(LAMBDA CST::RANGESUM
 (#<Parameter CST::SELF #<P-Class CST::INTEGER>
  #<Parameter CST::HIGH #<S-Class CST::OBJECT>)
 (#<Parameter CST::CONTINUATION #<Cont-Type #<S-Class CST::OBJECT>>>)
 ()
 (((LOCAL 435) #<S-Class CST::OBJECT>)
  ((LOCAL 434) #<S-Class CST::OBJECT>)
  ((LOCAL 433) #<S-Class CST::OBJECT>)
  ((LOCAL 432) #<S-Class CST::OBJECT>)
  ((LOCAL CST::MIDDLE) #<S-Class CST::OBJECT>)
  ((LOCAL 431) #<S-Class CST::OBJECT>)
  ((LOCAL 430) #<S-Class CST::OBJECT>)
  ((LOCAL 429) #<S-Class CST::OBJECT>)
  ((LOCAL CST::SELF) #<P-Class CST::INTEGER>)
  ((LOCAL CST::HIGH) #<S-Class CST::OBJECT>)
  ((LOCAL CST::CONTINUATION) #<Cont-Type #<S-Class CST::OBJECT>))
 (ASSERT-TYPE #<P-Class CST::INTEGER> (LOCAL CST::SELF))
 (APPLY ((LOCAL 429))
        (#<Built-In-Selector CST::=> (LOCAL CST::SELF) (LOCAL CST::HIGH)))
 (IF :FALSE (LOCAL 429) 2587)
 (MOVE (LOCAL 430) (LOCAL CST::SELF))
 (JUMP 2611)
 (LABEL 2587)
 (APPLY ((LOCAL 431)) ((GLOBAL CST::AVERAGE) (LOCAL CST::SELF) (LOCAL CST::HIGH)))
 (MOVE (LOCAL CST::MIDDLE) (LOCAL 431))
 (TOUCH (LOCAL CST::MIDDLE))
 (NCONCURRENTLY
  (((APPLY ((LOCAL 433))
           (#<Built-In-Selector CST::+> (LOCAL CST::MIDDLE) #<Integer 1>))
    (APPLY ((LOCAL 434)) ((GLOBAL CST::RANGESUM) (LOCAL 433) (LOCAL CST::HIGH))))
   ((APPLY ((LOCAL 432))
           ((GLOBAL CST::RANGESUM) (LOCAL CST::SELF) (LOCAL CST::MIDDLE))))))
 (APPLY ((LOCAL 435)) (#<Built-In-Selector CST::+> (LOCAL 432) (LOCAL 434)))
 (MOVE (LOCAL 430) (LOCAL 435))
 (LABEL 2611)
 (MOVE (CONT-REF LOCAL CST::CONTINUATION) (LOCAL 430)))
```

Figure 5-4. Initial Rangesum Hcode

This hcode is the final output of the initial phase. The lambda is comprised of the two parameters (self and high), a return (continuation), no display parameters, a list of local variables, and a representation of the hcode digraph.

Next, the compiler starts the iterative optimizations. The first successful one is local type specialization, which uses type dataflow analysis to detect the fact that local 429 always holds a boolean value, so it changes local 429's type to `boolean`.

```
(LAMBDA CST::RANGESUM
 (#<Parameter CST::SELF #<P-Class CST::INTEGER>
  #<Parameter CST::HIGH #<S-Class CST::OBJECT>)
 (#<Parameter CST::CONTINUATION #<Cont-Type #<S-Class CST::OBJECT>>>)
 ()
 (((LOCAL 435) #<S-Class CST::OBJECT>)
  ((LOCAL 434) #<S-Class CST::OBJECT>)
  ((LOCAL 433) #<S-Class CST::OBJECT>)
  ((LOCAL 432) #<S-Class CST::OBJECT>)
  ((LOCAL CST::MIDDLE) #<S-Class CST::OBJECT>)
  ((LOCAL 431) #<S-Class CST::OBJECT>)
  ((LOCAL 430) #<S-Class CST::OBJECT>)
  ((LOCAL 429) #<S-Class CST::OBJECT>)
  ((LOCAL CST::SELF) #<P-Class CST::INTEGER>)
  ((LOCAL CST::HIGH) #<S-Class CST::OBJECT>)
  ((LOCAL CST::CONTINUATION) #<Cont-Type #<S-Class CST::OBJECT>)
  ((LOCAL CST::CONTINUATION) #<P-Class CST::CONTEXT>)
  ((LOCAL CST::CONTINUATION) #<Disp-Type #<S-Class CST::OBJECT>))
 (ASSERT-TYPE #<P-Class CST::INTEGER> (LOCAL CST::SELF))
 (APPLY ((LOCAL 429))
        (#<Built-In-Selector CST::=> (LOCAL CST::SELF) (LOCAL CST::HIGH)))
 (IF :FALSE (LOCAL 429) 2587)
 (MOVE (LOCAL 430) (LOCAL CST::SELF))
 (JUMP 2611)
 (LABEL 2587)
 (APPLY ((LOCAL 431)) (#<Selector CST::AVERAGE> (LOCAL CST::SELF) (LOCAL CST::HIGH)))
 (MOVE (LOCAL CST::MIDDLE) (LOCAL 431))
 (TOUCH (LOCAL CST::MIDDLE))
 (APPLY ((LOCAL 433)) (#<Built-In-Selector CST::+> (LOCAL CST::MIDDLE) #<Integer 1>))
 (APPLY ((LOCAL 434)) (#<Selector CST::RANGESUM> (LOCAL 433) (LOCAL CST::HIGH)))
 (APPLY ((LOCAL 432)) (#<Selector CST::RANGESUM> (LOCAL CST::SELF) (LOCAL CST::MIDDLE)))
 (APPLY ((LOCAL 435)) (#<Built-In-Selector CST::+> (LOCAL 432) (LOCAL 434)))
 (MOVE (LOCAL 430) (LOCAL 435))
 (LABEL 2611)
 (MOVE (CONT-REF (LOCAL CST::CONTINUATION) (LOCAL CST::CONTINUATION)) (LOCAL 430)))
```

Figure 5-5. Hcode after Initial Transformations

The *nconcurrently* statement has been broken into its threads, and two variables assigned to hold the continuation. The two new continuation variables have the same name as the single old continuation variable, which is still present, but the compiler does not get confused over variable name conflicts.

```
(LAMBDA CST::RANGESUM
 (#<Parameter CST::SELF #<P-Class CST::INTEGER>
  #<Parameter CST::HIGH #<S-Class CST::OBJECT>)
 (#<Parameter CST::CONTINUATION #<Cont-Type #<S-Class CST::OBJECT>>>)
 ()
 (((LOCAL 435) #<S-Class CST::OBJECT>)
  ((LOCAL 434) #<S-Class CST::OBJECT>)
  ((LOCAL 433) #<S-Class CST::OBJECT>)
  ((LOCAL 432) #<S-Class CST::OBJECT>)
  ((LOCAL CST::MIDDLE) #<S-Class CST::OBJECT>)
  ((LOCAL 431) #<S-Class CST::OBJECT>)
  ((LOCAL 430) #<S-Class CST::OBJECT>)
  ((LOCAL 429) #<P-Class CST::BOOLEAN>)
  ((LOCAL CST::SELF) #<S-Class CST::OBJECT>)
  ((LOCAL CST::HIGH) #<S-Class CST::OBJECT>)
  ((LOCAL CST::CONTINUATION) #<Cont-Type #<S-Class CST::OBJECT>)
  ((LOCAL CST::CONTINUATION) #<P-Class CST::CONTEXT>)
  ((LOCAL CST::CONTINUATION) #<Disp-Type #<S-Class CST::OBJECT>))
 (ASSERT-TYPE #<P-Class CST::INTEGER> (LOCAL 435))
 (APPLY ((LOCAL 429)) (#<Built-In-Selector CST::=> (LOCAL 435) (LOCAL CST::HIGH)))
 (IF :FALSE (LOCAL 429) 2587)
 (JUMP 2611)
 (LABEL 2587)
 (APPLY ((LOCAL 431)) (#<Selector CST::AVERAGE> (LOCAL 435) (LOCAL CST::HIGH)))
 (APPLY ((LOCAL 433)) (#<Built-In-Selector CST::+> (LOCAL 431) #<Integer 1>))
 (APPLY ((LOCAL 434)) (#<Selector CST::RANGESUM> (LOCAL 433) (LOCAL CST::HIGH)))
 (APPLY ((LOCAL 432)) (#<Selector CST::RANGESUM> (LOCAL 435) (LOCAL 431)))
 (APPLY ((LOCAL 435)) (#<Built-In-Selector CST::+> (LOCAL 432) (LOCAL 434)))
 (LABEL 2611)
 (MOVE (CONT-REF (LOCAL CST::CONTINUATION) (LOCAL CST::CONTINUATION)) (LOCAL 435)))
```

Figure 5-6. Locally Optimized Hcode

This hcode has been fully optimized using the optimizations in the original Optimist compiler. Note that due to move elimination the self parameter is no longer stored in the old self local; instead, a new local numbered 435 is now used to hold the self value.

Afterwards, the standard dataflow optimizations described in [21] remove a few moves and a touch to yield the hcode in Figure 5-6. Then the constant folder realizes through type inference that only one possible method of the rangesum and average selectors could be called, so it replaces the method calls with direct function calls (Figure 5-7).

```
(LAMBDA CST::RANGESUM
 (#<Parameter CST::SELF #<P-Class CST::INTEGER>
  #<Parameter CST::HIGH #<S-Class CST::OBJECT>)
 (#<Parameter CST::CONTINUATION #<Cont-Type #<S-Class CST::OBJECT>>>)
 ()
 (((LOCAL 435) #<S-Class CST::OBJECT>)
  ((LOCAL 434) #<S-Class CST::OBJECT>)
  ((LOCAL 433) #<S-Class CST::OBJECT>)
  ((LOCAL 432) #<S-Class CST::OBJECT>)
  ((LOCAL CST::MIDDLE) #<S-Class CST::OBJECT>)
  ((LOCAL 431) #<S-Class CST::OBJECT>)
  ((LOCAL 430) #<S-Class CST::OBJECT>)
  ((LOCAL 429) #<P-Class CST::BOOLEAN>)
  ((LOCAL CST::SELF) #<S-Class CST::OBJECT>)
  ((LOCAL CST::HIGH) #<S-Class CST::OBJECT>)
  ((LOCAL CST::CONTINUATION) #<Cont-Type #<S-Class CST::OBJECT>)
  ((LOCAL CST::CONTINUATION) #<P-Class CST::CONTEXT>)
  ((LOCAL CST::CONTINUATION) #<Disp-Type #<S-Class CST::OBJECT>))
 (ASSERT-TYPE #<P-Class CST::INTEGER> (LOCAL 435))
 (APPLY ((LOCAL 429)) (#<Built-In-Selector CST::=> (LOCAL 435) (LOCAL CST::HIGH)))
 (IF :FALSE (LOCAL 429) 2587)
 (JUMP 2611)
 (LABEL 2587)
 (APPLY ((LOCAL 431)) ((LAMBDA CST::AVERAGE) (LOCAL 435) (LOCAL CST::HIGH)))
 (APPLY ((LOCAL 433)) (#<Built-In-Selector CST::+> (LOCAL 431) #<Integer 1>))
 (APPLY ((LOCAL 434)) ((LAMBDA CST::RANGESUM) (LOCAL 433) (LOCAL CST::HIGH)))
 (APPLY ((LOCAL 432)) ((LAMBDA CST::RANGESUM) (LOCAL 435) (LOCAL 431)))
 (APPLY ((LOCAL 435)) (#<Built-In-Selector CST::+> (LOCAL 432) (LOCAL 434)))
 (LABEL 2611)
 (MOVE (CONT-REF (LOCAL CST::CONTINUATION) (LOCAL CST::CONTINUATION)) (LOCAL 435)))
```

Figure 5-7. Hcode after Global Constant Propagation

The constant propagator found that the average and rangesum method calls would always invoke the same methods, so it replaced them with function calls.

```
(LAMBDA CST::AVERAGE
 (#<Parameter CST::SELF #<P-Class CST::INTEGER>
  #<Parameter CST::B #<P-Class CST::INTEGER>)
 (#<Parameter CST::CONTINUATION #<Cont-Type #<S-Class CST::OBJECT>>>)
 ()
 (((LOCAL 424) #<P-Class CST::INTEGER>)
  ((LOCAL 423) #<P-Class CST::INTEGER>)
  ((LOCAL CST::SELF) #<P-Class CST::INTEGER>)
  ((LOCAL CST::B) #<P-Class CST::INTEGER>)
  ((LOCAL CST::CONTINUATION) #<P-Class CST::CONTEXT>)
  ((LOCAL CST::CONTINUATION) #<Disp-Type #<S-Class CST::OBJECT>))
 (ASSERT-TYPE #<P-Class CST::INTEGER> (LOCAL CST::SELF))
 (APPLY ((LOCAL 423)) (#<Built-In-Selector CST::+> (LOCAL CST::SELF) (LOCAL CST::B)))
 (APPLY ((LOCAL 424)) (#<Built-In-Selector CST::/> (LOCAL 423) #<Integer 2>))
 (MOVE (CONT-REF (LOCAL CST::CONTINUATION) (LOCAL CST::CONTINUATION)) (LOCAL 424)))
```

Figure 5-8. Optimized Average Hcode

The average method for integers has been optimized in an attempt to inline it inside rangesum..

Next, the optimizer attempts to inline the average and rangesum functions. Due to the antirecursion restrictions, it cannot inline rangesum inside itself, but it is more successful with average. In order to inline average, it first optimizes it, yielding the hcode in Figure 5-8. Then it checks that the inlining heuristics are satisfied—they are because the optimized average contains only two primitive calls. Average does not perform any computation after it replies, so all of the requirements for inlining have been satisfied. Therefore, the optimizer inlines average inside rangesum to produce the hcode in Figure 5-9, which is optimized to the hcode in Figure 5-10 at end of the general optimizations.

```
(LAMBDA CST::RANGESUM
 (#<Parameter CST::SELF #<P-Class CST::INTEGER>
  #<Parameter CST::HIGH #<S-Class CST::OBJECT>)
 (#<Parameter CST::CONTINUATION #<Cont-Type #<S-Class CST::OBJECT>>>)
 ()
 (((LOCAL 435) #<S-Class CST::OBJECT>)
  ((LOCAL 434) #<S-Class CST::OBJECT>)
  ((LOCAL 433) #<S-Class CST::OBJECT>)
  ((LOCAL 432) #<S-Class CST::OBJECT>)
  ((LOCAL 431) #<S-Class CST::OBJECT>)
  ((LOCAL 429) #<P-Class CST::BOOLEAN>)
  ((LOCAL CST::HIGH) #<S-Class CST::OBJECT>)
  ((LOCAL CST::CONTINUATION) #<P-Class CST::CONTEXT>)
  ((LOCAL CST::CONTINUATION) #<Disp-Type #<S-Class CST::OBJECT>)
  ((LOCAL 424) #<P-Class CST::INTEGER>)
  ((LOCAL 423) #<P-Class CST::INTEGER>)
  ((LOCAL CST::SELF) #<P-Class CST::INTEGER>)
  ((LOCAL CST::B) #<P-Class CST::INTEGER>)
  ((LOCAL 455) #<S-Class CST::OBJECT>))
 (ASSERT-TYPE #<P-Class CST::INTEGER> (LOCAL 435))
 (APPLY ((LOCAL 429)) (#<Built-In-Selector CST::=> (LOCAL 435) (LOCAL CST::HIGH)))
 (IF :FALSE (LOCAL 429) 2979)
 (JUMP 2611)
 (LABEL 2979)
 (MOVE (LOCAL CST::B) (LOCAL CST::HIGH))
 (MOVE (LOCAL CST::SELF) (LOCAL 435))
 (TOUCH (LOCAL CST::B))
 (TOUCH (LOCAL CST::SELF))
 (ASSERT-TYPE #<P-Class CST::INTEGER> (LOCAL CST::SELF))
 (APPLY ((LOCAL 423)) (#<Built-In-Selector CST::+> (LOCAL CST::SELF) (LOCAL CST::B)))
 (APPLY ((LOCAL 424)) (#<Built-In-Selector CST:://> (LOCAL 423) #<Integer 2>))
 (MOVE (LOCAL 455) (LOCAL 424))
 (MOVE (LOCAL 431) (LOCAL 455))
 (APPLY ((LOCAL 433)) (#<Built-In-Selector CST::+> (LOCAL 431) #<Integer 1>))
 (APPLY ((LOCAL 434)) ((LAMBDA CST::RANGESUM) (LOCAL 433) (LOCAL CST::HIGH)))
 (APPLY ((LOCAL 432)) ((LAMBDA CST::RANGESUM) (LOCAL 435) (LOCAL 431)))
 (APPLY ((LOCAL 435)) (#<Built-In-Selector CST::+> (LOCAL 432) (LOCAL 434)))
 (LABEL 2611)
 (MOVE (CONT-REF (LOCAL CST::CONTINUATION) (LOCAL CST::CONTINUATION)) (LOCAL 435)))
```

Figure 5-9. Rangesum with Average Inlined
The integer average method has just been inlined into rangesum.

```
(LAMBDA CST::RANGESUM
 (#<Parameter CST::SELF #<P-Class CST::INTEGER>
  #<Parameter CST::HIGH #<S-Class CST::OBJECT>)
 (#<Parameter CST::CONTINUATION #<Cont-Type #<S-Class CST::OBJECT>>>)
 ()
 (((LOCAL 435) #<S-Class CST::OBJECT>)
  ((LOCAL 434) #<S-Class CST::OBJECT>)
  ((LOCAL 433) #<P-Class CST::INTEGER>)
  ((LOCAL 432) #<S-Class CST::OBJECT>)
  ((LOCAL 429) #<P-Class CST::BOOLEAN>)
  ((LOCAL CST::HIGH) #<S-Class CST::OBJECT>)
  ((LOCAL CST::CONTINUATION) #<P-Class CST::CONTEXT>)
  ((LOCAL CST::CONTINUATION) #<Disp-Type #<S-Class CST::OBJECT>)
  ((LOCAL 424) #<P-Class CST::INTEGER>)
  ((LOCAL 423) #<P-Class CST::INTEGER>))
 (ASSERT-TYPE #<P-Class CST::INTEGER> (LOCAL 435))
 (APPLY ((LOCAL 429)) (#<Built-In-Selector CST::=> (LOCAL 435) (LOCAL CST::HIGH)))
 (IF :FALSE (LOCAL 429) 2541)
 (JUMP 2611)
 (LABEL 2541)
 (ASSERT-TYPE #<P-Class CST::INTEGER> (LOCAL 435))
 (APPLY ((LOCAL 423)) (#<Built-In-Selector CST::+> (LOCAL 435) (LOCAL CST::HIGH)))
 (APPLY ((LOCAL 424)) (#<Built-In-Selector CST:://> (LOCAL 423) #<Integer 2>))
 (APPLY ((LOCAL 433)) (#<Built-In-Selector CST::+> (LOCAL 424) #<Integer 1>))
 (APPLY ((LOCAL 434)) ((LAMBDA CST::RANGESUM) (LOCAL 433) (LOCAL CST::HIGH)))
 (APPLY ((LOCAL 432)) ((LAMBDA CST::RANGESUM) (LOCAL 435) (LOCAL 424)))
 (APPLY ((LOCAL 435)) (#<Built-In-Selector CST::+> (LOCAL 432) (LOCAL 434)))
 (LABEL 2611)
 (MOVE (CONT-REF (LOCAL CST::CONTINUATION) (LOCAL CST::CONTINUATION)) (LOCAL 435)))
```

Figure 5-10. Rangesum after General Optimizations
The rangesum hcode is now at the "Optimized Hcode" stage in Figure 3-4.

The MDP-specific optimizations remove the assert-type hcode, reduce the division to a shift, and insert enter and exit hcodes to yield the final hcode in Figure 5-11.

```
(LAMBDA CST::RANGESUM
 (#<Parameter CST::SELF #<P-Class CST::INTEGER>
  #<Parameter CST::HIGH #<S-Class CST::OBJECT>)
 (#<Parameter CST::CONTINUATION #<Cont-Type #<S-Class CST::OBJECT>>>)
 ()
 (((LOCAL 435) #<S-Class CST::OBJECT>)
  ((LOCAL 434) #<S-Class CST::OBJECT>)
  ((LOCAL 433) #<P-Class CST::INTEGER>)
  ((LOCAL 432) #<S-Class CST::OBJECT>)
  ((LOCAL 429) #<P-Class CST::BOOLEAN>)
  ((LOCAL CST::HIGH) #<S-Class CST::OBJECT>)
  ((LOCAL CST::CONTINUATION) #<P-Class CST::CONTEXT>)
  ((LOCAL CST::CONTINUATION) #<Disp-Type #<S-Class CST::OBJECT>)
  ((LOCAL 424) #<P-Class CST::INTEGER>)
  ((LOCAL 423) #<P-Class CST::INTEGER>))
 (ENTER)
 (APPLY ((LOCAL 429)) (#<Built-In-Selector CST::=> (LOCAL 435) (LOCAL CST::HIGH)))
 (IF :FALSE (LOCAL 429) 2547)
 (JUMP 2611)
 (LABEL 2547)
 (APPLY ((LOCAL 423)) (#<Built-In-Selector CST::+> (LOCAL 435) (LOCAL CST::HIGH)))
 (APPLY ((LOCAL 424)) (#<Built-In-Selector CST::ASH> (LOCAL 423) #<Integer -1>))
 (APPLY ((LOCAL 433)) (#<Built-In-Selector CST::+> (LOCAL 424) #<Integer 1>))
 (APPLY ((LOCAL 434)) ((LAMBDA CST::RANGESUM) (LOCAL 433) (LOCAL CST::HIGH)))
 (APPLY ((LOCAL 432)) ((LAMBDA CST::RANGESUM) (LOCAL 435) (LOCAL 424)))
 (APPLY ((LOCAL 435)) (#<Built-In-Selector CST::+> (LOCAL 432) (LOCAL 434)))
 (LABEL 2611)
 (MOVE (CONT-REF (LOCAL CST::CONTINUATION) (LOCAL CST::CONTINUATION)) (LOCAL 435))
 (EXIT))
```

Figure 5-11. Final Hcode

This is the final hcode produced before it is compiled into MDP assembly language.

Compilation Phase

The compilation phase compiles each hcode in Figure 5-11 into MDP assembly instructions and then peephole-optimizes and emits the resulting code to produce the MDPSim file in Figure 5-12. There is no need to describe the transformations here, as an appropriate example is in [21].

The definitions of the label numbers in Figure 5-12 contain expressions of the form `LABEL cObject=(5&mX)<<sX|(5&mY)<<sY|(5&mZ)<<sZ|(5&m3)<<s3|(5&m4)<<s4|(5&m5)<<s5`. This expression means that `cObject` is class with serial number 5. Nevertheless, since objects should be distributed throughout the J-Machine, the bits in the class serial number 5 are permuted to map the low-order bits onto the bits denoting the x, y, and z network coordinates of an object. This is done by the first half of the expression, `(5&mX)<<sX|(5&mY)<<sY|(5&mZ)<<sZ`; `mX`, `mY`, `mZ`, `sX`, `sY`, and `sZ` are constants defined by the operating system and depend on the dimensions of the J-Machine. The second half of the expression, `(5&m3)<<s3|(5&m4)<<s4|(5&m5)<<s5`, maps the rest of the class serial number bits onto the remaining bits. A similar expression, `REF REV fSum=ID:(-2&mX)<<sX|(-2&mY)<<sY|(-2&mZ)<<sZ|(-2&mS)<<sS`, is used to map objects onto nodes.

```
LABEL cObject=(5&mX)<<sX|(5&mY)<<sY|(5&mZ)<<sZ|(5&m3)<<s3|(5&m4)<<s4|(5&m5)<<s5
LABEL cClass=(8&mX)<<sX|(8&mY)<<sY|(8&mZ)<<sZ|(8&m3)<<s3|(8&m4)<<s4|(8&m5)<<s5
LABEL cStandard_Class=(3&mX)<<sX|(3&mY)<<sY|(3&mZ)<<sZ|(3&m3)<<s3|(3&m4)<<s4|(3&m5)<<s5
LABEL cPrimitive_Class=(2&mX)<<sX|(2&mY)<<sY|(2&mZ)<<sZ|(2&m3)<<s3|(2&m4)<<s4|(2&m5)<<s5
LABEL cDistributed_Class=(4&mX)<<sX|(4&mY)<<sY|(4&mZ)<<sZ|(4&m3)<<s3|(4&m4)<<s4|(4&m5)<<s5
LABEL cSymbol=(7&mX)<<sX|(7&mY)<<sY|(7&mZ)<<sZ|(7&m3)<<s3|(7&m4)<<s4|(7&m5)<<s5
LABEL cNull=(6&mX)<<sX|(6&mY)<<sY|(6&mZ)<<sZ|(6&m3)<<s3|(6&m4)<<s4|(6&m5)<<s5
LABEL cFunct=(17&mX)<<sX|(17&mY)<<sY|(17&mZ)<<sZ|(17&m3)<<s3|(17&m4)<<s4|(17&m5)<<s5
LABEL cSelector=(9&mX)<<sX|(9&mY)<<sY|(9&mZ)<<sZ|(9&m3)<<s3|(9&m4)<<s4|(9&m5)<<s5
LABEL cMagnitude=(18&mX)<<sX|(18&mY)<<sY|(18&mZ)<<sZ|(18&m3)<<s3|(18&m4)<<s4|(18&m5)<<s5
LABEL cCharacter=(10&mX)<<sX|(10&mY)<<sY|(10&mZ)<<sZ|(10&m3)<<s3|(10&m4)<<s4|(10&m5)<<s5
LABEL cNumber=(19&mX)<<sX|(19&mY)<<sY|(19&mZ)<<sZ|(19&m3)<<s3|(19&m4)<<s4|(19&m5)<<s5
LABEL cReal=(20&mX)<<sX|(20&mY)<<sY|(20&mZ)<<sZ|(20&m3)<<s3|(20&m4)<<s4|(20&m5)<<s5
LABEL cInteger=(11&mX)<<sX|(11&mY)<<sY|(11&mZ)<<sZ|(11&m3)<<s3|(11&m4)<<s4|(11&m5)<<s5
LABEL cBoolean=(12&mX)<<sX|(12&mY)<<sY|(12&mZ)<<sZ|(12&m3)<<s3|(12&m4)<<s4|(12&m5)<<s5
LABEL cFalse=(13&mX)<<sX|(13&mY)<<sY|(13&mZ)<<sZ|(13&m3)<<s3|(13&m4)<<s4|(13&m5)<<s5
LABEL cTrue=(14&mX)<<sX|(14&mY)<<sY|(14&mZ)<<sZ|(14&m3)<<s3|(14&m4)<<s4|(14&m5)<<s5
LABEL cFloat=(15&mX)<<sX|(15&mY)<<sY|(15&mZ)<<sZ|(15&m3)<<s3|(15&m4)<<s4|(15&m5)<<s5
LABEL cFunction=(16&mX)<<sX|(16&mY)<<sY|(16&mZ)<<sZ|(16&m3)<<s3|(16&m4)<<s4|(16&m5)<<s5
LABEL c_Closure=(21&mX)<<sX|(21&mY)<<sY|(21&mZ)<<sZ|(21&m3)<<s3|(21&m4)<<s4|(21&m5)<<s5
LABEL cContext=(22&mX)<<sX|(22&mY)<<sY|(22&mZ)<<sZ|(22&m3)<<s3|(22&m4)<<s4|(22&m5)<<s5
LABEL cDisplacement=(23&mX)<<sX|(23&mY)<<sY|(23&mZ)<<sZ|(23&m3)<<s3|(23&m4)<<s4|(23&m5)<<s5
LABEL cContinuation=(24&mX)<<sX|(24&mY)<<sY|(24&mZ)<<sZ|(24&m3)<<s3|(24&m4)<<s4|(24&m5)<<s5
LABEL cGlobal=(25&mX)<<sX|(25&mY)<<sY|(25&mZ)<<sZ|(25&m3)<<s3|(25&m4)<<s4|(25&m5)<<s5
LABEL cDistobj=(26&mX)<<sX|(26&mY)<<sY|(26&mZ)<<sZ|(26&m3)<<s3|(26&m4)<<s4|(26&m5)<<s5
```

```
REF REV selPLUS=TAG0:subSEL<<subtagN|(0&mX)<<sX|(0&mY)<<sY|(0&mZ)<<sZ|(0&m3)<<s3|(0&m4)<<s4|(0&m5)<<s5
REF REV selEQUAL=TAG0:subSEL<<subtagN|(1&mX)<<sX|(1&mY)<<sY|(1&mZ)<<sZ|(1&m3)<<s3|(1&m4)<<s4|(1&m5)<<s5
REF REV selAsh=TAG0:subSEL<<subtagN|(2&mX)<<sX|(2&mY)<<sY|(2&mZ)<<sZ|(2&m3)<<s3|(2&m4)<<s4|(2&m5)<<s5
REF REV fRangesum=ID:(-1&mX)<<sX|(-1&mY)<<sY|(-1&mZ)<<sZ|(-1&mS)<<sS
REF REV fSum=ID:(-2&mX)<<sX|(-2&mY)<<sY|(-2&mZ)<<sZ|(-2&mS)<<sS


            MODULE cObject
            DC          MSG:hdrCopyable|cStandard_Class<<offsetN|5
            DC          TAG0:subCLASS<<subtagN|cObject
            DC          MSG:cObject<<offsetN|2
            DC          1
            DC          TAG0:subCLASS<<subtagN|cObject
            END

            MODULE cClass
            DC          MSG:hdrCopyable|cPrimitive_Class<<offsetN|6
            DC          TAG0:subCLASS<<subtagN|cClass
            DC          NIL
            DC          2
            DC          TAG0:subCLASS<<subtagN|cClass
            DC          TAG0:subCLASS<<subtagN|cObject
            END

            MODULE cStandard_Class
            DC          MSG:hdrCopyable|cPrimitive_Class<<offsetN|7
            DC          TAG0:subCLASS<<subtagN|cStandard_Class
            DC          NIL
            DC          3
            DC          TAG0:subCLASS<<subtagN|cStandard_Class
            DC          TAG0:subCLASS<<subtagN|cClass
            DC          TAG0:subCLASS<<subtagN|cObject
            END
```

... MODULEs for the rest of the classes deleted ...

```
            MODULE selPLUS
            DC          MSG:hdrCopyable|cSelector<<offsetN|3
            DC          {selPLUS}
            DC          0
            END

            MODULE selEQUAL
            DC          MSG:hdrCopyable|cSelector<<offsetN|3
            DC          {selEQUAL}
            DC          0
            END

            MODULE selAsh
            DC          MSG:hdrCopyable|cSelector<<offsetN|3
            DC          {selAsh}
            DC          0
            END


            MODULE fRangesum
            DC          MSG:hdrCopyable|cFunction<<offsetN|28
            DC          {fRangesum}
            DC          6
            MOVE        [2,A3],R0                       ;  3
            MOVE        [2,A3],R3                       ;  3.5
            EQUAL       R3,[3,A3],R1                    ;  4
            BT          R1,^L001                        ;  4.5
            ADD         R3,[3,A3],R1                    ;  5
            ASH         R1,-1,R3                        ;  5.5
            ADD         R3,1,R2                         ;  6
            MOVE        R2,R0                           ;  6.5
            CALL        objectNode                      ;  7
            DC          MSG:msgApplyFunction|6          ;  8
            SEND20      R1,R0                           ;  9
            DC          {fRangesum}                     ; 10
            SEND20      R0,R2                           ; 11
            SEND0       [3,A3]                          ; 11.5
            MOVE        6,R0                            ; 12
            SEND2E0     [1,A1],R0                       ; 12.5
            WTAG        R0,6,R0                         ; 13
            MOVE        R0,[6,A1]                       ; 13.5
            MOVE        [2,A3],R0                       ; 14
            CALL        objectNode                      ; 14.5
            DC          MSG:msgApplyFunction|6          ; 15
            SEND20      R1,R0                           ; 16
            DC          {fRangesum}                     ; 17
            SEND0       R0                              ; 18
            SEND20      [2,A3],R3                       ; 18.5
            MOVE        7,R0                            ; 19
            SEND2E0     [1,A1],R0                       ; 19.5
            WTAG        R0,6,R0                         ; 20
            MOVE        R0,[7,A1]                       ; 20.5
            MOVE        [7,A1],R2                       ; 21
            ADD         R2,[6,A1],R1                    ; 21.5
            MOVE        R1,[2,A3]                       ; 22
```

```
L001:    MOVE     [4,A3],R2                    ; 23
         BNIL     R2,^L002                     ; 23.5
         DC       MSG:msgReply|4               ; 24
         SEND20   R2,R0                        ; 25
         SEND0    R2                           ; 25.5
         SEND0    [5,A3]                       ; 26
         SENDE0   [2,A3]                       ; 26.5
L002:    SUSPEND                               ; 27
         END

         MODULE fSum
         DC       MSG:hdrCopyable|cFunction<<offsetN|10
         DC       {fSum}
         DC       5
         MOVE     0,R0                         ;  3
         CALL     objectNode                   ;  3.5
         DC       MSG:msgApplyFunction|6       ;  4
         SEND20   R1,R0                        ;  5
         DC       {fRangesum}                  ;  6
         SEND0    R0                           ;  7
         SEND0    0                            ;  7.5
         SEND0    [2,A3]                       ;  8
         SEND0    [3,A3]                       ;  8.5
         SENDE0   [4,A3]                       ;  9
         SUSPEND                               ;  9.5
         END

DOWNLOAD cObject
DOWNLOAD cClass
DOWNLOAD cStandard_Class
DOWNLOAD cPrimitive_Class
DOWNLOAD cDistributed_Class
DOWNLOAD cSymbol
DOWNLOAD cNull
DOWNLOAD cFunct
DOWNLOAD cSelector
DOWNLOAD cMagnitude
DOWNLOAD cCharacter
DOWNLOAD cNumber
DOWNLOAD cReal
DOWNLOAD cInteger
DOWNLOAD cBoolean
DOWNLOAD cFalse
DOWNLOAD cTrue
DOWNLOAD cFloat
DOWNLOAD cFunction
DOWNLOAD c_Closure
DOWNLOAD cContext
DOWNLOAD cDisplacement
DOWNLOAD cContinuation
DOWNLOAD cGlobal
DOWNLOAD cDistobj
DOWNLOAD selPLUS
DOWNLOAD selEQUAL
DOWNLOAD selAsh
DOWNLOAD fRangesum
DOWNLOAD fSum

RUN
```

Figure 5-12 MDPSim Output File

Except for Cosmos, this file contains all code and data necessary to run sum on a J-Machine. The file starts with class number definitions, which are followed by definitions of the classes themselves, including the class hierarchy. The selectors are defined next, followed by code and MDPSim statements that download all of the code, selector, and class modules to the simulated J-Machine. The RUN command runs the J-Machine until all modules have been loaded.

Only the functions and selectors necessary to run the program have been compiled. For example, neither average method has been included because, after optimization, neither is necessary to run sum. Similarly, all method dispatches have been optimized out, so there is no need to include the definition of the rangesum selector.

Running Rangesum

Before rangesum can be run on MDPSim, a file holding the calls that will be done needs to be defined; the file that was used is shown in Figure 5-13. Each MESSAGE directive defines an `ApplyFunction` message that can be used to call the sum function. The argument is the third word of the message, while the fourth and fifth words contain a magic continuation that cause the `Reply` message to be printed by MDPSim in the listener window. The MESSAGE definitions can also be entered into MDPSim manually.

Once the calls file is written, MDPSim can be started and used to run sum on a sample input. An example session is shown in Figure 5-14, in which the input 10 is tried on sum, and the statistics observed. The results will be discussed in more detail in Chapter 7.

```
MESSAGE sum1
MSG:msgApplyFunction|5
{fSum}
1
IONODE
0
END

MESSAGE sum10
MSG:msgApplyFunction|5
{fSum}
10
IONODE
0
END

MESSAGE sum50
MSG:msgApplyFunction|5
{fSum}
50
IONODE
0
END
```

Figure 5-13. Rangesum Call File

Three messages have been defined for calling the sum function with the arguments 1, 10, and 50. `IONODE` is an integer constant predefined by MDPSim and denotes the address of the MDP serving as the I/O node between the J-Machine and the outside world. In MDPSim, the I/O node simply prints every message it receives.

```
MDPSim -x 2 -y 2 -msize 0x1000 ::Cosmos:Cosmos.m RangeSum.mdp RangeSum.calls

Message-Driven Processor Simulator
Version 7.0 Rev B
Accompanies MDP Architecture Document 11B
Written by Waldemar Horwat
Architecture Updates by Brian Totty and Jerry Larivee
UROPs for Bill Dally

4 MDPs present.

@0..3|watch fault all
@0..3|resetstats
@0..3|inject sum10@1
@0..3|run
Fault:  @ 1:       (faultXlate0)
Fault:  @ 1: {BBBW} $008B =                 DC      fltXLATE              ;XLATE
Fault:  @ 1:       (lookupBinding)
Fault:  @ 1: {BBBW} $00C6 =                 DC      fltLookupBinding      ;$06
Fault:  @ 1:       (enterBinding)
Fault:  @ 1: {BBBW} $00C5 =                 DC      fltEnterBinding       ;$05
Fault:  @ 2:       (blockSend)
Fault:  @ 2: {BBBW} $00C2 =                 DC      fltBlockSend          ;$02
Fault:  @ 2:       (faultLimit0)
Fault:  @ 2: {BBBW} $0088 =                 DC      fltLimit              ;LIMIT
Fault:  @ 1:       (allocObject)
Fault:  @ 1: {BBBW} $00C4 =                 DC      fltAllocObject        ;$04
Fault:  @ 1:       (lookupBinding)
Fault:  @ 1: {BBBW} $00C6 =                 DC      fltLookupBinding      ;$06
Fault:  @ 1:       (blockMove)
Fault:  @ 1: {BBBW} $00C1 =                 DC      fltBlockMove          ;$01
Fault:  @ 1:       (faultLimit0)
Fault:  @ 1: {BBBW} $0088 =                 DC      fltLimit              ;LIMIT
Fault:  @ 1:       (objectNode)
Fault:  @ 1: {BBBW} $00D3 =                 DC      fltObjectNode         ;$13
Fault:  @ 2:       (faultXlate0)
Fault:  @ 2: {BBBW} $008B =                 DC      fltXLATE              ;XLATE
Fault:  @ 2:       (lookupBinding)
Fault:  @ 2: {BBBW} $00C6 =                 DC      fltLookupBinding      ;$06
Fault:  @ 2:       (enterBinding)
Fault:  @ 2: {BBBW} $00C5 =                 DC      fltEnterBinding       ;$05
Fault:  @ 3:       (blockSend)
Fault:  @ 3: {BBBW} $00C2 =                 DC      fltBlockSend          ;$02
Fault:  @ 3:       (faultLimit0)
```

```
Fault:  @ 3: {BBBW} $0088 =                     DC      fltLimit                ;LIMIT
Fault:  @ 2:        (allocObject)
Fault:  @ 2: {BBBW} $00C4 =                     DC      fltAllocObject          ;$04
Fault:  @ 2:        (lookupBinding)
Fault:  @ 2: {BBBW} $00C6 =                     DC      fltLookupBinding        ;$06
Fault:  @ 2:        (blockMove)
Fault:  @ 2: {BBBW} $00C1 =                     DC      fltBlockMove            ;$01
Fault:  @ 2:        (faultLimit0)
Fault:  @ 2: {BBBW} $0088 =                     DC      fltLimit                ;LIMIT
Fault:  @ 2:        (faultXlate0)
Fault:  @ 2: {BBBW} $008B =                     DC      fltXLATE                ;XLATE
Fault:  @ 2:        (lookupBinding)
Fault:  @ 2: {BBBW} $00C6 =                     DC      fltLookupBinding        ;$06
Fault:  @ 2:        (faultXlate0)
Fault:  @ 2: {BBBW} $008B =                     DC      fltXLATE                ;XLATE
Fault:  @ 2:        (objectNode)
Fault:  @ 2: {BBBW} $00D3 =                     DC      fltObjectNode           ;$13
Fault:  @ 1:        (faultXlate0)
Fault:  @ 1: {BBBW} $008B =                     DC      fltXLATE                ;XLATE
Fault:  @ 2:        (objectNode)
Fault:  @ 2: {BBBW} $00D3 =                     DC      fltObjectNode           ;$13
Fault:  @ 1:        (lookupBinding)
Fault:  @ 1: {BBBW} $00C6 =                     DC      fltLookupBinding        ;$06
Fault:  @ 0:        (faultXlate0)
Fault:  @ 0: {BBBW} $008B =                     DC      fltXLATE                ;XLATE
Fault:  @ 2:        (faultCFut0)
Fault:  @ 2: {BBBW} $008D =                     DC      fltCFUT                 ;CFUT
Fault:  @ 0:        (lookupBinding)
Fault:  @ 0: {BBBW} $00C6 =                     DC      fltLookupBinding        ;$06
Fault:  @ 2:        (faultXlate0)
Fault:  @ 2: {BBBW} $008B =                     DC      fltXLATE                ;XLATE
Fault:  @ 1:        (enterBinding)
Fault:  @ 1: {BBBW} $00C5 =                     DC      fltEnterBinding         ;$05
Fault:  @ 3:        (blockSend)
Fault:  @ 3: {BBBW} $00C2 =                     DC      fltBlockSend            ;$02
Fault:  @ 2:        (lookupBinding)
Fault:  @ 2: {BBBW} $00C6 =                     DC      fltLookupBinding        ;$06
Fault:  @ 0:        (enterBinding)
Fault:  @ 0: {BBBW} $00C5 =                     DC      fltEnterBinding         ;$05
Fault:  @ 3:        (faultLimit0)
Fault:  @ 3: {BBBW} $0088 =                     DC      fltLimit                ;LIMIT
Fault:  @ 3:        (blockSend)
Fault:  @ 3: {BBBW} $00C2 =                     DC      fltBlockSend            ;$02
Fault:  @ 1:        (allocObject)
Fault:  @ 1: {BBBW} $00C4 =                     DC      fltAllocObject          ;$04
Fault:  @ 3:        (faultLimit0)
Fault:  @ 3: {BBBW} $0088 =                     DC      fltLimit                ;LIMIT
Fault:  @ 0:        (allocObject)
Fault:  @ 0: {BBBW} $00C4 =                     DC      fltAllocObject          ;$04
Fault:  @ 1:        (lookupBinding)
Fault:  @ 1: {BBBW} $00C6 =                     DC      fltLookupBinding        ;$06
Fault:  @ 0:        (lookupBinding)
Fault:  @ 0: {BBBW} $00C6 =                     DC      fltLookupBinding        ;$06
Fault:  @ 1:        (blockMove)
Fault:  @ 1: {BBBW} $00C1 =                     DC      fltBlockMove            ;$01
Fault:  @ 0:        (blockMove)
Fault:  @ 0: {BBBW} $00C1 =                     DC      fltBlockMove            ;$01
Fault:  @ 1:        (faultLimit0)
Fault:  @ 1: {BBBW} $0088 =                     DC      fltLimit                ;LIMIT
Fault:  @ 0:        (faultLimit0)
Fault:  @ 0: {BBBW} $0088 =                     DC      fltLimit                ;LIMIT
Fault:  @ 1:        (faultXlate0)
Fault:  @ 1: {BBBW} $008B =                     DC      fltXLATE                ;XLATE
Fault:  @ 0:        (objectNode)
Fault:  @ 0: {BBBW} $00D3 =                     DC      fltObjectNode           ;$13
Fault:  @ 1:        (objectNode)
Fault:  @ 1: {BBBW} $00D3 =                     DC      fltObjectNode           ;$13
Fault:  @ 0:        (objectNode)
Fault:  @ 0: {BBBW} $00D3 =                     DC      fltObjectNode           ;$13
Fault:  @ 1:        (objectNode)
Fault:  @ 1: {BBBW} $00D3 =                     DC      fltObjectNode           ;$13
Fault:  @ 3:        (objectNode)
Fault:  @ 3: {BBBW} $00D3 =                     DC      fltObjectNode           ;$13
Fault:  @ 2:        (faultXlate0)
Fault:  @ 2: {BBBW} $008B =                     DC      fltXLATE                ;XLATE
Fault:  @ 0:        (faultCFut0)
Fault:  @ 0: {BBBW} $008D =                     DC      fltCFUT                 ;CFUT
Fault:  @ 1:        (faultCFut0)
Fault:  @ 1: {BBBW} $008D =                     DC      fltCFUT                 ;CFUT
Fault:  @ 3:        (objectNode)
Fault:  @ 3: {BBBW} $00D3 =                     DC      fltObjectNode           ;$13
Fault:  @ 2:        (lookupBinding)
Fault:  @ 2: {BBBW} $00C6 =                     DC      fltLookupBinding        ;$06
Fault:  @ 3:        (faultCFut0)
Fault:  @ 3: {BBBW} $008D =                     DC      fltCFUT                 ;CFUT
Fault:  @ 0:        (objectNode)
Fault:  @ 0: {BBBW} $00D3 =                     DC      fltObjectNode           ;$13
Fault:  @ 1:        (objectNode)
Fault:  @ 1: {BBBW} $00D3 =                     DC      fltObjectNode           ;$13
Fault:  @ 0:        (objectNode)
Fault:  @ 0: {BBBW} $00D3 =                     DC      fltObjectNode           ;$13
Fault:  @ 1:        (objectNode)
```

```
Fault:  @ 1: (BBBW) $00D3 =                DC       fltObjectNode        ;$13
Fault:  @ 2:        (objectNode)
Fault:  @ 2: (BBBW) $00D3 =                DC       fltObjectNode        ;$13
Fault:  @ 0:        (faultCFut0)
Fault:  @ 0: (BBBW) $008D =                DC       fltCFUT              ;CFUT
Fault:  @ 1:        (faultCFut0)
Fault:  @ 1: (BBBW) $008D =                DC       fltCFUT              ;CFUT
Fault:  @ 3:        (objectNode)
Fault:  @ 3: (BBBW) $00D3 =                DC       fltObjectNode        ;$13
Fault:  @ 2:        (objectNode)
Fault:  @ 2: (BBBW) $00D3 =                DC       fltObjectNode        ;$13
Fault:  @ 3:        (objectNode)
Fault:  @ 3: (BBBW) $00D3 =                DC       fltObjectNode        ;$13
Fault:  @ 2:        (faultCFut0)
Fault:  @ 2: (BBBW) $008D =                DC       fltCFUT              ;CFUT
Fault:  @ 3:        (faultCFut0)
Fault:  @ 3: (BBBW) $008D =                DC       fltCFUT              ;CFUT
Fault:  @ 0:        (objectNode)
Fault:  @ 0: (BBBW) $00D3 =                DC       fltObjectNode        ;$13
Fault:  @ 0:        (objectNode)
Fault:  @ 0: (BBBW) $00D3 =                DC       fltObjectNode        ;$13
Fault:  @ 1:        (faultXlate0)
Fault:  @ 1: (BBBW) $008B =                DC       fltXLATE             ;XLATE
Fault:  @ 0:        (faultCFut0)
Fault:  @ 0: (BBBW) $008D =                DC       fltCFUT              ;CFUT
Fault:  @ 2:        (objectNode)
Fault:  @ 2: (BBBW) $00D3 =                DC       fltObjectNode        ;$13
Fault:  @ 2:        (objectNode)
Fault:  @ 2: (BBBW) $00D3 =                DC       fltObjectNode        ;$13
Fault:  @ 2:        (faultCFut0)
Fault:  @ 2: (BBBW) $008D =                DC       fltCFUT              ;CFUT
Fault:  @ 2:        (faultXlate0)
Fault:  @ 2: (BBBW) $008B =                DC       fltXLATE             ;XLATE
Fault:  @ 2:        (faultXlate0)
Fault:  @ 2: (BBBW) $008B =                DC       fltXLATE             ;XLATE
Fault:  @ 1:        (faultXlate0)
Fault:  @ 1: (BBBW) $008B =                DC       fltXLATE             ;XLATE
Fault:  @ 2:        (faultXlate0)
Fault:  @ 2: (BBBW) $008B =                DC       fltXLATE             ;XLATE
Tick  1543   Received priority 0 message:
    OBJ:$801D9804 u=1 f=0 offset=$00766=Reply length=$0004
    INT:$0000FC00 = 64512
    INT:$00000000 = 0
    INT:$00000037 = 55

@0..3)stats
1544 ticks executed.
Dynamic Instruction Usage:
    STOP:   2887  47.13%
    READ:    737  12.03%
   WRITE:    500   8.16%
   READR:    163   2.66%
    SEND:    160   2.61%
      DC:    143   2.33%
      BR:    130   2.12%
   XLATE:    123   2.01%
     ROT:    117   1.91%
     ADD:    104   1.70%
     AND:     98   1.60%
  WRITER:     88   1.44%
      BT:     71   1.16%
   SEND2:     70   1.14%
    BNIL:     69   1.13%
     NOP:     64   1.04%
      BF:     57   0.93%
    LDIP:     54   0.88%
     SUB:     52   0.85%
 SUSPEND:     50   0.82%
     XOR:     48   0.78%
    CALL:     48   0.78%
    WTAG:     44   0.72%
   LDIPR:     40   0.65%
      EQ:     37   0.60%
   SENDE:     26   0.42%
   EQUAL:     25   0.41%
  SEND2E:     24   0.39%
   CHECK:     22   0.36%
    RTAG:     21   0.34%
      OR:     15   0.24%
      GT:     14   0.23%
     ASH:     10   0.16%
   ENTER:      7   0.11%
      GE:      4   0.07%
   BNNIL:      4   0.07%
     NEG:      0   0.00%
     NOT:      0   0.00%
     FFB:      0   0.00%
   INVAL:      0   0.00%
   PROBE:      0   0.00%
     LSH:      0   0.00%
     NEQ:      0   0.00%
     MUL:      0   0.00%
```

```
     MULH:      0   0.00%
   NEQUAL:      0   0.00%
    CARRY:      0   0.00%
     HALT:      0   0.00%
       BZ:      0   0.00%
       LE:      0   0.00%
      BNZ:      0   0.00%
       LT:      0   0.00%

     STOP:   2887  47.13%
     Move:   1488  24.29%
      ALU:    407   6.64%
   Branch:    331   5.40%
  Network:    330   5.39%
    Field:    204   3.33%
       DC:    143   2.33%
    Fault:    142   2.32%
    Assoc:    130   2.12%
      NOP:     64   1.04%
    Other:      0   0.00%

  Foregnd:   3239  52.87%
    Total:   6126

Fault Usage:
          objectNode:    21  26.25%
        faultXlate0:    14  17.50%
      lookupBinding:    11  13.75%
         faultCFut0:    10  12.50%
        faultLimit0:     8  10.00%
          blockSend:     4   5.00%
        allocObject:     4   5.00%
       enterBinding:     4   5.00%
          blockMove:     4   5.00%

              Total:    80

The xlate hit ratio is 109 out of 123 ( 88.62%).

376 words sent in 51 messages on priority 0.
  Average message size:   7.37.
  16.29 instructions/word (8.61 foreground instructions/word)
  120.12 instructions/message (63.51 foreground instructions/message)
No priority 1 words sent.

@0..3}
```

Figure 5-14. MDPSim Transcript

This transcript shows a MDPSim session in which the user loads the rangesum assembly code and calls the sum function with the argument 10 on a 2x2x1-node J-Machine with COSMOS using only internal memory (`-msize 0x1000`). Since watching faults was enabled, MDPSim prints each fault encountered at each MDP as it is running. The fault message gives the number of the MDP on which the fault occurred, the number of the fault vector, and the name of the fault; the {BBBW} is additional MDPSim breakpoint and watchpoint information. Finally, after 1544 steps the answer 55 is produced and displayed.

The dynamic instruction statistics for the run are also shown. About half of the time is spent distributing the functions to all of the nodes; the second time sum is called with the argument 10, it only takes 893 ticks to produce the answer (a tick is the time it takes every node to execute one instruction; MDPSim assumes that every instruction runs in the same amount of time).

Chapter 6. Debugging

Optimist II, Cosmos, and the Concurrent Smalltalk applications are large programs, and debugging them is an important consideration. I will not discuss the process of debugging Optimist II itself; standard Common Lisp and CLOS techniques such as building firewalls and providing print routines for important data structures were used.

The primary approach to debugging MDP code I took is prevention. I made sure that the Cosmos design was sound before running it. The criticality criteria were very helpful in avoiding re-entrancy and double fault problems. Nevertheless, while the prevention approach was successful on Cosmos itself, it cannot be the sole debugging method used on the Concurrent Smalltalk programs. Instead, a combination of debugging means at various levels has been provided.

Debugging Concurrent Smalltalk Code

The first line of defense is the Optimist II compiler itself. The compiler will complain when it detects errors such as incorrect function argument counts or bad types, if types are declared.

The second line of defense is the interpreter in the Optimist II compiler. The interpreter can be used to run Concurrent Smalltalk programs before they are downloaded into MDPSim or onto a J-Machine. The interpreter provides nearly complete checking of Concurrent Smalltalk programs, so it should catch most of the remaining bugs. However, the interpreter will not catch bugs which occur only on large data sets, nor will it find Cosmos's or the Optimist II code generator's bugs.

Debugging MDP Code on MDPSim

Debugging becomes considerably more difficult once the code is in assembly language form. Fortunately, Cosmos does include some facilities for debugging Concurrent Smalltalk programs.

The third line of defense is comprised of the safety features built into the MDP architecture. Type and bounds checking were extremely valuable when debugging Cosmos, as they catch most common type errors when they happen and prevent runaway programs from doing too much damage to the machine state. Without these facilities debugging Cosmos and Concurrent Smalltalk programs could have been intractable.

The fourth line of defense consists of safety checks built into a number of critical places in Cosmos. These checks include:

- A check in the CFUT handler that distinguishes real cfutures from uninitialized variables, together with the initialization of memory and globals to values that will cause CFUT faults.

- Checks in the XLATE and INVADR handlers for references to primitive, nonexistent, or deleted objects. Without these checks, such references would generate messages that wander about the J-Machine forever.

- A check in the `Return` handler to make sure that the context was expecting the value that was returned. This check catches the extremely elusive bug of replying to the same continuation twice, as the second reply message may overwrite a variable in the context after it has been reallocated to a completely unrelated function. The bug will be caught eventually, even if the second function stores a cfuture into the same context location, because then there will still be two replies to the same context location, and the cycle will repeat itself. Of course, by the time the bug will be caught, the original evidence may be gone, but at least there will be some indication of a problem.

- A check in the `Co` routine for a reference to a nonexistent constituent of a distributed object.

- A HALT on any reference out of bounds of any object except in `BlockMove` and `BlockSend`.

- HALT instructions on any type or overflow faults that occur in the course of execution of Concurrent Smalltalk programs.

Furthermore, MDPSim does its part to make debugging easier. Once the operating system is loaded, memory used by the operating system code is read and write-protected (it may only be executed) to catch any runaway references to it. Since dereferencing `NIL` is a common mistake in the MDP's unchecked mode, physical memory locations 0 through 3 have been protected from all accesses to catch any routines that dereference nonexistent objects. Moreover, MDPSim immediately halts if a message is sent to a nonexistent node.

MDPSim includes the `HALT` instruction which is not present on the MDP. The `HALT` instruction immediately halts the simulated J-Machine without altering any state. However, the `HALT` instruction can almost be emulated on the J-Machine—executing `HALT` will cause either an `INVINST` or a `CATASTROPHE` fault, which can be intercepted.

Moreover, the newest MDPSim [25] includes *hazard detection*—MDPSim 7.0 will complain and optionally stop the program if it detects an unsafe programming construct such as referencing the FIR register if it could have been altered by an asynchronous interrupt or sending a message when the F bit is set (a network send fault could be catastrophic in this case). Clearly MDPSim cannot discover all such possible bugs, but it can provide considerable assistance in uncovering sporadic asynchronous bugs.

Finally, MDPSim is *deterministic*—running the same program twice will always yield identical results. Thus, if an inexplicable bug occurs, it can always be reproduced. Moreover, earlier snapshots in time can be examined by running the same session again in MDPSim. On the Macintosh version of MDPSim, the entire session is automatically saved, making reproducing it easy.

Debugging MDP Code on a J-Machine

Debugging code on a real J-Machine is still harder than debugging it with MDPSim. Cosmos currently does not include any facilities specifically designed for such debugging other than the ones described above, but such facilities are being added in the true J-Machine version of it. The primary facilities consist of a set of mousetraps to catch weird conditions such as hardware errors and a set of fault handlers that interact with the host through the diagnostic port. Unfortunately, it is impossible to examine an MDP's state without destroying some register values, so debugging on the hardware is much harder.

Assuming one can stop the computation at a safe point, it is possible to get a dump of all memory and most registers on each MDP in a J-Machine. What does one do with a huge dump of the state of a J-Machine? One possible course of action would be to examine it using MDPSim's debugging facilities. Another possibility is periodically checkpointing the computation on the J-Machine by saving images. If a crash occurs, earlier images can be examined or restarted to determine the cause of the crash.

Summary

Debugging Concurrent Smalltalk code, while not especially easy, is not impossible. Several lines of defense against bugs in Concurrent Smalltalk programs are provided. It is highly recommended to try to find bugs in the earlier steps of the compilation process because the tools at those levels are more robust and informative (but not as faithful to the J-Machine).

Although Cosmos includes many checks for the common Concurrent Smalltalk programming errors, Cosmos does not protect itself from itself—it does not detect corruption in its data structures. Fortunately, segmentation by the MDP ensures that those data structures could only be corrupted by Cosmos itself, as well-compiled Concurrent Smalltalk programs cannot reference data outside their segments. Cosmos was mainly debugged by design, with only minor debugging necessary once the operating system was written.

MDPSim also helps in debugging MDP code by providing watchpoints, breakpoints, the HALT instruction, hazard detection, and determinism, which allows any bug to be reproduced.

Chapter 7. Performance Measurements

Both Cosmos and the code output by Optimist II were optimized for speed. This chapter presents some measurements that determine just how fast compiled Concurrent Smalltalk runs on a J-Machine. Both theoretical derivations and real measurements are presented and compared. Both calculations indicate that the average grain size (the ratio of useful instructions executed to messages sent) for running Concurrent Smalltalk on a J-Machine is between 50 and 70 instructions, and the average number of instructions executed per method is about 100 instructions. This is a pity if the average method only performs a few instructions' worth of real computation, yet, since Cosmos and the code output by Optimist II are already heavily optimized, it does not seem likely that incremental changes will reduce these numbers much further.

In addition to the above figures, various other statistics are presented. The static and dynamic instruction use frequencies were collected to identify areas in which the MDP's hardware performance could be improved; no major surprises were found there. These frequencies indicate that the MDP spends an average of about 2 cycles per instruction; this number increases to 4 if slow external DRAM is used to hold the user program and data.

Finally, the network load is analyzed. The network should not become saturated until more than 343 MDPs are put together; if a larger J-Machine is to be built, either the network will have to be made faster, the operating system slower, or considerable attention will have to be paid to locality.

7.1. Derived Times

This section presents some rough estimates of the overhead on the J-Machine. A number of assumptions are made when making these estimates; the results of actual measurements will be reported in the next section to verify those assumptions.

Cosmos Estimates

The instruction counts needed for various important Cosmos services are shown in Table 7-1. The counts are approximate, but usually accurate to within a few instructions. The counts listed may not be completely correct due to approximations in some routines.

Table 7-1. Selected Cosmos Routine Instruction Counts

Routine	Instruction Count	Description
Method and Control Managers		
Apply	3+ApplySelector or 5+ApplyFunction	Dispatch a general Apply message.
ApplyFunction	4	Dispatch an ApplyFunction message.
ApplySelector	≥23 (≥15+LookupMethodU)	Dispatch an ApplySelector message.
LookupMethod	8+LookupMethodU	Lookup a method given a class and a selector.
LookupMethodU	8 on cache hit, 40+SaveStateID023+message latency on cache miss.	Internal core of LookupMethod.
CFUT Fault	≈30+2msize if context available on queue (14+SaveStateID023)	Save state when a cfuture was read from the context.
Reply	27 if process is restarted; 12 if not.	Process a reply message.
RestartContext	20	Unconditionally restart a context.
Context Manager		
SaveStateID023	14 if message already saved in context and new context available in queue 16+2msize if context available on queue; 17+2msize+AllocNextObject otherwise.	Save the ID registers and the message in the context, save the context, and suspend.
Global Object Manager		
NewObject	≈37+2msize+Reply (21+SaveStateID023+Reply)	Allocate a remote object.
ClassOf	15 to 25 (10+TypeOf)	Return the class of an object.
TypeOf	5 to 15, depending on tag (5 for integers, 11 for ordinary user objects, varies for others)	Internal core of ClassOf.
ObjectNode	9 for primitive objects; 4 for ordinary user objects; 32 for distributed objects.	Return the node most likely to contain the object or a random node if the object is primitive.
Co	38 (49 when the object has more constituents than there are nodes)	Return the ID of the *n*th constituent of a distributed object.

PreferredConst	27 (12 when the object has more constituents than there are nodes)	Return the ID of a nearby constituent of a distributed object.
MigrateObject	≈62+AllocObject+LookupBinding+ 2size (may vary if more or fewer contexts are restarted)	Receive and install an object and restart a context waiting for it.
UpdateHome	≥11+LookupBinding	Update a migrated object's home BRAT entry.
Unlock	9	Unlock an object.
Local Object Manager		
NewLocalObject	3+AllocNextObject	Allocate a local object of the given class.
AllocNextObject	12+AllocObject+EnterBinding	Allocate a local object using the next ID and the given header word.
DeallocateObject	11+PurgeBinding	Deallocate a local, unlocked object.
BRAT Manager		
EnterBinding	26 (35 if no free BRAT entries were available; may also compact heap)	Allocate a new BRAT binding.
LookupBinding	14+5*n*, where *n* is the number of links traversed in linked list.	Lookup a binding in the BRAT.
PurgeBinding	2+DeleteBinding	Delete a binding from the BRAT and the XLATE table.
DeleteBinding	23+5*n*, where *n* is the number of links traversed in linked list.	Delete a binding from the BRAT.
Heap Manager		
AllocObject	20; may also compact heap	Allocate an object on the heap.
CompactHeap	varies from 2*N* to 10*N* or more, where *N* is the size of the heap.	Compact the heap.
Utilities		
Divide	from 40 for small numbers to 400 for large numbers.	Divide two 32-bit numbers and return the quotient and remainder.
Faults		
Early Fault	8	Penalty for reading data from message queue too fast.
Send Fault	8	Penalty for sending data into network too fast.

Some Definitions:

size is the size of the object.

msize is the size of the message in the queue. If the message has already been saved and the Q flag is false, msize is defined to be -1 for the purposes of the above timings. If msize is mentioned in a time expression, the current process is suspended and later restarted; the time does not include the time between the suspension and the resumption because other processes are assumed to execute then.

User Program Estimates

In contrast to the counts in Table 7-1, an examination of the rangesum method in Figure 5-12 shows that it takes about 13 instructions[1] to execute a function or method call and about 8 instructions to return a reply and suspend (see Table 7-2). Thus, the typical time the MDPs spend in user code to execute a function call and return is about 21 instructions; perhaps a few more instructions are used for primitives, but the user code execution time is seldom more than 30 instructions per function invocation. Hence, estimating conservatively, any

---

[1] There are several NOPs not shown in the listing caused by alignment around DCs.

Table 7-2. Selected User Action Instruction Counts

Action	Instruction Count	Description
Function or Method Call	≈11+nargs. May be higher if arguments must be touched or lower if many SEND2s are used.	Call a function or a method. The time does not include the CFUT fault or reply time.
Reply with Suspend	8–10	Return a reply to the caller.
Primitive	1–4 for instructions and up to 400 or more for system calls.	Perform a primitive operation such as an addition or a conditional.

Nargs is the number of arguments sent in the application message.

time above 30 instructions per function or method invocation is spent in the operating system[1].

Analysis

A juxtaposition of the main figures from Tables 7-1 and 7-2 reveals that a typical program will spend about 70% of its active time in the operating system and 30% of the time in user code. Furthermore, the program will take about 100 instructions per function invoked, except for tail-forwarded functions which will only take about 25 instructions each. About 20 extra instructions should be added for each method dispatch that the compiler is unable to optimize out. To derive these estimates the following system of accounting is used: the work ascribed to a function invocation consists of all work needed to call the function on the originating node plus all work needed to dispatch the function on the called node, but not including the work done by the called function to call other functions.

Standard Invocations

Each non-tail-forwarded function invocation requires the processing of an `ObjectNode` call, a function message send, a reply message, and optionally a cfuture fault on the originating node, and a function dispatch and a reply on the called node. Assuming that the average function call has two arguments, the total operating system work for the above activity is:

`ObjectNode` + `ApplyFunction`
+ c(CFUT fault + restarting Reply) + (1-c)(non-restarting Reply)
= $9 + 4 + 69c + 12(1-c)$
= $25 + 57c$ instructions.

c is the probability that a cfuture will be referenced before being replaced by the returned value. This probability can vary over a wide range depending on the branching factor of the program call graph. c is 1.0 for a recursive factorial program and 0.5 for a recursive fibonacci or rangesum program. If a branching factor between 1 and 2 is assumed, c will be somewhere between 0.5 and 1.0; suppose it is 0.75, which results in 68 instructions executed in the operating system per function invocation.

The total user code work is
Function call + Primitives called by function + Reply with Suspend.
The time spent executing primitives will vary greatly depending on the application; 10 instruction seems reasonable for most cases, although it will be higher if the user program calls `Divide` or allocates objects. Substituting this number and the average number of arguments yields a total user code work of

---

[1] Tail-forwarded calls are cheaper because the net cost of a tail-forwarded call is one call and no return, which is about 15 user code instructions.

13 + 10 + 9 = 32 instructions.

Thus, the total amount of work taken to process one function invocation is 100 instructions, out of which about 10 instructions (the primitives) could be construed as being "useful" work and the rest overhead. This figure does not include any object migration or XLATE miss overhead. These results should not be interpreted as implying that an MDP running Cosmos has a performance 10 times slower than a comparable processor in a sequential computer because sequential computers also have a considerable function calling and parameter passing overhead.

Tail-Forwarded Invocations

Tail-forwarded applications are considerably more efficient. Using the accounting method outlined above results in ascribing

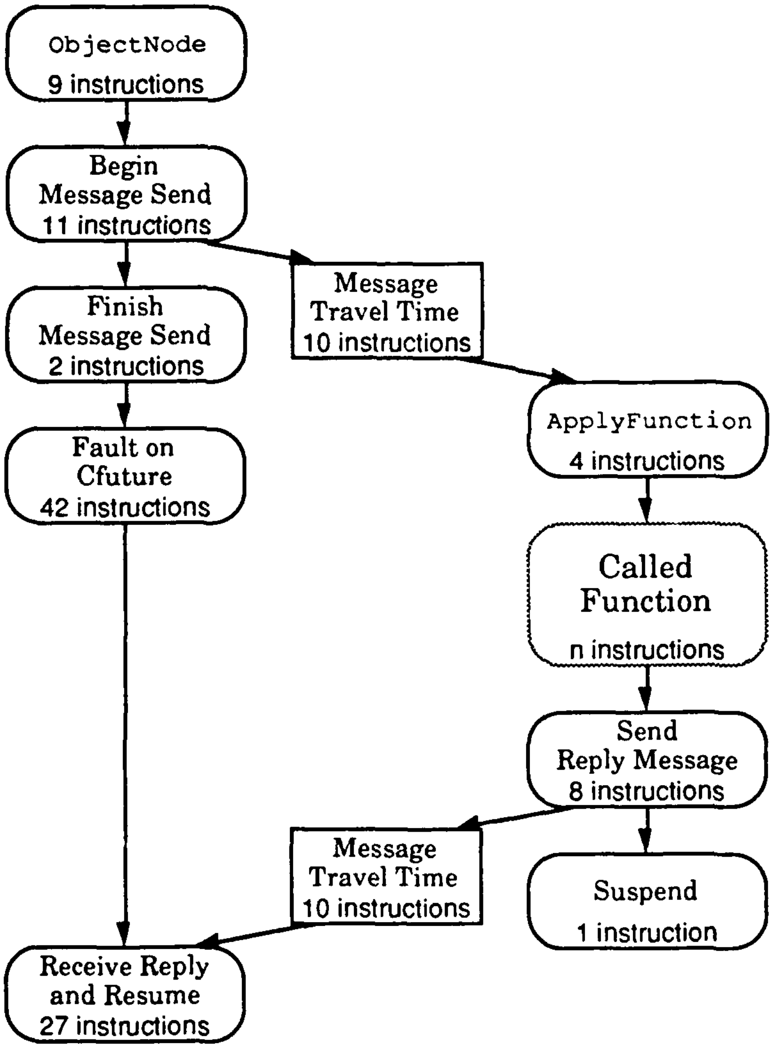


Figure 7-1. Function Invocation Latency

The latency of the network is estimated at about 10 instruction times (20 cycles) to send a message between two randomly chosen nodes on a 4096-node machine.

If *n* is the time taken by the called function, the latency of invoking a function is 9+11+10+4+8+10+27+*n* = 79+*n* instructions unless the called function takes fewer than 12 instructions, in which case the latency is 9+11+2+42+27 = 91 instructions.

`ObjectNode` + `ApplyFunction` = 13 instructions

operating system overhead and

Function call + Primitives called by function = 23 instructions

user code work. The total work done is 36 instructions, out of which again 10 instructions is "useful" work.

Latency

The preceding analysis calculated the total amount of work needed per function invocation in a program, which determines throughput on a fully loaded system in which each processor is busy; however, another important component of performance is latency. It turns out that the latency of a function invocation can be lower than the amount of work done by the function invocation because two processors (the caller and the callee) can execute much of the function invocation in parallel.

Assuming no other activity in the system, a non-tail-forwarded function invocation will consist of the caller sending a message to the callee. Then the callee evaluates the function, while the caller takes a cfuture fault (or calls another function, but this won't matter). Unless the called function is very short, the caller will finish the cfuture fault processing and then idle before it gets the reply message from the callee. Finally, the callee replies to the caller, which restarts the calling process.

As can be seen in Figure 7-1, the latency of a function call is 79 instructions in addition to the time taken to execute the function; if the function takes fewer than 12 instructions to execute, the overall latency is 91 instructions. These numbers are less than the total amount of work done by the system (104 instructions).

Summary

The results above indicate that the number of instructions needed to process a function invocation for Cosmos running on a J-Machine should be about 100 instructions, with the notable exception of tail-forwarded functions, which require only about 36 instructions. The instruction counts may be higher if many primitive calls are made or if the operating system faults often.

7.2. Measurements

Grain Size and Machine Load

To attempt to measure the J-Machine's performance and grain size, I ran several programs, including factorial **(Figure 7-2);** rangesum **as listed in Chapter 5;** rangesum2 **(Figure 7-3), which is a version of** rangesum **which builds and traverses a data structure; and** sort **(Figure 7-4), which generates and sorts an array of *n* pseudo-random numbers using the Batcher parallel sort technique described on page 112 of [28].**

```
(defun fact (n)
  (if (zero? n)
    1
    (* n (fact (- n 1)))))
```

Figure 7-2. Factorial Program

```
(defclass pair (object)
  car
  cdr)

(defun cons (x y):pair
  (put-car-cdr (new pair) x y))

(defmethod put-car-cdr pair (x y):pair
  (cset car x)
  (cset cdr y)
  self)

(defun make-countlist (low:integer high:integer)
  (if (> low high) (halt))
  (if (= low high)
    low
    (let ((middle (// (+ low high) 2)))
      (cons (make-countlist low middle)
            (make-countlist (+ middle 1) high)))))

(defmethod reduce pair (op:funct)
  (op (reduce car op) (reduce cdr op)))

(defmethod reduce integer (op:funct)
  self)

(defun add (x y)
  (+ x y))

(defun reduce-add (tree)
  (reduce tree add))

(defmethod ramp integer ()
  (make-countlist 0 self))

(defmethod rangesum2 integer ()
  (reduce-add (ramp self)))
```

Figure 7-3. Rangesum2 Program

This program exercises several Concurrent Smalltalk object facilities such as allocating objects and traversing trees.

```
(defclass distarray (distobj)
  value)

(defmethod initialize distarray (low,high:integer f:funct)
  (if (= low high)
    (cset (get-value (co group low)' (f low))
    (clet ((middle (// (+ low high) 2)))
      (concurrently
       (initialize group low middle f)
       (initialize group (+ middle 1) high f)))))

(defun make-distarray (n modulus)
  (clet ((da (new distarray n)))
    (initialize da 0 (- n 1) (lambda (x) (mod (* x x x) modulus)))
    da))


(defmethod sort-exchanges distarray (low,high,p,r,d:integer)
  (if (<= low high)
    (if (= low high)
      (clet ((low2 (+ low d)))
        (clet ((v1 (get-value (co group low)))
               (v2 (get-value (co group low2))))
          (if (> v1 v2)
            (concurrently
             (cset (get-value (co group low)) v2)
             (cset (get-value (co group low2)) v1)))))
      (clet ((middle (// (+ low high) 2)))
        (concurrently
         (sort-exchanges group low middle p r d)
         (sort-exchanges group (+ middle 1) high p r d))))))

(defmethod sort-q distarray (p,q,r,d:integer)
  (sort-exchanges group 0 (- (logical-limit self) (+ d 1)) p r d)
  (if (<> p q)
    (sort-q group p (// q 2) p (- q p))))

(defmethod sort-p distarray (half,p:integer)
  (sort-q group p half 0 p)
  (if (> p 1)
    (sort-p group half (// p 2))
    group))

(defmethod sort distarray ()
  (clet ((half (ash 1 (- (integer-length (- (logical-limit self) 1)) 1))))
    (sort-p group half half)))


(defun sort-distarray (n modulus)
  (sort (make-distarray n modulus)))
```

Figure 7-4. Sort Program

Sort-distarray, given the values of *n* and *modulus*, sorts an array of *n* pseudo-random numbers. The *i*th pseudo-random number is equal to i^3mod *modulus*. The Batcher sort algorithm is used, as presented on page 112 of [28].

Measurements were done on a 4-node and a 16-node simulated J-Machine. The results of the trials are summarized in Table 7-3.

The grain size is the third number in the working instructions executed column. The time to process one function invocation is approximately twice the grain size unless tail-forwarding is used extensively. Except for sorting 4 numbers and the trivial factorial case, the results indicate function invocation times of between 81 and 162 instructions, which means that the estimate of 100 in the previous section was about right. Many of the functions in the sort sample program are tail-forwarded, so the average function invocation time for that example is less than twice the grain size. In addition, the sort program has a grain size higher than predicted in the previous section. This is probably due to frequent calls to the multiplication, division[1], and co primitives as well as to distribution of large code objects; the grain size does decrease for larger input values.

---

[1] A division by 2 is just a single ASH instruction, but the division in make-distarray requires a complete Divide call.

Table 7-3. Performance Measurements

Program	# MDPs	Input	Invocations	Start-up	Total Instructions Executed			Working Instructions Executed			% Busy	Net Wds Sent	Net Msgs Sent	Avg Msg Size*
factorial	4	0	1	cold	95	8.64	47.50	17	1.55	8.50	18	11	2	5.50
	2×2	10	11	cold	5949	30.82	212.46	2001	10.37	71.46	34	193	28	6.89
				warm	3407	28.16	154.86	1078	8.91	49.00	32	121	22	5.50
rangesum	4	10	21	cold	6985	18.43	134.33	3364	8.88	64.69	48	379	52	7.29
	2×2			warm	3122	12.10	72.60	1737	6.73	40.40	56	258	43	6.00
		50	101	cold	17017	12.71	80.27	11585	8.65	54.65	68	1339	212	6.32
				warm	11998	9.85	59.10	9395	7.71	46.28	78	1218	203	6.00
				hot	10982	9.02	54.10	8841	7.26	43.55	81	1218	203	6.00
rangesum2	4	10	21	cold	15365	24.08	174.60	6194	9.71	70.39	40	638	88	7.25
	2×2			warm	5470	14.47	86.83	3067	8.11	48.68	56	378	63	6.00
		50	101	cold	27971	13.40	84.76	19559	9.37	59.27	70	2088	330	6.33
				warm	23232	12.57	75.18	16401	8.88	53.08	71	1848	309	5.98
				hot	21418	11.78	70.69	15767	8.67	52.04	74	1818	303	6.00
sort	4	4		cold	57939	30.27	298.65	23982	12.53	123.62	41	1914	194	9.87
	2×2			warm	35168	36.79	256.70	14655	15.33	106.97	42	956	137	6.98
		29		cold	351144	14.85	101.22	269019	11.38	77.55	77	23647	3469	6.82
				warm	289336	12.94	86.55	232974	10.42	69.69	81	22361	3343	6.69
	16	4		cold	201681	95.90	979.03	24026	11.42	116.63	12	2103	206	10.21
	4×4	29		cold	868586	32.56	238.56	295483	11.08	81.15	34	26679	3641	7.33
		100		cold	2612981	18.61	126.35	1469377	10.46	71.05	56	140436	20680	6.79

*The average message length includes the address word sent at the beginning of each message. That word is kept as the message is routed through the network but removed before the message is inserted into the queue on the destination node.

The working instruction counts are instruction counts with all STOP instructions executed in background loops removed; they represent the useful work done in the system.

The three numbers in the total instructions executed and working instructions executed columns give the absolute numbers of instructions executed, the numbers of instructions per word of network traffic, and the number of instructions per network message, in that order.

A cold startup indicates that the program was executed just after it was loaded; a considerable portion of the running time is spent on distributing the functions to all nodes that need them.

A warm startup indicates that the program was executed after the functions it needed were already installed on every node.

A hot startup indicates the third trial of the program on the particular input. This time may be less than the warm startup time because the previous trials have preallocated enough standard contexts on the MDPs to let the program run without the need to allocate any more contexts. Warm and hot startup times are probably the most representative of the J-Machine's performance on larger problems.

The geometry of the J-Machine does not have much of an effect on a program simulated under MDPSim. It is unimportant anyway for the small sizes simulated above.

Using inputs much larger than 50 for the range sums or 100 for the sort generated too much concurrency and caused the message queues to overflow. See Chapter 8 for a possible solution to this problem.

Another pattern in Table 7-3 is that the percentage of the J-Machine that is busy is higher for the larger problems, which was to be expected. Also, the warm and hot start programs tended to exhibit more concurrency than the cold start ones[1]; apparently there is some wasted time during the initial code distribution phase.

---

[1] This was not the case in a slightly earlier version of Cosmos, possibly because it was less efficient and therefore had more work to do.

Comparison with Dataflow

Ellen Spertus made a few performance numbers available for her implementation of dataflow on the J-Machine [34]. I compared her timings with those obtained by Optimist II/Cosmos on the same examples. The program used was the factorial function listed in Figure 7-5.

The dataflow interpreter took 431 steps to compute the factorial of 4. The Concurrent Smalltalk version of the factorial program took 725 steps to execute from a cold start but only 265 steps from a hot start. The dataflow interpreter allocates code statically and references absolute addresses, so every timing is effectively a hot start. The dataflow interpreter took 628 steps to compute three factorials of 4 in parallel, while the Concurrent Smalltalk code took 399 steps to complete the task. Thus, for this simple example the Concurrent Smalltalk/Optimist II/Cosmos combination is faster than dataflow, but not by much. However, Concurrent Smalltalk is more dynamic than the current dataflow system in [34].

```
(defun fact (n)
  (if (<= n 1)
    1
    (* n (fact (- n 1)))))
```

Figure 7-5. Factorial Program used in Dataflow

Network Load

As seen in Table 7-3, the network loading is usually between one word every 8 instructions and one word every 20 instructions, with the earlier figure dominating as the J-Machine utilization approaches 100%. If an average MDP instruction length is taken to be 2.0 cycles, this implies that a program could inject words into the network as fast as one word every 16 cycles on every MDP.

Suppose that we run one of the above programs on a J-Machine organized as a $k \times k \times k$ mesh. Let $N = k \times k \times k$ be the number of nodes. To a first-order approximation, the capacity of the network is $3N$ half-word-hops/cycle[1], or $1.5N$ word-hops/cycle. Assuming random sources and destinations, a message will have to travel an average of $k/3$ nodes on each of the three dimensions, so the expected distance the message has to travel is $3k/3 = k$ nodes. Hence, the network's theoretical capacity is the delivery of $1.5N/k = 1.5k^2$ words per cycle. On the other hand, the program offers $N/16$ words/cycle to the network, which means that unless locality is exploited or the program slowed down, there will be an upper bound on the size of the J-Machine which can run Cosmos.

A mesh loaded at about 30% of its theoretical capacity should be able to route messages without excessive delays [32]. To calculate the maximum k, set

$$0.3 \times 1.5k^2 = k^3/16$$

$$k = 7.2.$$

Thus, the network should not become a critical resource until a J-Machine with over $7^3 = 343$ nodes is built. If the network routing speed is doubled, network loading should not be problematic until the J-Machine exceeds $14^3 = 2744$ nodes. On the other hand, should the Cosmos operating system be sped up somehow, the critical size might fall below 343 nodes. Serious attention to locality will have to be paid if a J-Machine larger than a few hundred nodes is built; conversely, if only a small J-Machine is built, it may not be adequate for testing algorithms for exploiting locality because almost any algorithm will work.

---

[1] The J-Machine network can transmit half a word between every pair of adjacent MDPs on every cycle.

Table 7-4. Static Instruction Frequencies

Instruction	Count	Freq.	Instruction	Count	Freq.
DC	440	19.33%	XOR	13	0.57%
READ	324	14.24%	EQUAL	11	0.48%
WRITE	210	9.23%	ENTER	10	0.44%
NOP	173	7.60%	SENDE	10	0.44%
WRITER	104	4.57%	SEND2E	10	0.44%
READR	88	3.87%	NEG	9	0.40%
BR	86	3.78%	BZ	9	0.40%
SEND	80	3.51%	BNZ	9	0.40%
ROT	64	2.81%	PROBE	7	0.31%
HALT	64	2.81%	EQ	7	0.31%
ADD	59	2.59%	LT	5	0.22%
AND	50	2.20%	GT	5	0.22%
BT	46	2.02%	NOT	4	0.18%
CALL	46	2.02%	GE	4	0.18%
BF	42	1.85%	BNNIL	4	0.18%
SUB	39	1.71%	FFB	4	0.18%
CHECK	32	1.41%	LSH	3	0.13%
OR	29	1.27%	RTAG	3	0.13%
XLATE	29	1.27%	NEQUAL	1	0.04%
BNIL	25	1.10%	STOP	1	0.04%
LDIPR	23	1.01%	INVAL	1	0.04%
SEND2	22	0.97%	LE	1	0.04%
LDIP	21	0.92%	MUL	0	0.00%
SUSPEND	20	0.88%	MULH	0	0.00%
ASH	15	0.66%	CARRY	0	0.00%
WTAG	14	0.62%	NEQ	0	0.00%

Instruction	Count	Freq.	Instruction	Count	Freq.
Move	726	31.90%	Bit Field	116	5.10%
DC	440	19.33%	Fault	90	3.95%
ALU	256	11.25%	Other	64	2.81%
Branch	221	9.71%	Assoc. Table	47	2.07%
NOP	173	7.60%	STOP	1	0.04%
Network	142	6.24%			
Total	2276				

The above table includes the static instruction frequencies in the Cosmos kernel and the MDP runtime system. The second table categorizes the instructions according to their kinds. Each DC is counted twice because it occupies as much space as two normal instructions. 173 NOPs had to be inserted to align instructions to word boundaries around DCs and at branch entry points.

Instruction Frequencies

I collected data on the frequencies of various MDP instructions to provide another estimate of what the MDP is doing most of the time. Table 7-4 shows a histogram of the static instruction use in Cosmos and the MDP runtime routines, while Table 7-5 shows dynamic instruction use in the cold-start sort trial running on 16 MDPs on an input value of 100. Combined with the results from Table 7-6, which show the memory reference frequencies, these tables contain enough information to deduce the approximate[1] number of cycles taken per MDP instruction.

As shown in Table 7-6, a 16-MDP J-Machine will achieve somewhere between 1.87 and 3.48 cycles per working instruction when running the sort program on an input of 100. The internal-memory-only cycles-per-working-instruction number varied between 1.8 and 2.0 for other trials, while the external-memory cycles-per-working-instruction number varied between 3.0 and 3.9.

Table 7-5. Dynamic Instruction Frequencies

Instruction	Count	Freq.
STOP	1143604	43.77%
READ	309073	11.83%
WRITE	169577	6.49%
ROT	78272	3.00%
READR	76641	2.93%
AND	71150	2.72%
XLATE	67230	2.57%
DC	63981	2.45%
BF	53595	2.05%
SEND	47474	1.82%
ASH	43161	1.65%
OR	41948	1.61%
BR	40800	1.56%
WRITER	39085	1.50%
SEND2	30783	1.18%
ADD	26468	1.01%
NOP	23633	0.90%
SUB	21840	0.84%
BNIL	20786	0.80%
SUSPEND	20679	0.79%
WTAG	20406	0.78%
BT	19497	0.75%
LDIP	19032	0.73%
CALL	18557	0.71%
LDIPR	17712	0.68%
GT	15714	0.60%
CHECK	14401	0.55%
EQ	14293	0.55%
LT	11816	0.45%
SEND2E	11220	0.43%
NEG	10886	0.42%
RTAG	9859	0.38%
SENDE	9594	0.37%
XOR	5637	0.22%
FFB	5624	0.22%
NOT	5444	0.21%
EQUAL	4725	0.18%
LE	4296	0.16%
ENTER	1117	0.04%
BNZ	886	0.03%
LSH	786	0.03%
GE	502	0.02%
BZ	475	0.02%
BNNIL	420	0.02%
MUL	200	0.01%
PROBE	74	0.00%
NEQUAL	28	0.00%
NEQ	0	0.00%
MULH	0	0.00%
CARRY	0	0.00%
HALT	0	0.00%
INVAL	0	0.00%

Instruction	Count	Freq.
STOP	1143604	43.77%
Move	594376	22.75%
ALU	283732	10.86%
Branch	136459	5.22%
Bit Field	123724	4.73%
Network	119750	4.58%
Assoc. Table	68421	2.62%
DC	63981	2.45%
Fault	55301	2.12%
NOP	23633	0.90%
Other	0	0.00%
Foreground	1469377	56.23%
Total	2612981	

This particular problem (Sort creating and sorting an array of 100 numbers on 16 MDPs) only kept an average of 56% of the MDPs busy at a time—about 44% of the instructions executed are STOP. Although the frequency of the STOP instruction varies widely, the relative frequencies of the other instructions are typical for an MDP program.

---

[1] Some of the instruction row buffer dynamics were simplified and all branches were assumed to take 3 cycles, even though sometimes they may take fewer cycles.

Table 7-6. Memory Access Frequencies

Operating System memory usage:
Reads: 394430 (0.15/instruction, 0.27/working instruction)
Writes: 152756 (0.06/instruction, 0.10/working instruction)
Fetches: 2295682 (0.88/instruction, 1.56/working instruction)

Heap memory usage:
Reads: 152262 (0.06/instruction, 0.10/working instruction)
Writes: 138807 (0.05/instruction, 0.09/working instruction)
Fetches: 317299 (0.12/instruction, 0.22/working instruction)

Total memory usage:
Reads: 546692 (0.21/instruction, 0.37/working instruction)
Writes: 291563 (0.11/instruction, 0.20/working instruction)
Fetches: 2612981 (1.00/instruction, 1.78/working instruction)

3.48 cycles/working instruction
1.87 cycles/working instruction without external RAM

The numbers above indicate the number of memory references (reads, writes, and fetches) done to the operating system (everything except the heap) and heap areas of memory by Sort running on 16 MDPs with an input of 100. The numbers for the other sample programs are similar. The cycles per instruction figures were calculated by adding the instruction frequencies from Table 7-5 weighted by the instruction times together with the memory usage frequencies weighted by memory access times.

The 4096-word internal memory contains all of the operating system data and code and a small portion of the heap (about 2100 words). The rest of the heap (65536 words) lies in slow external memory. When running on a real J-Machine, the sort program will achieve somewhere between 1.87 and 3.48 cycles per working instruction depending on how much of the program and data resides in the internal memory portion of the heap.

Considering that internal memory read, write, and fetch times average 1, 0, and 1/8 cycles[1], respectively, while external memory read, write, and fetch times are 6, 5, and 3 cycles[2], respectively, a loss of only a factor of two in performance by placing the user program and data in external memory is surprisingly low. The reason for such a low cycles-per-working-instruction figure when the user program and data are in external memory is the high Cosmos overhead. The MDP spends most of its time executing Cosmos code, which decreases the cycles-per-working-instruction number from what it would otherwise have been. For the same reason, changes that would reduce Cosmos overhead at the expense of user program size are undesirable in most cases.

---

[1] The write time is 0 because it is absorbed by the execution of the `WRITE` instruction—`WRITE` does not require any extra cycles when writing to memory as opposed to a register. Eight instructions can be fetched in one cycle for an effective fetch time of 1/8 cycle per instruction; the branch instruction cycle counts already include the overhead for fetching the next set of instructions.

[2] Two instructions are fetched at a time from external memory in 6 cycles, for an effective fetch time of 3 cycles per instruction.

7.3. Conclusion

Context Switching Performance

A large component of the current operating system overhead time is the time taken to save and restore contexts, especially in the CFUT fault handler. One possibility to increase the speed of the CFUT fault handler is to not save data registers and not copy the message upon a CFUT fault [11]. Not saving data registers would reduce the fault handler's time by 4 instructions[1], while not copying the message would reduce it by 6 more instructions. However, these gains would come at a price—the size of the object code would increase because the compiler could not effectively allocate variables to registers; it is not clear whether the savings in the operating system overhead would outweigh the increased time spent executing user code, especially if the user code lies in external DRAM, while the operating system lies in fast internal SRAM.

Summary

Both the derived and measured data indicate that the grain size for running Concurrent Smalltalk on the J-Machine is 50 to 70 instructions. Since most functions involve two messages (one apply message and one return message), the average number of instructions needed to process a function call is between 100 and 140; actually, it is probably closer to 100 because of tail forwarding.

When running entirely from internal memory, the MDP executes one instruction about every two cycles; if user programs and data have to be accessed from external memory, that count increases to about four cycles per instruction. The network load was calculated assuming a fast program (two cycles per instruction) injecting messages into the network at the fastest observed rate (one word every eight instructions) and utilizing 100% of the J-Machine's processors. If the messages are sent randomly under the above conditions, the J-Machine network will saturate when a J-Machine with over 343 MDPs is built. Of course, most programs will not be as fast, but some crafted library routines could impose network loads as high as indicated above To prevent network saturation, either the network will have to be made faster, the program slower, or some means of exploiting locality invented.

---

[1] The reduction would be 8 instructions if the data registers did not have to be restored by the reply handler; however, it is difficult for the reply handler to distinguish the cases in which it has to restore registers because some unanticipated fault like overflow happened from the cases in which it doesn't; the extra instructions needed to make this decision would make this optimization not worthwhile.

Chapter 8. Future Evolution

Although working Concurrent Smalltalk programs have been demonstrated, the Concurrent Smalltalk programming system is by no means complete. Some suggestions for improvements were discussed throughout the previous chapters—more optimizations could be added to the compiler, distributed objects could be distributed more uniformly, and storage used by free BRAT entries and free standard contexts could be placed back into the heap's free storage pool.

Nevertheless, the possible modifications are by no means limited to the minor ones listed there. The Concurrent Smalltalk programming system is still an evolving research and demonstration vehicle, and many issues still have to be addressed before it becomes a truly general-purpose system. This chapter lists these issues together with potential approaches for addressing them.

The first section lists features that were left out of the Concurrent Smalltalk implementation that are desirable in a full system. These features are useful in many specialized applications, but the system can work without them.

The second section lists the resource management concerns raised by the implementation of Cosmos. These concerns include load balancing, garbage collection, name space reuse, fanout bottlenecks, and parallelism control. A few ideas are suggested about handling the fanout bottleneck and parallelism control problems, but many of these issues are still in the research stage.

The third section outlines a few changes that could be made to the MDP architecture that would improve the performance of Cosmos and compiled Concurrent Smalltalk programs.

8.1. Features

This section lists additional features that would be desirable in the Concurrent Smalltalk environment. The most obvious ones are the current omissions from Cosmos: futures, arrays, floating point numbers, and overriding primitive methods. In addition, the performance of Concurrent Smalltalk loops could be improved.

Arrays

Arrays are already fully implemented in Optimist II—Optimist II can interpret and compile code containing arrays. Cosmos, however, does not currently support arrays. When implemented, they will be added in the form of MDP runtime code in the Runtime.m Cosmos file. Ideally four different kinds of arrays will be provided: strings, bit arrays, integer arrays, and general object arrays. Strings can pack four characters per word, bit arrays can pack thirty-two booleans per word, while integer arrays can, depending on the range of integers supplied, pack 1, 2, 4, 8, 16, or 32 integers per word.

I expect arrays to be placed in self-contained objects fitting on single nodes rather than trees. This will limit arrays to about 200 words each because larger objects will overflow message queues when migrated. If large arrays are desired, distributed array classes should be defined, and perhaps `new-simple-array`, `new-integer-array`, `new-string`, and `new-boolean-array` could automatically allocate distributed arrays if their size arguments are large enough.

Enough primitives have been provided in Concurrent Smalltalk to support almost all common array operations efficiently. The `map` and `init` methods treat arrays dataflow-style, allowing elements of arrays to be defined in terms of other elements of the arrays. Cfutures could be used in unpacked arrays to prevent elements of arrays from being read before written; if an array is packed, a bitmap of valid elements, perhaps stored in a context, could be attached to it.

Although implementing arrays well on the J-Machine is not particularly difficult, it is quite time-consuming and was omitted from this thesis for this reason.

Overriding Primitive Selectors

Concurrent Smalltalk allows user programs to override primitive selectors such as + and <, thereby allowing the implementation of additional number types such as complex numbers and matrices which respond to the traditional numeric operations. While Optimist II permits selectors to be overridden in its interpreter, Cosmos does not support this facility, again because this feature would be too time-consuming to implement.

Adding the ability to override primitive selectors will not be as easy as adding arrays, but, fortunately, all the hardware building blocks needed are present in Architecture 11B. When an instruction is executed on a word with a type not supported by the hardware for that instruction, the MDP faults. When a system call such as `Divide` is done on words with unsupported types, the operating system halts. All of the type-related fault handlers and halts will have to be implemented; they will have to decode the operation which caused the fault and emulate it by performing a standard message send. This emulation will require a lot of attention to little details and will be error-prone. Also, the context will have to be enlarged.

For an example of the complexity involved, suppose the user overrides the = method to support complex numbers. One of the consequences might be that a `BNZ` instruction somewhere in the program faults `ID` because it was called on a complex number instead of an integer. The fault handler will have to call the = method to compare the complex number against zero. In order to make this call, it has to save the entire state of computation in the context plus two more words: a return IP back to the fault handler and a slot into which the result (`true` or `false`) should be written. When the fault handler regains control, it will examine the slot and either take the branch in the user program or let execution continue with the next instruction. Due to CFUT-handling in the MDP's architecture (specifically, because WRITE does not fault on cfutures), primitive selectors can never return cfutures.

Another issue is what to do with commutative operations such as +. One might add an integer to a complex number or a complex number to an integer, and it would be nice not to have to override the `integer` method for + to implement complex numbers. To implement this cleanly, the fault handler for ADD would have to try adding the arguments in one order, and, if no method matched, reverse the arguments and try again. If no method matched a second time, it would halt.

Finally, a few minor modifications may have to be done to Optimist II's back end to support overriding primitive selectors.

Long Integers

Once overriding primitive selectors is supported, it will not be particularly difficult to implement a bignum package for the MDPs and watch how many microseconds it takes a J-Machine to compute the factorial of 1000.

Futures

Optimist II currently provides most of the support needed for full futures, although some modifications would still be necessary. The major changes would be to the operating system. The changes would be similar to those needed to implement primitive selector overriding—FUT fault vectors would have to be defined and emulate all possible cases.

Floating Point Numbers

There are four different ways to implement floating point facilities on the J-Machine. Ranging from the easiest to the hardest and most exotic, they are:

1. Emulate operations on the FLOAT data type through software fault handlers. This approach would provide IEEE-compatible[1], single-precision floating point number capability. Unfortunately, this approach would be very slow because of the large instruction decoding and floating point packing and unpacking overheads. The advantages of this approach are simplicity, transparency, and IEEE compatibility, if desired.

2. Store floating point numbers as two words each. One word would be the exponent and the other the mantissa. The precision would be intermediate between single precision and double precision. Floating point operations could be inlined, and micro-optimization techniques [12] could be applied. The advantage of this approach is speed without the need for extra hardware. The disadvantages are that this approach would need object inlining to be implemented by Concurrent Smalltalk (otherwise this technique would be even slower than technique 1), the floating point number format is nonstandard, and the use of floating point numbers would be cumbersome. Since floating point variables would take two words instead of one, they would have to be declared as such to avoid losing efficiency, and a variable could not efficiently support both floating-point and non-floating-point values. The last restriction is a major problem because floating-point versions of all of the methods operating on general objects might have to be written and used to achieve good performance.

3. The third possibility would be the inclusion of a floating-point unit on the MDP. The unit would require no significant software-visible architectural changes; the arithmetic instruc-

---

[1] ANSI-IEEE standard 754.

tions would simply start working on words tagged `FLOAT`, and maybe a `DIV` instruction and a few control registers would appear. The disadvantage of this approach would be the inclusion of a hardware floating point unit on the MDP, which would increase the hardware's complexity. The advantages would be speed, simplicity, transparency, and IEEE compatibility, if desired.

4. The last possibility would be addition of RAP [19] chips to the J-Machine network. RAP chips are custom chips that contain a large number of serial floating point units, achieving estimated peak performance of 300 MFLOPS per chip. Under this approach, an MDP would send floating point calculations to a friendly neighborhood RAP, which would do the calculations and respond to the continuation it was given. This approach would work well if the floating point calculations were grouped and did not have to be mixed with symbolic processing. If the MDPs were to perform mainly symbolic processing with occasional floating point instructions, the message overheads would make this approach inefficient. This approach would require a large investment in operating system and runtime software, and it is not immediately clear that it would be faster than approach 3, although the potential payoff is large.

True Loops

Loops are currently not implemented particularly efficiently in Concurrent Smalltalk. It is not clear whether this inefficient implementation will hurt program performance; it does, of course, depend on how often loops are used. Using iterators and similar abstractions to step through arrays and other data structure is usually preferred to using loops because iterators might execute in parallel, while loops are inherently sequential. Nevertheless, there might be some situations where sequential loops are needed.

The primary reason for the current, inefficient implementation of loops is the need to ensure that a loop does not execute for a long time uninterrupted, preventing other messages from being executed at the node and maybe even causing a message queue overflow. Currently Optimist II compiles a loop into a function which calls itself tail-recursively, which is a fairly large penalty to pay for tight loops. The implementation could be improved to a true loop inside a lambda if the code inside the loop either made at least one full-fledged function call per iteration or tail-recursed every few iterations; either case takes care of the message problem. Some experimentation is needed in this area to determine the best course of action.

Inline Objects

The largest feature change to the Optimist II compiler would be the addition of inline objects. This would be a difficult and error-prone process because all cases have to be handled well; these cases include passing an inline object to a function that does not expect one, storing inline objects in contexts, creating pointers to inline objects, and altering inline objects. It is likely that if inline objects were implemented, several versions of each function would be compiled. One version would be unoptimized, while the others would support inline objects as arguments and results. The constant folder would then try to convert unoptimized function calls to optimized function calls in the same way it currently converts method calls to function calls.

8.2. Resource Management

Concurrent Smalltalk presents the programmer with an ideal model of a machine with an unlimited number of processors and an unlimited amount of memory; unfortunately, real computers are limited in both the number of processors and the size of memory. Several resource management problems result from the discrepancy between the Concurrent Smalltalk ideal and the hardware reality. These problems include reusing memory that can no longer be accessed and simulating an unlimited number of processors with a fixed, finite number. Additionally, there are a few bottlenecks in the current system that can be ignored in small implementations but will become important in large-scale systems.

Heap Compaction

The current design of the Cosmos heap compactor compacts the entire MDP heap when a storage allocation request exceeds available free memory. This approach works, but it has two significant disadvantages, both related to the long time it takes to compact the memory:

1. On a small J-Machine, the MDP will effectively stop responding until the heap compaction is done. In the few tens of thousands of instruction it takes the MDP to do the heap compaction, the other MDPs may run out of things to do and all wait for the stopped MDP. The heap compaction will effectively stop the entire computer. Soon after the first MDP finishes its heap compaction, another MDP may starts its own compaction, and the process will repeat.

2. On a large J-Machine, a heap compaction on one MDP will not be enough to stop the other MDPs from running; instead, they will continue to run longer and are likely to send enough messages to the compacting MDP that its incoming message queue overflows. The poor MDP now does not know what to do because it has no free memory into which to put the extra messages.

Finally, the current heap compactor does not compact BRAT entries or standard contexts, but it could compact them with a little additional effort.

An incremental heap compactor would address both of the serious disadvantages of the current heap compactor. It might even be possible to run the incremental heap compactor in the MDP's background mode, although the lack of a separate set of fault vectors and a full set of registers would pose serious detriments.

Fanout Bottlenecks

Cosmos currently assigns one node as a "home" of an object; with few exceptions, if a different node needs a copy of that object, it turns to the home node to get it, and the home node takes care of supplying the object. Unfortunately, sometimes many nodes want to use the same object simultaneously. Accesses to mutable objects are serialized anyway, so having a home node for a mutable object is not such a bad idea; however, there is no reason why accesses to immutable objects should be unnecessarily serialized. On the contrary, functions are immutable objects, and it would be nice if a function's home node did not have to send a copy of the code to every other node on a 65536-MDP computer.

One solution to this bottleneck would be to assign several home nodes to each immutable object; perhaps the more popular the object would be, the more home nodes it would have. When another node needed a copy of that object, it could ask the closest home node. If one home node were made special and all the others allowed to purge their copies of the object because they could get it from the special home node, this scheme would become a distribution tree. Brian Totty presents an analysis of distribution trees in [38].

Cosmos also serializes the allocation of distributed objects at one node because of the need to give each distributed object a unique ID. The allocation process could be parallelized by splitting the ID space and making several nodes responsible for allocating distributed objects, one for each chunk of addressing space.

Garbage Collection

Garbage collection on the J-Machine is currently an open research problem. Parallel garbage collection algorithms exist, but they may not work well on the J-Machine. For example, the parallel garbage collection algorithm in [29] requires a node to keep track of all of the local IDs it sends to other nodes, which would be unfeasible for two reasons. First, each MDP spends a considerable amount of its time sending data onto the network, and its performance would suffer if it had to record every ID sent. Second, most local IDs become known to other nodes in the J-Machine, degenerating the algorithm's performance.

Perhaps the best solution is a simple mark-and-sweep algorithm run on all MDPs in parallel; after all, the combined MDPs have a considerable amount of processing power. Unfortunately, this approach has three potential problems:

1. The mark-and-sweep garbage collector has to stop the J-Machine, and it might be difficult to stop all processors and allow the messages in the network to land somewhere, especially if the messages in the network are blocked because some node is out of memory and queue space.

2. The J-Machine network bandwidth may be insufficient for a mark-and-sweep garbage collection.

3. There may not be enough room on the MDPs for the intermediate storage needed by the algorithm. In particular, if all the MDPs immediately start marking their root sets, all message queues will quickly overflow with mark messages. This is a parallelism control problem.

Load Management

The purpose of load management on the J-Machine is to distribute a parallel computation evenly throughout the processors while keeping network congestion low. Load management is a very broad current research area. Cosmos and Optimist II include limited attempts to balance the load—Optimist II distributes the objects it compiles evenly among the nodes of the J-Machine, and Cosmos allocates new objects on random nodes and evaluates applications on primitive objects on random nodes to prevent the entire computation from taking place on one node. Nevertheless, these are only initial steps to addressing the load management issues. The following are at least some of the load management concerns that should be addressed on a large J-Machine:

- The current system for allocating objects may have to be reevaluated. At least theoretically, the current system should perform quite well if all objects are about the same size. If the nodes on which objects are allocated are always picked randomly, memory usage on all nodes will remain within a few standard deviations of the average memory usage, so even on a large J-Machine the probability that a single node's memory overflows can be made exponentially small. On the other hand, real programs may allocate objects with a large variation of sizes, and they may wish to allocate objects on specific nodes to take advantage of locality. Both of these conditions may overflow memory on some nodes while other nodes still have a considerable amount of free memory.

- An analogous issue to the one above is handling message queue overflows. Due to the queues' small sizes and the large variance in the sizes of messages, it is difficult to make queue overflows statistically unlikely. Instead, mechanisms have to be introduced to handle them. These mechanisms should not allocate extra local memory because a queue overflow is most likely to happen when little or no memory is available because it is being compacted.

- MDPSim assumes that the MDPs are connected by a crossbar network, so all MDPs are equally far apart from each other. This is a good approximation on a small J-Machine—on a 64-node J-Machine organized as 4×4×4 MDPs, no two processors are more than 9 links apart, while the expected distance between two random nodes is only 3 links. On the other hand, on a 65536-node J-Machine organized as 64×32×32, locality becomes an important issue; if objects continue to be allocated randomly, the network will become hopelessly congested.

There are two general approaches to distributing the load evenly. One approach is to make objects very mobile and hope that they will redistribute themselves to exploit locality. When a portion of the J-Machine becomes congested, it could simply throw objects at the rest of the J-Machine. JOSS hints [38] were an example of a technique that could be used by this approach. While this approach is simple, it does suffer from some disadvantages. In particular, if load management decisions are made often, they cannot be too time-consuming to prevent excessive overhead. Also, when an object migrates often, it is difficult for a node to send a message to it. In JOSS, if a node does not know where an object is, it sends the message to the object's home node instead, which forwards it to the object. If an object is not at the home node, then both the home node and the object's current node are congested with messages addressed to the object. JOSS attempted to correct this problem through the use of hints, but JOSS-style hints may be ineffective because all first-time users of an object must still first reference the home node to get to the object.

The other approach is making objects on the J-Machine relatively static and redistributing them to balance the load only occasionally. This is the approach taken in Cosmos. Objects are free to move around the J-Machine for short periods of time, but an object's home node asks the object to return to it when it another node sends a message to the object via the home node. Hence, objects tend to remain where they were first created. As long as the object allocator allocates objects well, the load will remain roughly balanced. Any small dynamic imbalances that arise can be handled by the garbage collector, which could have the power to truly change an object's home node by renaming all of the IDs in the entire J-Machine pointing to the object.

Controlling Parallelism

In addition to load balancing, which distributes a fixed amount of work among the MDPs, it will also be necessary to throttle the amount of work being done by the J-Machine as a whole. A simple example illustrates this point.

```
(Defmethod fib Integer ()
 (if (<= self 2)
  1
  (+ (fib (- self 1)) (fib (- self 2)))))
```

Figure 8-1. A Doubly-Recursive Fibonacci Program

Consider the doubly-recursive Fibonacci program in Figure 8-1. When run on a sequential computer, the program traverses the computation tree of the Fibonacci function in a depth-first order (Figure 8-2), taking only *O(n)* space but exponential time to compute *Fib*(*n*). On the other hand, when run on the J-Machine, each invocation of fib except the tail ones attempts to evaluate the two recursive calls in parallel. In effect, the computer traverses the computation tree in breadth-first order (Figure 8-3). This is good if there are many processors, because then the function is computed in only *O(n)* time. Unfortunately, this manner of computation requires an exponential amount of both main memory and message queue space. Thus, a parallel computer can fail if a program exhibiting *too much* parallelism is run on it.

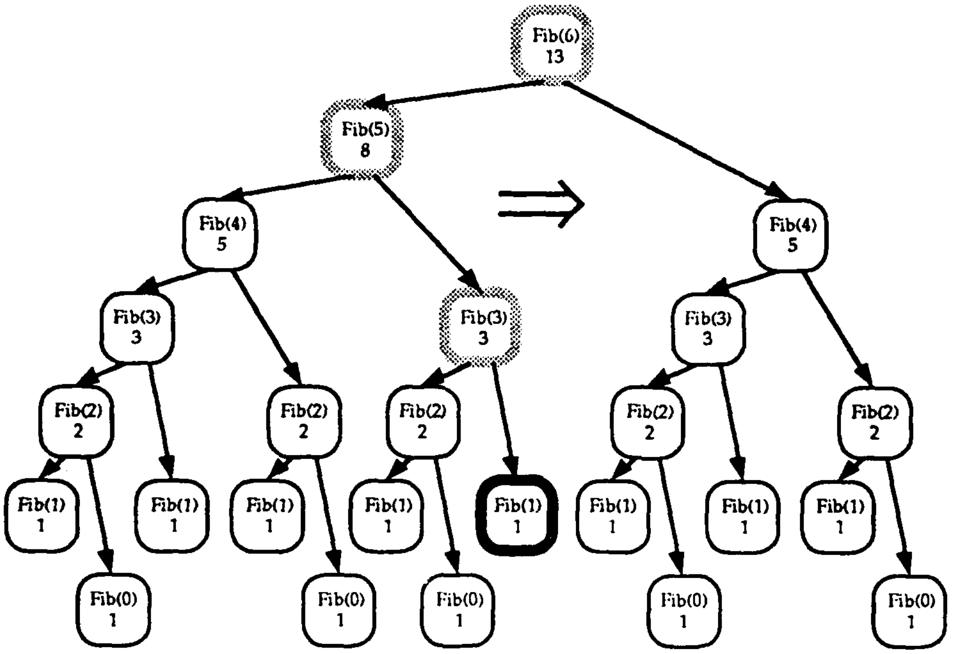


Figure 8-2. Progress of a Sequential Computation

Although the computation consists of a large number *N* of function invocations, a sequential computer traverses the computation tree in depth-first order, so only *O(log N)* functions are active at any particular time (bold gray), and the "wavefront" of computation consists of only a single invocation (bold black). *O(log N)*space is required to run the program and constant-size message queues suffice because the wavefront is at most one invocation.

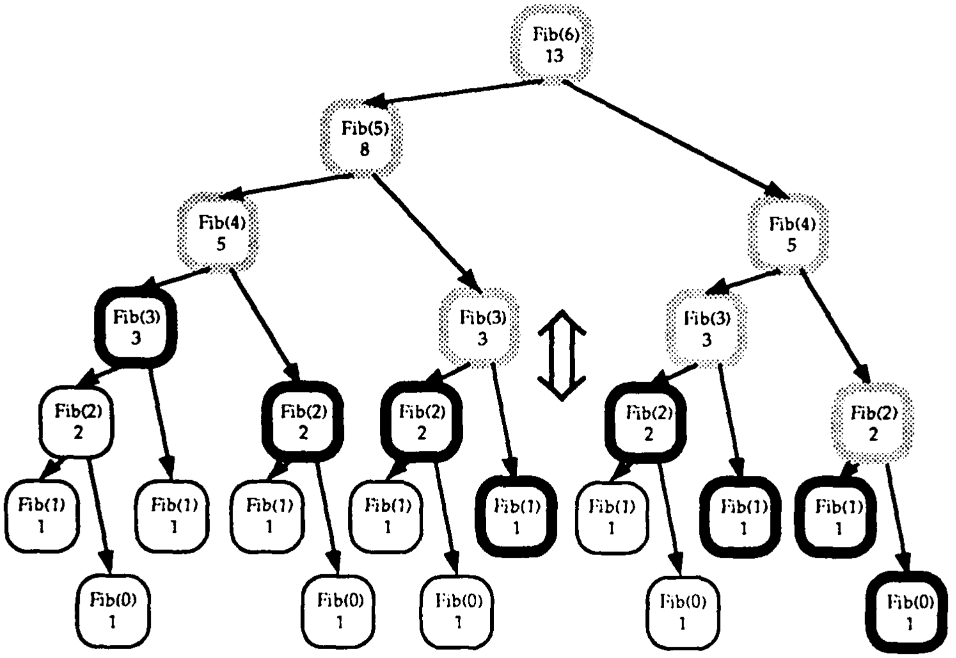


Figure 8-3. Progress of a Parallel Computation

A parallel computation tends to evaluate the computation tree in breadth-first order, which requires the storage of most of the function invocations in the computation tree at about the half-way point. Thus, the computation requires *O(N)* space, and, moreover, the "wavefront" can also become as large as *O(N)*. Hence, the computation also requires *O(N)* message queue space. The computation will exceed the parallel computer's memory if *N* is large compared with the number of processors.

When the compiled code for Fibonacci is run on a simulated 4-node J-Machine, *Fib(11)* is the largest value that can be computed. An attempt to compute *Fib(12)* results in queue overflows; enlarging the message queues or spilling them into main memory would not help much because the storage needs grow exponentially.

Fortunately, it appears that a solution to this problem does exist. Why not change from evaluating the computation tree in a breadth-first fashion to a depth-first fashion when all of the processors on the J-Machine are busy? A seven-instruction change to the compiled code for Fib (Figure 8-4) accomplishes just what is needed. The change forces sequential evaluation of Fib's two recursive calls if the local message queue is more than a quarter full. Thus, the computation grows exponentially until all MDPs are saturated. From then on until the answer is ready, all MDPs are busy computing the problem without increasing the space requirements. After the change was made, the Fib program could calculate answers for much larger inputs.

The simple change in Figure 8-4 is not a panacea, though. The change allows enough parallelism for the message queues to be a quarter full *on the average* throughout the J-Machine. Unfortunately, in practical simulations the sizes of the queues vary widely—the queues on some processors might be empty, while other MDPs may have queues that are more than half full. It is easy to see why this might happen—an MDP with a nearly empty queue is not throttled down and will happily send messages to an MDP with a nearly full queue. Due to this variance, the queues overflowed anyway if the threshold for inhibiting parallelism was set to half of the queue size. To summarize, it seems that this approach for controlling parallelism will work, but it may have to be combined with load balancing to keep the variance in queue sizes low.

Name Spaces

The scarcity of IDs in the 32-bit name space is also an important consideration on the J-Machine. After allowing for flags and nonuniform usage of the name space, 32 bits allow only about a billion objects to be named on the J-Machine. Furthermore, if the name space is not reused, a J-Machine could run out of names in less than a second—each node is limited to creating only about 32000 objects before exhausting its name space.

To solve this problem, object IDs could be collected and reused by the garbage collector[1]. The garbage collector could compact the ID space, which would also permit an ID-renaming load balancer almost for free. However, even this approach might not be enough. If the J-Machine is implemented using technology of the 1990's, it may well have enough physical memory to overflow the 32-bit name space even with garbage collection. At that point the only reasonable solution will be to increase the word size, perhaps to 64 bits.

---

[1] An approach that almost works and does not require a garbage collector is to test each candidate ID in the ID-generation routine. If the ID names an existing object, the ID-generator simply chooses another ID. Unfortunately, this approach does not work for immutable objects because some copies of such objects could exist with the home node not knowing about them. Keeping the home node informed about copies of its immutable objects would cause bottlenecks of its own, not the least of which are the space needed to store such information and the network bandwidth used to maintain it.

```
         MODULE   fFib
         DC       MSG:hdrCopyable|cFunction<<offsetN|32
         DC       {fFib}
         DC       5
         MOVE     [2,A3],R3
         LT       R3,2,R1
         BF       R1,^L001
         MOVE     1,R1
         BR       ^L002
L001:    SUB      R3,2,R2
         MOVE     R2,R0
         CALL     objectNode
         DC       MSG:msgApplyFunction|5
         SEND20   R1,R0
         DC       {fFib}
         SEND20   R0,R2
         MOVE     5,R0
         SEND2E0  [1,A1],R0
         WTAG     R0,6,R0
         MOVE     R0,[5,A1]
         MOVE     QHL,R1
         WTAG     R1,INT,R1
         DC       63
         AND      R1,$3FF,R1
         LE       R1,R0,R1
         BT       R1,^Empty
         MOVE     [5,A1],R0
Empty:   SUB      R3,1,R2
         MOVE     R2,R0
         CALL     objectNode
         DC       MSG:msgApplyFunction|5
         SEND20   R1,R0
         DC       {fFib}
         SEND20   R0,R2
         MOVE     6,R0
         SEND2E0  [1,A1],R0
         WTAG     R0,6,R0
         MOVE     R0,[6,A1]
         MOVE     [6,A1],R2
         ADD      R2,[5,A1],R1
L002:    MOVE     [3,A3],R2
         BNIL     R2,^L003
         DC       MSG:msgReply|4
         SEND20   R2,R0
         SEND0    R2
         SEND2E0  [4,A3],R1
L003:    SUSPEND
         END
```

Figure 8-4. Modified Fib Assembly Language Function

When the incoming message queue is at least a quarter full, the modified Fib function throttles down the parallelism by waiting until the result of the first recursive call has been received before starting the second one. The modification is shown in bold. No parallelism penalty other than the execution of six extra instructions is paid when the J-Machine is not saturated.

8.3. Architectural Considerations

Some architectural modifications could be made that would streamline execution of MDP code in critical sections in the operating system.

Minor Instruction Set Changes

One set of optimizations with a relatively large payoff would be allowing MOVE instructions from the ID registers directly to memory and XLATE instructions directly from memory into address and ID register pairs. Introducing these instructions would cut the number of instructions needed to save and restore ID registers on context switches by half, and it would accelerate allocation and deallocation of fast contexts.

A large part of the operating system is still spent saving and restoring state in fault handlers. Also, most faults point the FIP register to the instruction after the one that faulted, while most fault handlers (with the notable exception of CALL) would rather resume the instruction that faulted, requiring the FIP to point to the instruction that faulted. Backing up the FIP by one instruction takes five or seven instructions depending on whether a free register is available. Pointing the FIP to the instruction that faulted (except for CALL faults) or making an extra shadow FIP register that points to the instruction that faulted would reduce the number of instructions needed in important fault handlers such as CFUT, EARLY, and SEND—the EARLY and SEND fault handlers would be reduced from eight instructions to one!

Other critical resources which are near the limits of their capacities are the message queues and the XLATE table. The message queues can only be made to hold 1024 words, and the XLATE table cannot hold more than 512 bindings. If a MDP has 65536 words of memory, it might be beneficial to have a 2048-binding XLATE table or a message queue that could hold 4096 words, especially if large objects are frequently transmitted over the network.

Another critical resource in the XLATE table is the key space. The XLATE table is a popular associative cache in the operating system, and it is used for a variety of purposes. Unfortunately, there are only 16 tags on the MDP, and tag conflicts exist among the keys XLATEd by the users of the XLATE table. For example, class/selector pairs had to be tagged `INST1` because all of the "normal" tags were already taken. A future version of the operating system might run out of key tags for the XLATE table. Possible solutions to this problem include, but are not limited to, providing several XLATE tables or using more than one word as a key.

Finally, one instruction is seldom used and could be removed. The `INVAL` instruction is used only once in Cosmos in the heap compactor, and since a heap compaction takes a long time anyway, emulating `INVAL` in software would neither be difficult nor harm performance.

Fast Context Saves and Restores

Perhaps a more ambitious project would be to attempt to improve the MDP's context-switching time by supporting in hardware a shadow image of the registers in memory. In other words, the registers would act as a cache for a context in memory. When a context switch occurred, the modified registers would be written back into memory and a new register set loaded from the new context. Quick register saving and restoring for fault handling is even more important than fast context switching, and this approach might be generalized to support fault handling as well by allocating a context to each fault handler that wanted one.

8.4. Conclusion

A number of desirable features for future inclusion in Cosmos or Optimist II were described, including arrays, full futures, overriding primitive selectors, floating point numbers, and large integers. Implementing arrays and primitive selector override facilities should not present major difficulties, although it will be time-consuming. Several approaches for implementing floating point numbers were discussed, including two software approaches—a fast and dirty one and a clean but slow one—as well as two hardware approaches—including a floating point unit on every MDP and including RAP chips in the J-Machine network.

In addition, a number of resource management issues were discussed, ranging from heap compaction, garbage collection, load management, and ID reuse to fanout bottlenecks and parallelism control. New methods may have to be developed to support efficient garbage collection on the J-Machine, but once garbage collection is done, ID reuse and load management may be obtained for free. Parallelism control is a serious issue in many applications. An application that tries to operate on a large data set in parallel or explore a large search tree will quickly overflow the entire J-Machine's queue capacity. One approach to solving this problem was explored—if Concurrent Smalltalk code switches to evaluating itself sequentially when the local queue size exceeds a threshold, the total queue size on the J-Machine appears to remain bounded, although individual queues may still overflow. This approach shows some promise for solving the parallelism control problem.

Finally, a few changes to the MDP architecture were proposed. Allowing direct `MOVE`s to and from `ID` registers and providing the right `FIP` value after a fault would save instructions in many critical Cosmos code sections.

While Cosmos and Optimist form a workable system as they are now, much fine-tuning remains to be done. Due to a lack of time, a few features of Concurrent Smalltalk have not been fully implemented. The door is now open for experimenting with the difficult problems of load management, concurrency control, and garbage collection. These areas have not been studied very much in the context of fine-grain parallel computers, and there is room for both practical and theoretical results.

Chapter 9. Conclusion

Optimist II

Optimist II is a second-generation optimizing compiler for Concurrent Smalltalk, and the first to implement nearly the entire revised Concurrent Smalltalk language. Optimist II builds upon Optimist by adding an interactive prototyping and debugging environment and a few new classes of optimizations. The introduction of global optimizations was especially valuable in making Concurrent Smalltalk easy to use efficiently and the runtime system easy to write. The greatest advantage of global optimizations is that they permit the programmer to divide a system into self-contained abstractions without suffering a performance penalty for doing so. There is a trend in modern programming languages towards global optimizations[1], and Optimist II shows that they are both feasible and desirable for a language like Concurrent Smalltalk.

Cosmos

Cosmos is an optimized operating system for the J-Machine. In addition to performing the necessary services to keep the J-Machine running, it includes facilities for function and method calls; local and global object allocation, disposal, and migration; method lookup tables; distributed object creation and addressing; and various utilities. A few interesting programming techniques were used: an infinite loop broken by a fault is used for block moves and sends, and an addressing scheme was developed for distributed objects that allows easy addressing of constituents while at the same time distributing them throughout the J-Machine and allowing efficient implementation of an operation that returns a nearby constituent.

Cosmos was fairly difficult to write due to the constant specter of re-entrancy problems and double faults. These errors were the most common problems in JOSS [38]. Nevertheless, with the aid of the criticality system those difficulties were overcome. Unfortunately, the casualty of this battle with re-entrancy is ease of modification of the Cosmos kernel—the kernel is now one compact piece of code. Nevertheless, it should not be necessary to make extensive modifications to that kernel in order to add the features mentioned in Chapter 8.

Debugging

An important consideration when designing a complicated computer system today is ensuring that it is debuggable. The hardware world has been buzzing with ideas such as design-for-test for a few years now; yet, these ideas are just as applicable to software. Thus, Cosmos includes consistency checks in strategic locations which detect common errors that may be committed by Concurrent Smalltalk programs. However, even with those checks debugging a Concurrent Smalltalk in assembly language is unpleasant and not as interactive as it could be; for this reason, Optimist II includes an interpreter which can be used to get a Concurrent Smalltalk program working before it is run on a J-Machine.

Performance Measurements

Performance measurements on a simulated J-Machine indicate that the grain size (the number of instructions executed in response to a message) averages about 60 instructions. Since most functions invocations involve two messages (tail-forwarded invocations being an impor-

---

[1] For example, C++ [37] allows functions to be declared *inline*, recommending that the compiler inline them in other functions.

tant exception), the average number of instructions needed to process a function call is about 100 to 120; the number is lower if many tail-forwarded invocations are made.

The MDP executes one instruction about every two cycles when running entirely from internal memory; when the user program and data are located in external memory, that count only doubles to about four cycles per instruction even though the external memory is 5 times slower for writing, 3 times slower for reading, and about 24 times slower for fetching instructions. The reason for the unusually low cycles-per-instruction number when the user program and data are located in external memory is the high operating system overhead; since the operating system is always in internal memory, running operating system code out of internal memory tends to pull the cycles-per-instruction number down.

Under good conditions the MDPs can saturate the network on J-Machines larger than 343 nodes, although most programs will not execute fast enough for the network to saturate until significantly larger J-Machines are used. To prevent network saturation, either the network will have to be made faster, the program slower, or some means of exploiting locality invented.

Future Work

Many ideas for future work and research were outlined in Chapter 8. The short-term goals are twofold: first, to fill the remaining holes in the implementation of Concurrent Smalltalk; in particular, arrays will be useful for running real Concurrent Smalltalk applications; second, to write some nontrivial Concurrent Smalltalk programs and see how well they can utilize the J-Machine's power. In addition, the load management and parallelism control issues in Chapter 8 should be explored. So far development of the Concurrent Smalltalk environment has been done without much feedback from applications because until very recently it was not possible to run any applications on even a simulated J-Machine. Now that the compiler and the operating system are operative, it will be possible to close the loop and provide concrete measures of the J-Machine's performance on real problems.

Hopes

Optimist II and Cosmos are but an early step in an evolving base of software for the J-Machine. My hopes are that the J-Machine will evolve into a computer competitive with today's fastest computers on numerical codes and surpassing them on less-structured but nonetheless computation-intensive Artificial Intelligence applications.

When I originally wrote this thesis in early 1989, I wrote that I was hoping to be able to run a Concurrent Smalltalk program on a set of *real* MDPs. Two years later, during the summer of 1991, this wish came true.

Appendix A. Concurrent Smalltalk Reference

A.1. Introduction

Concurrent Smalltalk (CST) is a concurrent descendant of Smalltalk. It is an object-oriented programming language developed for multiple instruction/multiple data concurrent computers such as the J-Machine. It is an interesting language for a message-passing concurrent computer because it encourages locality and disciplines the use of message-passing.

Goals

Concurrent Smalltalk is a high-level language intended for general-purpose programming of the J-Machine. It was created and revised with the following goals in mind:

- **Expressiveness.** Concurrent Smalltalk must be expressive enough to support the parallel programming paradigms we desire to research on the J-Machine. In particular, it must support object-oriented programming and fine-grained parallelism. Also, since a large part of the Concurrent Smalltalk runtime system is written in itself, Concurrent Smalltalk must support higher-order features such as reasoning about classes of objects.

- **Consistency.** Features which would interact destructively with other features were left out. For example, `become`, although a useful Smalltalk-80 construct, would confuse the type semantics so it was left out.

- **Simplicity.** Concurrent Smalltalk should be as simple as possible. In order to reach the goal of simplicity, Concurrent Smalltalk should consist of a few orthogonal concepts. It is very important that Concurrent Smalltalk contain *no surprises*—one should be able to tell what a program should do by reading it. Features involving action at a distance (i.e. having a statement invisibly affect another statement far away) were intentionally excluded.

- **Familiarity.** Programmers familiar with existing languages should be able to carry over their experience to Concurrent Smalltalk. Also, corresponding features should act in the same ways, which reinforces the "no surprises" philosophy. On the other hand, Concurrent Smalltalk is most similar to Smalltalk-80, Common Lisp, and Scheme in this respect. Hence, static scoping is used for variables.

- **Efficiency.** It is important to be able to compile Concurrent Smalltalk programs into efficient machine code. An efficient implementation allows a programmer to concern himself primarily with algorithms and implementation rather than performance tuning. Concurrent Smalltalk is not a tightly bound low-level language in order to give the compiler latitude in optimizing code.

- **Commonality.** The sets of built-in classes and methods presented in this language specification are by no means minimal. However, the built-in classes are frequently used and were included in order to provide a common base for Concurrent Smalltalk programs. The inclusion of frequently used classes has three advantages:
  - The built-ins are implemented only once, saving time and effort.
  - The built-ins provide a consistent functional and naming specification.
  - The built-ins can be optimized for efficiency.

Format

BNF

The syntax of commands is presented in BNF. Literals are presented in bold, while non-terminals and metasymbols are plain. There are two enhancements to the BNF syntax:

The {expr1 | expr2 | ... | exprn} form specifies that each expr can appear at most once, but they can appear in any order. The symbol ≡ expr form is a macro used for readability. It specifies that whenever symbol appears, it should be replaced by expr before any productions are done.

Methods and Functions

The declarations of methods and functions are presented in a syntax similar to that used by `defmethod`. To give an example,

(move what:robot **x, y, z**:integer **theta**:float) :result **Method**

declares a method called `move` of class `robot` that takes a receiver argument `what` of class `robot`, three `integer` arguments, `x`, `y`, and `z`, and a `float` argument, `theta`. That method returns an object of class `result`.

Sometimes an *abstract class* like `number` is declared that has no direct instance objects; instead, every object of class `number` is also an object of one of `number`'s subclasses. Methods of an *abstract class* may or may not have definitions for that class. A method that does not have a definition for the abstract class is called an *abstract method*. For example, + is an abstract method of class `number`; there exists no generic method to add two arbitrary numbers. Instead, when + is called on two numbers, the definition of + for either the class `integer` or the class `float` is used. Were a third `number` subclass, `complex-number`, defined, it would have to define its own + method. On the other hand, the `zero?` method of class `number` is not abstract because it uses the = method (a method defined on all `numbers`). Thus, `complex-number` does not have to define its own `zero?` method.

Abstract methods are indicated by the words **Abstract Method** on the right side of the declaration line.

Optional statements are extensions to the basic Concurrent Smalltalk language. They are not guaranteed to be present in all implementations of Concurrent Smalltalk, but if an implementation supports the capabilities described by optional statements, it should use the described syntax.

A.2. Syntax

Tokens

A Concurrent Smalltalk token is an arbitrarily long string composed of the characters `A-Z`, `a-z`, `0-9`, `_`, `!`, `?`, `%`, `+`, `-`, `*`, `/`, `.`, `<`, `=`, `>`, `&`, `@`, and `^`. The characters `!`, `?`, `&`, and `@` may not be used at the beginning of a token, and a token may not be composed entirely of periods (`.`) or underscores (`_`). Also, tokens beginning with an underscore (`_`) or a percent sign (`%`) are reserved for system purposes and macros and should not be used by user programs. Case is not significant.

A token is considered to be a number if it consists entirely of the characters `0-9`, `_`, `+`, `-`, `/`, `.`, `E`, or `I`; it contains at least one digit; it begins with `+`, `-`, or a digit; and it does not end with a digit. These rules are borrowed from Common Lisp. `E` introduces an exponent, while `I` can be used for complex numbers if they are implemented. Any token that is not a number is an identifier.

Identifiers

Concurrent Smalltalk uses static scoping of identifiers. Local identifiers shadow identical global identifiers, and the meaning of an identifier can be determined by its location in the text of the program. Global identifiers are introduced by the following top-level statements and their derivatives:

- `Defconstant`, to define a constant;
- `Defglobal`, to define a global variable;
- `Defclass`, to define a class;
- `Defselector`, to define a method selector;
- `Define`, to define one global identifier in terms of another.

The syntactic sugar `defmethod` expands into, among other statements, a `defselector`, defining a global identifier. Similarly, `defun` expands into a `defconstant` statement that defines the function.

Except for classes, the above categories share a single name space. Redefining a global identifier causes an error or a warning unless the new definition is identical to the old one. Class names have lower precedence than other global identifiers, so a global constant can shadow a class name.

All macros are global; however, macros are also in a name space separate from the one shared by the above categories. Since macros match patterns instead of just names, two macros may share the same name. If more than one pattern is applicable, one is chosen at the implementation's discretion. Whenever a macro is applicable, it is expanded, unless one of the literals specified in the macro pattern is shadowed by a local identifier.

Local identifiers are introduced by the following statements and their derivatives:

- `Lambda` and `defun` introduce the names of formal parameters.
- `Method-Lambda` and `defmethod` introduce `self`, `group`, names of the instance variables, and names of the formals. If a name conflict occurs between the formals, instance variables, `self`, and `group`, the results are unspecified.
- `Let`, `clet`, `mv-clet`, and `mv-let` introduce names of the locals.
- `Lambda`, `method-lambda`, `defun`, `defmethod`, `block`, and `loop` introduce the names of continuations.

All of the shadowing rules are summarized in Figure A-1.

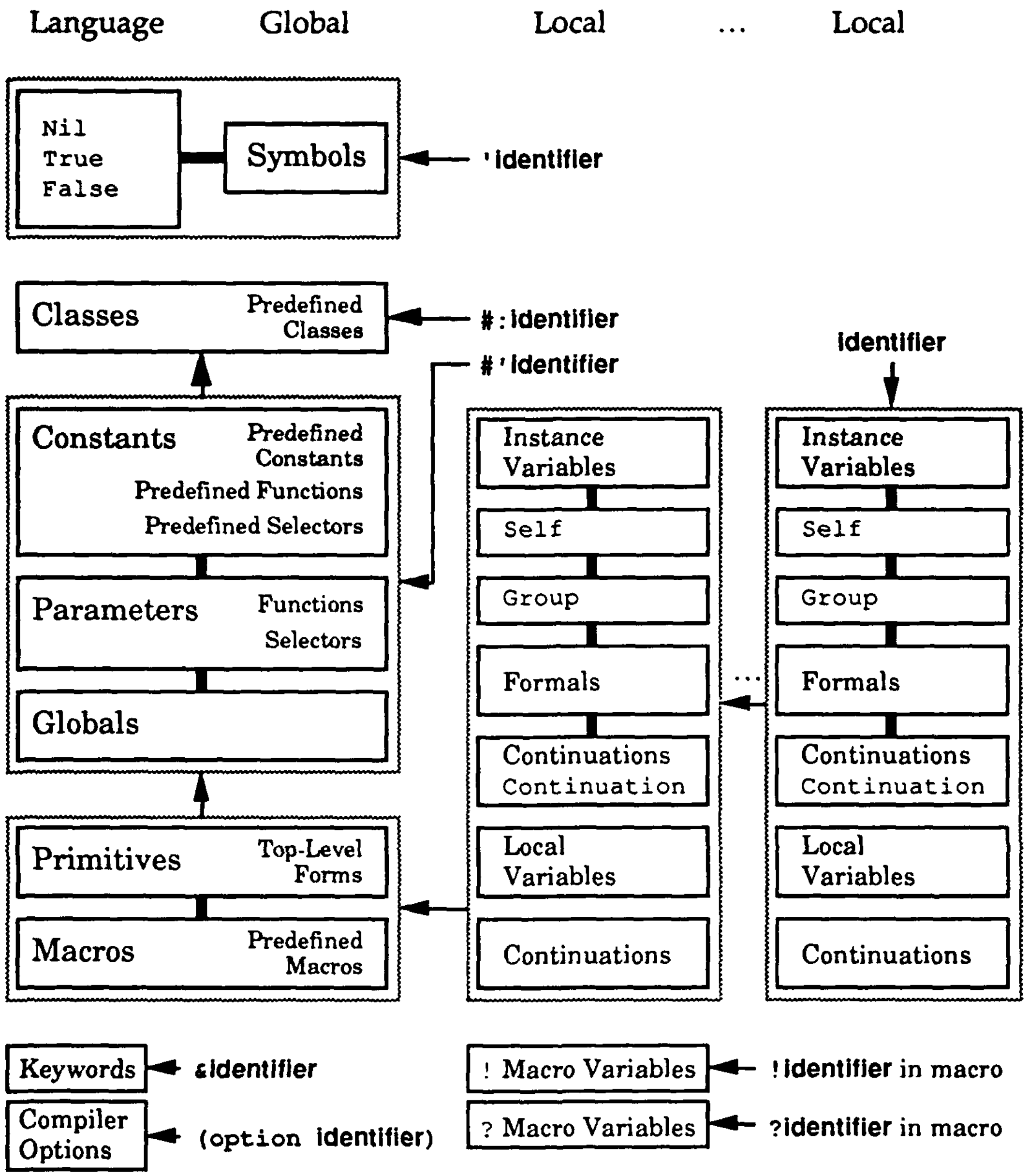


Figure A-1. Scopes of Identifiers

The scopes of various kinds of identifiers are shown above. Except for macros, sets of identifiers connected by thick lines are mutually exclusive and may not contain duplicate names. To find the meaning attributed to an identifier, follow the arrows from the bold pattern indicating the identifier's usage to the first box that contains the identifier. For example, if `i` is encountered in a program, it is first checked to be a local in the innermost scope, then a local in the next innermost scope, and so on until the global scope is reached. If `i` is not a valid macro pattern, it is checked against the globals, parameters, and constants, and finally classes. On the other hand, if `#:i` is encountered, `i` is checked against the names of classes only. `#'i` searches only globals, parameters, and constants, both user and predefined.

Identifiers that are not defined globally[1] or in any enclosing scope are defined as globals. They must be defined before they are used. The exceptions to this rule are identifiers enclosed in `quote` or `class` statements listed below.

(**`global`** identifier) **`Primitive`**
#'identifier

`Global` returns the *global* identifier identifier, which, if already defined, must be a global (not a class). If identifier is not already defined globally, it is defined as a global.

(**`class`** identifier) **`Primitive`**
#:identifier

`Class` returns the *global* class class. Since classes are in a separate name space from other globals, no error occurs if there is already a global identifier defined with the same name as identifier.

Symbols

(**`quote`** (**`nil`** | **`true`** | **`false`** | identifier | number | character | string)) **`Primitive`**
'(**`nil`** | **`true`** | **`false`** | identifier | number | character | string)

Symbols can be specified by preceding including them in a `quote` form as above, which can be abbreviated by a quote mark (`'`). When presented with an identifier, the `quote` expression evaluates to a symbol. Any valid identifier except `nil`, `true`, and `false` can be used—symbols cannot be captured by any scope, nor can they be globally redefined. `Nil`, `true`, and `false` are treated specially—`(quote nil)` returns the null object `nil`, while `(quote true)` returns the boolean `true` and `(quote false)` returns the boolean `false`. `(quote` number`)`, `(quote` character`)`, and `(quote` string`)` just returns the number number, character, or string.

Constants

A few constants are predefined. These are listed in Table A-1 below. In addition, any number can be specified by just including the number. Characters can be specified by preceding them with `#\`. Strings can be specified by enclosing them in double quotes ("). Double quotes can be included inside strings by preceding them with `\`.

Table A-1. Predefined Constants

Constant	Value	Class
`TRUE`	`True`	`True`[2]
`FALSE`	`False`	`False`[3]
`NIL`	`Nil`	`Null`
`End-of-file`	An end-of file object	`Object`

---

[1] It is up to the implementation to define the meaning of a global definition here. When a file is compiled, an implementation might choose to read all of the definitions in the file and then compile the code, or it could compile the file incrementally. In the latter case forward-referenced identifiers will be considered undefined.

[2] Since `true` is a global constant, `#:true` has to be used to refer to class `true`. Also, class `true` is a subclass of class `boolean`.

[3] Since `false` is a global constant, `#:false` has to be used to refer to class `false`. Also, class `false` is a subclass of class `boolean`.

Comments

Comments may be placed anywhere in source files. A comment starts with a semicolon (;) and is terminated by the end of the line. Comments are treated as if they were line breaks by the reader.

A.3. Programs

program ::= (top-level | statement)*

A Concurrent Smalltalk program is a sequence of top-level forms. Additionally, an implementation may allow `begin` and `if` as top-level forms if test is a constant expression and statements in body, consequent, and alternative are top-level forms. Other statements may also be allowed at the top level by extended implementations. Statements at the top level are executed sequentially as if they were enclosed in a `begin`.

Constant Expressions

Constant expressions are expressions that have to be evaluated at compile time. A constant expression can include any expression or function call, except that constant expressions may not produce distributed objects as values and may not call functions that use futures.

Global Definitions

All constants, parameters, and globals reside in a single name space; in general, redefining an identifier with a different meaning causes an error. Macros reside in a separate name space and do not conflict with each other or any other global objects (although they may be shadowed by local static scoping).

(**defconstant** name[:type] value) **Top-level Primitive**

`Defconstant` defines a constant named name. The constant can be any valid Concurrent Smalltalk type. If type is specified, the value must have that type. Once a constant is defined, it may not be changed (another constant is accepted, though, if it has the same value). Constants encountered in methods are replaced by their values at compile time. Value must be a constant expression. Predefined constants are listed in Table A-1.

Language primitives and built-in functions and selectors are defined as global constants.

(**defparameter** name[:type] value) **Top-level Primitive**

`Defparameter` defines a parameter named name. The parameter can have any valid Concurrent Smalltalk type. If type is specified, the value, if present, must have that type. If no type is specified, type is assumed to be `object`, the most general type. Parameters encountered in methods are replaced by their values at compile time. Value must be a constant expression. Unlike constants, parameters may be redefined at the top level, but their types may not be changed. The value of a parameter may not be changed by a running program.

User functions and selectors defined using `defun`, `defmethod`, and `defselector` are defined as parameters. Hence, they may be redefined.

(**defglobal** name[:type] [value]) **Top-level Primitive**

`Defglobal` defines a global named name. The global can have any valid Concurrent Smalltalk type. If type is specified, the value, if present, must have that type. If no type is specified, type is assumed to be `object`, the most general type. Value must be a constant expression.

A global may be defined several times, but only the value from the first definition is used. Nevertheless, all definitions of a global must have the same type.

(define name name) **Top-level Primitive**

This primitive defines the first name as an alias for the object specified by the second name. For example, if the second name refers to a global, after this primitive is executed, both names will refer to the same global.

(undef name) **Top-level Primitive**

This primitive removes the top-level definition of name, if any. It should be used with caution, as it is possible to bring the system into an inconsistent state using `undef`.

A.4. Classes

Built-in Classes

A few classes are predefined. These are listed in Table A-2, and their hierarchy is shown in Figure A-2. The `defclass` primitive can be used to define other classes, which may be based on the built-in ones shown in bold in Figure A-2.

Defining New Classes

(defclass class {class-declaration} superclasses instance-var-spec*) **Top-level Primitive**

class ::= name
superclasses ::= (class+)
instance-var-spec ::= typed-names | (typed-names {instance-var-declaration})
typed-names ::= name (, name)* [: type]
instance-var-declaration ≡ **&inline** | **&not-inline** |
&reader names |
&writer names | **&cwriter** names |
&cas-er names
names ::= name | (name*)

`Defclass` defines a new Concurrent Smalltalk class. A class is a template for specifying objects and methods. Each object belonging to the class contains the instance variables defined in the class definition as well as the instance variables inherited from its superclasses, if any.

In the class definition, class is the new class name. It is followed by an optional declaration, described later, the class's superclass list, and finally the additional instance variables declared by the class.

Class Inheritance

Each user-defined class must have at least one superclass, but it may have more than one. A class inherits the instance variable and method definitions from its superclasses. It may add its own instance variables and methods, and it may attempt to override existing methods. If a class is overriding a method, the new method must be a subtype of the existing one.

A simple form of multiple inheritance is allowed. Two or more superclasses may be specified for a class under the following conditions:

- There must be no instance variable conflicts among the superclasses. Formally, this requirement is satisfied if and only if out of the superclasses s_1, s_2, ... s_n provided there is one s_i such that if v is an instance variable of s_j, $1 \le j \le n$, then v is an instance variable of s_i or one of its superclasses.

- There must be no inherited method conflicts among the superclasses. Formally, this means that if selector s is associated with method m_i for superclass s_i and method m_j for s_j, then m_i and m_j are the same method (Textual equivalence of the method code is not enough; m_i and m_j must "point" to the same method).

The class then inherits all of the instance variables and all methods from all of its superclasses.

Instance Variables

After the superclasses in the class declaration is a list of new or redefined instance variables. Instance variables without any type are given the type `object`. An instance variable may be specified to have the same name as an instance variable of one of the superclasses. If so, the specified type must be a subtype of the original instance variable's type, and either both or neither must be inline.

An instance variable may be declared `&inline` or `&not-inline`. These are hints to the compiler that the variable's object should be placed inline or on the heap (not inline). These hints only apply if the variable's type is an inline class. The compiler is free to ignore these declarations.

Reader and Writer Methods

A few methods are automatically defined when a class `c` is defined. For each instance variable `x` of `c`, two functionally identical methods are defined, named `x` and `get-x`, that, when called on an object `o` of class `c`, return the value of `x` in `o`. These methods are called *reader* methods; two are defined in order to avoid name conflicts with instance variables. Similarly, a *writer* method `put-x` is defined that, when called on an instance object `o` of `c` and a new value `v` of `x`, and assigns `v` to `x` in `o` and returns `o`. Furthermore, a *cwriter* method `cput-x` is defined that behaves just like the writer method `put-x` except that it is not strict—it does not necessarily touch its second argument `v`. Finally, a *cas-er* method `cap-x` is defined that performs an atomic compare-and-put operation: `(cap-x o comparison replacement)` checks whether the value of instance variable `x` in `o` is `eq` to the value of comparison. If so, it stores the value of replacement in `x` and returns `true`; otherwise, it returns `false`.

If it is desirable to produce reader, writer, cwriter, or cas-er methods with names different from the defaults, the `&reader`, `&writer`, `&cwriter`, and `&cas-er` options can be used to specify the new names. More than one method name may be specified for an instance variable. If `&reader`, `&writer`, `&cwriter`, or `&cas-er` is used, the corresponding default method name is not defined. For example, if `&writer` is used with an empty list of names, the corresponding writer method name is not defined.

Class Definition Options

class-declaration ≡ **`&inline`** | **`&not-inline-default`** |
`&immutable` |
`&predicate` names

A class definition allows several options which are described in more detail below.

A class may be declared *inline*, which means that, whenever possible, objects of that class are allocated inside other objects or in local variables instead of on the heap. `&not-inline-default` is an option for inline classes.

Objects of an *immutable* class declared with the `&immutable` option may be shallow-copied at any time at the system's discretion, which can lead to significant performance improvements. They are also often passed by value to methods and functions. It is not necessary that no methods ever write to instance variables, but only that the effects of such writes not be visible outside the class data abstraction. The compiler is free to ignore `&immutable` declarations.

The `&predicate` option defines the name or names of the class predicate. A class predicate is a function that returns `true` when called on an object of the specified class or its subclasses and `false` on all other objects. The default name of a class predicate is obtained by concatenating a question mark (`?`) to the end of the class name, so `(integer? x)` tests whether `x` is an `integer`.

Inline Classes

When a class is declared `&inline`, instance objects of that class are often *inlined*—allocated inside other objects or local variables. No method dispatching takes place on inlined objects because the compiler knows the exact types of inlined objects—inline class methods are converted to functions. Declaring a class `&inline` does not alter its semantics except for a few additional restrictions on its usage. The compiler is free to ignore `&inline` declarations.

Subclasses of inline classes can be declared under the following restrictions: A subclass of an inline class may not declare any additional instance variables, and it may not override any methods. The only superclasses allowed for inline classes are classes with no instance variables.

Normally all formals, locals, and instance variables declared with inline classes are inlined by default. However, that default can be overridden for individual variables by declaring them `&not-inline`[1]. The default can be overridden for all variables by declaring the class `&not-inline-default`, in which case individual variables can be inlined by declaring them `&inline` and giving them the proper type.

Inline classes are useful for representing small objects such as `floats` and `locks` which require more than one word but for which ordinary object overhead is prohibitive. In general, it is pointless to declare a class inline unless it is immutable or its instance objects are rarely passed to methods other than the inline class's.

---

[1] Another way to override this default is to declare the variable's type as `object`.

A.5. Methods and Functions

Introduction

Methods and functions are the basic blocks of computation in Concurrent Smalltalk. Each method and function can accept a number of arguments, which are assigned to the formals for the duration of the execution of the body of the method or function. Furthermore, a method has a special first argument, called the receiver, which contains an object of the method's class on which the method was called. In general, methods and functions execute concurrently unless explicitly synchronized. This is true even if they are accessing shared objects.

Formals

formal-spec ::= typed-opt-names | (typed-opt-names {formal-declaration})
typed-opt-names ::= opt-name (, opt-name)* [: type]
opt-name ::= name | _
formal-declaration ≡ **`&value`** | **`&inline`** | **`&not-inline`** | **`&no-leak`** | **`&name`** name

A method's or function's formals are listed when the method or function is declared. Each name specifies the name of a formal. Typed-opt-names specifies one or more names separated by commas followed by an optional type. The character _ can be used to indicate an unnamed formal; unnamed formals accept arguments but cannot be referenced from within the method or function. If type is not present, it defaults to `object`. If the long form of a formal-spec is used, the formals in typed-opt-names can be declared using declarations.

Arguments are passed by value, just as in Smalltalk-80, Scheme, and Common Lisp. The types of the arguments to the method or function must be subtypes of the types of the corresponding formals. A method or a function may assign a value to a formal, which only changes the method's or function's local value. Of course, a method or a function is also free to mutate a formal using some other method; such changes *are* visible to the outside. This kind of mutation corresponds to communication via shared objects.

A formal may be declared `&value`, which means that, at the implementation's discretion, the method or function may be passed a shallow copy of the argument when it is called. Thus, not only is the formal passed by value, but its first-level structure may also be passed by value. All formals declared using an `&immutable` class are automatically declared `&value`. `&value` declarations are especially useful to improve efficiency of inline classes.

A formal may also be declared `&inline` or `&not-inline`. These are hints to the compiler that the formal's object should be placed inline or on the heap. These hints only apply if the formal's type is an inline class. The compiler is free to ignore these declarations.

Declaring a formal `&no-leak` is a hint to the compiler that the value of the formal is not passed out of the method or function, and it will not be referenced after the method or function returns. Thus, the implementation is free to perform a shallow deallocate on the value of the formal when the function returns. This declaration is especially helpful for arguments of type `funct`. The compiler is free to ignore this declaration.

`&name` can be used to name an anonymous function or method. The name is saved for debugging purposes. `&name` is only allowed in a `lambda` or a `method-lambda`.

Return Values

return-specs ::= : type | : : (return-spec*)
return-spec ::= typed-names | (typed-names {return-declaration})
typed-names ::= name (, name)* [: type]
return-declaration ≡ **&value**

A method's or function's return specification may be listed when the method or function is declared. Most methods and functions return only one value. For these functions, the short form, consisting of a colon (`:`) followed by the return type, is adequate. If the return type is `object`, the entire return specification can be omitted altogether.

The long form of declaring a method's or function's return types uses the double colon (`::`) notation and allows explicit naming of the return continuation. The name is called a *continuation name*. Continuation names are lexically scoped and may be referenced in the body of the method or function. The syntax and semantics of continuation declarations are analogous to those of formals, and the continuation names reside in the same namespace as formal and variable names. The only declaration allowed is `&value`. If the short form is used, a default continuation name `continuation` is used. Some implementations may also allow returning multiple values. Multiple values do not all have to be returned at the same time, but all have to be returned at most once by the time the method or function finishes.

Since the implicit `return` statement at the end of a method's or function's body returns its value to `continuation`, it is an error to allow execution to "fall through" the method or function to the implicit `return` statement unless one of the continuations is named `continuation`.

Method and Function Declarations

funct-declaration ≡ **&non-strict** |
(&inline | **&not-inline)** |
&side-effect-free

The following declarations are allowed for methods and functions:

- The `&non-strict` declaration specifies that the arguments do not have to be touched before the body of the method or function begins executing. Thus, the method or function may at the compiler's discretion receive cfutures in the formals. This declaration is useful mainly for inline functions.

- The `&inline` and `&not-inline` declarations specify that the method or function should or should not be included inline at the points where it is called. This declaration is only a hint, and the compiler does not have to obey it.

- The `&side-effect-free` declaration is a hint to the compiler that the method or function does not perform any visible side effects on its arguments or on the global environment. This information lets the compiler better schedule calls to the method or function. This directive is also useful on methods and functions that do perform side effects; it tells the compiler that those side effects are not essential. One example of a method that falls into this category is a method operating on an immutable class of complex numbers that allows redundant representations in rectangular or polar form. The method could side effect a complex number to calculate its polar representation from its rectangular one, but that side effect is not essential for the program to work correctly.

The Calling Process

When a function or a method is called, the values of the arguments are computed and assigned to the formals. The formals are touched unless the function is declared `&non-strict`. After all formal values are evaluated, execution of the method's expressions proceeds as if the expressions were enclosed in an implicit `block`—initially the first expression is evaluated, then the second one, and so forth. The value of the implicit `block`, which is the value of the last expression, is returned to the caller unless an `exit` or `return` statement is encountered first.

Scoping of Local Variables

Local variables are statically scoped. Any `lambda`, `method-lambda`, `future`, or `lazy-future` created within a method or a function is a full closure and may reference and alter the method's or function's local variables. Similarly, the method or function may alter its locals, and such changes will be visible to any `lambda`, `method-lambda`, `future`, or `lazy-future` nested within it.

If concurrency and efficiency are desired, however, such sharing should be avoided whenever possible. A `lambda`, `method-lambda`, `future`, or `lazy-future` should declare its own temporaries for local computations instead of using ones belonging to an outer static scope. If a method or function wants to pass values into a closure, it should initialize the appropriate temporaries before the closure is created and not change those temporaries afterwards. The closure should not change those temporaries either, unless it wants to pass a result back to the method or function that created it.

Functions

(**lambda** (formal-spec*) [return-specs] {funct-declaration} body) **Primitive**

`Lambda` defines and returns an anonymous function. Formal-spec* is a list of the function's formals and their types. Return-specs specifies the function's return type, or, if it returns multiple values, the number of return values and their types. The function may also have declarations, as explained above. Body is a list of statements that form the body of the function.

(**defun** name (formal-spec*) [return-specs] {funct-declaration} body) **Top-level Macro**

`Defun` defines a global function with name name, formals as specified in formal-spec*, return values defined by return-specs, optional declarations funct-declaration, and body body.

Methods

(**method-lambda** class (formal-spec*) [return-specs] {funct-declaration} body) **Macro**

`Method-lambda` returns a method of class class. The resulting method does not have a selector. Nevertheless, it can be called as a function if the first argument is an instance object of class. The other parameters are as in `lambda`.

Method-lambda also introduces into the scope of body the names of the instance variables of an object of class class as well as two special variables: `self` and `group`. `Self` refers to the first argument of the method call, also known as the receiver object. If class is a subclass of `distobj`, `group` refers to the group name of the distributed object of which `self` is a constituent.

(**defselector** selector) **Top-level Primitive**
selector ::= name

`Defselector` defines name as a selector. This primitive is rarely used explicitly, as all undefined names are assumed to be selectors by default.

(**add-method** selector class value) **Top-level Primitive**

`Add-method` associates a method with its class and selector. When selector is called with a receiver object that belongs to class, value is called. Value should be a function or a method.

(**method** selector class) **Primitive**

`Method` performs the inverse of the `add-method` operation—it returns the method associated with selector and class. If no method is associated with selector and class, `method` returns `nil`.

(**defmethod** selector class (formal-spec*) [return-specs] **Top-level Macro**
{funct-declaration}
body)

`Defmethod` defines a global method with the given selector and class. The rest of the syntax is analogous to `defun`.

When a method is called, the values of the selector and arguments are computed, and the method associated with the selector and the class of the receiver object is found. Of course, this method may have been defined for a superclass of the class of the receiver object (i.e. it may be inherited). It is an error occurs if no such method exists. Otherwise, the process of calling a method is the same as that of calling a function.

A.6. Statements

value ::= statement
expression ::= statement
body ::= statement*

In the definitions below, the non-terminals *value*, *expression*, and *statement* all refer to statements, although *value* usually denotes a side-effect-free statement that is executed for its return value, *expression* denotes a statement that may have side effects but is executed mainly for its return value, and *statement* denotes a statement that is executed mainly for its side effects. A *body* is a sequence of statements executed one after another just like in `begin`; the value of a body is the value of its last statement.

Futures and CFutures

Futures and cfutures (context futures) are the main means of achieving concurrency in Concurrent Smalltalk. Both futures and cfutures are promises to produce some value at a later time. *Forcing* a future means forcing the future to fulfill its promise and return its value. Analogously, *touching* a cfuture forces it to calculate and return its value. A force implies a touch, so a force never returns a cfuture.

There are two main differences between futures and cfutures. These are outlined below:

- Futures are guaranteed not to be forced unless they are explicitly forced, while cfutures are not guaranteed not to be touched—they may be touched at any time at the compiler's and operating system's discretion. In an extreme case, cfutures may be touched as soon as they are created, leading to a sequential implementation of Concurrent Smalltalk (except for futures).

- CFutures are generated by almost all primitive operations, while futures are generated only by the `future` and `lazy-future` primitives and their derivatives.

- CFutures are always *eager*—if left alone, they will tend to evaluate to their values. Normal futures, on the other hand, may be eager or lazy. A *lazy* future may not begin to evaluate to its value until it is forced; if it is never forced, it may never be evaluated.

The rationale behind creating two kinds of futures is to allow the use of cfutures for most tasks where parallelism is desirable but guaranteed parallelism is not necessary for the correct operation of the program. CFutures are intended to be very cheap—they can be created and touched in a few assembly language instructions. Futures, on the other hand, are reserved for the cases like normal-order evaluation where the semantics of delayed evaluation are necessary for the program to run correctly. Futures are much more expensive than cfutures in terms of space and time.

Both futures and cfutures may have values of complicated expressions as their promises. For example, if `(f 3)=30`, `(g 7)=49`, and `(h 30 49)=79`, during the execution of the statement

```
(cset a (h (f 3) (g 7)))
```

a may be computed in arbitrary order, and `f` and `g` need not have returned values by the time the next statement is executed. If a is later touched, it will assume the value `79`.

The semantics become more complicated if the functions `f`, `g`, and `h` have side effects. The order of evaluation of arguments of function calls is undefined and may be parallel, so `f` and `g` may be evaluated in parallel. Furthermore, if `h` is declared `&non-strict` (as many builtins are), the evaluation of `h` may overlap with the evaluation of its arguments. If, say, `h` does not use the value of its second argument until late in its execution, `h` may already be executing while `(g 7)` is still being calculated. Finally, if `h` can return without ever requesting the value of its second argument, `(g 7)` may never be completely evaluated (since cfutures are eager, it will keep evaluating, but the entire program may finish before it is done). A good example of this phenomenon is `(and b false)`, where the program can proceed without ever determining the value of `b`.

Argument Evaluation

Unless a method or a function is declared &non-strict, method and function calls are strict with respect to cfutures but not futures—the arguments of a method or function are guaranteed not to be cfutures when the method or function begins evaluation. For example, assuming no futures are used, in

```
(cset a (h (f 3) (g 7)))
(cset b (k 10))
(touch a)
(cset c (l 10))
```

(f 3) and (g 7) are guaranteed to be done evaluating before (h 30 49) begins evaluating. Also, (f 3), (g 7), and (h 30 49) are guaranteed to be done evaluating before the evaluation of (l 10) is started. However, (k 10) can be evaluated concurrently with any of (f 3), (g 7), (h 30 49), or (l 10).

The arguments of functions are evaluated concurrently. This means they may be evaluated sequentially, in parallel, or any combination of the two. Using side effects can sometimes lead to deadlock. For example, suppose that the function `release-lock` releases a global lock and `acquire-lock` waits until the lock is released and then acquires it. Further, suppose that `global-lock` is originally acquired. Then, the expression

```
(h (release-lock global-lock) (acquire-lock global-lock))
```

can lead to deadlock because the implementation might choose to evaluate `acquire-lock` sequentially before `release-lock`.

Concurrent evaluation order is also distinct from an arbitrary sequential order. For example, suppose that `c` is a local variable with an initial value of 0 and consider the value of the expression

```
(cset a (+ (cset c (+ 1 c)) (cset c (+ 1 c))))
(touch a)
(touch c)
```

Under sequential evaluation of arguments, the final value of `a` would always be 3 and the final value of `c` would always be 2 when this expression completes. Under concurrent evaluation of arguments, the final value of `c` could be 1 if, say, both increments were done before either assignment to `c`. In this case, `a` would get the value 2.

(touch expression**)** **Primitive**
(touch expression***)** **Macro**

If expression is not a cfuture, `touch` does nothing. Otherwise, `touch` waits until the value of the cfuture is available and then returns that value. It should be kept in mind that if `touch` is used in a subexpression, other subexpressions may or may not continue evaluating while this `touch` is waiting. Also, a `touch` in a subexpression does not guarantee that the entire expression will not yield a cfuture, as is demonstrated in one of the examples above.

If more than one expression is specified, `touch` touches them all and returns the value of the last one. If no expressions are specified, `touch` returns `nil`.

`Touch` does not have any effect on futures.

(**force** expression) **Primitive**
!expression **Macro**
(**force** expression*) **Macro**

If expression is not a future or a cfuture, `force` does nothing. Otherwise, `force` waits until the value of the future or cfuture is available and then returns that value. That value is guaranteed not to be a future or a cfuture.

If more than one expression is specified, `force` forces them all and returns the value of the last one. If no expressions are specified, `force` returns `nil`. The !expression form is a shorthand for `(force` expression`)`.

(**future** expression) **Primitive**
(**lazy-future** expression) **Primitive**

`Future` and `lazy-future` both return futures that promise to evaluate expression when `forced`. The futures are guaranteed to evaluate in parallel with all other processes unless explicitly synchronized. `Future` and `lazy-future` differ in that `future` begins evaluating its expression immediately, while `lazy-future` waits until it is `forced` before it starts evaluating its expression. In any case, expression is evaluated at most once, no matter how many times it is `forced`.

Caveats: The actual time when a future is forced is sometimes rather fuzzy, especially in the presence of inlined primitives and side-effect-free functions, so the guarantee in the previous paragraph may not apply in the code just before a future is forced (the extent of this fuzzy section of code is still to be determined). Also, futures should not return objects of classes that can be inlined—doing this may force the future immediately at any point. These caveats should not present problems unless futures have intricate side effect dependencies.

Application Statement

(funct arg*) **Primitive**
funct ::= expression
arg ::= expression

The first item of an application statement is either a method selector or a function. If it is a selector, the method corresponding to the selector and the class of the first argument is called using the arguments provided. If it is a function, it is applied to the specified arguments. The first item can also be any expression that evaluates to an object of type `funct`. The value of the application statement is either the return value or a cfuture promising that value.

The order of evaluation of arguments is not specified; in fact, some of them may be (but are not guaranteed to be) evaluated concurrently. The arguments are not guaranteed to be touched before being passed to the funct—some of them may be passed to the funct as futures or even cfutures (However, all user-defined methods and functions not explicitly declared `:non-strict` will touch their arguments before their code begins executing). For example, `(cset a (+ 0 a))` does not touch `a`, and `(and b false)` does not touch `b`.

Type Assertion

(:type expression) **Primitive**

The type assertion statement asserts that the type of expression's value is a subtype of type. It returns expression's value. The compiler is not required to generate an error if expression evaluates to a value that is not a subtype of type, but it may do so.

Variable Bindings

(**clet** (binding-spec*) body) **Primitive**
binding-spec ::= typed-opt-names | (typed-opt-names {variable-declaration} [value])
typed-opt-names ::= opt-name (, opt-name)* [: type]
opt-name ::= name | _
variable-declaration ≡ **&inline** | **&not-inline**

`Clet` creates local variable bindings and evaluates body within the scopes of those bindings. Each name specifies the name of a new variable. Typed-opt-names specifies one or more names separated by commas followed by an optional type. The character _ can be used to indicate an unnamed local variable; unnamed local variables can be used to evaluate the initial value expression without binding a name in the static scope. If type is not present, it defaults to `object`. If the long form of a binding-spec is used, the variables in typed-opt-names can be declared using declarations and can be given an initial value. Value, the initial value is an expression evaluated outside the scope of the `clet`. Each initial value is evaluated only once, even if it is assigned to more than one variable. The new variables are bound concurrently. Their initial values may be evaluated concurrently, and they are not guaranteed to be touched by the time body begins executing—in body the new variables may still contain cfutures.

A variable may be declared `&inline` or `&not-inline`. These are hints to the compiler that the variable's object should be placed inline or on the heap. These hints only apply if the variable's type is an inline class. The compiler is free to ignore these declarations.

The value returned by a `clet` is the value returned by the last statement in body.

(**let** (binding-spec*) body) **Macro**

`Let` is the same as `clet` except that all newly-bound variables are touched before body begins executing. As with `clet`, the initial values are evaluated concurrently.

(**cset** name expression) **Primitive**

`Cset` sets the variable name to expression. The variable gets either the touched value of expression or a cfuture promising to evaluate expression. The value returned by a `cset` is the value of expression.

(**set** name expression) **Macro**

`Set` sets the variable name to the value of expression. The value is touched before it is assigned to the variable, so the variable will not contain a cfuture or a future after this statement. The value returned by a `set` is the touched value of expression.

(**cas** name comparison replacement) **Primitive**
comparison ::= expression
replacement ::= expression

`CAS` (compare-and-set) is an atomic[1] operation that checks whether the value of variable name is `eq` to the value of comparison. If so, the value of replacement is stored in variable name and `cas` returns `true`; otherwise, `cas` returns `false`. The value of variable name is never a cfuture when `cas` completes.

---

[1] In the current implementation, in order for `cas` to be atomic, neither name nor replacement can be a future. If replacement could be a future, it should be forced before a `cas` is done. There is no easy solution if name could be a future. Fortunately, there is usually little reason to store a future in a semaphore.

Multiple Values

The constructs below are used for receiving multiple values from methods and functions. Multiple values may not be supported by all implementations of Concurrent Smalltalk.

(mv-cset (name*) (funct arg*)) **Optional Primitive**

MV-cset sets the variables name* to the multiple values returned by (funct arg*). The variables get either the touched return values of expression or cfutures promising to evaluate them. Some of the return values may be available before others. Mv-cset returns nil.

(mv-set (name*) (funct arg*)) **Optional Macro**

MV-set is just like mv-cset except that it touches all variables in name* before continuing.

(mv-clet (mv-binding-spec*) (funct arg*) body) **Optional Macro**
mv-binding-spec ::= typed-opt-names | (typed-opt-names {variable-declaration})
typed-opt-names ::= opt-name (, opt-name)* [: type]
opt-name ::= name | _
variable-declaration ≡ **&inline** | **&not-inline**

(mv-let (mv-binding-spec*) (funct arg*) body) **Optional Macro**

MV-clet and mv-let are just like clet and let except that they initialize the new variables to the values returned by (funct arg*).

Syntactic Sugar

[arg*] **Macro**

This form is equivalent to (get arg*).

(cset (funct arg*) expression) **Macro**

When the first argument of a cset is a function or a method call, cset is desugared into another function or a method call. The above forms are converted to (funct' arg* expression), where the identifier funct' is obtained by appending the characters cput- to the beginning of the identifier funct, unless:

funct is get, in which case funct' is cput;
funct is get-x, in which case funct' is cput-x (x is any sequence of characters);
funct is put, put-x, cput, cput-x, cap, or cap-x, in which case an error occurs.

Funct must be a function name or a method selector. It may not be an expression or a variable reference. (funct arg*) may, however, be a macro or contain macros; these macros are expanded before the above conversion takes place.

For example, `(cset (first sequence) 3)` is converted to `(cput-first sequence 3)`, while `(cset [big-array 7] 12)` is converted to `(cput big-array 7 12)`.

(set (funct arg*) expression) **Macro**

When the first argument of a set is a function or a method call, set is desugared into another function or a method call. The above forms are converted to (funct' arg* expression), where the identifier funct' is obtained by appending the characters put- to the beginning of the identifier funct, unless:

funct is get, in which case funct' is put;
funct is get-x, in which case funct' is put-x (x is any sequence of characters);
funct is put, put-x, cput, cput-x, cap, or cap-x, in which case an error occurs.

Funct must be a function name or a method selector. It may not be an expression or a variable reference. (funct arg*) may, however, be a macro or contain macros; these macros are expanded before the above conversion takes place.

For example, `(set (first sequence) 3)` is converted to `(put-first sequence 3)`, while `(set [big-array 7] 12)` is converted to `(put big-array 7 12)`.

(cas (funct arg*) comparison replacement) **Macro**

When the first argument of a `cas` is a function or a method call, `cas` is desugared into another function or a method call. The above form is converted to (funct' arg* comparison replacement), where the identifier funct' is obtained by appending the characters `cap-` (compare-and-put) to the beginning of the identifier funct, unless:

funct is `get`, in which case funct' is `cap`;
funct is `get-x`, in which case funct' is `cap-x` (x is any sequence of characters);
funct is `put`, `put-x`, `cput`, `cput-x`, `cap`, or `cap-x`, in which case an error occurs.

Funct must be a function name or a method selector. It may not be an expression or a variable reference. (funct arg*) may, however, be a macro or contain macros; these macros are expanded before the above conversion takes place.

Flow of Control

(begin body) **Primitive**

`Begin` evaluates the statements in body sequentially, touching each one except the last before it begins the next, and returns the untouched value returned by the last one. If there are no statements in body, `begin` returns `nil`.

(nconcurrently statement*) **Macro**
(concurrently statement*) **Macro**
?statement **Macro**

These macros evaluate the statements in statement* concurrently and return `nil`. `Concurrently` waits until all statements have finished executing before returning, while `nconcurrently` does not. ?statement is an abbreviation for (`nconcurrently` statement).

(nparallel statement*) **Macro**
(parallel statement*) **Macro**

These macros evaluate the statements in statement* in parallel and return `nil`. `Parallel` waits until all statements have finished executing before returning, while `nparallel` does not. The parallelism is guaranteed, which makes `parallel` a much more expensive statement than `concurrently`. In most cases `concurrently` should be used instead unless parallel semantics are explicitly required.

(if test consequent [alternative]) **Primitive**
test ::= expression
consequent ::= expression
alternative ::= expression

`If` evaluates the test expression, which must return either `true` or `false`. If it returns `true`, the consequent expression is evaluated and its value returned; otherwise, the alternative expression, if any, is evaluated and its value returned. `If` is not guaranteed to touch the test value. However, it is guaranteed to evaluate only the appropriate arm of the conditional.

Loops

(while test body) **Macro**
test ::= expression

`While` evaluates the test expression, which must return either `true` or `false`. As long as it returns `true`, body is evaluated and test reevaluated. When test evaluates to `false`, `while` returns `nil`.

`(repeat body until test)` **Macro**
test ::= expression

`Repeat` first evaluates body and then the test expression, which must return either `true` or `false`. As long as it returns `false`, `repeat` goes back to evaluating body. When test evaluates to `true`, `repeat` returns `nil`.

Primitive Control

`(block continuation body)` **Macro**

`Block` is just like begin except that it allows the use of `return` and `reply` statements to leave it. The statements in body are evaluated as in a `begin`. Continuation specifies the block's continuation for use in `return` and `reply` statements.

`(loop continuation body)` **Macro**

`Loop` defines a loop body. The statements in body are evaluated as in a `begin`, except that after the last statement in body has been evaluated, the first statement is evaluated again, and so on. The loop does not terminate unless an explicit `return` or `reply` statement is encountered. Continuation specifies the loop's continuation for use in `return` and `reply` statements.

Returning Values

Since the last expression in the method code is implicitly returned to `continuation`, the statements below are necessary only if it is desired to return a value from the middle of a method or function, if a `block` or `loop` should be terminated, if multiple values are being returned, or if a value is returned to a continuation with a name other than `continuation`. `Reply` and `exit` should be used with caution, as `exit` may cause the caller to hang, while `reply` may cause the caller to crash if two replies are inadvertently sent. *Care must be taken to reply to each continuation at most once*—sending a second reply to a continuation will almost certainly cause a system crash, and it is quite difficult to protect the system against this type of error. When using `reply` it is important to remember that there is an implicit `reply` of the last expression in the method code to `continuation`.

Continuations

Continuations are introduced by `lambda`, `method-lambda`, `defun`, `defmethod`, `block`, `loop`, `future`, `lazy-future`, `parallel`, and `nparallel`. The continuations defined by `future`, `lazy-future`, `parallel`, and `nparallel` are not externally accessible. `Lambda`, `method-lambda`, `defun`, and `defmethod` define the default continuation `continuation` unless told otherwise. They also reply to `continuation` if allowed to complete executing without an intervening `exit`. Thus, care must be taken when using nested function and method definitions to make sure that `reply` and `return` reply to the right continuation.

Continuation manipulation can become quite complicated, and not all features have to be supported by all implementations. A minimal implementation only has to allow replying to the innermost construct that defines continuations; hence, an implementation may restrict non-local replies. Furthermore, an implementation does not have to support replying out of a `future`, `lazy-future`, `parallel`, or `nparallel` statement, since these also introduce continuations. A more sophisticated implementation may allow replies to all continuations accessible in the current lexical scope. Finally, an advanced implementation may choose to make continuations first-class values of class `#:continuation` and allow them to be stored in variables.

`(exit)` **Primitive**

`Exit` is a statement that hangs, never returning a value. In most cases `exit` can be thought of as exiting the current method or function, but it does not necessarily do so if used in a `cset`, `concurrently`, `nconcurrently`, `parallel`, `nparallel`, `block`, `loop`, `future`, or `lazy-future` statement, `let` or `clet` bindings, or some other statement that permits parallel execution without synchronization.

(reply expression) **Macro**
(reply (continuation expression)*) **Primitive**

The first variant of `reply` evaluates expression and sends its value to `continuation`. Execution then proceeds with the next statement of the current method, if any. `Reply` is not strict—it may reply a future or a cfuture. The value of a `reply` statement is `nil`.

The second variant of `reply` is used to return values to named continuations. The `reply` takes an even number of arguments; within each pair, the first argument is the continuation name and the second one its value.

(return expression) **Macro**
(return (continuation expression)*) **Macro**

`Return` is equivalent to a `reply` followed by an `exit`—the values of the expressions are sent to the caller, and the execution of the method or function terminates subject to the caveats in the `exit` statement description.

(return-value-expected?) :boolean **Function**
(return-value-expected? continuation) :boolean **Function**

`Return-value-expected?` returns `true` if the caller of the method or function is expecting a reply for continuation (or `continuation` if continuation is not specified). It is not guaranteed to return `false` otherwise, so an implementation that always returns `true` is acceptable.

A.7. Built-in Methods and Functions

Built-in Classes

Built-in classes are provided for reasons of efficiency and convenience. Many methods on built-in classes are compiled into single assembly language instructions instead of method calls, improving their speed greatly. Other built-in classes may be defined by methods written in assembly language and linked with the programs generated by the compiler. Arrays may be defined this way. The built-in classes are listed in Table A-2, and their hierarchy is shown in Figure A-2.

Table A-2. Built-in Classes

Class	Metaclass	Values
`Null`[†]	`Primitive-Class`	`Nil`
`Symbol`	`Primitive-Class`	Symbols, including `nil`, but not `true` and `false`
`True`[†]	`Primitive-Class`	The boolean `true`
`False`[†]	`Primitive-Class`	The boolean `false`
`Boolean`	`Primitive-Class`	The booleans `true` and `false`
`Character`	`Primitive-Class`	ASCII characters
`Small-Integer`	`Primitive-Class`	Integers representable in a machine word[1]
`Large-Integer`	`Primitive-Class`	Integers not representable as `Small-Integers`
`Integer`	`Primitive-Class`	Arbitrary-sized integers
`Float`	`Primitive-Class`	Floating-point numbers[2]
`Real`	`Standard-Class`	Real numbers
`Number`	`Standard-Class`	Arbitrary numbers
`Magnitude`	`Standard-Class`	Numbers, characters, and booleans
`Primitive-Class`	`Primitive-Class`	Primitive classes defined by Concurrent Smalltalk
`Standard-Class`	`Primitive-Class`	Standard (non-distributed) classes
`Distributed-Class`	`Primitive-Class`	Distributed classes
`Class`[†]	`Primitive-Class`	General classes
`Function`	`Primitive-Class`	Functions, methods, and closures
`Funct`	`Primitive-Class`	Functions, methods, closures, and method selectors
`System-stream`	`Primitive-Class`	System-defined streams
`Stream`	`Standard-Class`	Sources of input or destinations for output
`Simple-Lock`	`Primitive-Class`	Very cheap and simple locks
`Queueing-Lock`	`Primitive-Class`	More expensive locks that queue pending tasks
`Lock`	`Standard-Class`	General locks
`Integer-Array`	`Primitive-Class`	Small arrays of integers
`String`	`Primitive-Class`	Small arrays of characters
`Boolean-Array`	`Primitive-Class`	Small arrays of booleans
`Simple-Array`	`Primitive-Class`	Small arrays of arbitrary objects
`Array`	`Standard-Class`	Arrays of arbitrary objects
`Collection`	`Standard-Class`	Indexed collections of objects
`Distobj`	`Distributed-Class`	All distributed objects
`Object`	`Standard-Class`	All first-class values

The *metaclass* of a class is the class of the class object itself. Metaclasses govern certain aspects of class behavior such as inheritance and the action of `new`. Only classes having `standard-class` or `distributed-class` as a metaclass permit user-defined subclasses. At the implementation's discretion some classes with `primitive-class` as a metaclass may

---

[†] This class name conflicts with another global name, so it has to be preceded with `#:` whenever it is used.

[1] Currently a machine word is 32 bits, so the `small-integer` range is -2147483648 to 2147483647.

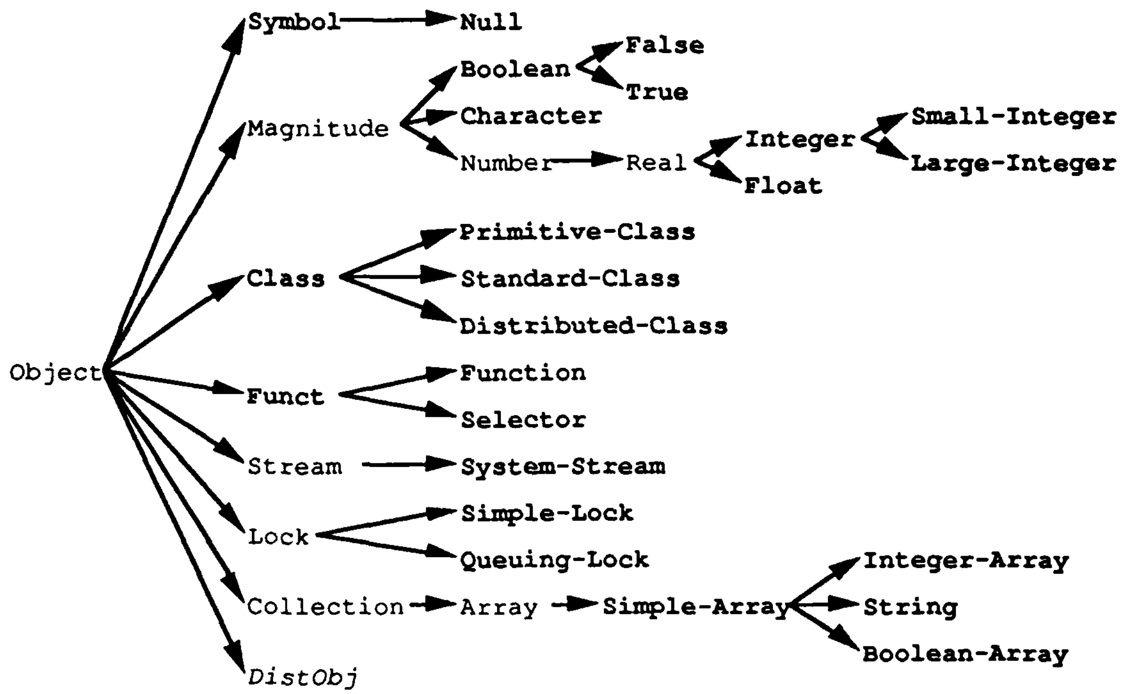


Figure A-2. Hierarchy of built-in classes

The superclasses are shown to the left of their subclasses. All classes are subclasses of `object`. Classes with metaclass `primitive-class` are shown in bold, classes with metaclass `standard-class` are shown in standard type, and the one class with metaclass `distributed-class` is shown in italic. User-defined classes may be defined as subclasses of any of the classes having `standard-class` or `distributed-class` as a metaclass.

actually be instances of `standard-class` or `distributed-class`, but portable programs should not rely on this.

Built-in Methods

Built-in methods are provided for the basic arithmetic and logical operations. The methods are explained in the following sections. Since some built-in method calls compile into assembly language instructions, some restrictions are necessary on the use of their selectors. Specifically, if any other methods are defined using the selectors in Table A-3, they must obey the identities listed in Table A-4.

Redefining Restricted Selectors

If a restricted selector is called with an argument that is not one of the built-in classes it recognizes, the actual method for the class is found and executed, *possibly after some of the identities in Table A-4 have been applied.* Thus, it is possible to define a class of type, say, `complex`, and define a method * for numbers of that type. That method will be called whenever * is used on a number of type `complex`, regardless of whether that number is the first or second argument. If both complex numbers and quaternions are defined, the complex * method should be prepared to handle a quaternion as the second argument, while the quaternion * method should be prepared to handle a complex number as the second argument. The `reverse` methods have been added to handle the case of a non-built-in object being the second argument of a noncommutative operation. The <>, <=, and >= methods should never be redefined, as they are never called. Redefine =, >, and < instead.

---

[2] Floating point numbers may not be implemented in all Concurrent Smalltalk implementations.

The associative restricted selectors allow an arbitrary number of arguments; they compile into pairwise invocations of the corresponding methods. The grouping order is not specified.

Methods declared with restricted selectors should not have side effects.

The identities in Table A-3 have been carefully selected to allow efficient implementation of primitive operations without sacrificing functionality. Some identities have been omitted on purpose. For example, * does not have to be commutative in general, nor does `(* a 0)` have to equal `0`. Not requiring these identities allows * to be used to multiply quaternions and matrices.

The restricted selectors `not`, `and`, `or`, and `xor` may not be distinguishable from `lognot`, `logand`, `logor`, and `logxor` on all implementations. Redefining these should be avoided; if they must be redefined, only one set should be redefined.

Table A-3. Restricted Selectors

```
not and or xor lognot logand logor logxor
< <= > >= = <>
neg + - reverse-- * // reverse-// mod reverse-mod
ash reverse-ash integer-length
```

Table A-4. Identities among Primitive Methods

- + is associative and commutative.
- 0 is an identity for +.
- (- a b) = (reverse-- b a).
- (- a b) = (+ a (neg b)).
- (neg (neg a)) = a.
- * is commutative with scalar constants and associative.
- 1 is an identity for *.
- (* a -1) = (neg a).
- (* a 2^e) = (ash a e).
- (// a (neg b)) = (neg (// a b)).
- (// a 2^e) = (ash a -e).
- (// a b) = (reverse-// b a).
- (mod a (neg b)) = (neg (mod a b)).
- (mod a b) = (reverse-mod b a).
- (ash a b) = (reverse-ash b a).
- (ash 0 a) = 0.
- (ash a 0) = a.
- (not (not a)) = a.
- and, or, and xor are associative and commutative.
- (and a false) = false.
- (and a true) = a.
- (or a false) = a.
- (or a true) = true.
- (xor a false) = a.
- (xor a true) = (not a).
- (lognot (lognot a)) = a.
- logand, logor, and logxor are associative and commutative.
- (logand a 0) = 0.
- (logand a -1) = a.
- (logor a 0) = a.
- (logor a -1) = -1.
- (logxor a 0) = a.
- (logxor a -1) = (lognot a).
- (< a b) = (not (>= a b)).
- (> a b) = (not (<= a b)).
- (= a b) = (not (<> a b)).
- (< a b) = (> b a).
- (<= a b) = (>= b a).
- (= a b) = (= b a).
- (<> a b) = (<> b a).

A.8. System and Object Operations

Objects

`(new c:standard-class) :object` **Method**

`New`, when applied to a standard class, creates and returns a new instance object of the specified class. The object is not initialized. Some implementations may restrict the new argument to be a constant expression.

Copiers

`(deep-copy o:object) :object` **Method**

`Deep-copy` returns a copy of the object. Any of the object's instance variables are also recursively copied using `deep-copy`. If the class of the object is immutable, `deep-copy` may just return the object it received. `Deep-copy` may fail to terminate on circular object references.

`(shallow-copy o:object) :object` **Method**

`Shallow-copy` returns a copy of the object without copying any of the object's instance variables. If the class of the object is immutable, `shallow-copy` may just return the object it received.

`(copy o:object) :object` **Method**

`Copy` is the most appropriate copying routine for a given object. It defaults to `shallow-copy`.

Deallocators

In addition to waiting for garbage collection, the following methods can be used to explicitly deallocate the storage for an object. Accessing an object after it has been deallocated causes an error.

`(deep-dispose o:object) :null` **Method**

`Deep-dispose` deallocates the object's storage. Any of the object's instance variables are also recursively disposed using `deep-dispose`. `Deep-dispose` should not be used on circular or multiple object references.

`(shallow-dispose o:object) :null` **Method**

`Shallow-dispose` deallocates the object's storage without disposing any of the object's instance variables.

`(dispose o:object) :null` **Method**

`Dispose` is the most appropriate deallocating routine for a given object. It defaults to `shallow-dispose`.

Class Inquiries

`(class-of o:object) :class` **Method**

`Class-of`, when applied to an object, returns its class.

(class-kind? o:object c:class) :boolean **Method**
(class-member? o:object c:class) :boolean **Method**

`Class-kind?` returns a boolean value that specifies whether the given object is an instance of the given class or one of its subclasses. `Class-member?` is just like `Class-kind?` except that it returns `true` only if the object is a direct instance of the given class.

(subclass? c1, **c2**:class) :boolean **Method**

`Subclass?` returns `true` if `c1` is a subclass of `c2` and `false` otherwise.

A.9. Distributed Objects

distobj **Class**

`Distobj` is the distributed object class.

Group and Constituents

A distributed object consists of a *group* name and one or more *constituent objects.* The constituent objects act just like normal objects except that they inherit methods and instance variables from the class `distobj` and they respond to the `group` and `get-group` messages. A group name indicates the entire collection of distributed objects. When a method is called on the group name, it is processed by one of the distobj's constituent objects, as though the method were called on that constituent object. The identity of the constituent object receiving the message is left unspecified; implementations are encouraged to heuristically pick different constituent objects for different calls to the group, thereby facilitating concurrency for distributed object operations. When a constituent object is processing a method, `self` is the constituent object, not the group name.

Creation

(**new** c:distributed-class **n-constituents**:integer) :distobj **Method**

`New`, when applied to a distributed class, creates and returns a new distributed object of the specified class with the given logical number of constituents. The constituents are not initialized.

The distributed object that is created may contain more constituents than n-constituents. The runtime system determines an appropriate physical number of constituents for the distributed object that is at least as large as n-constituents. The additional constituents should be prepared to respond to messages sent to the distributed object.

Operations

(**co** o:distobj **n**:integer) :distobj **Method**

`Co` returns the nth constituent object of the distributed object. `O` can be either the group object or any of its constituents. `N` must be between 0 and the physical number of constituent objects in the distobj minus one.

(**logical-limit** o:distobj) :integer **Method**

`Logical-limit` is the logical number of constituent objects in the distributed object.

(**physical-limit** o:distobj) :integer **Method**

`Physical-limit` is the physical number of constituent objects in the distributed object. The constituent objects are numbered between 0 and `physical-limit` minus one, inclusive. `Physical-limit` is never less than `logical-limit`.

(**index** o:distobj) :integer **Method**

`Index` is the number of a particular constituent object in a distributed object. `Index` ranges between 0 and `physical-limit` minus one, inclusive.

`group`:distobj **`Instance Variable`**
`(group` o:distobj**`)`** :distobj **`Method`**
`(get-group` o:distobj**`)`** :distobj

`Group` is the inverse of `co`—it returns the group object of the given distributed object. `o` can be either the group object or any of its constituents; if `o` is already a group object, `group` just returns it. `Get-group` is functionally equivalent to `group`; it is provided to avoid name conflicts with the `group` variable inside distributed object methods.

A.10. Logical and Arithmetic Operations

Comparisons

(eq o1, o2:object) :boolean	**Function**
(neq o1, o2:object) :boolean	**Function**

`Eq` returns `true` if the two objects are indistinguishable—there is no legal way of distinguishing `o1` from `o2`. For mutable objects this means that `o1` and `o2` are the same object. For immutable objects, `eq` may in addition return `true` if `o1` and `o2` are different objects that contain the same data.

`Eq` may return unusual results for inline classes—an instance object of an inline class is not necessarily `eq` to itself, but `eq` will never return `true` on distinguishable objects.

`Neq` is the logical negation of `eq`.

(= o1, o2:object) :boolean	**Method**
(<> o1, o2:object) :boolean	**Method**

These comparisons return `true` if `o1` is equal to or not equal to `o2`, respectively. Equality means numeric equality for numbers. It defaults to `eq` or `neq` for other objects, but the `=` method can be overridden to specify a different criterion for a particular class.

(< m1, m2:magnitude) :boolean	**Abstract Method**
(<= m1, m2:magnitude) :boolean	**Abstract Method**
(> m1, m2:magnitude) :boolean	**Abstract Method**
(>= m1, m2:magnitude) :boolean	**Abstract Method**

These comparisons return `true` if `m1` is less than `m2`, `m1` is less than or equal to `m2`, `m1` is greater than `m2`, or `m1` is greater than or equal to `m2`, respectively. For the purposes of comparison, `false` is considered to be less than `true`. It is an error to use <, <=, >, or >= to compare an object from one direct subclass of `magnitude` with one of another direct subclass of `magnitude`—a `boolean` cannot be compared with an `integer`.

(max m1, m2:magnitude) :magnitude	**Method**
(min m1, m2:magnitude) :magnitude	**Method**

`Max` returns the greater of m1 and m2, while `min` returns the lesser one. Both `max` and `min` use one of the comparison operations above to decide which is the greater or lesser, and the same caveats as above apply.

Logical Operations

(not b:boolean) :boolean	**Method**

`Not` returns the logical negation of `b`.

(and (b:boolean)*) :boolean	**Method**

`And` returns the logical AND of its arguments. If no arguments are specified, `and` returns `true`.

(or (b:boolean)*) :boolean	**Method**

`Or` returns the logical inclusive OR of its arguments. If no arguments are specified, `or` returns `false`.

(xor (b:boolean)*) :boolean **Method**

`Xor` returns the logical exclusive OR of its arguments. If no arguments are specified, `xor` returns `false`.

(sc-and (b:boolean)*) :boolean **Macro**
(sc-or (b:boolean)*) :boolean **Macro**

These are short-circuit versions of `and` and `or`. They evaluate arguments sequentially from left to right only as far as necessary for the answer to be unambiguously determined.

Arithmetic Operations

For most binary arithmetic operations, the class of the result is the class of the most general argument. For example, if two `integers` are added, the result is an `integer`, but if an `integer` and a `float` are added, the result is a `float`. User-defined classes may define other numeric subclasses, in which case they have to handle appropriate coercions themselves—if a `number` is added to a member of a user-defined subclass of `number`, the + method for the user-defined subclass will have to dispatch on the type of its second argument.

(zero? n:number) :boolean **Method**

`Zero?` returns `true` if `n` is zero and `false` otherwise.

(neg n:number) :number **Abstract Method**

`Neg` returns the negation of `n`. The class of the result value is the same as the class of `n`.

(+ (n:number)*) :number **Abstract Method**

\+ returns the sum of its arguments. If no arguments are specified, + returns 0.

(- n1, n2:number) :number **Abstract Method**

\- returns the difference of its arguments, `n1-n2`.

(* (n:number)*) :number **Abstract Method**

* returns the product of its arguments. If no arguments are specified, * returns 1.

(/ n1, n2:number) :number **Abstract Method**

/ returns the quotient of its arguments, `n1/n2`. If `n1` and `n2` are both `integers` and `n1` is not exactly divisible by `n2`, the result is a `float`. If `n2` is zero, either an error occurs or some representation of infinity is substituted as an answer.

(// n1, n2:integer) :integer **Method**

// returns the integer quotient of its arguments rounded towards $-\infty$, $\lfloor n1/n2 \rfloor$. If `n2` is zero, either an error occurs or some representation of infinity is substituted as an answer. Having // round towards $-\infty$ allows the use of `ash` to divide when the divisor is an integral power of two.

(mod n1, n2:integer) :integer **Method**

`Mod` returns the nonnegative remainder of dividing `n1` by `n2`, $n1-n2*\lfloor n1/n2 \rfloor$. If `n2` is zero, either an error occurs or some representation of an indeterminate number is substituted as an answer. Having `mod` return the nonnegative remainder allows the use of `logand` to find the remainder when the divisor is an integral power of two. When the remainder is nonzero, its sign is always the same as the sign of the divisor `n2`. Also, `(+ (mod n1 n2) (* n2 (// n1 n2))) ≡ n1`.

(ash n1:integer **n2**:integer**)** :integer **Method**
(ash n1:float **n2**:integer**)** :float **Method**

Ash returns n1 multiplied by two raised to the n2th power, $n1*2^{n2}$. If n1 is a float, no rounding takes place; however, if n1 is an integer and n2 is negative, the result is rounded towards $-\infty$.

(integer-length n:integer**)** :integer **Method**

Integer-length returns the bit "size" of n1. For positive n this is $\lceil \log_2(n+1) \rceil$, while for negative n it is equal to $\lceil \log_2(-n) \rceil$.

Bitwise Logical Operations

(lognot b:boolean**)** :boolean **Method**
(logand (b:boolean)+**)** :boolean **Method**
(logor (b:boolean)+**)** :boolean **Method**
(logxor (b:boolean)+**)** :boolean **Method**

(lognot b:integer**)** :integer **Method**
(logand (b:integer)***)** :integer **Method**
(logor (b:integer)***)** :integer **Method**
(logxor (b:integer)***)** :integer **Method**

These methods perform bitwise logical operations. When called on booleans, they perform the same operations as not, and, or, and xor, respectively. When called on integers, they perform the corresponding operations bitwise on semi-infinite two's complement representations of the integers, treating 0 as false and 1 as true. The integers do not have to be internally stored in the two's complement form; all that is necessary is that these operations act as if they were. When supplied with no arguments, logand returns -1, while logor and logxor return 0.

A.11. Locks

Locks are used to synchronize processes. A lock can be *acquired* by only one process at a time, and the acquiring operation is atomic. After a process has acquired a lock, it can proceed to perform whatever exclusive operations it wants to do. When it is done, it should *release* the lock to make it available again. If a process attempts to acquire a lock that is busy (acquired), it will wait until the lock is available.

Two built-in lock classes are provided: `simple-lock` and `queueing-lock`. `Simple-lock` is a very cheap and fast implementation intended for situations in which a lock is not acquired for long periods of time and there is little contention for the lock. `Simple-locks` are adequate for most purposes. `Queueing-locks` are heavy-duty locks for use in situations where there may be significant contention for a lock.

Lock Operations

(new-simple-lock) :simple-lock **Function**
(new-queueing-lock) :queueing-lock **Function**

`New-simple-lock` creates a new simple lock, while `new-queueing-lock` creates a new queueing lock. The lock is initially available.

(init l:simple-lock**)** :null **Method**
(init l:queueing-lock**)** :null **Method**

`Init` reinitializes the lock, making it available regardless of its previous state.

(acquire l:lock**)** :*null* **Abstract Method**

`Acquire` acquires the lock. If the lock is busy, `acquire` waits until the lock is available before acquiring it and returning.

(release l:lock**)** :null **Abstract Method**

`Release` releases the lock. If the lock is already available, `release` signals an error.

(busy? l:lock**)** :boolean **Abstract Method**

`Busy?` returns `true` if the lock is busy and `false` otherwise.

(with-locks ((l:lock)*) body**)** **Macro**

`With-locks` first acquires all of the locks listed, in the order in which they are listed, then evaluates body, and finally releases all of the locks. It returns the value of body.

A.12. Strings and Arrays

Strings and arrays are the primitive data structures for keeping track of indexed collections of data. All primitive strings and arrays are subclasses of the class `array`. The subclasses of class array can be implemented as arrays, but implementations are encouraged to pack `integer-arrays`, `strings`, and `boolean-arrays` to conserve space and time.

Creating Arrays

(**new-simple-array size**:integer) :simple-array **Method**

`New-simple-array` creates a new simple array of arbitrary objects. `Size` specifies the number of elements in the array; the elements are numbered 0 through `size-1`. The array's elements are not initialized.

(**new-integer-array size**:integer **low, high**:integer) :integer-array **Method**

`New-integer-array` creates a new array of integers in the range between `low` and `high`, inclusive. `Low` must be less than or equal to `high`. `Size` specifies the number of elements in the array; the elements are numbered 0 through `size-1`. The array's elements are not initialized.

(**new-string size**:integer) :string **Method**

`New-string` creates a new array of characters, also called a string. `Size` specifies the number of elements in the array; the elements are numbered 0 through `size-1`. The array's elements are not initialized.

(**new-boolean-array size**:integer) :boolean-array **Method**

`New-boolean-array` creates a new array of booleans. `Size` specifies the number of elements in the array; the elements are numbered 0 through `size-1`. The array's elements are not initialized.

Operations on Entire Arrays

(**fill a**:array **value**) :array **Abstract Method**

`Fill` destructively writes `value` to every element of the given array. If the array is an `integer-array`, a `string`, or a `boolean-array`, the value must have the correct type and, in the case of `integer-array`, it must be in the range specified when the array was created; otherwise, the results are unspecified. `Fill` returns the updated array.

(**init a**:array **f**:funct) :array **Abstract Method**

`Init` concurrently calls `f` on integers between 0 and the size of `a` minus one, inclusive, and stores the results in the corresponding elements of `a`. If `f` or any other function tries to read an element of `a`, it will wait until the value is available. It is an error for `f` or any other function to try to alter the values of elements of `a` before `init` returns. `Init` returns the a array after all calls to `f` have returned.

(**map src**:array **dst**:array **f**:funct) :array **Abstract Method**

`Map` concurrently calls `f` on each element of the `src` array and stores the results in the corresponding elements of the `dst` array. The sizes of the two arrays must be equal. If `src` is a `simple-array`, so must be `dst`. `Src` and `dst` may be the same array. If `f` or any other function tries to read an element of the `dst` array, it will wait until the value is available. It is an error for `f` or any other function to try to alter the values of elements of the `dst` array before `map` returns. `Map` returns the `dst` array after all calls to `f` have returned.

(for-each a:array **f**:funct) :array **Abstract Method**
(nfor-each a:array **f**:funct) :array **Abstract Method**

Both of the above methods concurrently call `f` on each element of the array and then return the array without modifying it. `Nfor-each` does not wait until any of the calls to `f` return, while `for-each` does.

Accessing Arrays

[a:array **pos**:integer**]** :object **Abstract Method**
(get a:array **pos**:integer) :object

`Get` returns the element at position `pos` of the given array. `Get` signals an error if `pos` is outside the bounds of the array. The results of accessing an uninitialized element are unspecified.

(set [a:array **pos**:integer**] value**:object) :array **Abstract Method**
(put a:array **pos**:integer **value**:object) :array

`Put` destructively writes `value` at position `pos` of the given array. `Value` is not touched. `Put` signals an error if `pos` is outside the bounds of the array. If the array is an `integer-array`, a `string`, or a `boolean-array`, the value must have the correct type and, in the case of `integer-array`, it must be in the range specified when the array was created; otherwise, the results are unspecified. `Put` returns the updated array.

(size a:array) :integer **Abstract Method**

`Size` *returns the size of the array*, as specified when the array was created.

A.13. Input and Output

Streams

Streams are sources and sinks of data. A stream is usually a connection to a terminal or to a file, but other uses of streams are possible. Concurrent Smalltalk defines a general class `stream` as well as a specific implementation of streams, `system-stream`. Other user-defined stream classes may be defined as subclasses of `stream`.

Operations on General Streams

Reading

(read-stream-char s:stream) :object **Abstract Method**

`Read-stream-char` reads a character from stream s and returns it. If there is no more input available on the stream, `read-stream-char` returns `nil`.

(read-stream-line s:stream) :object **Abstract Method**

`Read-stream-line` atomically reads a line from stream `s` and returns it in the form of a string (without the trailing line terminator). If there is no more input available on the stream, `read-stream-line` returns `nil`.

(read-stream s:stream) :object **Abstract Method**

`Read-stream` reads some representation of a Concurrent Smalltalk object from stream `s` and returns it. If there is no more input available on the stream, `read-stream` returns the constant `end-of-file`.

end-of-file:object **Constant**

This unique constant is returned when `read-stream-object` encounters an end of file.

(stream-char-ready? s:stream) :boolean **Abstract Method**

`Stream-char-ready?` returns `true` if a character is ready to be read from stream `s`. It is not guaranteed to return `false` otherwise, so an implementation that always returns `true` is acceptable.

Writing

(write-stream-char s:stream **ch**:character) :null **Abstract Method**

`Write-stream-char` writes character `ch` onto stream `s`.

(write-stream-string s:stream **string**:string) :null **Method**

`Write-stream-string` writes string `string` onto stream `s`. `Write-stream-string` is equivalent to calling `write-stream-char` on each character in `string` except that `string` is written atomically.

(write-stream s:stream (o:object)*) :null **Method**

`Write-stream` writes some representation of the given Concurrent Smalltalk objects onto stream `s`. It uses `print` to format objects it does not know about. Care should be taken when writing circular structures to make sure that `write-stream` terminates.

(display-stream s:stream (o:object)*) :null **Method**

`Display-stream` writes some representation of the given Concurrent Smalltalk objects onto stream `s`. Strings and characters are written literally, without escape characters. Care should be taken when writing circular structures to make sure that `display-stream` terminates.

Atomicity

(split s:stream) :stream **Abstract Method**

`Split` returns a new stream that can be used for atomic writing to `s`. Anything written to the returned stream is atomically written onto `s` when `join` is called on the returned stream.

(join s:stream) :null **Abstract Method**

`Join` joins `s` back to a stream from which it was split. It is an error to call `join` on a stream not returned by split or to call it more than once on such a stream.

Input and Output Streams

terminal-stream:system-stream **Global**

`Terminal-stream` is the `system-stream` used for interaction with the terminal.

(read-char) :object **Function**
(read-line) :object **Function**
(read) :object **Function**
(char-ready?) :boolean **Function**

(write-char ch:character) :null **Function**
(write-string string:string) :null **Function**
(write (o:object)*) :null **Function**
(display (o:object)*) :null **Function**

(split-terminal) :stream **Function**

These functions are the terminal equivalents of the general stream methods above.

Formatting

(print o:object **s**:stream) :null **Abstract Method**

`Print` is used for formatting arbitrary objects for the purposes of `write-stream`. `Print` should output some readable representation of object `o` onto stream `s`.

(display-print o:object **s**:stream) :null **Method**

`Display-print` is used for formatting arbitrary objects for the purposes of `display-stream`. `Display-print` should output some readable representation of object `o` onto stream `s`, avoiding escape characters where possible.

A.14. Macros

Concurrent Smalltalk provides a macro facility which can be used to extend the language. A macro consists of a pattern, an optional guard, and a replacement. The pattern can contain variables or literals (a literal is an identifier). If it matches with an expression and the guard is satisfied, that expression is replaced by the replacement, which can be either another pattern or a Common Lisp function.

```
(defmacro pattern [guard] replacement)                    Top-level Macro
pattern ::= literal | ?name | !name | (pattern* [pattern . pattern]) | @pattern
replacement ::= r-pattern | lisp-statement
r-pattern ::= literal | ?name | !name | (r-pattern* [r-pattern . r-pattern])
guard ::= &guard lisp-statement
lisp-statement ::= #L lisp
```

The macro pattern is a nested list of literals and macro variables. Variables are preceded by question marks (`?`) or exclamation points (`!`). Question-mark variables can match identifiers, numbers, and lists, while exclamation-point variables can only match identifiers. The dotted notation at the end of a list indicates that the rest of the list should match the pattern after the dot. When a pattern is matched to a candidate statement, all instances of the same variable have to match identical forms. The pattern can be as simple as `?x`, which will match any statement.

If an `@` symbol precedes a pattern, the form to which the pattern would match is macro-expanded before it is matched to the pattern. To avoid infinite loops, `@` should not be the first symbol in a macro pattern.

The guard, if present, is a Common Lisp statement that returns a boolean value. If the value returned is `true`, the macro replacement is substituted for the pattern; if not, the macro is not expanded. The values of the `?` and `!` variables are bound in a Common Lisp scope just outside the statement, so the Common Lisp statements can refer to the matched values of the variables just by referring to the correct variable names (including the leading `?` or `!`).

Replacement can be either another pattern or another block of Common Lisp statements. If replacement is a pattern, the values of the macro variables are substituted in it, and the replacement pattern replaces the original pattern in the code. If replacement is a Common Lisp statement, it is expected to return a list which replaces the original pattern in the code. As in the case of a guard, the Common Lisp statement has access to the matched values of the macro variables.

The macro replacement pattern can be another macro. Macros are expanded until the resulting form does not satisfy any of the existing macro patterns and guards. When several macros match a form, the form is expanded using the macro that was most recently defined.

A.15. Environment

Errors

`(error (msg:object)*)` **Function**

`Error` signals a run-time error. The arguments, if any, should contain descriptive information about the error. The interpretation of the arguments' values is implementation-dependent.

`(halt)` **Function**

`Halt` halts execution of the current program due to a run-time error. Debugging information about the function or method in which the halt took place may be printed.

Utilities

`(include "file-name")` **Top-level Primitive**

`Include` reads the definitions in the file named file-name, as if that file were included in place of the `include` primitive.

Options

`(pragma ...)` **Top-level Primitive**

`Pragma` is a general compiler declaration and can contain any implementation-dependent information.

`(declare option value)` **Top-level Primitive**

`Declare` sets the compiler option named option to the value specified. Value must be a legal value for the option; most compiler options are booleans, and for these value must evaluate to either `true` or `false`. Value must be a constant expression.

`(option option)` **Primitive**

`Option` returns the compile-time value of the specified compiler option.

Appendix B. Using Optimist II

This appendix describes the procedure for using the Optimist II compiler on a Macintosh II to compile Concurrent Smalltalk programs. In addition, a few helpful non-standard Concurrent Smalltalk features implemented by Optimist II are described.

Starting the Compiler

To start the compiler, load the image containing the compiler and the Common Lisp environment. If such an image is not available, load Common Lisp, PCL, the Loop macro, and the Optimist.Lisp file. Execute the `(optimist:compile-optimist)` command to compile and load the compiler, or, if it was already compiled, use `(optimist:load-optimist)` to load the compiler.

The compiler provides only one useful external Lisp function. It is `(interactive-cst)`. Typing `(interactive-cst)` will enter an interactive Concurrent Smalltalk listener loop.

Top-Level Commands

Utility Commands

(begin body**)** **Top-Level Primitive**

Due to constraints in the compiler, a select few forms such as `include` and `defclass` (but not all of the top-level primitives; most of the primitives listed as **top level** really only require that they not be included in any function or method) must be present at the top level. However, sometimes it is desirable to emit sequences of those directives as results of macros; to allow this, a special form of `begin` was provided. If `begin` appears at the top level, every form inside it is also evaluated at the top level.

(set name expression**)** **Top-Level Macro**

`Set` normally sets the variable name to the value of expression. However, if it is placed at the top level, it is also allowed to create a new global variable name if one does not exist already. Thus, at the top level, `set` acts as either `set` or `defglobal`, depending on whether the global variable name already exists.

(include) **Top-Level Primitive**

`Include`, when passed no file argument, will let the user interactively choose a text file and then include it. This feature is only available on the Macintosh version of Optimist II.

Viewing Objects

While the listener loop is active, any Concurrent Smalltalk command will be immediately evaluated, and the results displayed in the listener window. The resulting object may be displayed in a somewhat strange syntax; for example, integers may be displayed as `#<Integer 5>`, and booleans as `#<True>` or `#<False>`. The following commands may be used to show the internal structure of objects:

(show o:object**)** :object **Top-Level Primitive**

`Show` shows as a side effect the Optimist II internal representation of an object. If the object is a function, its hcodes are shown; if the object is a complex object, some of its structure may be shown. The output is controlled by the CLOS show generic function. The value of the show directive is the object itself, so the object is usually printed normally after it is shown.

Please note that the hcodes shown for a function are only an approximation of the actual hcode data structure used internally to represent the function. Some of the more esoteric fields are not shown, and sometimes a function may have two variables with the same name, which leads to confusing output. Variable names were included for human readability only; Optimist II does not use them internally. It is able to keep the variables distinct regardless of their names. Also, since a digraph is a nonlinear structure, pseudo-hcodes such as jump labels in conditionals and jump, label, and break hcodes are inserted into the output to make it readable.

(describe o:object) :object **Top-Level Primitive**

`Describe` describes as a side effect the Optimist II internal representation of an object. It is just like `show` except that the information displayed is longer and more detailed.

(show-hcode f:function [**#L**lisp-function-name]) :function **Top-Level Primitive**

`Show-hcode` calls the Optimizer's non-MDP-specific optimizations to optimize the function and shows the resulting hcode. `Show-hcode` may invoke global optimizations and try to inline the functions called by f, so this directive may take some time to execute. When the `progress` option is true (the default), progress information is displayed in the listener window while this directive is executing. Detailed progress can be obtained by setting the `detailed-progress` option. `Show-hcode` performs no side effects on the Concurrent Smalltalk environment, and it does not do a treewalk of the Concurrent Smalltalk program. `Show-hcode` returns f as its result.

If lisp-function-name is provided, instead of showing the optimized hcode, `show-hcode` calls the Lisp function lisp-function-name with the optimized hcode as an argument. Describe-dinodes is a useful Lisp function that will describe the compiled hcode in a little more detail.

`Show-hcode` will not optimize a selector. If viewing optimized method code is desired, the method must be extracted explicitly using the Concurrent Smalltalk `method` primitive.

(show-mdp-hcode f:function [**#L**lisp-function-name]) :function **Top-Level Primitive**

`Show-mdp-hcode` is just like `show-hcode` except that it also performs the MDP-specific hcode optimizations.

(show-asm f:function [**#L**lisp-function-name]) :function **Top-Level Primitive**

`Show-asm` compiles the function f all the way to assembly code and prints the resulting MDPSim-compatible text. If lisp-function-name is supplied, it is assumed to be a lisp function and called with the assembly language module as its only argument.

Compiling Programs

(compile f:object ["output-file-name"]) :object **Top-Level Primitive**

`Compile` compiles and treewalks the Concurrent Smalltalk data structures starting with f as a root. Normally f is a function, in which case it is compiled to assembly language along with any other functions that it might need. If output-file-name is specified, the MDPSim file is written to a new file named output-file-name; otherwise, the output is sent to the listener. When the `progress` option is true (the default), progress information is displayed in the listener window while this directive is executing. Detailed progress can be obtained by setting the `detailed-progress` option.

Options

As described in Section A.15, Concurrent Smalltalk options can be set using the `declare` Concurrent Smalltalk primitive and examined using the `option` primitive. The options currently provided by Optimist II are listed in Table B-1.

Table B-1. Options

Option	Default	Action
n-nodes	4	Define the number of nodes of a simulated J-Machine. This option only affects Optimist's internal interpreter; the compiled code is generic and will work on a J-Machine of any size (as long as the dimensions are powers of two).
precise	false	Inhibit optimizations that would affect the semantics of futures and lazy-futures in a few esoteric cases. If following precise Concurrent Smalltalk semantics is not important, disabling this option can produce significant performance improvements.
delete-dead-defs	true	Remove assignments to variables that will not be used again.
delete-moves	true	Try to remove unnecessary move statements.
delete-touches	true	Try to remove unnecessary touch statements.
vflow-optimizations	true	Calculate dataflow information and use it to perform a variety of optimizations such as changing x←y=0, branch if x false sequences to BNE instructions.
fold-constants	true	Fold constants. For example, replace 1+2 by 3. Also remove conditional branches when it can be determined that the condition is always true or always false.
fold-global-constants	true	Fold constants globally. For example, replace a call through a selector with a call of the method when the method can be determined using type analysis. This option is relevant only when fold-constants is true.
forward-tails	true	Enable the altering of application hcodes immediately followed by returns into tail-forwarded applications which allow the process to be deallocated and the answer directly forwarded to the caller. This is the equivalent of tail recursion.
merge-code	true	Merge common pieces of code wherever possible.
inline	true	Inline small functions.
inline-size-cutoff	12	Set the size cutoff for automatically deciding whether to inline a function. Increasing this number causes larger functions to be inlined.
optimize-built-ins	true	Perform local built-in optimizations such as changing multiplications to shifts.
compact-vars	true	Compact variables in the context to use as few slots as possible.
reg-variables	true	Assign variables to registers whenever possible.
lru-register-allocation	true	Use the least-recently-used algorithm to allocate temporary registers during code generation.
frame-touches	true	Accumulate information about which variables are touched and optimize touches when the variables are known to be touched.
frame-regs	true	Keep track of variables in the registers during code generation and use values from the registers instead of from memory whenever possible.
frame-migrate	true	Keep track of whether it is possible for the instance object to have migrated away. Don't force it if it could not have migrated away.
lazy-ivar-access	true	Don't XLATE the instance object if there are no references to it.
lazy-contexts	true	Don't allocate a context unless it is actually used.
fast-contexts	true	Use fast contexts whenever possible.

optimize-send-self	true	Send message to the current node if the receiver is self and it is not atomic.
fast-apply	true	Use `ApplyFunction` and `ApplySelector` instead of `Apply` whenever possible.
compact-sends	true	Try to combine `SEND`s and `SENDE`s into `SEND2`s and `SEND2E`s.
compact-DCs	true	Try to align `DC`s on word boundaries whenever possible.
delete-locals	true	Delete local variables in an intermediate stage of the compilation. This makes no difference in the final output, but makes the hcode look prettier and may speed up code generation.
warn-free-references	false	Emit a warning every time a free reference is found in a method or function.
progress	true	Print progress reports.
detailed-progress	false	Print very long progress reports.
permanent-definitions	false	Use `defconstant` instead of `defparameter` when compiling function and method definitions. When this option is set, a warning is emitted every time a free reference is found in a method or function regardless of the setting of warn-free-references.
print-pc	true	Print program counter values as comments in output.
lisp-break	true	Enter a Lisp break loop upon a Concurrent Smalltalk warning or error.

Appendix C. Using Cosmos

Loading Cosmos

To use Cosmos, launch MDPSim using the Cosmos.m file as an argument. You may also wish to specify the J-Machine's dimensions as arguments to MDPSim. Use `-x` *x* `-y` *y* `-z` *z*, where x, y, and z are integers; they should be powers of two. To avoid using too much memory, you may wish to allocate less memory per MDP with the `-msize` *mem* option.

When Cosmos.m is assembled by MDPSim, it will automatically load the operating system onto the MDPs and initialize the MDPs. This process may take anywhere from a few seconds to a few minutes depending on how many MDPs are present and the speed of the host computer.

Loading User Programs

Once Cosmos is ready, a user program compiled by Optimist II can be loaded. Use the MDPSim `INCLUDE` command to load the program generated by Optimist II. Keep in mind that Cosmos puts MDPSim into the case-sensitive mode, so the case of identifiers and commands matters; MDPSim recognizes commands which are either all upper case or all lower case characters.

```
MESSAGE fib4
MSG:msgApply|5
{fFib}
4
IONODE
0
END
```

Figure C-1. An Injected Application Message
This message calls the fFib function with the argument 4. The message itself can be injected by executing the command `INJECT fib4`. The 5 is the length of the message, `{fFib}` is Optimist II's output name for the function to be called (see the Optimist II output file if you are unsure about the name), 4 is the argument, and `IONODE` and 0 are magic numbers that cause the Reply message to be printed by MDPSim. More *than one one argument can be specified, as long as the length of the message (the 5) is increased appropriately.*

Once the user program has been loaded, it is a good idea to build a few templates for messages to be injected into the program. An application message should have the format shown in Figure C-1. If the messages will be used for several sessions, it might be appropriate to put them into a file and `INCLUDE` that file. Application messages should never be injected before the program is installed.

Instead of issuing the `INCLUDE` commands manually, you can also specify the files on MDPSim's command line, as was done in the example in Figure 5-14.

Running Programs

To run a program, execute the `INJECT` command on the message on which the program should be called and then `RUN` the program. Remember to specify the processor onto which `INJECT` should inject the application message; otherwise, `INJECT` will inject a copy of the message to every processor, and as many copies of the program will execute simultaneously as there are processors in the simulated system.

MDPSim allows statistics to be gathered about programs which are executed on it. If the statistics should only include data about the running program, they should be reset after the program is downloaded and before it is run. See the current MDPSim manual [25] and Figure 5-14 of this document for more details.

When you finish the desired program runs, use the `QUIT` command to exit the simulator and the quit menu item to exit MPW. In an emergency, command-shift-period can be used to abort MDPSim; command-period aborts the running MDP program and returns to MDPSim's command line (use control-C on UNIX machines).

Appendix D. MDP Architecture Summary

This appendix is a summary of the current version of the MDP architecture. A slightly obsolete full version of the architecture can be found in [16]. Many details have been simplified in order to keep this Appendix to a reasonable length.

Introduction

The Message-Driven Processor is a processing node for the J-Machine, a message-passing concurrent computer. The MDP is designed to provide support for fine-grained concurrent computation. Towards this goal the processor includes hardware for message queueing, low-latency message dispatching, and message sending. The same chip also contains a network interface and a router to allow the routing of messages throughout the network without any processor intervention.

The size of the MDP's register set is limited to minimize context-switching time. Much of the memory is on the chip to improve performance and reduce the chip's pin count and the chip count for the concurrent computer. Having memory on chip allows more flexibility in the use of memory than in designs with off-chip memory. For example, a portion of memory may be designated as a two-way set-associative cache to be used by the XLATE instruction. Nevertheless, since current technological limitations restrict the size of the on-chip memory to about 4096 words, an external memory interface has been provided to allow access to slow, off-chip DRAM.

The MDP is also designed to efficiently support object-oriented programming. Every MDP word consists of 32 data bits and a 4 bit tag that classifies the word as an integer, boolean, address, instruction, pointer, or other data. The MDP's four address registers include both base addresses and lengths, so all memory accesses are bounds checked. Normally the address registers point to objects, so, since absolute memory addressing is not allowed except by the operating system, memory references can only be made to objects relative to their beginnings. Having tags and no absolute references permits the use of garbage collection and transparent migration of objects to other MDP nodes on the network.

The MDP is almost completely message-driven. It is controlled by the messages arriving from the network that are automatically queued and processed. There are two priority levels to allow urgent messages to interrupt normal processing. There is also limited support for a background mode of execution when no messages are waiting in the queues.

Processor State

The processor state of the MDP is kept in a set of registers shown in Figure D-1. There are three independent copies of most registers for each of the two priorities of the MDP, allowing easy priority switches while keeping the integrity of the registers. The registers are symbolically represented as follows:

R0-R3	general-purpose data registers
A0-A3	address registers
ID0-ID3	ID registers
Q, M, U, F, I, P, B	flags
IP	instruction pointer register
FIR	faulted instruction register
FIP	faulted instruction pointer register
FOP0, FOP1	faulted operand registers
QBM	queue base/limit register
QHL	queue head/tail register

TBM	translation base/mask register
NNR	node number register
MAR	memory address register

The Q flag controls message queue access through register A3, while the M flag guards against inter-priority message deadlocks. Setting the U (unchecked mode) flag disables type and overflow faults. Setting the F (faulted) flag vectors all faults to the CATASTROPHE vector; this flag is often set in critical sections of fault handlers. Setting the I (interrupt) flag prevents higher-priority interrupts. The B and P flags encode the current priority level.

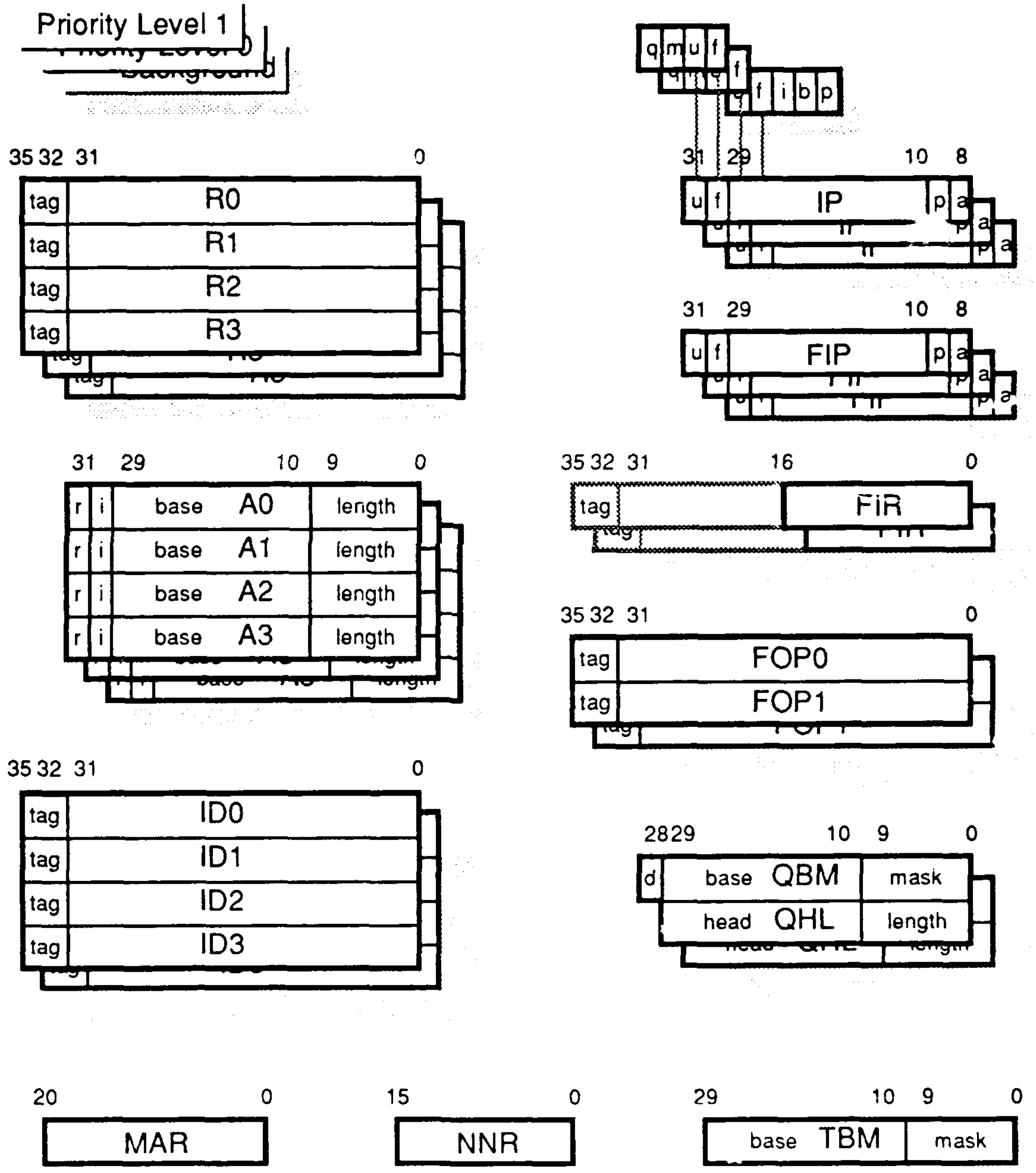


Figure D-1. The MDP Register Set.

Data Types

The data types that may be used in a word are shown in Figure D-2. All data types except FUT and CFUT may be moved, compared with EQ and NEQ, XLATEd and ENTERed, RTAGged, WTAGged, CHECKed, and executed. Executing a non-INST word causes it to be loaded into R0. Some data types allow additional operations, which are listed in detail in the description of the instruction set.

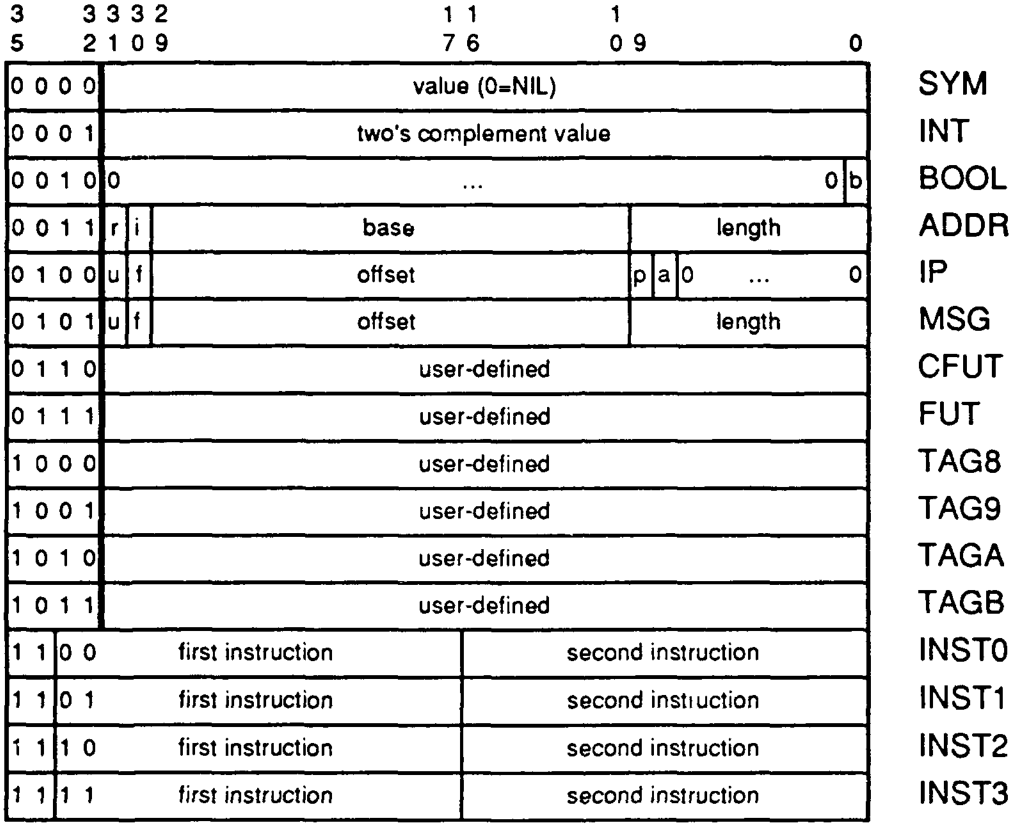


Figure D-2. The MDP Data Types.

- SYM contains an atomic symbol. EQUAL and NEQUAL are allowed on SYMbols. If the data portion of a symbol contains all zeroes, the word takes on the value of NIL. Cosmos renames SYM as TAG0 and inserts a subtag in bits 28 through 31 to distinguish between a few more types.
- INT contains a two's complement integer between -2^{31} and $2^{31}-1$, inclusive. All arithmetic, logical, and comparison operations are allowed on INTS.
- BOOL contains a boolean value, which is either true (b=1) or false (b=0). All logical and comparison operations are allowed on BOOLS; false is considered to be less than true.
- ADDR contains a base/length pair that may be loaded into either one of the address registers or QBM, QHL, or TBM. The uses of bits 30 and 31 vary among these registers.
- IP contains a value appropriate for loading into the IP.
- MSG is the header of a message. It is similar to an IP. Due to a shortage of tags, Cosmos also uses this tag under the name OBJ as an object header.
- CFUT contains a context future. Almost all operations fault on context futures. They are not meant to be MOVEable. CFUTS are used as placeholders for unavailable values to be computed in parallel by other processes; an attempt to read a CFUT before its value is available will fault, and the operating system will suspend the current process until the value is available.

- FUT is a standard future. FUTs may be moved, and their tags may be read and written, but they may not participate in any primitive operations such as addition or checking for equality. As with CFUTS, an attempt to use a FUT in a primitive operation will cause a fault, and the operating system will have to provide the appropriate value for the FUT.
- TAG8 through TAGB are tags for operating system-defined words. They cause faults on all primitive operations except EQ, NEQ, BNIL, and BNNIL. Cosmos renames these tags as ID, DID, TAGA, and FLOAT, respectively.
- INST0 through INST3 are tags for instructions. The two instructions in a word occupy a total of 34 bits, so two tag bits are also used to encode them.

Network Interface

Incoming messages are queued in *message queues* before being dispatched and processed. There are two message queues, one for each priority level. When a message arrives, register A3 is set up to point to it in the message queue, and execution begins at the address specified by the message header. A message may be processed as soon as its first word arrives; the processor does not wait until the entire message is present before processing it. Memory accesses to the message are checked to make sure that the processor does not try to access a word in the message before it arrives; if the processor tries to access a word too early, it waits until the word has arrived.

The SUSPEND instruction informs the hardware that the processing of the current message is done and that it should fetch the next message.

Message Transmission

The SEND, SEND2, SENDE, and SEND2E instructions are used to send messages. The first word sent specifies the node number of the destination node (i.e. the destination node's NNR value) in the low 16 bits. The SEND instruction will use the current node's NNR and the destination node number to find the relative offsets in the X and Y dimensions that the network controllers will use in routing the messages through the network. There are actually two flavors of each SEND instruction: SEND0, SEND20, SENDE0, and SEND2E0 send words of priority 0 messages, while SEND1, SEND21, SENDE1, and SEND2E1 send words of priority 1 messages. The priority of the message is independent of the priority of the process that is sending it.

The initial routing word is followed by a number of words which the network delivers verbatim to the destination node. The network does not examine the contents of these words. The message is terminated by a SENDE or SEND2E instruction, which send the last one or two, respectively, words of it and inform the network to actually transmit the message. The first word that arrives at the destination node (the second word actually sent since the routing word is only used by the network and doesn't arrive at the destination node) must be tagged MSG. It contains the length of that message including that word but not including the routing word preceding it. It also contains the initial value of the IP at which execution is supposed to start. The destination node will fault MSG if this word is incorrect.

The total time between the first SEND and the SENDE should be as short as possible to avoid blocking the network. For the same reason, *faults should be avoided while sending.*

Fault Processing

When a fault occurs, the instruction that caused the fault is saved in the FIR register, the current IP (which points one instruction beyond the faulting instruction) is saved in the FIP register, and the values of the instruction operands, if any, are saved in the FOP0 and FOP1 registers. If the fault occurred while fetching an instruction, the FIR is set to NIL and the FIP points to the instruction. The IP is then fetched from the memory location whose address is equal to the fault number plus the base of the fault vector table of the current priority. If the F bit was, the IP is loaded from the CATASTROPHE vector instead. The U, A, and F flags receive their new values from the loaded IP. The faults are listed in Table D-1.

Table D-1. MDP Faults

Name	Fault Number	Description
CATASTROPHE	$00	Double fault,bad vector, or other catastrophe.
INTERRUPT	$01	Interrupt pin has gone active.
QUEUE	$02	Message queue about to overflow.
SEND	$03	Send buffer full.
ILGINST	$04	Illegal instruction.
DRAMERR	$05	Double bit error in the external RAM.
INVADR	$06	Attempt to access data through address register with `I` bit set.
LIMIT	$07	Attempt to access object data past limit.
ADRTYPE	$08	Index in indexed addressing mode not tagged `INT`.
EARLY	$09	Attempt to access data in message queue before it arrived.
MSG	$0A	Bad message header.
XLATE	$0B	XLATE missed.
OVERFLOW	$0C	Integer arithmetic overflow.
CFUT	$0D	Attempted operation on a word tagged `CFUT`.
FUT	$0E	Attempted operation on a word tagged `FUT`.
TAG8	$0F	Attempted operation on a word tagged `TAG8`.
TAG9	$10	Attempted operation on a word tagged `TAG9`.
TAGA	$11	Attempted operation on a word tagged `TAGA`.
TAGB	$12	Attempted operation on a word tagged `TAGB`.
TYPE	$13	An operand or a combination of operands with a bad tag type used in an instruction.
	$14-$1F	Reserved for future faults.

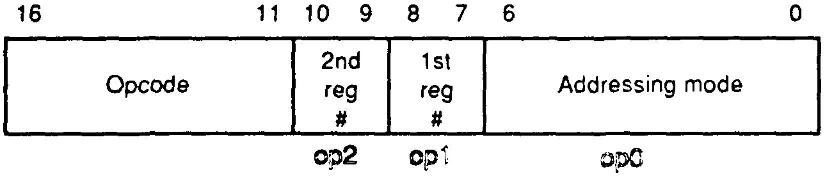
If multiple faults occur simultaneously the fault vector chosen is the one that has the highest precedence. Each fault is assigned a precedence by its fault number; lower fault numbers correspond to higher precedence.

Instruction Encoding

The program executed by the MDP consists of instructions and constants. A constant is any word not tagged `INST0` through `INST3` that is encountered in the instruction stream. When a constant word is encountered, that word is loaded into `R0` and execution proceeds with the next word (the assembler syntax for including a word in the code stream is `DC`).

Every instruction is 17 bits long. Two 17-bit instructions are packed into a word. Since a word has only 32 data bits, two tag bits are also used to specify the instructions. The instruction in the high part of the word is executed first, followed by the instruction in the low part of the word. As a matter of convention, if only one instruction is present in a word, it should be placed in the high part, and the low part of the word set to all zeros.

The format of an instruction is as follows:



The *opcode* field specifies one of 64 possible instructions. The other fields specify three operands; instructions that don't require three operands ignore some of the operand fields. Operands 1 and 2 must be data registers; their numbers (0 through 3) are encoded in the *1st* *reg #* and *2nd reg #* fields. Operand 2, if used, is always the destination of an operation and operand 1, if used, is always a source.

Normal Addressing Mode (bits 6 … 0)	Syntax	Addressing Mode
0 0 0 0 0 Rn	`Rn`	Data register Rn
0 0 0 0 1 An	`An`	Address register An
0 0 0 1 0 0 0	`NIL`	Immediate constant NIL (SYM:0)
0 0 0 1 0 0 1	`FALSE`	Immediate constant FALSE (BOOL:0)
0 0 0 1 0 1 0	`TRUE`	Immediate constant TRUE (BOOL:1)
0 0 0 1 0 1 1	`$80000000`	Immediate constant INT:$80000000
0 0 0 1 1 0 0	`$FF`	Immediate constant INT:$000000FF
0 0 0 1 1 0 1	`$3FF`	Immediate constant INT:$000003FF
0 0 0 1 1 1 0	`$FFFF`	Immediate constant INT:$0000FFFF
0 0 0 1 1 1 1	`$FFFFF`	Immediate constant INT:$000FFFFF
0 0 1 Rx An	`[Rx,An]`	Offset Rx in object An
0 1 imm	`imm`	Immediate imm (signed)
1 imm An	`[imm,An]`	Offset imm (unsigned) in object An

Figure D-3. The MDP Normal Addressing Modes.

The immediate constants are eight immediate values outside the range `INT:-16..INT:15`. They are provided for convenience and code density improvement. The $FF and $FFFF constants are useful for masking bytes and words, while the $3FF and $FFFFF constants may be used for masking lengths and addresses.

Operand 0 can be used as a source or a destination in an instruction. It can hold two possible encodings. A normal instruction has op0 address mode encodings as shown in Figure D-3. The register-oriented op0 mode is used only by three variants of the MOVE instruction. If an instruction uses the register-oriented op0, the encodings are as in Figure D-4.

Instruction Set Summary

The instructions supported by the MDP are summarized in Table D-2. The Types column specifies the types on which the instruction operates; if the arguments have different types, the instruction faults. Except for a MOVE to memory, all instructions fault when any of their operands are tagged CFUT. Also, except for MOVEs and SENDs, all instructions fault when any of their operands are tagged FUT.

6						0		
Register-Oriented Addressing Mode							**Syntax**	**Addressing Mode**
B	P	0	0	0	Rn		Rn	Data register Rn
B	P	0	0	1	An		An	Address register An
-	P	0	1	0	IDn		IDn	ID register IDn
B	P	0	1	1	0	0	FIP	Trapped Instruction pointer
-	P	0	1	1	0	1	FIR	Trapped Instruction register
-	P	0	1	1	1	0	FOP0	Trapped OP0 register
-	P	0	1	1	1	1	FOP1	Trapped OP1 register
-	P	1	0	0	0	0	QBM	Queue Base/Mask register
-	P	1	0	0	0	1	QHL	Queue Head/Length register
B	P	1	0	0	1	0	IP	Instruction Pointer register
-	-	1	0	0	1	1	TBM	Translation Base/Mask register
-	-	1	0	1	0	0	NNR	Node Number register
-	-	1	0	1	0	1	MAR	Memory Address Bus register
-	-	1	0	1	1	0		Unused (ILGINST fault)
-	-	1	0	1	1	1		Unused (ILGINST fault)
-	-	1	1	0	0	0	P	Priority Level flag
-	-	1	1	0	0	1	B	Background Execution flag
-	-	1	1	0	1	0	I	Interrupt flag
B	P	1	1	0	1	1	F	Fault flag
B	P	1	1	1	0	0	U	Unchecked flag
-	P	1	1	1	0	1	Q	A3 Queue flag
-	-	1	1	1	1	0		Unused (ILGINST fault)
-	-	1	1	1	1	1		Unused (ILGINST fault)

Figure D-4. The MDP Register Oriented Addressing Modes.
B and P represent the priority of the register being accessed XORed with the current priority. For example, 00 indicates the current priority, while 01 would let priority 1 access priority 0's registers, and 11 would let priority 1 access the background registers. The assembler syntax for specifying a register belonging to the other priority is the register name followed by a B to flip the B bit and/or a backquote (`) to flip the P bit.

Table D-2. MDP Instructions

Instruction		**Brief Description**	**Types**
General Movement and Type Instructions			
MOVE	Src,Rd	Rd ← Src. Src may be a register addressing mode.	All
MOVE	Rs,Dst	Dst ← Rs. Dst may be a register addressing mode.	All
MOVE	Src,IP	IP ← Src. Src may be a register addressing mode.	All

RTAG	Src,Rd	Rd ← INT:tag(Src)	All
WTAG	Rs,Src,Rd	Rd ← Src:Rs	All
CHECK	Rs,Src,Rd	Rd ← BOOL:tag(Rs)=Src	All

Arithmetic and Logical Instructions

NEG	Src,Rd	Rd ← -Src	INT
ADD	Rs,Src,Rd	Rd ← Rs+Src	INT
SUB	Rs,Src,Rd	Rd ← Rs-Src	INT
CARRY	Rs,Src,Rd	Rd ← Carry from the addition of Rs and Src	INT
MUL	Rs,Src,Rd	Rd ← Rs*Src	INT
MULH	Rs,Src,Rd	Rd ← High 32 bits of 64-bit unsigned product of Rs and Src	INT
ASH	Rs,Src,Rd	Rd ← Rs shifted left arithmetically by Src bits	INT
LSH	Rs,Src,Rd	Rd ← Rs shifted left logically by Src bits	INT
ROT	Rs,Src,Rd	Rd ← Rs rotated left by Src (mod 32) bits	INT
FFB	Src,Rd	Rd ← 31-position of leftmost bit of Rs differing from bit 31.	INT
NOT	Src,Rd	Rd ← NOT Src	INT, BOOL
AND	Rs,Src,Rd	Rd ← Rs AND Src	INT, BOOL
OR	Rs,Src,Rd	Rd ← Rs OR Src	INT, BOOL
XOR	Rs,Src,Rd	Rd ← Rs XOR Src	INT, BOOL
LT	Rs,Src,Rd	Rd ← BOOL:Rs<Src	INT, BOOL
LE	Rs,Src,Rd	Rd ← BOOL:Rs≤Src	INT, BOOL
GT	Rs,Src,Rd	Rd ← BOOL:Rs>Src	INT, BOOL
GE	Rs,Src,Rd	Rd ← BOOL:Rs≥Src	INT, BOOL
EQUAL	Rs,Src,Rd	Rd ← BOOL:Rs=Src	SYM, INT, BOOL
NEQUAL	Rs,Src,Rd	Rd ← BOOL:Rs≠Src	SYM, INT, BOOL
EQ	Rs,Src,Rd	Rd ← BOOL:Rs=Src (Pointer comparison only)	All
NEQ	Rs,Src,Rd	Rd ← BOOL:Rs≠Src (Pointer comparison only)	All

Network Instructions

SEND	Src	Send Src onto the network	All
SENDE	Src	Send Src onto the network and terminate message	All
SEND2	Rs,Src	Send Rs and Src onto the network	All
SEND2E	Rs,Src	Send Rs and Src onto the network and terminate message	All

Associative Lookup Table Instructions

XLATE	Rs,Dst,C	Dst ← associative lookup in the associative lookup table of Rs	All
ENTER	Src,Rs	Enter (Src, Dst) into the associative lookup table	All
PROBE	Src,Rd	Rd ← BOOL:Src is in the associative lookup table	All

Special Instructions

NOP		No operation
INVAL		Invalidate all relocatable address registers
SUSPEND		Terminate current process and fetch another message
CALL	Src	Call system routine numbered Src

Branches

BR	Src	Branch forward Src words	
BNIL	Rs,Src	Branch forward Src words if Rs is NIL	All
BNNIL	Rs,Src	Branch forward Src words if Rs is not NIL	All
BF	Rs,Src	Branch forward Src words if Rs is false	BOOL
BT	Rs,Src	Branch forward Src words if Rs is true	BOOL
BZ	Rs,Src	Branch forward Src words if Rs is zero	INT
BNZ	Rs,Src	Branch forward Src words if Rs is non-zero	INT

Appendix E. Optimist II Listing

This listing has been removed due to space constraints. For a copy of the source, please send mail to `waldemar@ai.mit.edu` or `billd@ai.mit.edu`. A slightly older, printed copy of the source can also be found in the original Master's thesis version of this document.

Appendix F. Cosmos Listing

Cosmos.i

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;                                             ;;;;
;;;;             MDP Operating System            ;;;;
;;;;                                             ;;;;
;;;;                 version 2.3                 ;;;;
;;;;                                             ;;;;
;;;;                  written by                 ;;;;
;;;;               Waldemar Horwat               ;;;;
;;;;                                             ;;;;
;;;;  Master's thesis under Prof. William Dally  ;;;;
;;;;                                             ;;;;
;;;;                March 28, 1989               ;;;;
;;;;                   May 1991                  ;;;;
;;;;                                             ;;;;
;;;;         Send problems and comments to       ;;;;
;;;;            waldemar@hx.lcs.mit.edu.         ;;;;
;;;;                                             ;;;;
;;;; Copyright 1989, 1990, 1991 Waldemar Horwat  ;;;;
;;;;                                             ;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

CASE ON

;+--------------------------------------------------------------------------------------+
;| Parameters                                                                           |
;+--------------------------------------------------------------------------------------+
;These parameters are used to customize Cosmos.  You can override the default settings of
;these parameters by using -d REALMODE=1, etc. command line options.

;REALMODE is true if the code should be compiled for a real J-Machine instead of MDPSim.
;This turns off the STOP instruction (this means you can't use RUN).
IF !Defined(REALMODE)
  LABEL REALMODE = 0
END

;FASTSIM is true if the loop that clears memory to CFUTs should be skipped.
IF !Defined(FASTSIM)
  LABEL FASTSIM = 1
END

;DEBUG is true if extra debugging code should be run.
IF !Defined(DEBUG)
  LABEL DEBUG = 1
END


;
;Equates
;

LABEL LogNNodes = LGNNODES
LABEL NNodes = 1<<LogNNodes

LABEL nFastContexts = 8                 ;Number of fast context to allocate.

;+--------------------------------------------------------------------------------------+
;| Memory Map                                                                           |
;+--------------------------------------------------------------------------------------+

LABEL Globals0Start = 0
LABEL Globals0End = $40
LABEL Globals1Start = $40
LABEL Globals1End = $80
LABEL ADR Faults0Start = $80
LABEL ADR Faults0End = $A0
LABEL ADR Faults1Start = $A0
LABEL ADR Faults1End = $C0
LABEL ADR CallsStart = $C0
LABEL ADR CallsEnd = $100
LABEL ADR Queue1Start = $100
LABEL ADR Queue1End = $100
LABEL ADR Queue0Start = $100
LABEL ADR Queue0End = $200
LABEL ADR XlateStart = $200
LABEL ADR XlateEnd = $400
LABEL BRATLenLog = 6
LABEL BRATLength = 1<<BRATLenLog
LABEL ADR BRATStart = $400
LABEL ADR BRATEnd = BRATStart+BRATLength
LABEL ADR HeapEnd = MEMSIZE


;+--------------------------------------------------------------------------------------+
;| Tags                                                                                 |
;+--------------------------------------------------------------------------------------+

LABEL TAG TAG0 = 0                      ;Immediate object tag.
LABEL TAG OBJ = MSG                     ;Objects and messages have the same tag.
LABEL TAG CS = INST1                    ;Class/selector.
```

```
;Subtags of TAG0:
LABEL subtagN = 28                  ;Subtag offset.
LABEL subtagL = 4                   ;Subtag length.
LABEL subtagM = (1<<subtagL)-1      ;Subtag mask.
LABEL subSYM = 0                    ;Symbol.
LABEL subCLASS = 1                  ;Class.
LABEL subSEL = 2                    ;Selector.
LABEL subCHAR = 3                   ;Character.


;+------------------------------------------------------------------------------+
;| Types                                                                        |
;+------------------------------------------------------------------------------+

;Address bits
LABEL lengthN = 0                   ;Length field offset.
LABEL lengthL = 10                  ;Length field length.
LABEL lengthM = (1<<lengthL)-1      ;Length field mask.
LABEL baseN = 10                    ;Base field offset.
LABEL baseL = 20                    ;Base field length.
LABEL baseM = (1<<baseL)-1          ;Base field mask.
LABEL invalidN = 30                 ;Invalid address.
LABEL invalid = 1<<invalidN
LABEL relN = 31                     ;Relocatable address.
LABEL rel = 1<<relN
LABEL disableN = 30                 ;Disable bit of QBM regs.
LABEL disable = 1<<disableN

;IP bits
LABEL absN = 8                      ;Absolute IP.
LABEL abs = 1<<absN
LABEL phaseN = 9                    ;IP phase bit.
LABEL phase = 1<<phaseN
LABEL offsetN = 10                  ;Offset field offset.
LABEL offsetL = 20                  ;Offset field length.
LABEL faultN = 30                   ;Fault flag.
LABEL fault = 1<<faultN
LABEL uncheckedN = 31               ;Unchecked mode flag.
LABEL unchecked = 1<<uncheckedN

;ID bits
LABEL homeNodeN = 0                 ;Home node.
LABEL homeNodeL = 16
LABEL homeNodeM = (1<<homeNodeL)-1
LABEL serialN = 16                  ;Serial number.
LABEL serialL = 15
LABEL serialM = (1<<serialL)-1
LABEL distobjMemberN = 31           ;Distributed object member flag.
LABEL distobjMember = 1<<distobjMemberN

;DID bits
LABEL initialNodeN = 0              ;Initial node.
LABEL initialNodeL = 11
LABEL initialNodeM = (1<<initialNodeL)-1
LABEL logStrideN = 11               ;2's complement lg(#nodes/#constituents)
LABEL logStrideL = 5
LABEL logStrideM = (1<<logStrideL)-1

;Class/Selector bits
LABEL csSelectorN = 0               ;Selector.
LABEL csSelectorL = 16
LABEL csSelectorM = (1<<csSelectorL)-1
LABEL csClassN = 16                 ;Class.
LABEL csClassL = 16
LABEL csClassM = (1<<csClassL)-1


;xyz bits
LABEL xN = 0                        ;X field offset.
LABEL xL = LGXNODES                 ;X field length.
LABEL xM = (1<<xL)-1                ;X field mask.
LABEL xMC = (1<<5-xL)-1             ;X complement field mask.
LABEL yN = 5                        ;Y field offset.
LABEL yL = LGYNODES                 ;Y field length.
LABEL yM = (1<<yL)-1                ;Y field mask.
LABEL yMC = (1<<5-yL)-1             ;Y complement field mask.
LABEL zN = 10                       ;Z field offset.
LABEL zL = LGZNODES                 ;Z field length.
LABEL zM = (1<<zL)-1                ;Z field mask.
LABEL zMC = (1<<6-zL)-1             ;Z complement field mask.

;These constants are used to fashion serial and node numbers for precompiled objects.
LABEL mX = xM
LABEL sX = 0
LABEL mY = yM<<xL
LABEL sY = yN-xL
LABEL mZ = zM<<xL+yL
LABEL sZ = zN-xL-yL
LABEL mS = serialM<<xL+yL+zL
LABEL sS = serialN-xL-yL-zL
;The nth object is stored at (n&mX)<<sX|(n&mY)<<sY|(n&mZ)<<sZ|(n&mS)<<sS

;These constants are used to fashion numbers for precompiled classes and selectors
;so as to distribute them evenly throughout the J-Machine.
LABEL m3 = xMC<<xL+yL+zL
LABEL s3 = -yL-zL
LABEL m4 = yMC<<xN+yL+zL
LABEL s4 = -zL
LABEL m5 = zMC<<xN+yN+zL
LABEL s5 = 0
;The nth object is stored at (n&mX)<<sX|(n&mY)<<sY|(n&mZ)<<sZ|(n&m3)<<s3|(n&m4)<<s4|(n&m5)<<s5

LABEL nodeMask = zM<<zN|yM<<yN|xM<<xN    ;Mask for generating random node numbers.

LABEL RandomSeedIncrement = 5<<zN-2|3<<yN-1|1<<xN
```

```
;Hardwired classes:
LABEL classPrimitiveClass = (2&mX)<<sX|(2&mY)<<sY|(2&mZ)<<sZ|(2&m3)<<s3|(2&m4)<<s4|(2&m5)<<s5
LABEL classStandardClass = (3&mX)<<sX|(3&mY)<<sY|(3&mZ)<<sZ|(3&m3)<<s3|(3&m4)<<s4|(3&m5)<<s5
LABEL classDistributedClass = (4&mX)<<sX|(4&mY)<<sY|(4&mZ)<<sZ|(4&m3)<<s3|(4&m4)<<s4|(4&m5)<<s5
LABEL classObject = (5&mX)<<sX|(5&mY)<<sY|(5&mZ)<<sZ|(5&m3)<<s3|(5&m4)<<s4|(5&m5)<<s5
LABEL classNull = (6&mX)<<sX|(6&mY)<<sY|(6&mZ)<<sZ|(6&m3)<<s3|(6&m4)<<s4|(6&m5)<<s5
LABEL classSymbol = (7&mX)<<sX|(7&mY)<<sY|(7&mZ)<<sZ|(7&m3)<<s3|(7&m4)<<s4|(7&m5)<<s5
LABEL classClass = (8&mX)<<sX|(8&mY)<<sY|(8&mZ)<<sZ|(8&m3)<<s3|(8&m4)<<s4|(8&m5)<<s5
LABEL classSelector = (9&mX)<<sX|(9&mY)<<sY|(9&mZ)<<sZ|(9&m3)<<s3|(9&m4)<<s4|(9&m5)<<s5
LABEL classCharacter = (10&mX)<<sX|(10&mY)<<sY|(10&mZ)<<sZ|(10&m3)<<s3|(10&m4)<<s4|(10&m5)<<s5
LABEL classInteger = (11&mX)<<sX|(11&mY)<<sY|(11&mZ)<<sZ|(11&m3)<<s3|(11&m4)<<s4|(11&m5)<<s5
LABEL classBoolean = (12&mX)<<sX|(12&mY)<<sY|(12&mZ)<<sZ|(12&m3)<<s3|(12&m4)<<s4|(12&m5)<<s5
LABEL classFalse = (13&mX)<<sX|(13&mY)<<sY|(13&mZ)<<sZ|(13&m3)<<s3|(13&m4)<<s4|(13&m5)<<s5
LABEL classTrue = (14&mX)<<sX|(14&mY)<<sY|(14&mZ)<<sZ|(14&m3)<<s3|(14&m4)<<s4|(14&m5)<<s5
LABEL classFloat = (15&mX)<<sX|(15&mY)<<sY|(15&mZ)<<sZ|(15&m3)<<s3|(15&m4)<<s4|(15&m5)<<s5
LABEL classFunction = (16&mX)<<sX|(16&mY)<<sY|(16&mZ)<<sZ|(16&m3)<<s3|(16&m4)<<s4|(16&m5)<<s5


;+---------------------------------------------------------------------------------------+
;| Objects                                                                               |
;+---------------------------------------------------------------------------------------+

LABEL objectHeader = 0
LABEL objectID = 1

;Object header bits:
LABEL hdrLengthN = 0                     ;Length field offset.
LABEL hdrLengthL = 10                    ;Length field length.
LABEL hdrLengthM = (1<<hdrLengthL)-1;    ;Length field mask.
LABEL hdrClassN = 10                     ;Class field offset.
LABEL hdrClassL = 16                     ;Class field length.
LABEL hdrClassM = (1<<hdrClassL)-1;      ;Class field mask.
LABEL hdrFastN = 26                      ;Fast context.
LABEL hdrFast = 1<<hdrFastN
LABEL hdrDeletedN = 27                   ;Free object.
LABEL hdrDeleted = 1<<hdrDeletedN
LABEL hdrCopyableN = 28                  ;Immutable copyable object.
LABEL hdrCopyable = 1<<hdrCopyableN
LABEL hdrPurgeableN = 29                 ;Purgeable object.
LABEL hdrPurgeable = 1<<hdrPurgeableN
LABEL hdrLockedN = 30                    ;Locked object.
LABEL hdrLocked = 1<<hdrLockedN
LABEL hdrMarkedN = 31                    ;Purgeable object marked by sweeper.
LABEL hdrMarked = 1<<hdrMarkedN


;Class objects:
LABEL oClassWord = 2                     ;Header word for objects of this class.
LABEL oClassNAllSupers = 3               ;Count of all superclasses for this class.
LABEL oClassAllSupers = 4                ;List of all superclasses for this class.

;Selectors:
LABEL oSelNMethods = 2                   ;Number of methods defined for this selector.
LABEL oSelMethods = 3                    ;List of class/method pairs for this selector.

;Functions:
LABEL oFunctionNArgs = 2                 ;Number of arguments or NIL
LABEL oFunctionCode = 3                  ;Code of function.

;Closures:
LABEL oClosureNArgs = 2                  ;Number of arguments or NIL
LABEL oClosureCode = 3                   ;Faulting instruction.
LABEL oClosureFunct = 4                  ;Function to be called.
LABEL oClosureDisplay = 5                ;Additional display arguments.

;Distobjs:
LABEL oDistobjGroup = 2                  ;OID of a distributed object.
LABEL oDistobjIndex = 3                  ;Constituent number of a constituent.
LABEL oDistobjLogicalLimit = 4           ;Logical number of constituents in a distributed object.


;+---------------------------------------------------------------------------------------+
;| Contexts
;+---------------------------------------------------------------------------------------+

LABEL contextHeader = 0
LABEL contextID = 1
;Context message and locals are in locations 2 through 15.
LABEL contextR0 = 16
LABEL contextR1 = 17
LABEL contextR2 = 18
LABEL contextR3 = 19
LABEL contextID0 = 20
LABEL contextID2 = 21
LABEL contextID3 = 22
LABEL contextIP = 23
LABEL contextNext = 24                   ;Next context in a chain.
                                         ;Also used to store NIL or next context number when waiting
                                         ;for an object, or zero when waiting for a cfuture.
LABEL contextSize = 25                   ;Size of a fast context.
;More locals may follow here.


;+---------------------------------------------------------------------------------------+
;| Messages
;+---------------------------------------------------------------------------------------+

;Apply message:
LABEL applyHeader = 0
LABEL applyFunct = 1
LABEL applyReceiver = 2

;Reply message:
LABEL replyHeader = 0
LABEL replyID = 1
LABEL replySlot = 2
LABEL replyValue = 3
```

```
;RestartContext message:
LABEL restartHeader = 0
LABEL restartID = 1

;NewObject message:
LABEL newObjHeader = 0
LABEL newObjClass = 1
LABEL newObjReplyID = 2
LABEL newObjReplySlot = 3

;Dispose message:
LABEL disposeHeader = 0
LABEL disposeID = 1

;DisposeBRAT message:
LABEL disposeBRATHeader = 0
LABEL disposeBRATID = 1

;LookupMethod message:
LABEL LLookupMethod = ID:(1<<30|(0&mX)<<sX|(0&mY)<<sY|(0&mZ)<<sZ|(0&mS)<<sS)
REF REV LookupMethod = LLookupMethod
LABEL lookMethSelector = 2
LABEL lookMethClass = 3
LABEL lookMethReplyID = 4

;MethodReply message:
LABEL methodReplyHeader = 0
LABEL methodReplyID = 1
LABEL methodReplyValue = 2

;RequestObject message:
LABEL reqObjHeader = 0
LABEL reqObjID = 1
LABEL reqObjReplyNode = 2

;UpdateHome message:
LABEL updtHomeHeader = 0
LABEL updtHomeID = 1
LABEL updtHomeNode = 2

;Unlock message:
LABEL unlockHeader = 0
LABEL unlockID = 1


LABEL msgAcknowledgeObject = 1<<offsetN


;+-----------------------------------------------------------------------------+
;| Globals
;+-----------------------------------------------------------------------------+

LABEL ADR TempDiv_Count = 4             ;Divide temporary.
LABEL ADR LimitOverride = 5             ;NIL or IP to which a limit fault should jump (one time only).
LABEL ADR FastContextQueue = 6          ;Queue of fast contexts.
LABEL ADR TempCH_R0 = 7                 ;CompactHeap temporary.
LABEL ADR FirstFree = 8                 ;Pointer to first free heap word.
LABEL ADR LastFree = 9                  ;Pointer to last free heap word plus one.
LABEL ADR BRATFree = 10                 ;Pointer to free BRAT links.
LABEL ADR LastObjectID = 11             ;ID of last object to be allocated.
LABEL ADR NextDistobjID = 12            ;ID of next distributed object to be allocated.
LABEL ADR SerialNode = 13               ;This node's serial number.

LABEL ADR NodeMask = 15                 ;The nodeMask constant.
LABEL ADR HeapStart = 16                ;Pointer to the beginning of the relocatable heap
LABEL ADR RandomSeed = 17               ;Random number seed.
LABEL ADR TempXLATE_R0 = 18             ;XLATE fault handler temporaries.
LABEL ADR TempXLATE_R1 = 19
LABEL ADR TempXLATE_R2 = 20
LABEL ADR TempXLATE_FIP = 21
LABEL ADR TempXLATE_FIR = 22
LABEL ADR TempXLATE_Temp = 23
LABEL ADR TempDO_FIP = 24               ;DisposeObject temporary
LABEL ADR TempNC_FIP = 25               ;NewContext temporaries.
LABEL ADR TempNC_ID2 = 26
LABEL ADR TempNC_R2 = 27
LABEL ADR TempNC_R3 = 28
LABEL ADR TempCH_FIP = 29               ;CompactHeap temporaries.
LABEL ADR TempCH_R2 = 30
LABEL ADR TempCH_R3 = 31
LABEL ADR TempCH_A3 = 32
LABEL ADR TempCH_ID3 = 33
LABEL ADR TempCH_Lock = 34
LABEL ADR TempCH_Src = 35
LABEL ADR TempANO_FIP = 36              ;AllocNewObject temporary.
LABEL ADR TempEB_Key = 37               ;EnterBinding temporary.
LABEL ADR TempLM_FIP = 38               ;LookupMethod temporaries
LABEL ADR TempINITM_Context = 39        ;InitializeMDP temporary.
LABEL ADR TempTOf_FIP = 40              ;ClassOf temporary.
LABEL ADR TempDiv_R2 = TempTOf_FIP      ;Divide temporaries.
LABEL ADR TempDiv_R3 = 41
LABEL ADR TempDiv_80000000 = 42
LABEL ADR TempNCl_FIP = TempDiv_R2      ;NewClosure temporary
LABEL ADR TempDealloc_FIP = 43          ;DeallocateObject temporary
```

```
;+-------------------------------------------------------------------------------------+
;| Fault Numbers                                                                       |
;+-------------------------------------------------------------------------------------+

LABEL VECTOR suspend = $00
LABEL VECTOR blockMove = $01
LABEL VECTOR blockSend = $02
LABEL VECTOR compactHeap = $03
LABEL VECTOR allocObject = $04
LABEL VECTOR enterBinding = $05
LABEL VECTOR lookupBinding = $06
LABEL VECTOR deleteBinding = $07
LABEL VECTOR purgeBinding = $08
LABEL VECTOR newLocalObject = $09
LABEL VECTOR allocNextObject = $0A
LABEL VECTOR allocNewObject = $0B
LABEL VECTOR newContext = $0C
LABEL VECTOR disposeContext = $0D
LABEL VECTOR disposeObject = $0E
LABEL VECTOR deallocateObject = $0F
LABEL VECTOR newObject = $10
LABEL VECTOR classOf = $11
LABEL VECTOR typeOf = $12
LABEL VECTOR objectNode = $13
LABEL VECTOR preferredConstituent = $14
LABEL VECTOR co = $15
LABEL VECTOR lookupMethod = $16
LABEL VECTOR lookupMethodU = $17
LABEL VECTOR divide = $18
LABEL VECTOR newClosure = $19
LABEL VECTOR callClosure = $1A


;+-------------------------------------------------------------------------------------+
;| XLATE Fault Codes                                                                   |
;+-------------------------------------------------------------------------------------+

LABEL objectXLATE = 0                   ;Find and bring the object here.
LABEL internalXLATE = 1                 ;Same as localXLATE but also works for classes and selectors.
LABEL localXLATE = 2                    ;Return the object address, its node number, or NIL if it is a constant.
LABEL restoreXLATE = 3                  ;Restore an address register from a saved ID value.


;+-------------------------------------------------------------------------------------+
;| Halt Codes                                                                          |
;+-------------------------------------------------------------------------------------+

LABEL haltFault0 = 0                    ;General priority 0 fault.
LABEL haltFault1 = 1                    ;General priority 1 fault.
LABEL haltFuture = 2                    ;Futures are not implemented yet.
LABEL haltOverflow = 3                  ;Bignums are not implemented yet.
LABEL haltType = 4                      ;Overriding built-in selectors is not implemented yet.
LABEL haltUser = 5                      ;Halt by user program.
LABEL haltRange = 6                     ;Range exceeded in a primitive operation.
LABEL haltCall = 7                      ;Undefined system call.
LABEL haltInvalidA1 = 8                 ;A1 invalid.
LABEL haltReply = 9                     ;Reply to a bad slot.
LABEL haltUninitVar = 13                ;An uninitialized variable was referenced.
LABEL haltTypeOf = 14                   ;Nonexistent or incorrectly tagged object passed to typeOf.
LABEL haltXLATE = 15                    ;Nonexistent or incorrectly tagged object is XLATEd.
LABEL haltBRATType = 18                 ;An object's BRAT entry is missing or mistyped.
LABEL haltBRATMissing = 17              ;An object's BRAT entry is missing.
LABEL haltBRATDelete = 18               ;Attempt to delete a missing BRAT entry.
LABEL haltClassType = 19                ;Incorrectly tagged word used as a class.
LABEL haltInternalType = 20             ;A non-CST-tagged word used as an object.
LABEL haltBRATFull = 21                 ;The BRAT is full.
LABEL haltMemFull = 22                  ;Memory is full.
LABEL haltApply = 23                    ;Attempt to apply an incorrectly tagged word.
LABEL haltHeap = 24                     ;Heap is in an inconsistent state during a compaction.
LABEL haltLimit = 25                    ;An object's limit is exceeded.
LABEL haltDiv0 = 26                     ;Division by zero.
```

Cosmos.m

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;                                            ;;;;
;;;;             MDP Operating System           ;;;;
;;;;                                            ;;;;
;;;;                 version 2.3                ;;;;
;;;;                                            ;;;;
;;;;                  written by                ;;;;
;;;;               Waldemar Horwat              ;;;;
;;;;                                            ;;;;
;;;;  Master's thesis under Prof. William Dally ;;;;
;;;;                                            ;;;;
;;;;                March 28, 1989              ;;;;
;;;;                   May 1991                 ;;;;
;;;;                                            ;;;;
;;;;        Send problems and comments to       ;;;;
;;;;           waldemar@hx.lcs.mit.edu.         ;;;;
;;;;                                            ;;;;
;;;; Copyright 1989, 1990, 1991 Waldemar Horwat ;;;;
;;;;                                            ;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


INCLUDE "Cosmos.i"

;Each routine and section of code has an attribute called criticality.  A criticality is
;a number between 0 and 7 with the following meanings:
;
;0:  All operations allowed.  Caller's registers not preserved.
;1:  Caller's registers preserved.  May suspend, so caller's globals are not preserved.
;2:  No suspending faults, no modification of context state.
;3:  No suspending faults, no modification of context state, no object migration.
;4:  No message sends, no object migration.
;5:  No heap compaction, no message sends.
;6:  No faults, no heap compaction, no message sends.
;7:  No priority 1 interrupts, no faults, no heap compaction, no message sends.
;
;A routine's criticality is no greater than the criticality of any of its components, subroutines,
;or fault handlers.  The criticality of a fault handler can be no greater than 5.
;Each fault handler's code starts at criticality 6 until the state in the fault registers is saved.
;Dereferencing an address register (other than A0 in absolute mode) has criticality at most 5 unless
;the code has criticality 5 and the address register is known to be valid.
;A routine that uses a global must have criticality at least 2.
;
;Faults that save registers should start in unchecked mode because a register might contain a CFUTure.


@0..NNodes-1

                MODULE
                ENTRY   ALL
                ORG     $400                    ;Reset IPB location.
                DC      InitializeMDP-(*+2)     ;Go initialize!
                BR      R0

                ORG     BRATEnd
OSStart:
```

```
                    ;######################
                    ;##                  ##
                    ;##  Fault Handlers  ##
                    ;##                  ##
                    ;######################

;+-------------------------------------------------------------------------------+
;| Crash on a general priority 0 or 1 or undefined system call fault.            |
;+-------------------------------------------------------------------------------+

Crash0:         HALT    haltFault0
Crash1:         HALT    haltFault1
CrashCall:      HALT    haltCall

fltCrash0 = IP:abs|fault|unchecked|Crash0<<offsetN
fltCrash1 = IP:abs|fault|unchecked|Crash1<<offsetN
fltCrashCall = IP:abs|fault|unchecked|CrashCall<<offsetN

;+-------------------------------------------------------------------------------+
;| Crash on a general future or type fault.                                      |
;+-------------------------------------------------------------------------------+

CrashFuture:    HALT    haltFuture
CrashType:      HALT    haltType

fltCrashFuture = IP:abs|fault|unchecked|CrashFuture<<offsetN
fltCrashType = IP:abs|fault|unchecked|CrashType<<offsetN

;+-------------------------------------------------------------------------------+
;| Handle the early or send fault by re-trying the operation.                    |
;+-------------------------------------------------------------------------------+
;|
;|Criticality 5.
;|

RetryHandler:   MOVE    R0,FOP0                 ;Save R0.  Criticality 6.
                MOVE    FIP,R0                  ;Back up FIP by one instruction.
                ROT     R0,-phaseN,R0
                SUB     R0,1,R0
                ROT     R0,phaseN,R0
                MOVE    R0,FIP
                MOVE    FOP0,R0                 ;Restore R0.
                MOVE    FIP,IP

fltEarly = IP:abs|fault|unchecked|RetryHandler<<offsetN
fltSend = IP:abs|fault|unchecked|RetryHandler<<offsetN

;+-------------------------------------------------------------------------------+
;| Handle a limit fault.  Halt unless LimitOverride was set, in which case clear it and |
;| jump to the override routine.  R0 and R1 are altered when LimitOverride is used.     |
;+-------------------------------------------------------------------------------+
;|
;|Criticality 5.
;|

LimitHandler:   MOVE    [LimitOverride,A0],R0   ;Criticality 6.
                BNIL    R0,^Limit_Halt          ;Halt unless LimitOverride was set.
                MOVE    NIL,R1                  ;Clear LimitOverride.
                MOVE    R1,[LimitOverride,A0]
                MOVE    R0,IP                   ;Go to the override routine.
Limit_Halt:     HALT    haltLimit

fltLimit = IP:abs|fault|unchecked|LimitHandler<<offsetN


;+-------------------------------------------------------------------------------+
;| Handle a CFUTure fault.                                                       |
;+-------------------------------------------------------------------------------+
;|
;|Criticality 1.
;|
;+-------------------------------------------------------------------------------+
;| Save the current state in the context, suspend, and allocate a new fast context for |
;| the next message.  Various entry points are provided depending on how much save has  |
;| to be saved.  These routines do not return.                                          |
;+-------------------------------------------------------------------------------+
;|
;|Entry: SaveStateID023 (ID0, ID2, ID3, and the message, if any, have to be saved.)
;|Entry: SaveStateID03 (ID0, ID3, and the message, if any, have to be saved.)
;|Entry: SaveState (No registers have to be saved.)
;|
;|Unchecked absolute non-fault mode required.
;|

CFUT_Halt:      HALT    haltUninitVar           ;An uninitialized variable was referenced.
CFUTHandler:    MOVE    R0,[contextR0,A1]       ;Save R0 and R1.  Criticality 6.
                MOVE    R1,[contextR1,A1]
                MOVE    R2,[contextR2,A1]
                MOVE    R3,[contextR3,A1]
                MOVE    FOP0,R3
                IF DEBUG
                 GT     R3,0,R0
                 BF     R0,^CFUT_Halt           ;Halt if an uninitialized variable was referenced.
                END
                MOVE    R3,[contextNext,A1]
                MOVE    FIP,R1                  ;Back up FIP by one instruction.
                MOVE    R1,F                    ;Criticality 2.
                ROT     R1,-phaseN,R1
                SUB     R1,1,R1
                ROT     R1,phaseN,R1
                MOVE    R1,[contextIP,A1]       ;Save IP, R2, R3, ID0, ID2, and ID3 in the context.
SaveStateID023: MOVE    ID2,R0                  ;Save ID0, ID2, and ID3 in the current context.
                MOVE    R0,[contextID2,A1]
```

```
SaveStateID03:    MOVE     ID0,R0                  ;Save ID0 and ID3 in the current context.
                  MOVE     R0,[contextID0,A1]
                  MOVE     Q,R0                    ;Check whether the message should be copied into the context.
                  BF       R0,^SaveState_Msg       ;Don't copy if A3 didn't point into the queue.
                  MOVE     16,R0                   ;Copy the message into the context:
                  SUB      R0,[0,A3],R0            ;Jump into the appropriate place in the copy code.
                  AND      R0,lengthM,R0
                  BR       R0
:                 MOVE     [15,A3],R0
                  MOVE     R0,[15,A1]
:                 MOVE     [14,A3],R0
                  MOVE     R0,[14,A1]
:                 MOVE     [13,A3],R0
                  MOVE     R0,[13,A1]
:                 MOVE     [12,A3],R0
                  MOVE     R0,[12,A1]
:                 MOVE     [11,A3],R0
                  MOVE     R0,[11,A1]
:                 MOVE     [10,A3],R0
                  MOVE     R0,[10,A1]
:                 MOVE     [9,A3],R0
                  MOVE     R0,[9,A1]
:                 MOVE     [8,A3],R0
                  MOVE     R0,[8,A1]
:                 MOVE     [7,A3],R0
                  MOVE     R0,[7,A1]
:                 MOVE     [6,A3],R0
                  MOVE     R0,[6,A1]
:                 MOVE     [5,A3],R0
                  MOVE     R0,[5,A1]
:                 MOVE     [4,A3],R0
                  MOVE     R0,[4,A1]
:                 MOVE     [3,A3],R0
                  MOVE     R0,[3,A1]
:                 MOVE     [2,A3],R0
                  MOVE     R0,[2,A1]
:                 MOVE     ID1,R0
                  MOVE     R0,ID3
SaveState_Msg:    MOVE     ID3,R0
                  MOVE     R0,[contextID3,A1]
SaveState:        MOVE     [FastContextQueue,A0],R0 ;Allocate a new fast context.
                  BNIL     R0,^AllocFastContext    ;There are no more.
                  XLATE    R0,objectXLATE,A1
                  MOVE     [contextNext,A1],R0     ;Unlink it.
                  MOVE     R0,[FastContextQueue,A0]
                  SUSPEND                          ;Criticality 1.


;+-----------------------------------------------------------------------------+
;| Allocate and initialize a new fast context to be used by the next message.  This  |
;| routine does not return.                                                     |
;+-----------------------------------------------------------------------------+
;|
;|Entry: AllocFastContext
;|
;|Unchecked absolute non-fault mode required.
;|

AllocFastContext: DC       OBJ:hdrLocked|contextSize
                  CALL     allocNextObject         ;Create the context object.
                  MOVE     ID2,R1                  ;Point A1 and ID1 to the new context.
                  XLATE    R1,objectXLATE,A1
                  SUSPEND


;+-----------------------------------------------------------------------------+
;| Suspend: if a slow context was used, deallocate it and replace it with a fast one.  |
;| This routine does not return.                                                |
;+-----------------------------------------------------------------------------+
;|
;|Call: suspend
;|
;|In:   AID1     Context.
;|
;|Criticality 0.
;|

Suspend:          MOVE     [contextHeader,A1],R0   ;Criticality 3.
                  ROT      R0,-hdrFastN,R0         ;Check whether this was a fast context.
                  BT       R0,^Suspend_Fast        ;Yes.
                  MOVE     ID1,R0                  ;No.  Dispose this context and allocate a new one.
                  CALL     disposeObject
                  BR       ^SaveState
Suspend_Fast:     SUSPEND

fltCFUT = IP:abs|fault|unchecked|CFUTHandler<<offsetN
fltSuspend = IP:abs|unchecked|Suspend<<offsetN


;+-----------------------------------------------------------------------------+
;| Handle an INVADR fault.  If the object is on this node, store its address in the    |
;| address register; if it is not on this node, bring it here.                  |
;+-----------------------------------------------------------------------------+
;|
;|Criticality 1.
;|

INVADRHandler:    MOVE     R1,[TempXLATE_R1,A0]    ;Save R1.  Criticality 6.
                  MOVE     FIR,R1
                  AND      R1,3,R1                 ;FIR contains the correct address register number,
                  LSH      R1,2,R1                 ;even when FIR=NIL.
                  BNZ      R1,R1                   ;Go to one of four handlers
                  MOVE     ID0,R1
                  PROBE    R1,R1                   ;Check the xlate cache first.
                  BNIL     R1,^INVADR_Miss0        ;Jump into the objectXLATE handler if missed.
                  MOVE     R1,A0
                  MOVE     FIR,R1
                  BNIL     R1,^INVADR_Rstrt2       ;If the FIR was NIL, don't back up the FIP.
```

```
                MOVE    FIP,R1
                BR      ^INVADR_Restart
:               HALT    haltInvalidA1
INVADR_Miss:    MOVE    R0,[TempXLATE_R0,A0]    ;Save R0, R2, and FIP.
                MOVE    FIP,R0
                MOVE    R0,[TempXLATE_FIP,A0]
                MOVE    R2,[TempXLATE_R2,A0]
                MOVE    R0,F                    ;Criticality 5.
                BR      ^XLATE_ToObject         ;Jump into the objectXLATE handler if missed.
:               MOVE    ID2,R1
                PROBE   R1,R1                   ;Check the xlate cache first.
                BNIL    R1,^INVADR_Miss2        ;Jump into the objectXLATE handler if missed.
                MOVE    R1,A2
                MOVE    FIP,R1
                BR      ^INVADR_Restart
INVADR_Miss2:   MOVE    ID2,R1
                BR      ^INVADR_Miss
:               MOVE    ID3,R1
                PROBE   R1,R1                   ;Check the xlate cache first.
                BNIL    R1,^INVADR_Miss3        ;Jump into the objectXLATE handler if missed.
                MOVE    R1,A3
                MOVE    FIP,R1
INVADR_Restart: ROT     R1,-phaseN,R1           ;Restart the instruction.
                SUB     R1,1,R1
                ROT     R1,phaseN,R1
                MOVE    R1,FIP
INVADR_Rstrt2:  MOVE    [TempXLATE_R1,A0],R1
                MOVE    FIP,IP

INVADR_Miss0:   MOVE    FIR,R1
                BNNIL   R1,^INVADR_M0_2
                MOVE    FIP,R1                  ;Advance the FIP if the FIR was NIL.
                ROT     R1,-phaseN,R1
                ADD     R1,1,R1
                ROT     R1,phaseN,R1
                MOVE    R1,FIP
INVADR_M0_2:    MOVE    ID0,R1
                BR      ^INVADR_Miss
INVADR_Miss3:   MOVE    ID3,R1
                BR      ^INVADR_Miss


;+-------------------------------------------------------------------------------+
;| Handle an XLATE fault.                                                        |
;| Two bits of the instruction are used to determine what to do.  The possible actions |
;| are:                                                                          |
;|  objectXLATE:  Return an ADDR containing the object's address.  If the object is not |
;|       on this node, bring it here.  Rs must be an ID, a DID, a class, or a selector. |
;|  localXLATE:  If Rs represents an object on this node, return its address; if Rs is |
;|       a constant, return NIL; otherwise, return the number of a node likely to |
;|       contain the object.  This mode can be used only when Rd is a data register.  Rs |
;|       must be an ID, DID, a class, a selector, or a constant.                 |
;|  internalXLATE:  Same as localXLATE except that treats futures as if they were |
;|       objects instead of constants.                                           |
;|  restoreXLATE:  Invalidate Rd by storing an invalid address there.  Of course, if |
;|       the XLATE table hits, the value associated with Rs is stored in Rd instead. |
;| XLATE should be made to fault on FUTures or CFUTures; this can be accomplished by |
;| calling XLATE in checked mode.                                                |
;+-------------------------------------------------------------------------------+
;|
;|The criticalities are as follows:
;| objectXLATE: Criticality 1 (criticality 5 if the object is known to reside on this node).
;| internalXLATE: Criticality 5.
;| localXLATE: Criticality 5.
;| restoreXLATE: Criticality 5.
;|

XLATEHandler:   MOVE    R0,[TempXLATE_R0,A0]    ;Save R0, R1, R2, FIP, and FIR.  Criticality 6.
                MOVE    R1,[TempXLATE_R1,A0]
                MOVE    R2,[TempXLATE_R2,A0]
                MOVE    FIP,R0
                MOVE    R0,[TempXLATE_FIP,A0]
                MOVE    FIR,R2
                MOVE    FOP1,R1
                MOVE    R0,F                    ;Criticality 5.
                ROT     R2,-9,R0
                AND     R2,7,R2                 ;Save the destination addressing mode in R2.
                AND     R0,3,R0
                BR      R0
:               BR      ^XLATE_ToObject         ;Get the object.
:               BR      ^XLATE_Internal         ;Go to the internal code.
:               BR      ^XLATE_Local            ;Go to the local code.
:               DC      ADDR:rel|invalid        ;RestoreXLATE:  Invalidate the address register.
XLATE_Result:   ROT     R2,1,R2                 ;Store R0 in the destination of the XLATE.  R1 contains the
                BR      R2                      ;value of Rs and is stored in the ID register.  R2 contains the
:               MOVE    [TempXLATE_R2,A0],R2    ;addressing mode from the XLATE instruction.
                MOVE    [TempXLATE_R1,A0],R1
                MOVE    [TempXLATE_FIP,A0],IP
:               MOVE    R0,R1
                MOVE    [TempXLATE_R2,A0],R2
                MOVE    [TempXLATE_R0,A0],R0
                MOVE    [TempXLATE_FIP,A0],IP
:               MOVE    R0,R2
                MOVE    [TempXLATE_R1,A0],R1
                MOVE    [TempXLATE_R0,A0],R0
                MOVE    [TempXLATE_FIP,A0],IP
:               MOVE    R0,R3
                BR      ^XLATE_R_Done
                DC      0
:               MOVE    R0,A0
                BR      ^XLATE_R_Done
                DC      0
:               MOVE    R0,A1
                BR      ^XLATE_R_Done
                DC      0
:               MOVE    R0,A2
                BR      ^XLATE_R_Done
                DC      0
```

```
:               MOVE    R0,A3
                MOVE    R1,ID3
XLATE_R_Done:   MOVE    [TempXLATE_R2,A0],R2
                MOVE    [TempXLATE_R1,A0],R1
                MOVE    [TempXLATE_R0,A0],R0
                MOVE    [TempXLATE_FIP,A0],IP

XLATE_L_TAG0:   AND     R0,subtagM,R0             ;If the value is a class, pretend it is an ID.
                SUB     R0,subCLASS,R0
                BZ      R0,^XLATE_I_SC
                SUB     R0,subSEL-subCLASS,R0
                BZ      R0,^XLATE_I_SC
                BR      ^XLATE_L_NIL

XLATE_ToObject: CHECK   R1,ID,R0
                BR      ^XLATE_Object

XLATE_Internal: CHECK   R1,FUT,R0                 ;XLATE_Internal is the same as XLATE_Local for values which
                BT      R0,^XLATE_I_SC            ;aren't futures.
XLATE_Local:    RTAG    R1,R0                     ;Dispatch on the tag of the object.
                BR      R0
:               ROT     R1,-subtagN,R0            ;TAG0
                BR      ^XLATE_L_TAG0
XLATE_L_NIL:    MOVE    NIL,R0                    ;INT
                BR      ^XLATE_Result
:               MOVE    NIL,R0                    ;BOOL
                BR      ^XLATE_Result
:               HALT    haltXLATE                 ;ADDR
:               HALT    haltXLATE                 ;IP
:               HALT    haltXLATE                 ;MSG / OBJ
:               HALT    haltXLATE                 ;CFUT
:               HALT    haltXLATE                 ;FUT.
XLATE_I_SC:     MOVE    R2,[TempXLATE_FIR,A0]     ;ID.  Save the FIR.
                BR      ^XLATE_L_ID
:               MOVE    R2,[TempXLATE_FIR,A0]     ;DID.  Save the FIR.
                BR      ^XLATE_L_DID
:               MOVE    NIL,R0                    ;TAGA
                BR      ^XLATE_Result
:               MOVE    NIL,R0                    ;FLOAT
                BR      ^XLATE_Result
:               HALT    haltXLATE                 ;INST0
:               HALT    haltXLATE                 ;INST1
:               HALT    haltXLATE                 ;INST2
XLATE_Halt:     HALT    haltXLATE                 ;INST3
XLATE_L_DID:    MOVE    R1,R2                     ;Save the DID.
                CALL    preferredConstituent      ;Get an ID from the DID.
                PROBE   R1,R0                     ;Check if the constituent ID is in the cache.
                BNIL    R0,^XLATE_L_ID
                ENTER   R2,R0                     ;If so, enter and return it.
                BR      ^XLATE_L_2
XLATE_L_ID:     CALL    lookupBinding             ;Look for a binding of the object on
                BNIL    R0,^XLATE_L_Miss          ;this node.
                CHECK   R0,INT,R2                 ;If an integer was found, it is the object's current
                BT      R2,^XLATE_L_2             ;node number.
                CHECK   R0,ADDR,R2
                BF      R2,^XLATE_L_Cxt
                MOVE    R0,[TempXLATE_Temp,A0]    ;Save R0.
                ROT     R0,-baseN,R2
                AND     R2,baseM,R2
                MOVE    [R2,A0],R0                ;Fetch the object's header and clear the marked flag in it.
                OR      R0,hdrMarked,R0
                XOR     R0,hdrMarked,R0
                MOVE    R0,[R2,A0]
                MOVE    [TempXLATE_Temp,A0],R0    ;Restore R0.
                ENTER   R1,R0                     ;Found such a binding.  Enter it in the XLATE table.
                BR      ^XLATE_L_2
XLATE_L_Miss:   MOVE    [NodeMask,A0],R0          ;Did not find a binding.  Extract the node number from
                AND     R0,R1,R0                  ;the ID and go return it.
                MOVE    NNR,R2                    ;If the node number is this node, halt because this is
                EQUAL   R0,R2,R2                  ;supposed to be the home node, yet it doesn't know where the
                BT      R2,^XLATE_Halt            ;object is.
XLATE_L_2:      MOVE    [TempXLATE_FIR,A0],R2     ;Go return the result in R0.
                BR      ^XLATE_Result

XLATE_L_Cxt:    CHECK   R0,ID,R2                  ;The ID is bound to a context.  Dereference the context and
                BF      R2,^XLATE_Halt            ;read its contextNext field.
                MOVE    R1,[TempXLATE_Temp,A0]    ;Save the object ID.
                MOVE    R0,R1
                CALL    lookupBinding
                MOVE    [TempXLATE_Temp,A0],R1    ;Restore the object ID.
                CHECK   R0,ADDR,R2
                BF      R2,^XLATE_Halt
                ROT     R0,-baseN,R2
                AND     R2,baseM,R2
                MOVE    contextNext,R0
                ADD     R2,R0,R2
                MOVE    [R2,A0],R0
                BNIL    R0,^XLATE_L_Miss          ;Miss if it was NIL.
                CHECK   R0,INT,R2                 ;If an integer was found, it is the object's current
                BF      R2,^XLATE_L_Cxt           ;node number; otherwise there is another context linked.
                BR      ^XLATE_L_2

XLATE_O_Access: MOVE    R0,[TempXLATE_Temp,A0]    ;Save R0.
                ROT     R0,-baseN,R2
                AND     R2,baseM,R2
                MOVE    [R2,A0],R0                ;Fetch the object's header and clear the marked flag in it.
                OR      R0,hdrMarked,R0
                XOR     R0,hdrMarked,R0
                MOVE    R0,[R2,A0]
                MOVE    [TempXLATE_Temp,A0],R0    ;Restore R0.
                ENTER   R1,R0                     ;Found a binding.  Enter it in the XLATE table and
                MOVE    [TempXLATE_R2,A0],R2      ;restart the XLATE (or address-faulted) instruction.
XLATE_O_Rebind: DC      phase
                MOVE    [TempXLATE_FIP,A0],R1
                SUB     R1,R0,R1
                MOVE    R1,[TempXLATE_FIP,A0]
                MOVE    [TempXLATE_R1,A0],R1
                MOVE    [TempXLATE_R0,A0],R0
```

```
                MOVE    [TempXLATE_FIP,A0],IP

                IF DEBUG
XLATE_O_2:       CHECK  R1,TAG0,R0                ;Classes and selectors are also objects and are
                 BF     R0,^XLATE_Halt_2          ;treated as if they  ere ID's.
                 ROT    R1,-subtagN,R0
                 AND    R0,subtagM,R0
                 SUB    R0,subCLASS,R0
                 BZ     R0,^XLATE_O_ID
                 SUB    R0,subSEL-subCLASS,R0
                 BZ     R0,^XLATE_O_ID
                END
XLATE_Halt_2:   HALT    haltXLATE
XLATE_Object:   BT      R0,^XLATE_O_ID            ;Dispatch on the tag of the object.
                CHECK   R1,DID,R0
                BF      R0,^XLATE_O_2
                MOVE    R1,R2                     ;Save the DID.
                CALL    preferredConstituent      ;Get an ID from the DID.
                PROBE   R1,R0                     ;Check if the constituent ID is in the cache.
                BNIL    R0,^XLATE_O_ID
                ENTER   R2,R0                     ;If so, enter and return it.
                MOVE    [TempXLATE_R2,A0],R2
                BR      ^XLATE_O_Rebind

                IF !DEBUG
XLATE_O_2:
                END
XLATE_O_ID:     CALL    lookupBinding             ;Look for a binding of the object.
                BNIL    R0,^XLATE_O_Miss
                MOVE    R2,[TempXLATE_Temp,A0]    ;Save the binding's address.
                CHECK   R0,ADDR,R2                ;If the object's address was found, return it.
                BT      R2,^XLATE_O_Access
                CHECK   R0,INT,R2                 ;If the object's current node was found, send a requestObject
                BT      R2,^XLATE_O_Point         ;message there.
                MOVE    [TempXLATE_Temp,A0],R2    ;Otherwise someone is already waiting for the object.
                BR      ^XLATE_O_Fetch            ;Append this context to the waiting queue.
XLATE_O_Point:  MOVE    [TempXLATE_Temp,A0],R2
                SEND0   R0                        ;Send a message requesting the object to the object's
                DC      MSG:msgRequestObject+3    ;current location.
                SEND0   R0
                MOVE    NNR,R0
                SEND2E0 R1,R0
                BR      ^XLATE_O_Fetch
XLATE_O_Miss:   DC      MSG:msgRequestObject+3    ;Send a message requesting the object to the object's
                AND     R1,[NodeMask,A0],R2       ;home.
                SEND20  R2,R0
                MOVE    NNR,R0
                SEND2E0 R1,R0                     ;However, if this node is supposed to be the object's home,
                EQUAL   R0,R2,R2                  ;halt because the object doesn't appear to exist.
                BT      R2,^XLATE_Halt_2
                MOVE    NIL,R2                    ;R2 being NIL means no one else is waiting for the object.
XLATE_O_Fetch:  MOVE    [TempXLATE_R0,A0],R0      ;Save state in the context.
                MOVE    R0,[contextR0,A1]
                MOVE    [TempXLATE_R1,A0],R0
                MOVE    R0,[contextR1,A1]
                MOVE    [TempXLATE_R2,A0],R0
                MOVE    R0,[contextR2,A1]
                MOVE    [TempXLATE_FIP,A0],R0
                ROT     R0,-phaseN,R0             ;Criticality 3.
                SUB     R0,1,R0                   ;Back up IP to point to the XLATE instruction.
                ROT     R0,phaseN,R0
                MOVE    R0,[contextIP,A1]         ;Save IP, R0-R3, ID0, ID2, and ID3 in the context.
                MOVE    R3,[contextR3,A1]
                MOVE    ID2,R0
                MOVE    R0,[contextID2,A1]
                MOVE    ID1,R0
                BNNIL   R2,^XLATE_O_Append
                MOVE    R2,[contextNext,A1]       ;Make a binding indicating that the context in ID1 is
                CALL    enterBinding              ;waiting for the object in R1.
XLATE_Suspend:  DC      SaveStateID03-(*+2)       ;Save the rest of the state and suspend.
                BR      R0
XLATE_O_Append: MOVE    [R2,A0],R3                ;Append the binding to the linked list headed by R2.
                MOVE    R3,[contextNext,A1]
                MOVE    R0,[R2,A0]
                BR      ^XLATE_Suspend

fltINVADR = IP:abs|fault|unchecked|INVADRHandler<<offsetN
fltXLATE = IP:abs|fault|unchecked|XLATEHandler<<offsetN
```

```
                           ;####################
                           ;##                ##
                           ;##  Heap Manager  ##
                           ;##                ##
                           ;####################

;+-------------------------------------------------------------------------------+
;| Copy the object pointed by A3 into the object pointed by A2.  The copy stops as soon |
;| as a limit fault is reached.  A2 and A3 are guaranteed not to be XLATEd, so they do  |
;| not have to correspond to the values in the ID registers.                            |
;| The objects are copied from the bottom up, so, if they overlap, the destination must |
;| start before the source.                                                             |
;| R0 can be a number smaller than 32 indicating the offset of the first word in each   |
;| object which should be copied; words with indices smaller than R0 are not copied.    |
;+-------------------------------------------------------------------------------+
;|
;|Call: blockMove
;|
;|In:    R0      Offset of first word to copy.
;|       A2      Destination object pointer.
;|       A3      Source object pointer.
;|
;|Criticality 5.
;|
;|Alters R0/R1.
;|

BlockMove:      MOVE    FIP,R1                  ;Criticality 6.
                MOVE    R1,[LimitOverride,A0]   ;Override the Limit fault for the duration of this routine.
                MOVE    R1,F                    ;Criticality 5.
                BR      R0
:               MOVE    [0,A3],R0
                MOVE    R0,[0,A2]
:               MOVE    [1,A3],R0
                MOVE    R0,[1,A2]
:               MOVE    [2,A3],R0
                MOVE    R0,[2,A2]
:               MOVE    [3,A3],R0
                MOVE    R0,[3,A2]
:               MOVE    [4,A3],R0
                MOVE    R0,[4,A2]
:               MOVE    [5,A3],R0
                MOVE    R0,[5,A2]
:               MOVE    [6,A3],R0
                MOVE    R0,[6,A2]
:               MOVE    [7,A3],R0
                MOVE    R0,[7,A2]
:               MOVE    [8,A3],R0
                MOVE    R0,[8,A2]
:               MOVE    [9,A3],R0
                MOVE    R0,[9,A2]
:               MOVE    [10,A3],R0
                MOVE    R0,[10,A2]
:               MOVE    [11,A3],R0
                MOVE    R0,[11,A2]
:               MOVE    [12,A3],R0
                MOVE    R0,[12,A2]
:               MOVE    [13,A3],R0
                MOVE    R0,[13,A2]
:               MOVE    [14,A3],R0
                MOVE    R0,[14,A2]
:               MOVE    [15,A3],R0
                MOVE    R0,[15,A2]
:               MOVE    [16,A3],R0
                MOVE    R0,[16,A2]
:               MOVE    [17,A3],R0
                MOVE    R0,[17,A2]
:               MOVE    [18,A3],R0
                MOVE    R0,[18,A2]
:               MOVE    [19,A3],R0
                MOVE    R0,[19,A2]
:               MOVE    [20,A3],R0
                MOVE    R0,[20,A2]
:               MOVE    [21,A3],R0
                MOVE    R0,[21,A2]
:               MOVE    [22,A3],R0
                MOVE    R0,[22,A2]
:               MOVE    [23,A3],R0
                MOVE    R0,[23,A2]
:               MOVE    [24,A3],R0
                MOVE    R0,[24,A2]
:               MOVE    [25,A3],R0
                MOVE    R0,[25,A2]
:               MOVE    [26,A3],R0
                MOVE    R0,[26,A2]
:               MOVE    [27,A3],R0
                MOVE    R0,[27,A2]
:               MOVE    [28,A3],R0
                MOVE    R0,[28,A2]
:               MOVE    [29,A3],R0
                MOVE    R0,[29,A2]
:               MOVE    [30,A3],R0
                MOVE    R0,[30,A2]
:               MOVE    [31,A3],R0
                MOVE    R0,[31,A2]
:               MOVE    32,R1
BM_MoveRest:    MOVE    [R1,A3],R0              ;Move the rest of the object.
                MOVE    R0,[R1,A2]
                ADD     R1,1,R1
                MOVE    [R1,A3],R0
                MOVE    R0,[R1,A2]
                ADD     R1,1,R1
                MOVE    [R1,A3],R0
                MOVE    R0,[R1,A2]
                ADD     R1,1,R1
                MOVE    [R1,A3],R0
                MOVE    R0,[R1,A2]
                BR      ^BM_MoveRest
```

```
fltBlockMove = IP:abs|fault|unchecked|BlockMove<<offsetN


;+-------------------------------------------------------------------------------+
;| Send the object pointed by A2.  The send stops as soon as a limit fault is reached.  |
;| A2 is guaranteed not to be XLATEd, so it does not have to correspond to the value in |
;| ID2.  R0 should be one of the following:                                      |
;|  0:  Words are sent starting from offset 1 in the object.                     |
;|  1:  Words are sent starting from offset 3 in the object.                     |
;|  2:  Words are sent starting from offset 5 in the object.                     |
;+-------------------------------------------------------------------------------+
;|
;|Call: blockSend
;|
;|In:   R0      Encoded offset of first word to send.
;|      A2      Source object pointer.
;|
;|Criticality 5.
;|
;|Alters R0/R1/A2.
;|

BlockSend:      MOVE    FIP,R1                  ;Criticality 6.
                MOVE    R1,[LimitOverride,A0]   ;Override the Limit fault for the duration of this routine.
                MOVE    R1,F                    ;Criticality 5.
                BR      R0
:               SENDO   [1,A2]
                SENDO   [2,A2]
:               SENDO   [3,A2]
                SENDO   [4,A2]
:               SENDO   [5,A2]
                SENDO   [6,A2]
                SENDO   [7,A2]
                SENDO   [8,A2]
                SENDO   [9,A2]
                SENDO   [10,A2]
                SENDO   [11,A2]
                SENDO   [12,A2]
                SENDO   [13,A2]
                SENDO   [14,A2]
                SENDO   [15,A2]
                SENDO   [16,A2]
                SENDO   [17,A2]
                SENDO   [18,A2]
                SENDO   [19,A2]
                SENDO   [20,A2]
                SENDO   [21,A2]
                SENDO   [22,A2]
                SENDO   [23,A2]
                SENDO   [24,A2]
                SENDO   [25,A2]
                SENDO   [26,A2]
                SENDO   [27,A2]
                SENDO   [28,A2]
                SENDO   [29,A2]
                SENDO   [30,A2]
                SENDO   [31,A2]
                MOVE    32,R0
BS_SendRest:    SENDO   [R0,A2]                 ;Sends more words of the object.
                ADD     R0,1,R0
                SENDO   [R0,A2]
                ADD     R0,1,R0
                SENDO   [R0,A2]
                ADD     R0,1,R0
                SENDO   [R0,A2]
                ADD     R0,1,R0
                BR      ^BS_SendRest

fltBlockSend = IP:abs|fault|unchecked BlockSend<<offsetN


;+-------------------------------------------------------------------------------+
;| Compact the node's heap, trying to free at least R1 words of memory.  Halt if this  |
;| much memory is not available.                                                 |
;+-------------------------------------------------------------------------------+
;|
;|Call: compactHeap
;|
;|In:   R0      Number of words needed.
;|
;|Criticality 3.
;|
;|Alters R0/R1/AID2.
;|

CompactHeap:    INVAL                           ;Criticality 6.  Invalidate all relocatable address registers.
                MOVE    R0,[TempCH_R0,A0]       ;Save FIP, R0, R2, R3, Q, A3, and ID3.
                MOVE    R2,[TempCH_R2,A0]
                MOVE    R3,[TempCH_R3,A0]
                MOVE    FIP,R0
                MOVE    R0,[TempCH_FIP,A0]
                MOVE    R0,F                    ;Criticality 3.
                MOVE    Q,R0
                BT      R0,^CH_Q
                MOVE    ID3,R0
CH_Q:           MOVE    R0,[TempCH_ID3,A0]
                MOVE    A3,R0
                MOVE    R0,[TempCH_A3,A0]
                MOVE    NIL,R3                  ;R3 will contain NIL for the duration of the xlate flush.
                MOVE    R3,[TempCH_Lock,A0]     ;Indicate that this is the first time the heap is compacted.
                MOVE    R3,Q                    ;Disable queue wraparound.
                DC      IP:abs|unchecked CH_2<<offsetN
                MOVE    R0,[LimitOverride,A0]   ;Override the Limit fault for the duration of this routine
                MOVE    -1,R2
                DC      ADDR:(XlateStart<<baseN)+XlateEnd-XlateStart
                MOVE    R0,A2
CH_FlushXlate:  ADD     R2,2,R2                 ;Check every entry in the XLATE table whether it contains
```

```
                MOVE    [R2,A2],R0              ;a relocatable ADDR.  If it does, replace it with NIL.
                CHECK   R0,ADDR,R1
                BF      R1,^CH_FlushXlate
                AND     R0,rel,R1
                BZ      R1,^CH_FlushXlate
                MOVE    R3,[R2,A2]
                BR      ^CH_FlushXlate

CH_2:           MOVE    [HeapStart,A0],R2       ;R2 is the source heap scanner.
                MOVE    R2,R3                   ;R3 is the destination heap scanner.
                BR      ^CH_Compact
CH_Compact2:    MOVE    [R2,A0],R0              ;Get the next object at the source.
                ROT     R0,-hdrLockedN,R1       ;Let it live if it is locked.
                BT      R1,^CH_Live
                ROT     R0,-hdrDeletedN,R1      ;Kill it if it is deleted.
                BT      R1,^CH_Die
                ROT     R0,-hdrMarkedN,R1       ;Purge it if is is marked.
                BF      R1,^CH_MarkLive
                ADD     R2,1,R1                 ;Read the object's ID into R1.
                MOVE    [R1,A0],R1
                CALL    deleteBinding           ;No need to purge the xlate table.
                MOVE    [R2,A0],R0
CH_Die:         AND     R0,hdrLengthM,R0        ;Skip the source heap scanner past the removed object.
                ADD     R2,R0,R2
CH_Compact:     GE      R2,[FirstFree,A0],R0    ;Check whether the entire heap was scanned.
                BF      R0,^CH_Compact2
                EQUAL   R2,[FirstFree,A0],R0    ;If so, then R2 must match FirstFree exactly.
                BF      R0,^CH_AlignError
                MOVE    R3,[FirstFree,A0]       ;Update FirstFree.
                MOVE    [LastFree,A0],R0
                SUB     R0,R3,R0                ;Check whether there is now enough room to satisfy the allocation
                GE      R0,[TempCH_R0,A0],R0    ;request.
                BT      R0,^CH_Done             ;Leave if so.
                MOVE    [TempCH_Lock,A0],R0
                MOVE    TRUE,R1                 ;If not, compact the heap again unless it was just compacted.
                MOVE    R1,[TempCH_Lock,A0]
                BNIL    R0,^CH_2
                HALT    haltMemFull             ;Give up if two successive compactions don't free enough space.

CH_Done:        MOVE    A1,R0                   ;Make sure that A1 is valid.
                ROT     R0,-invalid,R0
                BF      R0,^CH_A1Valid
                MOVE    ID1,R0                  ;If not, re-xlate it.
                XLATE   R0,objectXLATE,A1
CH_A1Valid:     MOVE    [TempCH_A3,A0],R0       ;Restore A3, Q, and ID3.
                MOVE    R0,A3
                MOVE    [TempCH_ID3,A0],R0
                EQ      R0,TRUE,R1
                BT      R1,^CH_DoneQ
                MOVE    R0,ID3
                MOVE    [TempCH_R2,A0],R2       ;Restore R2.
                BR      ^CH_Done2
CH_DoneQ:       MOVE    R0,Q
                MOVE    [TempCH_R2,A0],R2       ;Restore R2.
CH_Done2:       MOVE    [TempCH_R3,A0],R3       ;Restore R3 and return.
                MOVE    [TempCH_FIP,A0],IP

CH_MarkLive:    ROT     R0,-hdrPurgeableN,R1    ;If this object is purgeable, mark it so that it will be purged
                BF      R1,^CH_Live             ;on the next scan.
                OR      R0,hdrMarked,R0
                MOVE    R0,[R2,A0]
CH_Live:        AND     R0,hdrLengthM,R0        ;Store the length of the object in R0.
                ROT     R3,baseN,R1             ;Point A2 to the destination object.
                ADD     R1,R0,R1
                MOVE    R1,A2
                ROT     R2,baseN,R1             ;Point A3 to the source object.
                ADD     R1,R0,R1
                ADD     R2,R0,R2                ;Advance the source and destination scanners.
                ADD     R3,R0,R3
                EQUAL   R2,R3,R0                ;There is no need to move an object if the source and destination
                BT      R0,^CH_Compact          ;addresses are the same.
                MOVE    R1,A3
                MOVE    [objectID,A3],R1        ;Update the object's binding in the BRAT.
                MOVE    R2,[TempCH_Src,A0]      ;Save R2.
                CALL    lookupBinding
                CHECK   R0,ADDR,R1              ;Make sure that the binding is an ADDR.
                BF      R1,^CH_BRATError
                MOVE    A2,R0
                OR      R0,rel,R0
                MOVE    R0,[R2,A0]
                MOVE    0,R0
                CALL    blockMove               ;Move the object to its destination location.
                MOVE    [TempCH_Src,A0],R2      ;Restore R2.
                BR      ^CH_Compact

CH_AlignError:  HALT    haltHeap
CH_BRATError:   HALT    haltBRATType
fltCompactHeap = IP:abs|fault|unchecked|CompactHeap<<offsetN
```

```
;+-----------------------------------------------------------------------------+
;| Allocate and initialize a new heap object.  R0 contains the word to be stored as the |
;| first word of the object.  The length is extracted from R0, and the flags in the    |
;| high bits of R0 should be set to benign values.  R1 contains the ID for the object.  |
;| The object is not entered in the XLATE and the BRAT tables.                         |
;+-----------------------------------------------------------------------------+
;|
;|Call: allocObject
;|
;|In:   R0      First word of object.
;|      R1      ID of the object.
;|
;|Out:  AID2    ^Object.
;|      R0      ADDR pointing to object.
;|
;|Criticality 3.
;|
;|Alters R0/R2/R3/AID2.
;|

AllocObject:    MOVE    R0,R3                   ;Save R0.  Criticality 6.
                MOVE    R1,ID2                  ;Store the object's ID in ID2.
AO_Retry:       MOVE    [FirstFree,A0],R1       ;Advance the heap scanner.
                AND     R3,hdrLengthM,R2
                ADD     R1,R2,R2
                MOVE    [LastFree,A0],R0        ;Always leave three words on the heap in case a BRAT entry
                SUB     R0,3,R0                 ;needs to be allocated for this object.
                GT      R2,R0,R0                ;Check whether the heap overflowed.
                BF      R0,^AO_1
                AND     R3,hdrLengthM,R0        ;If it did, compact the heap, telling the compactor that
                ADD     R0,3,R0                 ;at least three plus the length of the object words are needed.
                MOVE    FIP,R2
                MOVE    R2,F                    ;Criticality 3.
                CALL    compactHeap
                MOVE    TRUE,R0
                MOVE    R0,F                    ;Criticality 6.
                MOVE    R2,FIP
                BR      ^AO_Retry               ;Go try the allocation again.
AO_1:           MOVE    R2,[FirstFree,A0]
                ROT     R1,baseN,R1             ;Create a base/address pair for the object.
                AND     R3,hdrLengthM,R0
                OR      R1,R0,R0
                OR      R0,rel,R0               ;Mark the object as relocatable.
                WTAG    R0,ADDR,R0
                MOVE    R0,A2                   ;Store a pointer to the object in A2.
                MOVE    ID2,R1
                MOVE    R3,[objectHeader,A2]    ;Write the object's header and ID.
                MOVE    R1,[objectID,A2]
                MOVE    FIP,IP

fltAllocObject - IP:abs|fault|unchecked|AllocObject<<offsetN
```

```
                                ;####################
                                ;##                ##
                                ;##  BRAT Manager  ##
                                ;##                ##
                                ;####################

;+------------------------------------------------------------------------------+
;| Enter a binding of R1 to R0 in the BRAT.  The BRAT should not have an existing |
;| binding of R1.  It may also be appropriate to enter the object into the xlate table. |
;+------------------------------------------------------------------------------+
;|
;|Call: enterBinding
;|
;|In:    R0       Data.
;|       R1       Key.
;|
;|Criticality 3.
;|
;|Alters R0-R3/AID2.
;|

EnterBinding:   MOVE    R0,R3                   ;Criticality 6.  Save data in R3.
                MOVE    R1,[TempEB_Key,A0]      ;Save the key.
EB_1:           ROT     R1,-BRATLenLog*4,R2     ;Calculate the hash code for the key.
                XOR     R1,R2,R2                ;The hash code is the XOR of the BRATLenLog-bit fields of
                ROT     R2,-BRATLenLog*2,R0     ;the key.
                XOR     R2,R0,R2
                ROT     R2,-BRATLenLog,R0
                XOR     R2,R0,R2
                MOVE    BRATLength-1,R0
                AND     R2,R0,R2                ;R2 contains a hash code between 0 and BRATLength-1.
                DC      BRATStart
                ADD     R2,R0,R0                ;R0 points to the head of the BRAT chain.
                MOVE    [BRATFree,A0],R2        ;R2 points to a free BRAT link.
                BNIL    R2,^EB_BRATFull         ;Compact the heap if the BRAT is full.
EB_2:           MOVE    [TempEB_Key,A0],R1
                MOVE    R1,[R2,A0]              ;Store the key in the link.
                MOVE    [R0,A0],R1              ;Save the second link in the chain in R1.
                MOVE    R2,[R0,A0]              ;Make this link be the first in the chain.
                ADD     R2,1,R2
                MOVE    R3,[R2,A0]              ;Store the data in the link.
                ADD     R2,1,R2
                MOVE    [R2,A0],R0              ;Put the next free link in BRATFree.
                MOVE    R0,[BRATFree,A0]
                MOVE    R1,[R2,A0]              ;Link with the second link in the chain.
                MOVE    FIP,IP

EB_BRATFull:    MOVE    [LastFree,A0],R2        ;Attempt to allocate three words from the back of the heap.
                SUB     R2,3,R2
                GE      R2,[FirstFree,A0],R1
                BF      R1,^EB_HeapFull         ;Go compact the heap if it was full.
                MOVE    R2,[LastFree,A0]
                ADD     R2,2,R2                 ;Store a NIL in the link word of the new entry and go
                MOVE    NIL,R1                  ;allocate this entry in the BRAT.
                MOVE    R1,[R2,A0]
                SUB     R2,2,R2
                BR      ^EB_2

EB_HeapFull:    MOVE    FIP,R2                  ;Save the FIP.
                MOVE    3,R0                    ;At least three free words are needed on the heap.
                MOVE    R2,F                    ;Criticality 3.
                CALL    compactHeap
                MOVE    TRUE,R0
                MOVE    R0,F                    ;Criticality 6.
                MOVE    R2,FIP                  ;Restore the FIP.
                MOVE    [TempEB_Key,A0],R1      ;Restore the key and go back to the beginning.
                BR      ^EB_1

fltEnterBinding = IP:abs|fault|unchecked|EnterBinding<<offsetN


;+------------------------------------------------------------------------------+
;| Lookup a binding of R1 in the BRAT.  Return the binding or NIL if there isn't any. |
;| Also return the absolute address of the binding in the BRAT so that it can be     |
;| modified.                                                                      |
;+------------------------------------------------------------------------------+
;|
;|Call: lookupBinding
;|
;|In:    R1       Key.
;|
;|Out:   R0       Data or NIL if none.
;|       R2       Absolute address of data in the BRAT (valid only when R0<>NIL).
;|
;|Criticality 5.
;|
;|Alters R0/R2.
;|

LookupBinding:  ROT     R1,-BRATLenLog*4,R2     ;Criticality 6.
                XOR     R1,R2,R2                ;Calculate the hash code for R1.
                ROT     R2,-BRATLenLog*2,R0     ;The hash code is the XOR of the four bytes of R1,
                XOR     R2,R0,R2                ;the same as the XLATE hash code.
                ROT     R2,-BRATLenLog,R0
                XOR     R2,R0,R2
                MOVE    BRATLength-1,R0
                AND     R2,R0,R2                ;R2 contains a hash code between 0 and BRATLength-1.
                DC      BRATStart-2
                ADD     R2,R0,R0
LB_Next:        ADD     R0,2,R0
                MOVE    [R0,A0],R0              ;Follow the linked list of BRAT entries starting with
                BNIL    R0,^LB_Done             ;the one in R0.  Leave if R0 is NIL.
                EQ      R1,[R0,A0],R2           ;Compare the key against R1.
                BF      R2,^LB_Next             ;Check the next entry if it doesn't match.
                ADD     R0,1,R2                 ;Otherwise return this entry's data.
                MOVE    [R2,A0],R0
                MOVE    FIP,IP
```

```
LB_Done:          MOVE     FIP,IP

fltLookupBinding = IP:abs|fault|unchecked|LookupBinding<<offsetN


;+------------------------------------------------------------------------------+
;| Delete a binding of R1 in the BRAT.  Halt if no such binding existed.        |
;| The purgeBinding entry point also purges the binding from the xlate table.   |
;+------------------------------------------------------------------------------+
;|
;|Call: deleteBinding
;|Call: purgeBinding
;|
;|In:    R1       Key.
;|
;|Criticality 5.
;|
;|Alters R0.
;|

PurgeBinding:     MOVE     NIL,R0                  ;Criticality 6.  Purge the object's binding from the XLATE table.
                  ENTER    R1,R0
DeleteBinding:    MOVE     R2,FOP0                 ;Criticality 6.  Save R2 and R3.
                  MOVE     R3,FOP1
                  ROT      R1,-BRATLenLog*4,R2
                  XOR      R1,R2,R2                ;Calculate the hash code for R1.
                  ROT      R2,-BRATLenLog*2,R0     ;The hash code is the XOR of the four bytes of R1,
                  XOR      R2,R0,R2                ;the same as the XLATE hash code.
                  ROT      R2,-BRATLenLog,R0
                  XOR      R2,R0,R2
                  MOVE     BRATLength-1,R0
                  AND      R2,R0,R2                ;R2 contains a hash code between 0 and BRATLength-1.
                  DC       BRATStart-2
                  ADD      R2,R0,R2
DB_Next:          ADD      R2,2,R0
                  MOVE     [R0,A0],R2              ;Follow the linked list of BRAT entries starting with
                  BNIL     R2,^DB_Halt             ;the one in R0.  Leave if R0 is NIL.
                  EQ       R1,[R2,A0],R3           ;Compare the key against R1.
                  BF       R3,^DB_Next             ;Check the next entry if it doesn't match.
                  ADD      R2,2,R2                 ;Otherwise delete this entry.
                  MOVE     [R2,A0],R3
                  MOVE     R3,[R0,A0]
                  MOVE     [BRATFree,A0],R3
                  MOVE     R3,[R2,A0]
                  SUB      R2,2,R2
                  MOVE     R2,[BRATFree,A0]
                  MOVE     FOP1,R3
                  MOVE     FOP0,R2
                  MOVE     FIP,IP
DB_Halt:          HALT     haltBRATDelete

fltPurgeBinding = IP:abs|fault|unchecked|PurgeBinding<<offsetN
fltDeleteBinding = IP:abs|fault|unchecked|DeleteBinding<<offsetN
```

```
                          ;##################################
                          ;##                              ##
                          ;##  Object and Context Manager  ##
                          ;##                              ##
                          ;##################################

;+--------------------------------------------------------------------------------+
;| Allocate and initialize a new object on the local heap and enter it in the XLATE and |
;| BRAT tables.  R0 contains the class of the object.                               |
;+--------------------------------------------------------------------------------+
;|
;|Call: newLocalObject
;|
;|In:   R0      Object's class.
;|
;|Out:  R0      Object's ID.
;|
;|Criticality 1.
;|
;|Alters R0-R3/AID2.
;|
;+--------------------------------------------------------------------------------+
;| Allocate and initialize a new object on the local heap and enter it in the XLATE and |
;| BRAT tables.  R0 contains the word to be stored as the first word of the object.  |
;| The length is extracted from R0, and the flags in the high bits of R0 should be set |
;| to benign values.  The object gets the next unused ID.                           |
;+--------------------------------------------------------------------------------+
;|
;|Call: allocNextObject
;|
;|In:   R0      First word of object.
;|
;|Out:  AID2    ^Object.
;|      R0      Object's ID.
;|
;|Criticality 3.
;|
;|Alters R0-R3/AID2.
;|
;+--------------------------------------------------------------------------------+
;| Allocate and initialize a new object on the local heap and enter it in the XLATE and |
;| BRAT tables.  R0 contains the word to be stored as the first word of the object.  |
;| The length is extracted from R0, and the flags in the high bits of R0 should be set |
;| to benign values.  R1 contains the ID to be used for the object.                 |
;+--------------------------------------------------------------------------------+
;|
;|Call: allocNewObject
;|
;|In:   R0      First word of object.
;       R1      Object's ID.
;
;|Out:  AID2    ^Object.
;       R0      Object's ID.
;|
;|Criticality 3.
;|
;|Alters R0-R3/AID2.
;|

NewLocalObject: MOVE     FIP,R2                  ;Criticality 6.
                MOVE     R2,F                    ;Criticality 1.
                XLATE    R0,objectXLATE,A2       ;Get the object's first word.
                MOVE     [oClassWord,A2],R0
                MOVE     [LastObjectID,A0],R1    ;Get the next object ID.
                ADD      R1,(1<<serialN)-1,R1
                ADD      R1,1,R1
                MOVE     R1,[LastObjectID,A0]    ;Advance the object ID counter
                BR       ^ANO_2

AllocNextObject: MOVE    [LastObjectID,A0],R1    ;Criticality 6.  Get the next object ID.
                ADD      R1,(1<<serialN)-1,R1
                ADD      R1,1,R1
                MOVE     R1,[LastObjectID,A0]    ;Advance the object ID counter
AllocNewObject: MOVE     FIP,R2                  ;Criticality 6.  Save FIP.
                MOVE     R2,F                    ;Criticality 5.
ANO_2:          MOVE     R2,[TempANO_FIP,A0]
                CALL     allocObject             ;Allocate the object.
                ENTER    R1,R0                   ;Put it into the xlate cache and the BRAT table.
                CALL     enterBinding
                MOVE     ID2,R0                  ;Load the object's ID into R0.
                MOVE     [TempANO_FIP,A0],IP

fltNewLocalObject = IP:abs|fault|unchecked|NewLocalObject<<offsetN
fltAllocNextObject = IP:abs|fault|unchecked|AllocNextObject<<offsetN
fltAllocNewObject = IP:abs|fault|unchecked|AllocNewObject<<offsetN


;+--------------------------------------------------------------------------------+
;  Allocate and initialize a new context.  If ID1 is non-NIL on entry, it points to a
;  context that should be deallocated; however, if A3 no longer points to the message,
;  before that context is deallocated, its locals in locations [illegible] through [illegible], inclusive,
;  are copied into the new context
;+--------------------------------------------------------------------------------+
;
;|Call: newContext
;|
;|In:   R0      First word of context, including desired length
;|      AID1    ^context or NIL if none already exists
;|
;|Out:  AID1    New context
;
;|Criticality 2.
;|
;|Alters R0/R1/AID1.
;
```

```
NewContext:     MOVE    FIP,R1                  ;Criticality 6.  Save R2, R3 and the FIP.
                MOVE    R1,[TempNC_FIP,A0]
                MOVE    R1,F                    ;Criticality 3.
                MOVE    R2,[TempNC_R2,A0]
                MOVE    ID2,R1                  ;Save ID2 in TempNC_ID2.
                MOVE    R1,[TempNC_ID2,A0]
                MOVE    R3,[TempNC_R3,A0]
                CALL    allocNextObject         ;Create the context object.
                MOVE    ID1,R0
                BNIL    R0,^NC_NoOldCxt
                MOVE    Q,R0
                BT      R0,^NC_HadMessage
                MOVE    [2,A1],R0               ;If A3 did not point to a message, copy the old context's
                MOVE    R0,[2,A2]               ;locals into the new context.
                MOVE    [3,A1],R0
                MOVE    R0,[3,A2]
                MOVE    [4,A1],R0
                MOVE    R0,[4,A2]
                MOVE    [5,A1],R0
                MOVE    R0,[5,A2]
                MOVE    [6,A1],R0
                MOVE    R0,[6,A2]
                MOVE    [7,A1],R0
                MOVE    R0,[7,A2]
                MOVE    [8,A1],R0
                MOVE    R0,[8,A2]
                MOVE    [9,A1],R0
                MOVE    R0,[9,A2]
                MOVE    [10,A1],R0
                MOVE    R0,[10,A2]
                MOVE    [11,A1],R0
                MOVE    R0,[11,A2]
                MOVE    [12,A1],R0
                MOVE    R0,[12,A2]
                MOVE    [13,A1],R0
                MOVE    R0,[13,A2]
                MOVE    [14,A1],R0
                MOVE    R0,[14,A2]
                MOVE    [15,A1],R0
                MOVE    R0,[15,A2]
NC_HadMessage:  CALL    disposeContext          ;Then dispose the old context.
NC_NoOldCxt:    MOVE    ID2,R1                  ;Point A1 and ID1 to the new context.
                XLATE   R1,objectXLATE,A1
                MOVE    [TempNC_ID2,A0],R2      ;Restore A2 and ID2.
                XLATE   R2,restoreXLATE,A2
                MOVE    [TempNC_R2,A0],R2       ;Restore R2 and R3.
                MOVE    [TempNC_R3,A0],R3
                MOVE    [TempNC_FIP,A0],IP

fltNewContext = IP:abs|fault|unchecked|NewContext<<offsetN


;+-----------------------------------------------------------------------------+
;| Deallocate a context, which may be either a fast context or a heap context. |
;+-----------------------------------------------------------------------------+
;|
;|Call: disposeContext
;|
;|In:   AID1     Context.
;|
;|Criticality 3.
;|
;|Alters R0-R2/AID1.
;|
;+-----------------------------------------------------------------------------+
;| Dispose an object.  If the object is locked, it is deleted as soon as the unlock |
;| message comes in.                                                           |
;+-----------------------------------------------------------------------------+
;|
;|Call: disposeObject
;|
;|In:   R0       Object.
;|
;|Criticality 3.
;|
;|Alters R0-R2.
;|

DisposeFastContext: MOVE [FastContextQueue,A0],R0 ;Criticality 6.
                MOVE    R0,[contextNext,A1]     ;Put the context back on the context queue.
                MOVE    ID1,R0
                MOVE    R0,[FastContextQueue,A0]
                MOVE    FIP,IP

DisposeContext: MOVE    [contextHeader,A1],R0   ;Criticality 6.  Check whether this was a fast context.
                ROT     R0,-hdrFastN,R0
                BT      R0,^DisposeFastContext  ;Yes.
                MOVE    ID1,R0                  ;No.  Deallocate a normal object.
DisposeObject:  MOVE    FIP,R2                  ;Criticality 6.
                MOVE    R2,F                    ;Criticality 3.
                XLATE   R0,localXLATE,R1        ;Get the object location into R1.
                BNIL    R1,^DO_Done             ;Exit if the object was a constant.
                MOVE    R2,[TempDO_FIP,A0]      ;Save the FIP.
                CHECK   R1,INT,R2
                BT      R2,^DO_Remote           ;Go send a Dispose message if the object is remote.
                MOVE    TRUE,R2                 ;Enter unchecked mode.
                MOVE    R2,U
                MOVE    ID2,R2
                MOVE    R0,ID2
                MOVE    R1,A2                   ;If the object is local, point AID2 to it.
                MOVE    [objectHeader,A2],R1    ;Can't delete a locked object.
                ROT     R1,-hdrLockedN,R1
                BT      R1,^DO_Locked
                AND     R0,[NodeMask,A0],R1
                MOVE    NNR,R0                  ;Check whether this is the object's home.
                EQUAL   R0,R1,R0
                BT      R0,^DO_Home
```

```
                MOVE    ID2,R1                  ;If not, send a message to the object's home to delete
                DC      MSG:msgDisposeBRAT+2    ;its BRAT entry.
                SEND20  R1,R0
                SENDE0  R1
DO_Home:        CALL    deallocateObject        ;Deallocate it.
                XLATE   R2,restoreXLATE,A2      ;Restore AID2.
                MOVE    [TempDO_FIP,A0],IP
DO_Done:        MOVE    R2,IP
DO_Remote:      MOVE    R0,R2                   ;Send a Dispose message to the object's node.
                SEND0   R1
                DC      MSG:msgDispose+2
                SEND2E0 R0,R2
                MOVE    [TempDO_FIP,A0],IP

DO_Locked:      ROT     R1,hdrLockedN-hdrDeletedN,R0
                OR      R0,1,R0                 ;If the object is locked, mark it as deleted but do not
                ROT     R0,hdrDeletedN,R0       ;delete it yet.
                MOVE    R0,[objectHeader,A2]
                MOVE    [TempDO_FIP,A0],IP

fltDisposeContext = IP:abs|fault|unchecked|DisposeContext<<offsetN
fltDisposeObject = IP:abs|fault|DisposeObject<<offsetN


;+--------------------------------------------------------------------------------------+
;| Execute a Dispose message.                                                           |
;+--------------------------------------------------------------------------------------+

Dispose:        MOVE    [disposeID,A3],R0       ;Criticality 2.
                CALL    disposeObject           ;Dispose the object.
                SUSPEND

msgDispose = Dispose<<offsetN


;+--------------------------------------------------------------------------------------+
;| Execute a DisposeBRAT message.  If the object was present on its home node, it is    |
;| disposed; otherwise, only the object's home BRAT entry is deleted.                   |
;+--------------------------------------------------------------------------------------+

DisposeBRAT:    MOVE    [disposeBRATID,A3],R1   ;Criticality 2.
                XLATE   R1,localXLATE,R2        ;Check if the object is here too.
                CHECK   R2,ADDR,R3
                BT      R3,^DBRAT_Here          ;If so, dispose it.
                CALL    purgeBinding            ;Purge the object's binding from the XLATE table and
                SUSPEND                         ;from the BRAT.
DBRAT_Here:     MOVE    R1,R0
                CALL    disposeObject           ;Dispose the object.
                SUSPEND

msgDisposeBRAT = DisposeBRAT<<offsetN


;+--------------------------------------------------------------------------------------+
;| Deallocate an object residing on this node.  The object must not be locked.          |
;+--------------------------------------------------------------------------------------+
;|
;|Call: deallocateObject
;|
;|In:   AID2    Object.
;|
;|Criticality 4.
;|
;|Alters R0/R1.
;|

DeallocateObject: MOVE  FIP,R1                  ;Criticality 6.  Save the FIP.
                MOVE    R1,[TempDealloc_FIP,A0]
                DC      OBJ:hdrDeleted
                MOVE    R1,F                    ;Criticality 4.
                OR      R0,[objectHeader,A2],R0 ;Set the deleted flag in the object header.
                MOVE    R0,[objectHeader,A2]
                MOVE    [objectID,A2],R1        ;Delete the object's binding from the BRAT and the xlate table.
                CALL    purgeBinding
                MOVE    NIL,R0
                MOVE    R0,ID2                  ;Clear ID2.
                MOVE    [TempDealloc_FIP,A0],IP

fltDeallocateObject = IP:abs|fault|unchecked|DeallocateObject<<offsetN
```

```
                    ;##############################
                    ;##                          ##
                    ;##  Global Object Manager   ##
                    ;##                          ##
                    ;##############################


;+-------------------------------------------------------------------------------+
;| Allocate and initialize a new object on the heap of a random node and enter it in  |
;| that node's XLATE and BRAT tables.    R0 contains the class of the object.         |
;+-------------------------------------------------------------------------------+
;|
;|Call: newObject
;|
;|In:   R0      Object's class.
;|
;|Out:  R0      Object's ID.
;|
;|Criticality 1.
;|
;|Alters R0/R1.
;|

NewObject:      MOVE    FIP,R1                  ;Criticality 6.
                MOVE    R1,F                    ;Criticality 3.
                MOVE    R1,[contextIP,A1]       ;Save the state in the context.
                MOVE    R2,[contextR2,A1]
                MOVE    R3,[contextR3,A1]
                MOVE    R0,R3                   ;Save the class in R3.
                DC      RandomSeedIncrement
                MOVE    [RandomSeed,A0],R2      ;Generate a random node number.
                ADD     R2,R0,R2                ;Advance the random node counter and return its new value.
                MOVE    R2,[RandomSeed,A0]
                AND     R2,[NodeMask,A0],R2
                DC      MSG:msgNewObject+4      ;Send a NewObject message to that node.
                SEND20  R2,R0
                MOVE    ID1,R2
                SEND20  R3,R2
                SENDE0  contextR0
                DC      CFUT:contextR0
                MOVE    R0,[contextNext,A1]     ;Tell the context to wait for the quasi-cfuture in R0.
                MOVE    R0,[contextR0,A1]       ;Store a cfuture in R0.
                DC      SaveStateID023-(*+2)    ;Go save the context and suspend.
                BR      R0

fltNewObject = IP:abs|fault|unchecked|NewObject<<offsetN


;+-------------------------------------------------------------------------------+
;| Execute a NewObject message.  Return the object's ID to the caller.               |
;+-------------------------------------------------------------------------------+

NewObjectM:     MOVE    [newObjClass,A3],R0     ;Criticality 0.
                CALL    newLocalObject          ;Allocate the object locally.
                MOVE    R0,R2
                DC      MSG:msgReply+4          ;Reply with the object's ID.
                MOVE    [newObjReplyID,A3],R1
                SEND20  R1,R0
                SEND0   R1
                SEND2E0 [newObjReplySlot,A3],R2
                SUSPEND

msgNewObject = NewObjectM<<offsetN


;+-------------------------------------------------------------------------------+
;| Return the class of an object.  TypeOf returns the class as an integer, while     |
;| classOf wraps it as a class.  The argument of TypeOf must not be a future.        |
;+-------------------------------------------------------------------------------+
;|
;|Call: classOf
;|Call: typeOf
;|
;|In:   R0      Object.
;|
;|Out:  R0      The object's class.
;|
;|Criticality 1.
;|
;|Alters R0/R1/AID2.
;|

ClassOf:        MOVE    FIP,R1                  ;Criticality 6.  Save the FIP.
                MOVE    R1,F                    ;Criticality 1.
                BNNIL   R0,^TOf_2               ;Force the argument.
TOf_2:          MOVE    R1,[TempTOf_FIP,A0]     ;Save the FIP in memory.
                CALL    typeOf                  ;Get the integer type and write its tag and subtag.
                MOVE    subCLASS,R1
                ROT     R1,subtagN,R1
                OR      R0,R1,R0
                WTAG    R0,TAG0,R0
                MOVE    [TempTOf_FIP,A0],IP

fltClassOf = IP:abs|fault|ClassOf<<offsetN


TypeOf:         RTAG    R0,R1                   ;Criticality 6.  Dispatch on the tag of the object.
                BR      R1
:               ROT     R0,-subtagN,R1          ;TAG0
                BR      ^TOf_TAG0
:               BR      ^TOf_Integer            ;INT
:               BT      R0,^TOf_True            ;BOOL
                BR      ^TOf_False
:               HALT    haltTypeOf              ;ADDR
:               HALT    haltTypeOf              ;IP
:               HALT    haltTypeOf              ;MSG / OBJ
:               HALT    haltTypeOf              ;CFUT
:               HALT    haltTypeOf              ;FUT
```

```
:                   MOVE    FIP,R1                  ;ID.  Save the FIP.
                    BR      ^TOf_Object
:                   MOVE    FIP,R1                  ;DID.  Save the FIP.
                    BR      ^TOf_Object
:                   HALT    haltTypeOf              ;TAGA
:                   BR      ^TOf_Float              ;FLOAT
:                   HALT    haltTypeOf              ;INST0
:                   HALT    haltTypeOf              ;INST1
:                   HALT    haltTypeOf              ;INST2
:                   HALT    haltTypeOf              ;INST3
TOf_Integer:        DC      classInteger            ;Return the integer class.
                    MOVE    FIP,IP
TOf_True:           DC      classTrue               ;Return the true class.
                    MOVE    FIP,IP
TOf_False:          DC      classFalse              ;Return the false class.
                    MOVE    FIP,IP
TOf_Float:          DC      classFloat              ;Return the float class.
                    MOVE    FIP,IP

TOf_Object:         MOVE    R1,F                    ;Criticality 1.
                    XLATE   R0,objectXLATE,A2
                    MOVE    [objectHeader,A2],R0    ;Extract the class from the object header.
                    WTAG    R0,INT,R0
                    ROT     R0,-hdrClassN,R0
                    AND     R0,hdrClassM,R0         ;R0 now contains the class.
                    MOVE    R1,IP

TOf_TAG0:           AND     R1,subtagM,R1           ;Dispatch on the subtag.
                    BR      R1
:                   BNNIL   R0,^TOf_Symbol          ;subSYM
                    BR      ^TOf_NIL
:                   MOVE    FIP,R1                  ;subCLASS.  Save the FIP.
                    BR      ^TOf_Object
:                   BR      ^TOf_Selector           ;subSEL
:                   BR      ^TOf_Character          ;subCHAR
:                   HALT    haltTypeOf
:                   HALT    haltTypeOf
:                   HALT    haltTypeOf
:                   HALT    haltTypeOf
:                   HALT    haltTypeOf
:                   HALT    haltTypeOf
:                   HALT    haltTypeOf
:                   HALT    haltTypeOf
:                   HALT    haltTypeOf
:                   HALT    haltTypeOf
:                   HALT    haltTypeOf
:                   HALT    haltTypeOf
TOf_Symbol:         DC      classSymbol             ;Return the symbol class.
                    MOVE    FIP,IP
TOf_NIL:            DC      classNull               ;Return the null class.
                    MOVE    FIP,IP
TOf_Selector:       DC      classSelector           ;Return the selector class.
                    MOVE    FIP,IP
TOf_Character:      DC      classCharacter          ;Return the character class.
                    MOVE    FIP,IP

fltTypeOf = IP:abs|fault|unchecked|TypeOf<<offsetN


;+----------------------------------------------------------------------------+
;| Return the node on which the object might reside.  If the object is a constant,  :
;| return a random node number.  If the object is a DID, return a random constituent.  :
;+----------------------------------------------------------------------------+
;|
;|Call: objectNode
;|
;|In:   R0      Object.
;|
;|Out:  R1      Number of node likely to contain object.  The number may not necessarily be
;|              tagged INTeger, and it may contain junk data in the high 16 bits.
;|
;|Criticality 5.
;|
;|Alters R0/R1.
;|

ObjectNode:         RTAG    R0,R1                   ;Criticality 6.
                    BR      R1                      ;Dispatch on the tag of the object.
:                   MOVE    [RandomSeed,A0],R1      ;TAG0
                    BR      ^ON_Random
:                   MOVE    [RandomSeed,A0],R1      ;INT
                    BR      ^ON_Random
:                   MOVE    [RandomSeed,A0],R1      ;BOOL
                    BR      ^ON_Random
:                   HALT    haltInternalType        ;ADDR
:                   HALT    haltInternalType        ;IP
:                   HALT    haltInternalType        ;MSG / OBJ
:                   HALT    haltInternalType        ;CFUT
:                   MOVE    R0,R1                   ;FUT.  Return the FUT; the node number is in the low
                    MOVE    FIP,IP                  ;16 bits
:                   MOVE    R0,R1                   ;ID.  Return the ID; the node number is in the low
                    MOVE    FIP,IP                  ;16 bits.
:                   MOVE    R0,R1                   ;DID
                    BR      ^RandomConst
:                   MOVE    [RandomSeed,A0],R1      ;TAGA
                    BR      ^ON_Random
:                   MOVE    [RandomSeed,A0],R1      ;FLOAT
                    BR      ^ON_Random
:                   HALT    haltInternalType        ;INST0
:                   HALT    haltInternalType        ;INST1
:                   HALT    haltInternalType        ;INST2
:                   HALT    haltInternalType        ;INST3
ON_Random:          DC      RandomSeedIncrement
                    ADD     R1,R0,R1                ;Advance the random node counter and return its new value.
                    MOVE    R1,[RandomSeed,A0]
                    AND     R1,[NodeMask,A0],R1
                    MOVE    FIP,IP
```

```
RandomConst:    MOVE    R2,FOP0
                ROT     R1,-(logStrideN+logStrideL),R2 ;R1 has s, the distobj initial node number, in bits 0..10
                ASH     R2,-16,R2               ;and e, 2's complement logStride, in bits 11..15.
                ASH     R2,logStrideL-16,R2     ;R2:=e.
                MOVE    -1,R0
                ASH     R0,R2,R2                ;R2:=e zeros in LSBs with the MSBs being ones.
                MOVE    [RandomSeed,A0],R0
                ADD     R0,7,R0                 ;Advance the random node counter and return its new value.
                MOVE    R0,[RandomSeed,A0]
                AND     R0,R2,R2
                BR      ^GetConst

fltObjectNode = IP:abs|fault|unchecked|ObjectNode<<offsetN


;+----------------------------------------------------------------------------+
;| Return the ID of the preferred constituent of a distributed object with the given    |
;| DID.                                                                       |
;+----------------------------------------------------------------------------+
;|
;|Call: preferredConstituent
;|
;|In:   R1      DID.
;|
;|Out:  R1      ID.
;|
;|Criticality 5.
;|
;|Alters R0/R1.
;|

PreferredConst: MOVE    R2,FOP0                 ;Criticality 6.
                ROT     R1,-(logStrideN+logStrideL),R2 ;R1 has s, the distobj initial node number, in bits 0..10
                ASH     R2,-16,R2               ;and e, 2's complement logStride, in bits 11..15.
                ASH     R2,logStrideL-16,R2     ;R2:=e.
                LE      R2,0,R0                 ;Jump to a faster routine if the distributed object is dense.
                BT      R0,^PrefCnst_Dense
                MOVE    -1,R0
                ASH     R0,R2,R2                ;R2:=e zeros in LSBs with the MSBs being ones.
                MOVE    [SerialNode,A0],R0
                AND     R0,R2,R2                ;R2:=masked serial node number.
GetConst:       DC      initialNodeM
                AND     R0,R1,R0
                OR      R0,R2,R2                ;R2:=serial constituent node number.
                DC      ID:~homeNodeM
                AND     R0,R1,R1                ;R1:=ID:serial number.
                AND     R2,xM,R0
                OR      R1,R0,R1                ;Store the x node number in R1.
                ROT     R2,-xL,R0
                AND     R0,yM,R0
                ROT     R0,yN,R0
                OR      R1,R0,R1                ;Store the y node number in R1.
                ROT     R2,-(xL+yL),R2
                AND     R2,zM,R2
                ROT     R2,zN,R2
                OR      R1,R2,R1                ;Store the z node number in R1.
                MOVE    FOP0,R2
                MOVE    FIP,IP

PrefCnst_Dense: DC      ID:~homeNodeM
                AND     R0,R1,R1                ;R1:=ID:serial number.
                MOVE    NNR,R0                  ;This is a dense distributed object; just use the current node.
                OR      R1,R0,R1
                MOVE    FOP0,R2
                MOVE    FIP,IP

fltPreferredConstituent = IP:abs|fault|unchecked|PreferredConst<<offsetN


;+----------------------------------------------------------------------------+
;| Return the ID of the nth constituent of a distributed object with the given DID.     |
;+----------------------------------------------------------------------------+
;|
;|Call: co
;|
;|In:   R0      n.
;|      R1      DID.
;|
;|Out:  R1      ID.
;|
;|Criticality 1.
;|
;|Alters R0-R2.
;|

Co:             MOVE    R2,FOP0                 ;Criticality 6.  Save R2 and R3.
                MOVE    R3,FOP1
                CHECK   R0,INT,R2
                BF      R2,^Co_BadType
                CHECK   R1,DID,R2
                BF      R2,^Co_BadType
                LT      R0,0,R2
                BT      R2,^Co_BadRange
                ROT     R1,-(logStrideN+logStrideL),R2 ;R1 has s, the distobj initial node number, in bits 0..10
                ASH     R2,-16,R2               ;and e, 2's complement logStride, in bits 11..15.
                ASH     R2,logStrideL-16,R2     ;R2:=e.
                LT      R2,0,R3
                BF      R3,^Co_Sparse           ;There is at most one constituent per node.
                NEG     R2,R2                   ;There are multiple constituents per node.
                MOVE    -1,R3
                ASH     R3,R2,R3
                NOT     R3,R3                   ;R3 contains the number of constituents per node minus one.
                AND     R0,R3,R3                ;Modulo n by the number of constituents per node to get the
                ROT     R3,serialN,R3           ;displacement to be added to the DID's serial number.
                ADD     R1,R3,R1
                NEG     R2,R2
                ASH     R0,R2,R0                ;Divide n by the number of constituents per node.
```

```
                  MOVE     0,R2                     ;Now assume that there is one constituent per node.
Co_Sparse:        FFB      R0,R3                    ;R3:=30-lg(n).
                  SUB      R3,R2,R3                 ;R3:=30-lg(node#).
                  ASH      R0,R2,R2                 ;R2:=node#.
                  MOVE     30-LogNNodes,R0          ;R0:=30-lg(NNodes).
                  GT       R3,R0,R0
                  BF       R0,^Co_BadRange
                  MOVE     FOP1,R3
                  BR       ^GetConst

Co_BadRange:      HALT     haltRange
Co_BadType:       CHECK    R0,FUT,R2                ;If either operand was a future, crash with the future fault;
                  BT       R2,^Co_Future            ;otherwise, crash with the type fault.
                  CHECK    R1,FUT,R2
                  BT       R2,^Co_Future
                  HALT     haltType
Co_Future:        HALT     haltFuture

fltCo = IP:abs|fault|unchecked|Co<<offsetN


;+--------------------------------------------------------------------------------------+
;| Execute a RequestObject message.  The ID passed in the message must be a word tagged |
;| ID, a class or a selector: it cannot be a future, constant, or distributed object.   |
;+--------------------------------------------------------------------------------------+

RequestObject:    MOVE     [reqObjID,A3],R3         ;Criticality 4.
                  XLATE    R3,localXLATE,R1         ;Is the object here?
                  CHECK    R1,ADDR,R2
                  BT       R2,^RO_Local             ;Yes.
RO_Resend:        DC       MSG:msgRequestObject+3   ;No.  Send a message requesting the object to the object's
                  SEND20   R1,R0                    ;likely location.
                  SEND0    R3
                  SENDE0   [reqObjReplyNode,A3]
                  SUSPEND

RO_Locked:        MOVE     NNR,R1                   ;Criticality 4.  Resend the message back to this node.
                  BR       ^RO_Resend
RO_Local:         MOVE     R3,ID2                   ;Criticality 5.  Point AID2 to the object.
                  MOVE     R1,A2
                  MOVE     [objectHeader,A2],R2
                  ROT      R2,-hdrLockedN,R0        ;Resend the message back to this node if the object is locked.
                  BT       R0,^RO_Locked
                  AND      R2,hdrLengthM,R3
                  DC       MSG:msgMigrateObject+1   ;The length of the message is one plus the length of the
                  ADD      R0,R3,R0                 ;object.
                  SUB      R3,1,R3
                  MOVE     [R3,A2],R3               ;Save the last word of the object in R3.
                  SUB      R1,1,R1                  ;Shorten the object's limit by one word.
                  MOVE     R1,A2
                  SEND20   [reqObjReplyNode,A3],R0  ;Send the message header.
                  SEND0    R2                       ;Send the words of the object.
                  MOVE     0,R0
                  CALL     blockSend
                  SENDE0   R3                       ;Send the last word of the object.
                  ROT      R2,-hdrCopyableN,R0      ;Leave the object here if it is copyable.
                  BT       R0,^RO_Copyable
                  MOVE     ID2,R1                   ;If the object is not copyable, purge it from the xlate table
                  AND      R1,[NodeMask,A0],R2      ;and from the BRAT, unless this is its home node, in which
                  MOVE     NNR,R3                   ;case purge it from the xlate table and replace its BRAT entry
                  EQUAL    R2,R3,R0                 ;to suggest that it is present on this node: messages requesting
                  BT       R0,^RO_Home              ;the object will keep cycling at this node until the object's
                  CALL     deallocateObject         ;new location is known.
RO_Copyable:      SUSPEND
RO_Home:          DC       OBJ:hdrDeleted
                  OR       R0,[objectHeader,A2],R0  ;Set the deleted flag in the object header.
                  MOVE     R0,[objectHeader,A2]
                  MOVE     NIL,R0
                  ENTER    R1,R0
                  CALL     lookupBinding
                  BNIL     R0,^RO_NoBinding
                  MOVE     R3,[R2,A0]               ;Pretend that the object is located at this node.
                  SUSPEND
RO_NoBinding:     HALT     haltBRATMissing

msgRequestObject = unchecked|RequestObject<<offsetN


;+--------------------------------------------------------------------------------------+
;| Execute an AcceptObject message.  Make this node the object's home.  The object's ID |
;| must reflect this node as the object's home.                                         |
;+--------------------------------------------------------------------------------------+

AcceptObject:     MOVE     [2+objectHeader,A3],R0   ;Criticality 3.  Read the object's header and ID.
                  MOVE     [2+objectID,A3],R1
                  CALL     allocNewObject           ;Allocate space for the object.
                  MOVE     A2,R1
                  DC       (-(1<<baseN)+1)*2        ;Decrease A2's base by two words and increase its limit likewise
                  ADD      R1,R0,R0                 ;because the object starts in the message two words late.
                  MOVE     R0,A2
                  MOVE     4,R0                     ;Copy the object into the heap starting from the fifth word
                  CALL     blockMove                ;of the message (third word of the object).
                  MOVE     [1,A3],R3
                  BNIL     R3,^AO_Done              ;Acknowledge the sender if an acknowledgement was requested.
                  DC       MSG:msgAcknowledgeObject+2
                  SEND20   R3,R0
                  SENDE0   [2+objectID,A3]
AO_Done:          SUSPEND

msgAcceptObject = unchecked|AcceptObject<<offsetN
```

```
;+-------------------------------------------------------------------------------+
;| Execute a MigrateObject message.  If the object is copyable, store a copy of it on  |
;| this node.  If the object is not copyable, store it on this node, lock it, and      |
;| inform the home node about the object's presence here.                               |
;+-------------------------------------------------------------------------------+

MigrateObject:  MOVE    [1+objectHeader,A3],R0  ;Criticality 3.  Read the object's header and ID.
                MOVE    [1+objectID,A3],R1
                ROT     R0,-hdrCopyableN,R0
                AND     R0,$FFFFFFF1,R0         ;Clear the purgeable, locked, and marked flags.
                BT      R0,^MO_Copyable         ;If the object is copyable, make this copy purgeable.
                AND     R1,[NodeMask,A0],R2     ;This object is noncopyable.
                MOVE    NNR,R3
                EQUAL   R2,R3,R2                ;Check whether this node is the object's home node.
                BT      R2,^MO_Noncopyable      ;If so, do nothing.
                MOVE    R0,R2                   ;Save R0.
                DC      MSG:msgUpdateHome+3     ;Otherwise, tell the home node about this object's location
                SEND20  R1,R0                   ;and lock the object until the home node replies.
                SEND2E0 R1,R3
                OR      R2,1<<(hdrLockedN-hdrCopyableN),R0
                BR      ^MO_Noncopyable         ;Lock this object.
MO_Copyable:    OR      R0,1<<(hdrPurgeableN-hdrCopyableN),R0
MO_Noncopyable: ROT     R0,hdrCopyableN,R0      ;Allocate storage for the object and put it into the xlate table.
                CALL    allocObject
                ENTER   R1,R0                   ;Criticality 5.
                MOVE    R0,R3
                CALL    lookupBinding           ;Check whether a binding for the object existed in the BRAT.
                BNIL    R0,^MO_Unexpected
                MOVE    R3,[R2,A0]              ;If one did exist, save its data in R2 and rebind the BRAT
                MOVE    R0,R2                   ;entry to point to the object.
                BR      ^MO_Expected
MO_Unexpected:  MOVE    R3,R0
                CALL    enterBinding            ;This obeys criticality 5 because allocObject allocates three
                MOVE    NIL,R2                  ;extra heap words.
MO_Expected:    MOVE    A2,R0                   ;Decrease A2's base by one word and increase its limit
                SUB     R0,(1<<baseN)-1,R0      ;because the object starts in the message one word late.
                MOVE    R0,A2
                MOVE    3,R0                    ;Copy the object into the heap starting from the fourth word
                CALL    blockMove               ;of the message (third word of the object).
                CHECK   R2,ID,R0
                BF      R0,^MO_Suspend          ;Leave if there are no contexts to restart.
                MOVE    [FastContextQueue,A0],R0
                MOVE    R0,[contextNext,A1]     ;Dispose the current context.
                MOVE    ID1,R0
                MOVE    R0,[FastContextQueue,A0]
MO_NextRestart: XLATE   R2,objectXLATE,A1       ;Restart contexts.
                MOVE    [contextNext,A1],R1
                CHECK   R1,ID,R3
                BF      R3,^Reply_Restart       ;Restart this context if there is only one to be restarted.
                DC      MSG:msgRestartContext+2 ;Otherwise send a message back to this node to restart this
                SEND20  R2,R0                   ;context and go restart the next one now.
                SENDE0  R2
                MOVE    R1,R2
                BR      ^MO_NextRestart
MO_Suspend:     SUSPEND

msgMigrateObject = unchecked|MigrateObject<<offsetN


;+-------------------------------------------------------------------------------+
;| Execute a RestartContext message.                                             |
;+-------------------------------------------------------------------------------+
;| Execute a Reply message.                                                      |
;+-------------------------------------------------------------------------------+

RestartContext: MOVE    [FastContextQueue,A0],R0
                MOVE    R0,[contextNext,A1]     ;Put the fast context back on the context queue.
                MOVE    ID1,R0
                MOVE    R0,[FastContextQueue,A0]
                MOVE    [replyID,A3],R0         ;Criticality 3.
                XLATE   R0,objectXLATE,A1       ;Xlate the reply context into A1.
                BR      ^Reply_Restart
Reply_2:        MOVE    R1,[R0,A2]              ;Store the value replied.
                MOVE    [FastContextQueue,A0],R0
                MOVE    R0,[contextNext,A1]     ;Put the fast context back on the context queue.
                MOVE    ID1,R0
                MOVE    R0,[FastContextQueue,A0]
                XLATE   R3,objectXLATE,A1       ;Xlate the reply context into A1.
Reply_Restart:  MOVE    FALSE,R0                ;Turn off A3 queue wraparound.
                MOVE    R0,Q
                MOVE    [contextID0,A1],R0      ;Restore the address and ID registers.
                XLATE   R0,restoreXLATE,A0
                MOVE    [contextID2,A1],R0
                XLATE   R0,restoreXLATE,A2
                MOVE    [contextID3,A1],R0
                XLATE   R0,restoreXLATE,A3
                MOVE    [contextR3,A1],R3       ;Restore the data registers.
                MOVE    [contextR2,A1],R2
                MOVE    [contextR1,A1],R1
                MOVE    [contextR0,A1],R0
                MOVE    [contextIP,A1],IP       ;Resume computation of the message.

Reply:          MOVE    [replyID,A3],R3         ;Criticality 3.
                XLATE   R3,objectXLATE,A2       ;Xlate the reply context into A2.
                MOVE    [replySlot,A3],R0
                WTAG    R0,CFUT,R0
                MOVE    [contextNext,A2],R1     ;Check whether the process was waiting for this slot.
                EQ      R1,R0,R2
                MOVE    [replyValue,A3],R1
                BT      R2,^Reply_2             ;Suspend if not.
                EQ      R0,[R0,A2],R2           ;Check the previous value from the slot and make
                BF      R2,^Reply_Halt          ;sure it was the proper cfuture.
                MOVE    R1,[R0,A2]              ;Store the new value there.
                SUSPEND
Reply_Halt:     HALT    haltReply

msgRestartContext = unchecked|RestartContext<<offsetN
msgReply = unchecked|Reply<<offsetN
```

```
;+---------------------------------------------------------------------------------------+
;| Execute an UpdateHome message.  Update the BRAT to contain the object's new home      |
;| location and send an Unlock message to the object to allow it to move again.          |
;+---------------------------------------------------------------------------------------+

UpdateHome:         MOVE    [updtHomeID,A3],R1          ;Criticality 2.
                    SEND0   [updtHomeNode,A3]           ;Send an Unlock message back to the object.
                    DC      MSG:msgUnlock+2
                    SEND2E0 R0,R1
                    CALL    lookupBinding               ;Look the object up in the BRAT.
                    BNIL    R0,^UH_Halt                 ;The BRAT entry should be present.
                    CHECK   R0,INT,R3
                    BF      R3,^UH_Waiting
                    MOVE    [updtHomeNode,A3],R0        ;Change the BRAT entry to reflect the
                    MOVE    R0,[R2,A0]                  ;object's new location.
                    SUSPEND
UH_Waiting:         CHECK   R0,ID,R3
                    BF      R3,^UH_Halt
                    XLATE   R0,objectXLATE,A2
                    MOVE    [contextNext,A2],R0
                    BNIL    R0,^UH_Halt                 ;The BRAT entry should be present.
                    CHECK   R0,INT,R3
                    BF      R3,^UH_Waiting
                    MOVE    [updtHomeNode,A3],R0        ;Change the BRAT entry to reflect the
                    MOVE    R0,[contextNext,A2]         ;object's new location.
                    SUSPEND
UH_Halt:            HALT    haltBRATMissing

msgUpdateHome = UpdateHome<<offsetN


;+---------------------------------------------------------------------------------------+
;| Execute an Unlock message.  Unlock the object to allow it to move again.  If the      |
;| object was marked deleted, dispose it now.                                            |
;+---------------------------------------------------------------------------------------+

Unlock:             MOVE    [unlockID,A3],R1            ;Criticality 2.
                    XLATE   R1,objectXLATE,A2           ;Find the object and clear its locked flag.
                    DC      ~hdrLocked
                    MOVE    [objectHeader,A2],R2
                    AND     R2,R0,R2
                    MOVE    R2,[objectHeader,A2]
                    ROT     R2,-hdrDeletedN,R2          ;If the object was marked deleted, dispose it now.
                    BF      R2,^Unlk_Done
                    MOVE    R1,R0
                    CALL    disposeObject
Unlk_Done:          SUSPEND

msgUnlock = unchecked|Unlock<<offsetN
```

```
                        ;########################
                        ;##                    ##
                        ;##   Method Manager   ##
                        ;##                    ##
                        ;########################


;+---------------------------------------------------------------------------------+
;| Return the ID of a method associated with the given class and selector.  The second |
;| entry point, lookupMethodU, can be used when the class has already been type-checked |
;| and coerced to be an integer.                                                   |
;+---------------------------------------------------------------------------------+
;|
;|Call: lookupMethod
;|Call: lookupMethodU
;|
;|In:   R0      Class (Tagged TAG0:subCLASS if lookupMethod is used, INT if lookupMethodU is used).
;|      R1      Selector (Tagged TAG0:subSEL).
;|
;|Out:  R2      ID of method or NIL if none.
;|
;|Criticality 1.
;|
;|Alters R0-R3/AID2.
;|

LookupMethod:   CHECK   R0,TAG0,R2              ;Criticality 6.
                BF      R2,^LM_Halt             ;Make sure that R0 is tagged as a class.
                WTAG    R0,INT,R0
                ROT     R0,-subtagN,R3
                AND     R0,csClassM,R0          ;Coerce it to an integer.
                EQUAL   R3,subCLASS,R3
                BF      R3,^LM_Halt
LookupMethodU:  ROT     R0,csClassN,R2          ;Criticality 6.  Shift the class to the high 16 bits.
                WTAG    R1,INT,R3
                AND     R3,csSelectorM,R3
                OR      R2,R3,R3                ;Make a class/selector pair in R3.
                WTAG    R3,CS,R3
                PROBE   R3,R2                   ;Get the cached ID of the method into R0, if there is one.
                BNIL    R2,^LM_SendMsg
                MOVE    FIP,IP

LM_Halt:        HALT    haltClassType

LM_SendMsg:     MOVE    FIP,R2                  ;Criticality 6.
                MOVE    R2,F                    ;Criticality 3.
                MOVE    R2,[contextIP,A1]       ;Save FIP in the context.
                MOVE    R3,[contextR0,A1]       ;Save the class/selector pair in R0 of the saved context.
                MOVE    R0,R3
                AND     R1,[NodeMask,A0],R2
                DC      TAG0:subCLASS<<subtagN
                OR      R0,R3,R3                ;Generate the class ID in R3.
                DC      MSG:msgApplyFunction+5
                SEND20  R2,R0                   ;Send a LookupMethod message asking to lookup the method and
                MOVE    ID1,R2                  ;send a reply back to the context.
                DC      LLookupMethod
                SEND20  R0,R1
                SEND2E0 R3,R2
                MOVE    NIL,R1
                MOVE    R1,[contextNext,A1]     ;The LookupMethod handler will return the method via a
                DC      SaveStateID023-(*+2)    ;MethodReply, which will reply into the R2 slot of the context.
                BR      R0                      ;There is no need to save the data registers in the context.

fltLookupMethod = IP:abs|fault|unchecked|LookupMethod<<offsetN
fltLookupMethodU = IP:abs|fault|unchecked|LookupMethodU<<offsetN


;+---------------------------------------------------------------------------------+
;| Execute a MethodReply message.                                                  |
;+---------------------------------------------------------------------------------+

MethodReply:    MOVE    [methodReplyID,A3],R0   ;Criticality 3.
                XLATE   R0,objectXLATE,A1       ;Xlate the reply context into A1.
                MOVE    [methodReplyValue,A3],R2 ;Get the method ID.
                MOVE    [contextR0,A1],R0       ;Enter it into the cache.
                ENTER   R0,R2
                MOVE    FALSE,R0                ;Turn off A3 queue wraparound.
                MOVE    R0,Q
                MOVE    [contextID0,A1],R0      ;Restore the address and ID registers.
                XLATE   R0,restoreXLATE,A0
                MOVE    [contextID2,A1],R0
                XLATE   R0,restoreXLATE,A2
                MOVE    [contextID3,A1],R0
                XLATE   R0,restoreXLATE,A3
                MOVE    [contextIP,A1],IP       ;Resume computation.

msgMethodReply = MethodReply<<offsetN
```

```
                                ;##################
                                ;##              ##
                                ;##  Utilities   ##
                                ;##              ##
                                ;##################


;+---------------------------------------------------------------------------------+
;| Divide R1 by R0.  Return the quotient and remainder.  The magnitude of the remainder |
;| is always less than the magnitude of the divisor, and the sign of the remainder is   |
;| the same as the sign of the divisor.  Halt if the divisor is zero.                   |
;+---------------------------------------------------------------------------------+
;|
;|Call: divide
;|
;|In:   R0      Divisor.
;|      R1      Dividend.
;|
;|Out:  R0      Quotient.
;|      R1      Remainder
;|
;|Criticality 1.
;|
;|Alters R0/R1.
;|

Divide:          MOVE    R2,[TempDiv_R2,A0]       ;Criticality 6.  Save R2 and R3.
                 MOVE    R3,[TempDiv_R3,A0]
                 CHECK   R0,INT,R2                ;Check for futures and bad types.
                 BF      R2,^Div_NonInteger
                 CHECK   R1,INT,R2
                 BF      R2,^Div_NonInteger
                 BZ      R0,^Div_Zero             ;Halt if the divisor is zero.
                 BNZ     R1,^Div_DividendNZ
                 MOVE    0,R0                     ;If the dividend was zero, return a zero quotient and
                 BR      ^Div_Done                ;remainder.
Div_DividendNZ:  LT      R0,0,R2                  ;R2 is true if the divisor is negative.
                 EQUAL   R1,$80000000,R3          ;TempDiv_80000000 is true if the dividend was $80000000.
                 MOVE    R3,[TempDiv_80000000,A0]
                 BF      R3,^Div_Normal
                 EQUAL   R0,-1,R3                 ;In this case, when the divisor is -1, the division overflows.
                 BT      R3,^Div_Overflow         ;The dividend is $80000000.
                 ADD     R1,R0,R1                 ;When the divisor is positive, add it to the dividend;
                 BF      R2,^Div_Normal           ;when the divisor is negative, subtract it from the dividend.
                 SUB     R1,R0,R1                 ;The reverse adjustment will be made on the quotient later.
                 SUB     R1,R0,R1
Div_Normal:      LT      R1,0,R3                  ;R3 is true if the dividend is negative.
                 BF      R2,^Div_DivisorPos
                 NEQUAL  R0,$80000000,R2          ;The divisor is -$80000000.
                 BT      R2,^Div_DivisorNeg       ;When the dividend is positive, the quotient is -1.
                 MOVE    0,R0                     ;When the dividend is negative, the quotient is 0.
                 BT      R3,^Div_Done1
                 MOVE    -1,R0
                 ADD     R1,$80000000,R1
                 BR      ^Div_Done1
Div_DivisorNeg:  NEG     R0,R0                    ;If the divisor was negative, negate both it and the dividend.
                 NEG     R1,R1
                 NOT     R3,R3
Div_DivisorPos:  BF      R3,^Div_DividendPos
                 NEG     R1,R1                    ;If the dividend is now negative, negate only it.
Div_DividendPos: MOVE    R2,FOP0                  ;Now both the divisor and the dividend should be positive
                 MOVE    R3,FOP1                  ;(and no greater than $7FFFFFFF).
                 GT      R0,R1,R2
                 MOVE    R0,R3                    ;Move the divisor to R3.
                 MOVE    0,R0                     ;From now on R0 contains the quotient.
                 BT      R2,^Div_Done3
                 FFB     R1,R2
                 MOVE    R2,[TempDiv_Count,A0]
                 FFB     R3,R2                    ;R2 contains the number of extra bits of magnitude in the
                 SUB     R2,[TempDiv_Count,A0],R2 ;dividend over the divisor.
                 LSH     R3,R2,R3                 ;Shift the divisor so that its most significant bit is in the
                 MOVE    R2,[TempDiv_Count,A0]    ;same position as the dividend's.
                 BR      ^Div_Loop_1

Div_Loop:        SUB     R2,1,R2
                 MOVE    R2,[TempDiv_Count,A0]
                 ADD     R0,R0,R0                 ;Shift the quotient to the left and the divisor to the right.
                 LSH     R3,-1,R3
Div_Loop_1:      LT      R1,R3,R2                 ;Try subtracting the shifted divisor from the dividend.
                 BT      R2,^Div_Loop_2
                 SUB     R1,R3,R1
                 ADD     R0,1,R0                  ;If successful, increment the quotient.
Div_Loop_2:      MOVE    [TempDiv_Count,A0],R2
                 BNZ     R2,^Div_Loop

Div_Done3:       MOVE    FOP1,R2                  ;If the dividend was negative, negate the quotient;
                 BF      R2,^Div_Done2            ;if the remainder was positive, subtract the remainder
                 NEG     R0,R0                    ;from the divisor and subtract one from the quotient to keep the
                 BZ      R1,^Div_Done2            ;remainder positive.
                 SUB     R0,1,R0
                 SUB     R3,R1,R1
Div_Done2:       MOVE    FOP0,R2
                 BF      R2,^Div_Done1
                 NEG     R1,R1                    ;If the divisor was negative, negate the remainder.
Div_Done1:       MOVE    [TempDiv_80000000,A0],R3
                 BF      R3,^Div_Done
                 MOVE    FOP0,R2                  ;The dividend was $80000000.  Perform the quotient adjustment.
                 SUB     R0,1,R0                  ;When the divisor was positive, subtract 1 from the quotient;
                 BF      R2,^Div_Done             ;When the divisor was negative, add 1 to the quotient.
                 ADD     R0,2,R0
Div_Done:        MOVE    [TempDiv_R2,A0],R2       ;Restore registers and return.
                 MOVE    [TempDiv_R3,A0],R3
                 MOVE    FIP,IP

Div_Zero:        HALT    haltDiv0
Div_Overflow:    HALT    haltOverflow

Div_NonInteger:  CHECK   R0,FUT,R2                ;If either operand was a future, crash with the future fault;
```

```
                BT      R2,^Div_Future          ;otherwise, crash with the type fault.
                CHECK   R1,FUT,R2
                BT      R2,^Div_Future
                HALT    haltType
Div_Future:     HALT    haltFuture

fltDivide = IP:abs|fault|unchecked|Divide<<offsetN
fltCrashOverflow = IP:abs|fault|unchecked|Div_Overflow<<offsetN


;+---------------------------------------------------------------------------------+
;| Allocate and initialize a new closure.                                          |
;+---------------------------------------------------------------------------------+
;|
;|Call: newClosure
;|
;|In:   R0      First word of object.
;|
;|Out:  AID2    ^Object.
;|      R1      Object's ID.
;|
;|Criticality 3.
;|
;|Alters R0-R3/AID2.
;|

NewClosure:     MOVE    FIP,R1                  ;Criticality 6.
                MOVE    R1,F                    ;Criticality 3.
                MOVE    R1,[TempNC1_FIP,A0]
                CALL    allocNextObject
                MOVE    R0,R1
                DC      (%11000100000100000|(callClosure&%100000)<<4|callClosure&%011111)<<17
                WTAG    R0,INST3,R0             ;Install the faulter instruction.
                MOVE    R0,[oClosureCode,A2]
                MOVE    [TempNC1_FIP,A0],IP

fltNewClosure = IP:abs|fault|unchecked|NewClosure<<offsetN


;+---------------------------------------------------------------------------------+
;| Call the function in a closure.  This routine does not return.                  |
;+---------------------------------------------------------------------------------+
;|
;|Call: callClosure
;|
;|Criticality 0.
;|

CallClosure:    MOVE    A0,R3                   ;Criticality 5.
                MOVE    R3,A2                   ;Copy A0 to A2.
                DC      MSG:msgApplyFunction
                MOVE    [oFunctionNArgs,A2],R1
                OR      R0,R1,R0
                MOVE    A3,R2                   ;Mask the length of the object pointed by A3
                OR      R2,lengthM,R2           ;to nArgs words.
                XOR     R2,lengthM,R2
                OR      R2,R1,R2
                MOVE    R2,A3
                SUB     R3,1,R2
                AND     R3,lengthM,R3
                SUB     R3,oClosureDisplay,R3   ;Put the number of display arguments in R3.
                ADD     R0,R3,R0                ;Update the length of the message.
                MOVE    NNR,R1
                SEND20  R1,R0                   ;Send the message back to this node.
                SEND0   [oClosureFunct,A2]      ;Send the real function.
                DC      IP:abs|unchecked|CallClosure_2<<offsetN
                MOVE    R0,[LimitOverride,A0]   ;Override the Limit fault.
                SEND0   [2,A3]                  ;Send the rest of the arguments.
                SEND0   [3,A3]
                SEND0   [4,A3]
                SEND0   [5,A3]
                SEND0   [6,A3]
                SEND0   [7,A3]
                SEND0   [8,A3]
                SEND0   [9,A3]
                SEND0   [10,A3]
                SEND0   [11,A3]
                SEND0   [12,A3]
                SEND0   [13,A3]
                SEND0   [14,A3]
                SEND0   [15,A3]
                MOVE    16,R0
CC1_SendRest:   SEND0   [R0,A3]                 ;Sends more arguments.
                ADD     R0,1,R0
                SEND0   [R0,A3]
                ADD     R0,1,R0
                SEND0   [R0,A3]
                ADD     R0,1,R0
                SEND0   [R0,A3]
                ADD     R0,1,R0
                BR      ^CC1_SendRest
CallClosure_2:  AND     R2,lengthM,R3           ;Get the last display argument.
                MOVE    [R3,A2],R3
                MOVE    R2,A2                   ;Decrement the length of the display by one.
                MOVE    2,R0
                CALL    blockSend               ;Send the display arguments.
                SENDE0  R3
                CALL    suspend

fltCallClosure = IP:abs|unchecked|CallClosure<<offsetN
```

```
                        ;########################
                        ;##                    ##
                        ;##  Control Manager   ##
                        ;##                    ##
                        ;########################

;+--------------------------------------------------------------------------------+
;| Execute an Apply, ApplyFunction, or ApplySelector message.                     |
;+--------------------------------------------------------------------------------+

Apply:          MOVE    [applyFunct,A3],R1      ;Criticality 0.  Get the funct.
                CHECK   R1,TAG0,R2              ;If it has tag 0, assume it is a selector.
                BT      R2,^ApplySelector
                CHECK   R1,ID,R2                ;If it has tag ID, assume it is a function.
                BT      R2,^ApplyFunction
                HALT    haltApply               ;Otherwise the message was invalid.

ApplyFunction:  MOVE    [applyFunct,A3],R0      ;Criticality 0.  Get the function.
                XLATE   R0,objectXLATE,A0
                DC      IP:oFunctionCode<<offsetN ;Start executing at the second word of the function.
                MOVE    R0,IP

ApplySelector:  MOVE    [applyReceiver,A3],R0   ;Criticality 0.  Get the receiver.
                PROBE   R0,R1                   ;Probe it, hoping it is an ID or DID.
                BNIL    R1,^AS_Miss
                MOVE    R1,A2                   ;If so, point A2 to the instance object.
                MOVE    R0,ID2
                MOVE    [objectHeader,A2],R0    ;Extract the class from the object header.
                WTAG    R0,INT,R0
                ROT     R0,-hdrClassN,R0
                AND     R0,hdrClassM,R0
AS_1:           MOVE    [applyFunct,A3],R1      ;Get the selector.
                CALL    lookupMethodU           ;R0 now contains INT:class.
                DC      IP:oFunctionCode<<offsetN ;Go execute the method.
                XLATE   R2,objectXLATE,A0
                MOVE    R0,IP

AS_Miss:        CALL    typeOf                  ;Call the real class-extraction routine.
                BR      ^AS_1

msgApply = Apply<<offsetN
msgApplyFunction = ApplyFunction<<offsetN
msgApplySelector = ApplySelector<<offsetN
```

```
                                ;######################
                                ;##                  ##
                                ;##  Initialization  ##
                                ;##                  ##
                                ;######################


;+--------------------------------------------------------------------------------+
;| Initialize the MDP.                                                            |
;+--------------------------------------------------------------------------------+

InitializeMDP:    DC       ADDR:invalid                    ;Clear the user address and ID registers.
                  MOVE     R0,A0
                  MOVE     R0,A1
                  MOVE     R0,A2
                  MOVE     R0,A3
                  MOVE     R0,A0B
                  MOVE     R0,A1B
                  MOVE     R0,A2B
                  MOVE     R0,A3B
                  MOVE     R0,A0B'
                  MOVE     R0,A1B'
                  MOVE     R0,A2B'
                  MOVE     R0,A3B'
                  MOVE     NIL,R0
                  MOVE     R0,ID0
                  MOVE     R0,ID1
                  MOVE     R0,ID2
                  MOVE     R0,ID3
                  MOVE     R0,ID0B
                  MOVE     R0,ID1B
                  MOVE     R0,ID2B
                  MOVE     R0,ID3B
                  MOVE     R0,ID0B'
                  MOVE     R0,ID1B'
                  MOVE     R0,ID2B'
                  MOVE     R0,ID3B'
                  MOVE     -1,R1                           ;Clear all globals to CFUT:-1.
                  WTAG     R1,CFUT,R1                      ;R1 contains CFUT:-1.
                  DC       64
IMDP_ClrGlobals:  SUB      R0,1,R0
                  MOVE     R1,[R0,A0]
                  BNZ      R0,^IMDP_ClrGlobals

                  DC       ADDR:Queue1Start<<baseN ;Initialize the queues.
                  MOVE     R0,QHL
                  DC       ADDR:Queue1Start<<baseN|(Queue1End-Queue1Start-1)
                  MOVE     R0,QBM
                  DC       ADDR:Queue0Start<<baseN
                  MOVE     R0,QHL
                  DC       ADDR:Queue0Start<<baseN|(Queue0End-Queue0Start-1)
                  MOVE     R0,QBM

                  MOVE     NIL,R2                          ;R2 contains NIL.
                  DC       ADDR:XlateStart<<baseN|(XlateEnd-XlateStart-1)
                  MOVE     R2,[LimitOverride,A0]           ;Initialize LimitOverride.
                  MOVE     R0,TBM                          ;Initialize the xlate table.
                  DC       ADDR:XlateStart<<baseN
                  MOVE     R2,[FastContextQueue,A0] ;Initialize FastContextQueue.
                  MOVE     R0,A2
                  DC       XlateEnd-XlateStart
IMDP_ClrXlate:    SUB      R0,1,R0                         ;Clear every entry in the table to NIL.
                  MOVE     R2,[R0,A2]
                  BNZ      R0,^IMDP_ClrXlate
                  MOVE     R2,[BRATFree,A0]                ;Initialize BRATFree.
                  DC       ADDR:BRATStart<<baseN
                  MOVE     R0,A2
                  DC       BRATEnd-BRATStart               ;Clear the BRAT.
IMDP_ClrBrat:     SUB      R0,1,R0
                  MOVE     R2,[R0,A2]
                  BNZ      R0,^IMDP_ClrBrat
                  DC       FixedHeapStart                  ;Initialize the heap.
                  MOVE     R0,[HeapStart,A0]
                  MOVE     R0,[FirstFree,A0]
                  MOVE     R0,R3
                  DC       HeapEnd
                  MOVE     R0,[LastFree,A0]
                  IF !FASTSIM
IMDP_ClrHeap:      MOVE    R1,[R3,A0]                      ;Clear the heap to CFUT:-1.
                   ADD     R3,1,R3
                   GE      R3,R0,R2
                   BF      R2,^IMDP_ClrHeap
                  END
                  MOVE     NNR,R2                          ;Initialize RandomSeed, and SerialNode.
                  MOVE     R2,[RandomSeed,A0]
                  DC       nodeMask
                  MOVE     R0,[NodeMask,A0]
                  AND      R2,xM,R3                        ;Calculate this node's serial number from the NNR value.
                  ROT      R2,-yN,R0
                  AND      R0,yM,R0
                  ROT      R0,xL,R0
                  OR       R3,R0,R3
                  ROT      R2,-zN,R0
                  AND      R0,zM,R0
                  ROT      R0,xL+yL,R0
                  OR       R3,R0,R3
                  MOVE     R3,[SerialNode,A0]
                  DC       ID:(nFastContexts-1)<<serialN
                  OR       R0,R2,R0                        ;Initialize LastObjectID and NextDistobjID.
                  MOVE     R0,[LastObjectID,A0]
                  MOVE     0,R0
                  MOVE     R0,[NextDistobjID,A0]
                  MOVE     nFastContexts,R3                ;Make nFastContexts fast contexts.
IMDP_MakeFast:    SUB      R3,1,R3
                  MOVE     R3,[TempINITM_Context,A0]   ;Save the number of fast contents yet to be made.
                  ROT      R3,serialN,R1
                  MOVE     NNR,R3                          ;Put the node number into the context ID.
                  OR       R1,R3,R1
```

```
                WTAG      R1,ID,R1
                DC        OBJ:hdrLocked|hdrFast|contextSize
                CALL      allocObject
                XOR       R0,rel,R0                    ;Make the fast context ADDRs nonrelocatable.
                ENTER     R1,R0
                CALL      enterBinding
                MOVE      [FastContextQueue,A0],R0
                MOVE      R0,[contextNext,A2]
                MOVE      ID2,R0
                MOVE      R0,[FastContextQueue,A0]
                MOVE      [TempINITM_Context,A0],R3
                BNZ       R3,^IMDP_MakeFast
                MOVE      R0,ID1B                      ;Initialize priority 0's AID1 to a fast context.
                MOVE      A2,R1
                MOVE      R1,A1B
                MOVE      [contextNext,A2],R0
                MOVE      R0,[FastContextQueue,A0]
                MOVE      [FirstFree,A0],R0            ;The real heap starts after the fast contexts.
                MOVE      R0,[HeapStart,A0]
                MOVE      FALSE,R0                     ;Enable message reception.
                MOVE      R0,I
IMDP_Background:
                IF !REALMODE
                 STOP                                  ;Do nothing in the background mode.
                END
                BR        ^IMDP_Background

OSEnd:
FixedHeapStart:


                ORG       Faults0Start
;Priority 0 faults:
                DC        fltCrash0                    ;CATASTROPHE
                DC        fltCrash0                    ;INTERRUPT
                DC        fltCrash0                    ;QUEUE
                DC        fltSend                      ;SEND
                DC        fltCrash0                    ;ILGINST
                DC        fltCrash0                    ;DRAMERR
                DC        fltINVADR                    ;INVADR
                DC        fltCrashType                 ;ADRTYPE
                DC        fltLimit                     ;LIMIT
                DC        fltEarly                     ;EARLY
                DC        fltCrash0                    ;MSG
                DC        fltXLATE                     ;XLATE
                DC        fltCrash0verflow             ;OVERFLOW
                DC        fltCFUT                      ;CFUT
                DC        fltCrashFuture               ;FUT
                DC        fltCrashType                 ;TAG8 - ID
                DC        fltCrashType                 ;TAG9 - DID
                DC        fltCrashType                 ;TAGA
                DC        fltCrashType                 ;TAGB - FLOAT
                DC        fltCrashType                 ;TYPE
                DC        fltCrash0                    ;$14
                DC        fltCrash0                    ;$15
                DC        fltCrash0                    ;$16
                DC        fltCrash0                    ;$17
                DC        fltCrash0                    ;$18
                DC        fltCrash0                    ;$19
                DC        fltCrash0                    ;$1A
                DC        fltCrash0                    ;$1B
                DC        fltCrash0                    ;$1C
                DC        fltCrash0                    ;$1D
                DC        fltCrash0                    ;$1E
                DC        fltCrash0                    ;$1F
;Priority 1 faults:
                DC        fltCrash1                    ;CATASTROPHE
                DC        fltCrash1                    ;INTERRUPT
                DC        fltCrash1                    ;QUEUE
                DC        fltCrash1                    ;SEND
                DC        fltCrash1                    ;ILGINST
                DC        fltCrash1                    ;DRAMERR
                DC        fltCrash1                    ;INVADR
                DC        fltCrash1                    ;ADRTYPE
                DC        fltCrash1                    ;LIMIT
                DC        fltCrash1                    ;EARLY
                DC        fltCrash1                    ;MSG
                DC        fltCrash1                    ;XLATE
                DC        fltCrash1                    ;OVERFLOW
                DC        fltCrash1                    ;CFUT
                DC        fltCrash1                    ;FUT
                DC        fltCrash1                    ;TAG8 - ID
                DC        fltCrash1                    ;TAG9 - DID
                DC        fltCrash1                    ;TAGA
                DC        fltCrash1                    ;TAGB - FLOAT
                DC        fltCrash1                    ;TYPE
                DC        fltCrash1                    ;$14
                DC        fltCrash1                    ;$15
                DC        fltCrash1                    ;$16
                DC        fltCrash1                    ;$17
                DC        fltCrash1                    ;$18
                DC        fltCrash1                    ;$19
                DC        fltCrash1                    ;$1A
                DC        fltCrash1                    ;$1B
                DC        fltCrash1                    ;$1C
                DC        fltCrash1                    ;$1D
                DC        fltCrash1                    ;$1E
                DC        fltCrash1                    ;$1F
;System calls:
                DC        fltSuspend                   ;$00
                DC        fltBlockMove                 ;$01
                DC        fltBlockSend                 ;$02
                DC        fltCompactHeap               ;$03
                DC        fltAllocObject               ;$04
                DC        fltEnterBinding              ;$05
                DC        fltLookupBinding             ;$06
                DC        fltDeleteBinding             ;$07
                DC        fltPurgeBinding              ;$08
```

```
                DC      fltNewLocalObject       ;$09
                DC      fltAllocNextObject      ;$0A
                DC      fltAllocNewObject       ;$0B
                DC      fltNewContext           ;$0C
                DC      fltDisposeContext       ;$0D
                DC      fltDisposeObject        ;$0E
                DC      fltDeallocateObject     ;$0F
                DC      fltNewObject            ;$10
                DC      fltClassOf              ;$11
                DC      fltTypeOf               ;$12
                DC      fltObjectNode           ;$13
                DC      fltPreferredConstituent ;$14
                DC      fltCo                   ;$15
                DC      fltLookupMethod         ;$16
                DC      fltLookupMethodU        ;$17
                DC      fltDivide               ;$18
                DC      fltNewClosure           ;$19
                DC      fltCallClosure          ;$1A
                DC      fltCrashCall            ;$1B
                DC      fltCrashCall            ;$1C
                DC      fltCrashCall            ;$1D
                DC      fltCrashCall            ;$1E
                DC      fltCrashCall            ;$1F
                DC      fltCrashCall            ;$20
                DC      fltCrashCall            ;$21
                DC      fltCrashCall            ;$22
                DC      fltCrashCall            ;$23
                DC      fltCrashCall            ;$24
                DC      fltCrashCall            ;$25
                DC      fltCrashCall            ;$26
                DC      fltCrashCall            ;$27
                DC      fltCrashCall            ;$28
                DC      fltCrashCall            ;$29
                DC      fltCrashCall            ;$2A
                DC      fltCrashCall            ;$2B
                DC      fltCrashCall            ;$2C
                DC      fltCrashCall            ;$2D
                DC      fltCrashCall            ;$2E
                DC      fltCrashCall            ;$2F
                DC      fltCrashCall            ;$30
                DC      fltCrashCall            ;$31
                DC      fltCrashCall            ;$32
                DC      fltCrashCall            ;$33
                DC      fltCrashCall            ;$34
                DC      fltCrashCall            ;$35
                DC      fltCrashCall            ;$36
                DC      fltCrashCall            ;$37
                DC      fltCrashCall            ;$38
                DC      fltCrashCall            ;$39
                DC      fltCrashCall            ;$3A
                DC      fltCrashCall            ;$3B
                DC      fltCrashCall            ;$3C
                DC      fltCrashCall            ;$3D
                DC      fltCrashCall            ;$3E
                DC      fltCrashCall            ;$3F

                END

BREAK HAZARDS                                   ;Break on hazards.
IF !REALMODE
  BREAK FAULT Faults0Start,Faults1Start         ;Break on catastrophic faults.
  BREAK READ WRITE OSStart..OSEnd-1             ;Protect operating system code.
  BREAK FETCH 0..OSStart                        ;Globals cannot be executed.
  IGNORE FETCH $400,$401                        ;The initialization code, however, can.
  BREAK READ WRITE Faults0Start..CallsEnd-1     ;Fault vectors are protected.
  STEP 300                                      ;Allow the operating system to write globals.
  BREAK FETCH $400,$401                         ;The initialization code is now gone.
  BREAK READ WRITE 0..3                         ;Locations 0 through 3 are not used for anything.
END

INCLUDE "Runtime.m"                             ;Load the run-time system.

IF !REALMODE
  RUN                                           ;Initialize the operating system.
END
```

Runtime.m

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;                                            ;;;;
;;;;              MDP Operating System          ;;;;
;;;;                                            ;;;;
;;;;                  version 2.3               ;;;;
;;;;                                            ;;;;
;;;;                  written by                ;;;;
;;;;                Waldemar Horwat             ;;;;
;;;;                                            ;;;;
;;;;  Master's thesis under Prof. William Dally ;;;;
;;;;                                            ;;;;
;;;;                March 28, 1989              ;;;;
;;;;                   May 1991                 ;;;;
;;;;                                            ;;;;
;;;;        Send problems and comments to       ;;;;
;;;;           waldemar@hx.lcs.mit.edu.         ;;;;
;;;;                                            ;;;;
;;;; Copyright 1989, 1990, 1991 Waldemar Horwat ;;;;
;;;;                                            ;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;The download header is appended to the beginning of every module that is downloaded.
                MODULE  DOWNLOADHEADER
                DC      MSG:msgAcceptObject|+2
                DC      IONODE
                END


                MODULE  LookupMethod
vCurrentClass = 6       ;Class number of superclass currently scanned.

Begin0:         DC      OBJ:hdrCopyable:classFunction<<hdrClassN:End0-Begin0
                DC      (LookupMethod)
                DC      5
                MOVE    [lookMethSelector,A3],R0
                XLATE   R0,objectXLATE,A2         ;Point AID2 to the selector object.
                MOVE    [lookMethClass,A3],R3     ;Store the class in R3.
                MOVE    [lookMethReplyID,A3],R0   ;Save the reply ID in the context.
                MOVE    R0,[lookMethReplyID,A1]
                MOVE    FALSE,R0                  ;Turn off A3 queue wraparound.
                MOVE    R0,Q
                MOVE    R0,[vCurrentClass,A1]
                MOVE    [oSelNMethods,A2],R2
                MOVE    oSelMethods,R1
                BZ      R2,^Miss1
Search1:        EQ      R3,[R1,A2],R0             ;Search the class/method associations for the
                BT      R0,^FoundMethod           ;class in R3.
                ADD     R1,2,R1
                SUB     R2,1,R2
                BNZ     R2,^Search1
Miss1:          XLATE   R3,objectXLATE,A3         ;If no association was found, scan the class's
                MOVE    0,R0                      ;superclasses.
Miss2:          ADD     R0,1,R0
                GE      R0,[oClassNAllSupers,A3],R1
                BT      R1,^MissAll               ;Return NIL if an association still wasn't found.
                MOVE    R0,[vCurrentClass,A1]
                ADD     R0,oClassAllSupers,R0
                MOVE    [R0,A3],R3
                MOVE    [oSelNMethods,A2],R2
                MOVE    oSelMethods,R1
Search2:        EQ      R3,[R1,A2],R0             ;Search the class/method associations for the
                BT      R0,^FoundMethod           ;class in R3.
                ADD     R1,2,R1
                SUB     R2,1,R2
                BNZ     R2,^Search2
                MOVE    [vCurrentClass,A1],R0
                BR      ^Miss2

MissAll:        MOVE    NIL,R2                    ;No method was found, so return NIL.
                BR      ^FoundMethod2
FoundMethod:    ADD     R1,1,R1                   ;Extract the method ID
                MOVE    [R1,A2],R2
FoundMethod2:   MOVE    [lookMethReplyID,A1],R1
                DC      MSG:msgMethodReply+3      ;Return a reply message with the method ID.
                SEND20  R1,R0
                SEND2E0 R1,R2
                SUSPEND
End0:
                END


;NewDistobj message:
LABEL newDistobjClass = 2
LABEL newDistobjSize = 3
LABEL newDistobjReturnID = 4
LABEL newDistobjReturnSlot = 5
LABEL newDistobjTemp = 6          ;Temporary

                MODULE  f_New_Distobj
Begin:          DC      OBJ:hdrCopyable:classFunction<<hdrClassN:End-Begin
                DC      (f_New_Distobj)
                DC      6
                MOVE    NNR,R1
                EQUAL   R1,0,R1
                BT      R1,^OnNode0
                DC      MSG:msgApplyFunction+6
                SEND20  0,R0                      ;If not, forward this message to node 0
                DC      (f_New_Distobj)
                SEND0   R0
```

```
                SEND0    [newDistobjClass,A3]
                SEND0    [newDistobjSize,A3]
                SEND0    [newDistobjReturnID,A3]
                SENDE0   [newDistobjReturnSlot,A3]
                SUSPEND
OnNode0:        MOVE     [newDistobjSize,A3],R0  ;Put max(size,1) into R0.
                GT       R0,0,R1
                BT       R1,^PositiveSize
                MOVE     1,R0
PositiveSize:   SUB      R0,1,R0                 ;Calculate lg(max(size,1)) and store it in R0.
                FFB      R0,R1
                MOVE     31,R0
                SUB      R0,R1,R0
                ADD      R0,-LogNNodes,R1        ;R1 contains -stride.
                NEG      R1,R3                   ;R3 contains stride.
                MOVE     -1,R2
                ASH      R2,R3,R2
                NOT      R2,R2
                DC       ADDR:64                 ;Point A2 to the global area.
                MOVE     R0,A2                   ;Criticality 2.
                MOVE     [RandomSeed,A2],R3
                ADD      R3,3,R3
                MOVE     R3,[RandomSeed,A2]
                AND      R2,R3,R2                ;R2 contains the offset.
                MOVE     [NextDistobjID,A2],R3   ;Get the ID for this distributed object.
                MOVE     -1,R0                   ;Advance the ID counter by the number of constituents
                ASH      R0,R1,R0                ;per node.
                NOT      R0,R0
                ADD      R0,1,R0
                ADD      R3,R0,R3
                MOVE     R3,[NextDistobjID,A2]
                SUB      R3,R0,R3                ;Criticality 1.
                ROT      R3,serialN,R3
                OR       R3,distobjMember,R3     ;Calculate the DID for this distributed object and store it
                OR       R3,R2,R3                ;in R3.
                NEG      R1,R2
                ASH      R2,logStrideN,R2
                AND      R2,(1<<logStrideN+logStrideL)-1,R2
                OR       R3,R2,R3
                WTAG     R3,DID,R3
                MOVE     LogNNodes,R2
                ADD      R2,R1,R2                ;Put lg(max(size,1)) into newDistobjTemp.
                MOVE     R2,[newDistobjTemp,A1]
                MOVE     0,R0
                MOVE     R3,R1                   ;Send a newDistobjTree message to the node that will contain the
                CALL     co                      ;first constituent of the distributed object.
                DC       MSG:msgApplyFunction|9
                SEND20   R1,R0
                DC       (fNewDistobjTree)
                SEND0    R0
                SEND0    [newDistobjClass,A3]
                SEND20   [newDistobjSize,A3],R3
                SEND0    0
                SEND0    [newDistobjTemp,A1]
                SEND0    [newDistobjReturnID,A3]
                SENDE0   [newDistobjReturnSlot,A3]
                SUSPEND
End:
                END
REF REV f_New_Distobj = ID:(1<<30|(1&mX)<<sX|(1&mY)<<sY|(1&mZ)<<sZ|(1&mS)<<sS)


;NewDistobjTree message:
LABEL newDistobjTreeClass = 2
LABEL newDistobjTreeSize = 3
LABEL newDistobjTreeID = 4
LABEL newDistobjTreeStart = 5
LABEL newDistobjTreeLogDelta = 6
LABEL newDistobjTreeReturnID = 7
LABEL newDistobjTreeReturnSlot = 8

                MODULE   fNewDistobjTree
Begin2:         DC       OBJ:hdrCopyable|classFunction<<hdrClassN:End2-Begin2
                DC       (fNewDistobjTree)
                DC       9
                MOVE     [newDistobjTreeLogDelta,A3],R3
                BZ       R3,^Leaf
                SUB      R3,1,R3
                MOVE     NNR,R1
                DC       MSG:msgApplyFunction|9
                SEND20   R1,R0                   ;Call fNewDistobjTree twice, each time on half of the range
                DC       (fNewDistobjTree)       ;of constituents.
                SEND0    R0
                SEND0    [newDistobjTreeClass,A3]
                SEND0    [newDistobjTreeSize,A3]
                SEND0    [newDistobjTreeID,A3]
                SEND20   [newDistobjTreeStart,A3],R3
                MOVE     9,R0
                SEND2E0  [contextID,A1],R0
                WTAG     R0,CFUT,R0              ;Make a cfuture in R0.
                MOVE     R0,[9,A1]
                MOVE     [newDistobjTreeID,A3],R1
                MOVE     [newDistobjTreeStart,A3],R0
                MOVE     1,R2
                ASH      R2,R3,R2
                ADD      R0,R2,R0
                MOVE     R0,[11,A1]
                CALL     co
                DC       MSG:msgApplyFunction|9
                SEND20   R1,R0
                DC       (fNewDistobjTree)
                SEND0    R0
                SEND0    [newDistobjTreeClass,A3]
                SEND0    [newDistobjTreeSize,A3]
                SEND0    [newDistobjTreeID,A3]
                SEND20   [11,A1],R3
                MOVE     10,R0
                SEND2E0  [contextID,A1],R0
```

```
                    WTAG     R0,CFUT,R0                  ;Make a cfuture in R0.
                    MOVE     R0,[10,A1]
                    MOVE     [9,A1],R0                   ;Force the cfutures.
                    MOVE     [10,A1],R0
                    BR       ^Return
Leaf:               MOVE     [newDistobjTreeClass,A3],R0
                    XLATE    R0,objectXLATE,A2           ;Get the header word for objects of this class and put it in R3.
                    MOVE     [oClassWord,A2],R3
                    MOVE     [newDistobjTreeID,A3],R1
                    MOVE     [newDistobjTreeStart,A3],R0
                    CALL     co
                    MOVE     R3,R0
                    CALL     allocNewObject
                    MOVE     [newDistobjTreeID,A3],R0
                    MOVE     R0,[oDistobjGroup,A2]     ;Initialize the group, index, and logical-limit instance variables.
                    MOVE     [newDistobjTreeStart,A3],R0
                    MOVE     R0,[oDistobjIndex,A2]
                    MOVE     [newDistobjTreeSize,A3],R0
                    MOVE     R0,[oDistobjLogicalLimit,A2]
Return:             MOVE     [newDistobjTreeReturnID,A3],R3
                    BNIL     R3,^GoSuspend
                    DC       MSG:msgReply|4
                    SEND20   R3,R0
                    SEND0    R3
                    SEND0    [newDistobjTreeReturnSlot,A3]
                    SENDE0   [newDistobjTreeID,A3]
GoSuspend:          SUSPEND
End2:
                    END
REF REV fNewDistobjTree = ID:(1<<30|(2&mX)<<sX|(2&mY)<<sY|(2&mZ)<<sZ|(2&mS)<<sS)

DOWNLOAD LookupMethod
DOWNLOAD f_New_Distobj
DOWNLOAD fNewDistobjTree
```

Bibliography

[1] Abelson, Harold and Sussman, Gerald J. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 1985.

[2] Agha, Gul. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, 1986.

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

[4] Athas, William C. "Fine Grain Concurrent Computation." California Institute of Technology Ph.D. Thesis, 1987.

[5] Atkinson, Robert G. "Hurricane: An Optimizing Compiler for Smalltalk." *Proceedings of the 1986 Object-Oriented Programming Systems, Languages, and Applications Conference*, September 1986.

[6] Bobrow, Daniel G., DeMichiel, Linda G., Gabriel, Richard P., Keene, Sonya E., Kiczales, Gregor, and Moon, David A. *Common Lisp Object System Specification*, Chapters 1 and 2. X3J13 Document 88-002R, June 1988.

[7] Burke, Glenn and Moon, David. "Loop Iteration Macro." MIT Laboratory for Computer Science Memo TM-169, January 1981.

[8] Chien, Andrew A. "Concurrent Aggregates (CA): An Object-Oriented Language for Fine-Grained Message-Passing Machines." MIT Artificial Intelligence Laboratory Technical Report 1248, July 1990.

[9] Chien, Andrew A. "The Concurrent Aggregates (CA) Language Report, Version 0.3." Working Draft, MIT Concurrent VLSI Architecture Group, April 1989.

[10] Chien, Andrew A. "The Particle-in-Cell Code (PIC): An Application Study for CST." MIT Concurrent VLSI Architecture Memo 16, December 1988.

[11] Dally, William J. "Fast Context Switching on the MDP." MIT Concurrent VLSI Architecture Memo 13, August 1988.

[12] Dally, William J. "Micro-Optimization of Floating-Point Operations." *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989.

[13] Dally, William J. *A VLSI Architecture for Concurrent Data Structures*. Kluwer, Boston, MA, 1987.

[14] Dally, William J. et al. "Architecture of a Message-Driven Processor." *Proceedings of the 14th ACM/IEEE Symposium on Computer Architecture*, June 1987, pp. 189-196.

[15] Dally, William J. et al. "The Message-Driven Processor." Hot Chips III Symposium, August 1991, p. 2.11–2.21.

[16] Dally, William J. et al. "Message-Driven Processor Architecture, Version 11." MIT Concurrent VLSI Architecture Memo 14; MIT Artificial Intelligence Laboratory Memo 1069, March 1988.

[17] Dally, William J. and Chien, Andrew A. "Object-Oriented Concurrent Programming in CST." *Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications*, Pasadena, CA, January 1988.

[18] Deutsch, L. Peter, and Schiffman, Allan M. "Efficient Implementation of the Smalltalk-80 System." *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*. 1983.

[19] Fiske, James Alexander Stuart. "A Reconfigurable Arithmetic Processor." MIT Master's Thesis in Electrical Engineering And Computer Science, December 1988.

[20] Goldberg, Adele and Robson, David. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.

[21] Horwat, Waldemar. "A Concurrent Smalltalk Compiler for the Message-Driven Processor." MIT Artificial Intelligence Laboratory Technical Report 1080, May 1988.

[22] Horwat, Waldemar, Chien, Andrew A., and Dally, William J. "Experience with CST: Programming and Implementation." *Proceedings of the 1989 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Portland, OR, June 1989.

[23] Horwat, Waldemar and Totty, Brian. "Message-Driven Processor Architecture, Version 10." MIT Concurrent VLSI Architecture Memo, March 1988.

[24] Horwat, Waldemar and Totty, Brian. "Message-Driven Processor Simulator, Version 5.0." MIT Concurrent VLSI Architecture Memo, December 1987.

[25] Horwat, Waldemar. "Message-Driven Processor Simulator, Version 7.0." MIT Concurrent VLSI Architecture Memo 38, May 1991.

[26] Johnson, Ralph E., Graver, Justin O., and Zurawski, Lawrence W. "TS: An Optimizing Compiler for Smalltalk." *Proceedings of the 1988 Object-Oriented Programming Systems, Languages, and Applications Conference*, September 1988.

[27] Keene, Sonya E. *Object-Oriented Programming in COMMON LISP: A Programmer's Guide to CLOS*. Addison Wesley, 1989.

[28] Knuth, Donald E. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Reading, MA, 1973.

[29] Lieberman, Henry and Hewitt, Carl. "A Real-Time Garbage Collector Based on the Lifetimes of Objects." *Communications of the ACM*, June 1983.

[30] Manning, Carl R. "ACORE: The Design of a Core Actor Language and its Compiler." MIT Master's Thesis in Computer Science, May 1987.

[31] Rees, Jonathan and Clinger, William (editors). "Revised^3 Report on the Algorithmic Language Scheme." MIT Artificial Intelligence Laboratory Memo 848a, September 1986.

[32] Song, Paul Y. "Design of a Network for Concurrent Message Passing Systems." MIT Master's Thesis in Computer Science, May 1988.

[33] Spertus, Ellen. "Dataflow Computation for the J-Machine." MIT Artificial Intelligence Laboratory Technical Report 1233, June 1990.

[34] Spertus, Ellen. "Preliminary Dataflow on the MDP." MIT Concurrent VLSI Architecture Memo 21, May 1989.

[35] Steele, Guy L. *Common Lisp: The Language*. Digital Press, Digital Equipment Corporation, 1984.

[36] Steele, Guy L. "Rabbit: A Compiler for Scheme." MIT Artificial Intelligence Technical Report 474, May 1978.

[37] Stroustrup, Bjarne. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1987.

[38] Totty, Brian K. "An Operating Environment for the Jellybean Machine." MIT Artificial Intelligence Laboratory Memo 1044, October 1988.

[39] Yokote, Yasuhiko, and Tokoro, Mario. "The Design and Implementation of ConcurrentSmalltalk." *Proceedings of the 1986 Object-Oriented Programming Systems, Languages, and Applications Conference*, September 1986.

[40] Yokote, Yasuhiko, and Tokoro, Mario. "Experience and Evolution of Concurrent-Smalltalk." *Proceedings of the 1987 Object-Oriented Programming Systems, Languages, and Applications Conference*, October 1987.

[41] Wills, D. Scott. "Multi-Model Execution on a Fine Grain Message Passing Substrate." MIT Concurrent VLSI Architecture Memo 20, January 1989.

[42] Wills, D. Scott. "Pi: A Parallel Architecture Interface for Multi-Model Execution." MIT Artificial Intelligence Laboratory Technical Report 1245, July 1990.

[43] Wulf, William M., Johnsson, Richard K., Weinstock, Charles B., Hobbs, Steven O., and Geschke, Charles, M. *The Design of an Optimizing Compiler.* American Elsevier, New York, 1975.

Concurrent Smalltalk Index