Technical Report 1233

Dataflow Computation for the J-Machine

Ellen Spertus

MIT Artificial Intelligence Laboratory

This blank page was inserted to preserve pagination.

Dataflow Computation for the J-Machine

Ellen Spertus

Abstract

The dataflow model of computation exposes and exploits parallelism in programs without requiring programmer annotation; however, instruction-level dataflow is too fine-grained to be efficient on general-purpose processors. A popular solution is to develop a "hybrid" model of computation where regions of dataflow graphs are combined into sequential blocks of code. I have implemented such a system to allow the J-Machine to run Id programs, leaving exposed a high amount of parallelism — such as among loop iterations. I describe this system and provide an analysis of its strengths and weaknesses and those of the J-Machine, along with ideas for improvement.

This report describes research done at the Artificial Intelligence Laboratory of the Massachussets Institute of Technology. Support was provided in part by the Advanced Research Projects Agency of the Department of Defense under contracts N00014-88K-0738 and N00014-87K-0825, in part by a National Science Foundation Presidential Young Investigator Award, grant MIP-8657531, with matching funds from General Electric Corporation and IBM Corporation, and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85K-0124.

Submitted to the Department of Electrical Engineering and Computer Science in May, 1990, in partial fulfillment of the requirements for the Degree of Bachelor of Science.

Acknowledgments

I have received help and encouragement from many people. First, I would like to thank the members of the MIT Concurrent VLSI Architecture group for their help. Julia Bernard, Andrew Chien, Stuart Fiske, Waldemar Horwat, John Keen, Richard Lethin, Mike Noakes, Peter Nuth, Lucien Van Elsen, Debby Wallach, and Scotty Wills have all helped me directly or indirectly in my work. They are all a great bunch of people. I am especially grateful to Scotty for getting Id World up on our machines, Mike for his all-around helpfulness, and Debby and Lucien for helping me with text and graphics formatting.

The members of the Computation Structures Group provided crucial help, generously sharing their equipment and expertise. I want to specifically thank Jonathan Young for his general helpfulness and Bradley Kuszmaul for describing his P-RISC research to me. I am especially grateful to Robert Iannucci, for developing his hybrid system and being willing to answer my questions, and Jamey Hicks, who got the hybrid system up on the current release of Id World.

Thanks are also due to Arvind, one of the major forces behind the dataflow group and also my academic advisor. He has been providing me with encouragement and advice for years, as well as teaching me about dataflow in 6.847.

I am very grateful to my thesis supervisor and head of the CVA group, Bill Dally. If it weren't for his faith in me, his encouragement and advice, his willingness to spend time and resources on me, and the wonderful working environment he provides, none of this would have been possible.

I am also grateful to my friend Nate Osgood for letting me bounce ideas off him and brainstorming with me, as well as for moral support.

Finally, I would like to thank my family. My parents, siblings, and grandparents have been wonderful role models, and they have unceasingly providing me with love and support. I am especially grateful to my father, who has given me encouragement he has given me since as far back as I can remember.

Contents

1 Introduction 1

1.1 Background . . . 2

1.1.1 Id . . . 3

1.1.2 Iannucci's Hybrid Architecture . . . 7

1.1.3 The J-Machine . . . 7

1.2 Overview . . . 8

2 Executing Hybrid Code on the J-Machine 9

2.1 Overview . . . 10

2.2 Data Structures . . . 11

2.2.1 Codeblocks . . . 11

2.2.2 The Data Stack . . . 12

2.2.3 Frames . . . 12

2.2.4 Continuations . . . 13

2.2.5 I-Structures . . . 15

2.3 Control Structure . . . 16

2.3.1 Execution Within a Codeblock . . . 16

2.3.2 Procedure Calls . . . 19

2.3.3 Loops . . . 20

2.4 Conclusion . . . 26

3 Compilation 27

3.1 Changes to Machine Code Generation . . . 30

3.1.1 Loops . . . 30
3.1.2 Procedure Calls . . . 30
3.2 Assembling Hybrid Code . . . 31
3.3 Convert Hybrid to Complex J . . . 33
3.3.1 Label Instruction . . . 34
3.3.2 Simple Arithmetic Instructions . . . 34
3.3.3 Complicated Arithmetic Instructions . . . 35
3.3.4 Move Instructions . . . 35
3.3.5 Test Instructions . . . 37
3.3.6 Continuation Instructions . . . 37
3.3.7 Procedure Linkage Instructions . . . 38
3.3.8 Conclusion . . . 39
3.4 Convert Complex J to Simple J . . . 42
3.4.1 Converting Literals to Tagged Literals . . . 42
3.4.2 Generating Suspensive Code . . . 42
3.4.3 Allocating MDP Registers . . . 44
3.4.4 Converting to Legal MDP Operands . . . 44
3.5 Convert Simple to ASM . . . 46
3.6 Conclusion . . . 47

4 Analysis **48**

4.1 Detailed Benchmark: Factorial . . . 48
4.1.1 The Dataflow Graph . . . 48
4.1.2 The Hybrid Code . . . 50
4.1.3 The MDP Code . . . 55
4.1.4 Load Balancing . . . 55
4.1.5 Dynamic Counts . . . 55
4.1.6 Throughput . . . 57
4.1.7 Conclusion . . . 57
4.2 Fibonacci . . . 58
4.3 Loop Parallelization . . . 59

4.4 Conclusion 62

5 Conclusion **63**

5.1 Improving MDP Code 64

5.2 Improving Hybrid Code 64

5.3 Strengths and Weaknesses of the J-Machine 65

5.4 Synchronization on Tokens 66

5.5 Conclusion 68

A MDP Program Examples **70**

A.1 MDP Code for Factorial 70

A.2 MDP Code for Fibonacci 76

A.3 MDP Code for Loop Example 84

B MDP Library Code **94**

B.1 General Library 94

B.2 I-Structure Routines 101

B.3 Loop Support 104

C Source Code **106**

C.1 Convert Hybrid to Complex J 106

C.2 Convert Complex J to Simple J 113

C.3 Convert Simple J to Assembly 124

List of Figures

1-1 A FSM Description of an I-Structure Location 5

2-1 Run-Time Data Structures 11
2-2 A Pointer to a Codeblock 12
2-3 A Non-Loop Procedure Frame 13
2-4 An I-Structure Descriptor and Storage 15
2-5 An I-Structure 16
2-6 A Statically Unschedulable Codeblock 17
2-7 Scheduling Quanta for Codeblock Example 18
2-8 Snapshots for Codeblock Example 18
2-9 Procedure Linkage Example 19
2-10 Possible Implementation of an Iteration Descriptor 21
2-11 A Loop Procedure Frame 22
2-12 Iteration Areas and Pointers 23
2-13 Loop Program Example 23
2-14 Pseudo-Code Produced for Loop Example 24
2-15 Snapshots for Loop Example 25

3-1 Structure of the Id-to-MDP Compiler 28
3-2 New and Modified Compiler Stages 29
3-3 A Non-Optimal J-Machine Calling Convention 32
3-4 The Ordering Specified by Successive Function Calls 33
3-5 The Hybrid-to-Complex-J Conversion of an Addition 35
3-6 The Template for Converting Absolute Value. 36

3-7 The Hybrid-to-Complex-J Conversion of a Test-1 37
3-8 Transformation of Get-Context 40
3-9 Transformation of Index-Current-Context 41
3-10 Intermediate Code Produced for Suspensive Pseudo-Operands 43
3-11 Compiler Register Allocation 45

4-1 Id Code for Factorial 49
4-2 A Dataflow Graph for Factorial 49
4-3 Hybrid Code for Factorial 52
4-4 Scheduling Quanta of Factorial Code 53
4-5 Frame Slots Used by Factorial Code 54
4-6 Id Code for Fibonacci 58
4-7 Plot of Ticks for Fibonacci 60
4-8 Id Code for Loop Example 61

5-1 A Monadic Node Using ETS 67
5-2 A Dyadic Node Using ETS 67
5-3 The Cfuture Handler for ETS 68

List of Tables

4.1 System Calls for (fact 4). 56
4.2 Dynamic Instruction Usage for (fact 4). 56
4.3 Throughput for Factorial 57
4.4 Timings for Fibonacci 59

Chapter 1

Introduction

If you can look into the seeds of time,
And say which grain will grow and which will not,
Speak.
— William Shakespeare, *Macbeth*, Act I, Scene iii, line 58.

This thesis describes a system I designed and implemented to allow programs written in the dataflow language Id to run on the J-Machine, a massively-parallel general-purpose computer. The system is functional and includes:

- A compiler that recognizes a significant portion of Id and produces J-Machine assembly code.
- Library routines to provide operating system functions, fault handlers, and language-specific features like I-structure storage.
- A strategy for aggressive loop parallelization.

I do not directly address the question of how to sequentialize portions of dataflow graphs. For this, I took advantage of the work done by Ken Traub on program partitioning [Traub 1988] and Robert Iannucci for his "dataflow / von Neumann hybrid" architecture and compiler [Iannucci 1988]. With some optimizations, my system simulates Iannucci's hybrid architecture on the J-Machine. In this document, I describe and justify my approach, detail my

transformations, analyze the results, and present my conclusions about the project and future research on dataflow computation for the J-Machine.

1.1 Background

A large amount of research has gone into developing and implementing the dataflow model of parallel computation. In order to exploit the parallelism revealed by dataflow techniques, special-purpose dataflow machines have been built that are unlike traditional von Neumann processors, using parallel machine languages and having token and I-structure memory. Because individual instructions are scheduled dynamically on dataflow processors, this leads to unnecessarily high run-time overhead. On the other hand, dataflow architectures, with their per instruction synchronization, are more tolerant than von Neumann machines at tolerating latency: If the data dependences allow some computation to be performed while the previously-executing task is waiting for data, the processor will be kept busy. The motivation for a hybrid architecture is to combine the latency toleration of a dataflow processor with the efficiency of a von Neumann processor. Often, enough is known at compile-time to specify a full ordering of a set of instructions, reducing the amount of run-time scheduling necessary. Hybrid architectures attempt to take advantage of this knowledge by delineating sequences of instructions whose order can be pre-determined, combining the exposed parallelism of dataflow with the efficiency of von Neumann computation.[1]

While combining instructions into sequential threads theoretically lessens the amount of run-time parallelism available, it can be more practical in that it minimizes scheduling overhead and allows the code to run on computers not dedicated to dataflow processing. Additionally, even dataflow computers do not attempt to exploit the maximum possible parallelism. For example, on Monsoon, a specific invocation of a procedure is generally not divided among processors but takes place on a single one. Instead, the parallelism comes from pipelining and from running iterations of one loop concurrently on separate processors [Papadopoulos and Culler 1990], a feature that is retained by hybrid architectures. In order to ensure that grouping instructions into threads does not lessen the ability to tolerate latency,

[1]This justification of hybrid architectures based on latency toleration is due to ideas in [Iannucci 1988, Chapters 1 and 2].

we obey "Iannucci's Injunction" that instructions within a thread may not have unbounded latency. Instructions with unbounded latency — such as procedure calls and global memory accesses — cause a thread to suspend, allowing another to execute.

My work includes a compiler back-end to allow dataflow programs to run on the J-Machine, a general-purpose massively-parallel computer. Although closer to the von Neumann model than dataflow architectures, the J-Machine has many of the necessary communication and naming primitives needed for dataflow computation. I built my back-end on top of the Id compiler developed by the Computation Structures Group at the MIT Laboratory for Computer Science [Traub 1986a], as augmented by Robert Iannucci to produce code for his hybrid architecture [Iannucci 1988]. My system transforms his hybrid code to run on the J-Machine.

1.1.1 Id

Id is a primarily functional language developed in the Computation Structures Group of the MIT Laboratory for Computer Science for programming dataflow and other parallel computers. [Nikhil 1988] is a reference for the latest version. All of its features are supported by my transformations, except for algebraic types, as they postdate Iannucci's compiler on which mine is based. A quick overview of pertinent features of the language is presented here.

Types

The only primitive types in Id are booleans, characters, numbers, character strings, and symbols.[2] Additionally, there are four pre-defined type constructors that take one or more types and create new types:

- array types: (1D_array t), (2D_array t), ...
- list types: (list t)
- tuple types: $(t_0, ..., t_n)$
- function types: $(t_0 \rightarrow t_1)$

[2] In the latest version of Id, booleans are not primitive but are defined with *algebraic types*, which we were unable to support, as described above.

Id is strongly-typed in that extensive compile-time and run-time type-checking is performed, but users rarely explicitly provide type information. Additionally, Id allows polymorphism.

Function Application

The application of function f with arguments $a_1, ..., a_n$ is written:

$$f\ a_1 ... a_n$$

Id also supports currying: If function f "expects" two arguments, fa, instead of being illegal as in most languages, returns a function that takes one argument. For example, if *plus* is defined as a function that takes two numbers and adds them, *plus 3* returns a function that takes one number as an argument and adds 3 to it. As will be seen later, currying causes additional overhead in run-time procedure linkage.

I-Structures

One major argument against purely functional languages is their suboptimal efficiency with arrays. Specifically, it is unnecessarily wasteful to copy an entire array when modifying one element. Filling in the n elements of a previously-empty array can take $O(n^2)$ time and space, as the entire array is recopied when each element is written. This problem was partially solved with I-structures, arrays with elements that can only be written to once. After being written to, reads take place as expected; subsequent writes are a run-time error. Because no copying is done, filling an array of I-structures takes $O(n)$ time. If a read takes place before a write, the read is silently deferred until the data is available. This process is illustrated in Figure 1-1. Out-of-bound accesses to I-structures cause run-time errors. The properties of I-structures guarantee deterministic behavior in legal programs[3]. While keeping Id from being purely functional, they greatly improve its efficiency without harming abstraction. Tuples and arrays, described above, are implemented as I-structures.

In addition to supporting user types, I-structures are used to create closures for currying

[3]Here and elsewhere, a *legal* program is one in which no compile-time or run-time errors occur.

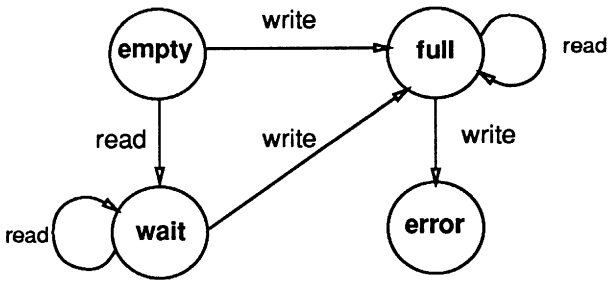


Figure 1-1: A FSM Description of an I-structure Location. Originally, an I-structure location is empty. Reads are silently deferred until data has arrived. Once data has been written, pending and subsequent read requests can be fulfilled. Writing a location more than once is a run-time error.

procedure calls. Whenever an argument is applied to a procedure, a check is made whether the argument supplied is the last one. If so, the procedure is invoked; otherwise, the argument is added to the I-structure list of arguments and saved into a closure.

Blocks

Blocks in Id provide a mechanism to bind names to values within the block's body. It is analogous to Lisp's *let* construct, except that, as in all Id constructs, the textual order of the statements is ignored. A block to compute the surface area of a cylinder, given its radius r and height h, could be written:

```
{ face = Pi * r * r;
  body = 2 * Pi * r * h
in
  2 * face + body }
```

Note that it is not always possible to statically determine the order in which statements in the "declaration" section of a block will execute. Consider the following example from [Traub 1989, page 2]:

```
{ p = x > 0;
  a = if p then bb else 3;
  b = if p then 4 else aa;
  aa = a + 5;
  bb = b + 6;
  c = a + b
```

```
in
  c};
```

If $x > 0$, the only possible order of evaluation is: p, b, bb, a, aa, c. If $x < 0$, the expressions must be evaluated in a different order: p, a, aa, b, c. This provides an example of an Id fragment in which the order of execution of statements cannot be determined at compile-time. This provides a theoretical limit on compile-time scheduling, beyond any practical limits based on insufficiently sophisticated compilers, because no compile-time scheduling exists.

Loops

The format of a loop statement is:

```
{for x <- eIndex do
  <statement> ;
        .
        .
  <statement>
finally e}
```

The keyword *next* is provided to refer to the next value of a loop iteration. For example, a loop to add the first n integers would be written:

```
{ sum = 0
in
  { for count <- 1 to n do
      next sum = sum + count
    finally sum }}
```

The semantics of Id are such that it is possible for multiple iterations of a loop to execute in parallel. Iannucci's compiler for the hybrid architecture has loops execute in "parallel" on a single processor, i.e. statements in the ith iteration may execute before statements in the jth iteration, as long as data dependences are respected. Inner loops are put in separate codeblocks and can be spawned to separate processors.

Many years have been spent developing and optimizing an Id compiler for the Tagged-Token Dataflow Architecture [Traub 1986a], a paper dataflow architecture. This compiler was the base of Iannucci's and of my research.

1.1.2 Iannucci's Hybrid Architecture

Development of hybrid architectures is an active area of research. See [Gaudiot and Bic 1989] for a summary of recent research in the area. One of the best known hybrid architectures is the EM-4 being developed at the Electrotechnical Laboratory in Japan [Sakai et al 1989]. I chose to base my work on Iannucci's system because of the ease with which I could access his compiler, developed at MIT, as well as its quality.

Iannucci's extensions to the Id compiler make use of information available at compile-time to create *scheduling quanta* (SQs), sequences of code within which the order is specified at compile-time. Invocation of a codeblock or procedure takes place on a single processor and generally consists of many SQs.[4] When a procedure is invoked, the instructions in the first SQ are executed sequentially, suspending at the end of the SQ or if a fault occurs, signifying that needed data is not ready. The execution of other SQs results from explicit forks.[5] The length of scheduling quanta is limited by the level of the compiler's analysis and by the requirements of Id. Arguments, local variables, and all but the most ephemeral of temporaries are stored within a *frame* allocated when the codeblock is invoked. My implementation for the J-Machine includes all of these characteristics. Further details about Iannucci's implementation and architecture will be provided as needed throughout the document. Henceforth, when I write "the hybrid architecture," I mean to refer to Iannucci's architecture.

1.1.3 The J-Machine

The target of my system is the J-Machine, a massively-parallel MIMD computer based on the Message-Driven Processor (MDP). Each processor has 260K (4K on chip) of 32-bit-word memory augmented with 4-bit tags. Tag types include booleans, integers, symbols, and *cfutures*. Cfutures generate faults on most operations. The MDPs communicate with each other through a low-latency network by sending messages. When a message arrives at a processor,

[4] To be exact, it is not always true that a procedure invocation executes on a single processor. More precisely, a *codeblock* invocation executes on a single processor. A procedure is usually one codeblock, but there are exceptions. When interior procedures are lambda-lifted out of a procedure definition, they constitute separate codeblocks, as do inner loops, so that they can be spawned among processors. Occasionally in the document, I provide simplified explanations whose exact details are fleshed out later.

[5] Throughout this document, I use "fork" to mean enabling a continuation on the current processor and "spawn" for enabling a continuation on another processor.

it is written into the message queue. When the message gets to the head of the queue, its first word is loaded into the instruction pointer, and a pointer to the base of the message is loaded into an address register so that subsequent words may be accessed. Execution continues sequentially until an explicit suspend instruction. The first J-Machine is expected to be built within a year and will have thousands of processors. For my research, I used a simulator of a 32-node J-Machine [Horwat and Totty 1987]. See [Dally et al 1988b] for a more complete description of the Message-Driven Processor.

1.2 Overview

In Chapter 2, I provide an overview of how the code is executed on the J-Machine, describing the run-time structures and control structure transformations. Chapter 3 describes my compiler and how it fits on top of the Id-to-hybrid compiler, as well as showing the code production templates. Chapter 4 provides benchmarks, including an extended example of the transformation and execution of a simple factorial program. Chapter 5 is the conclusion, presenting my retrospective opinions on the project and describing ways in which it could be improved. The appendices include program examples and source code.

Chapter 2

Executing Hybrid Code on the J-Machine

The villainy you teach me I will execute,
and it shall go hard,
but I will better the instruction.
— William Shakespeare, *The Merchant of Venice*, Act III, scene i, line 76.

Because Id is designed for dataflow processors — its name stands for Irvine Dataflow — its run-time demands are different from those of traditional imperative languages designed for von Neumann processors. On dataflow architectures, such as the Tagged-Token Dataflow Architecture and Monsoon, instructions are scheduled individually as soon as the data dependences have been satisfied. It would not be reasonable to attempt to imitate this on a non-dataflow architecture: When I hand-compiled Id programs onto the J-Machine with such a strategy, overhead was extremely high. For a typical dataflow instruction, such as plus, with two sources and two sinks, 20 MDP instructions were executed [Spertus 1989].

One of the major goals of compiling any language is to do as much work as possible at compile-time, leaving a minimum of work for run-time. Thus before running dataflow code on a von Neumann processor, the compiler should sequentialize sequences of instructions as much as possible. In [Traub 1988], a method of sequentializing regions of code into threads, or scheduling quanta (SQs), is presented. This lessens the amount of run-time overhead

considerably; however, it does not reduce it to zero. Because it cannot be determined statically what order the SQs must run in — if it were known, the SQs would already have been combined — some run-time scheduling is necessary. Specifically, SQs are explicitly forked as soon as the necessary data might be present. They may begin executing any time thereafter. Within a SQ, checks are performed to see if necessary data is present. If it is not, the SQ suspends, to try again once the data is received. Run-time support is necessary for these operations. In this chapter, I describe the run-time behavior of the programs at a detailed but relatively high level. I go into lower level detail in the following chapters.

2.1 Overview

Program execution on the J-Machine is based on the same ideas as on the hybrid architecture: Instructions are grouped into scheduling quanta subject to the following constraints:

1. The program yields the same results as pure dataflow computation.

2. No deadlocks are introduced.

3. An instruction with unbounded latency must not be within a SQ.

Because I work with the scheduling quanta produced by Iannucci's compiler, I inherit the assurance that the partitioning yields correct and terminating results [Iannucci 1988, Chapter 4].[1] As Iannucci did, I divide all unbounded-latency tasks into multiple phases so that other tasks can execute between initiation and fulfillment of a request.

When a codeblock is invoked, a contiguous region of memory called a *frame* is allocated for its arguments and scratch variables. The frame is given a unique global name. Because each invocation has its own data area, the same procedure can execute multiple times on one processor, with execution of the invocations interleaved. After a codeblock starts executing, it will probably fault on a slot in its frame — i.e. it will look for a value in a specific slot of the frame, but the data will not be present. In this case, a *continuation* is created encoding the code address and is stored into the offending slot. When the data arrives, the data will be

[1] It is not entirely true that I use the SQ divisions unchanged. As will be discussed in the next chapter, there are a few cases in which I tweak SQs.

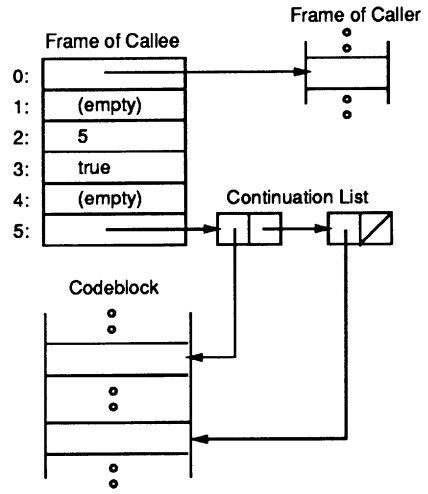


Figure 2-1: Run-Time Data Structures. Slots 1 and 4 of the callee's frame are empty, signifying that the corresponding data values have not arrived yet and have not been requested. The data for slots 0, 2, and 3 have arrived. Slot 0 points to the caller's frame so that the return value can be sent there. The data for slot 5 has not arrived. The presence of a continuation list indicates that instructions in the codeblock have tried to access slot 5. When the data arrives, the SQs indicated in the codeblock will be restarted.

written into the frame slot and the continuation will be re-enabled. When all of the SQs in a codeblock have successfully completed and any return values have been sent to the caller, the frame can be freed. These structures are shown in Figure 2-1. The following sections describe them in more detail.

2.2 Data Structures

2.2.1 Codeblocks

A codeblock consists of one or more scheduling quanta stored contiguously on each processor on which the procedure might be invoked. Unlike [Horwat 1989], code is distributed at load-time. The format of a pointer to a codeblock is shown in Figure 2-2. A user-defined tag value, CB, is used to indicate a pointer to a codeblock.[2] The low sixteen bits of the descriptor hold

[2]In this context, "user-defined" means defined by my dataflow system, as opposed to the hardware-specified tag types on the MDP. The MDP has 9 pre-defined tag types and 4 user-defined types.

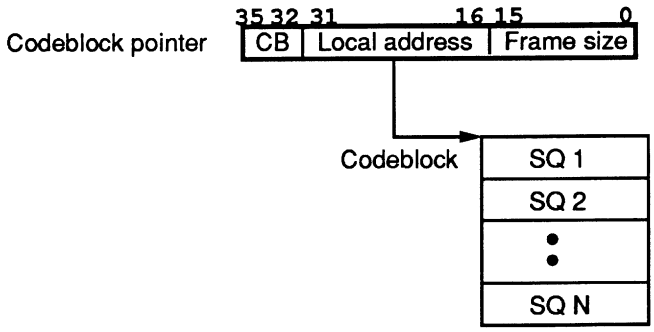


Figure 2-2: A Pointer to a Codeblock. The user-defined tag CB denotes a pointer to a codeblock. The low sixteen bits tell how large a frame must be allocated for the codeblock to execute. The high sixteen bits tell where the codeblock can be found.

the number of words of storage required for each invocation, and the high sixteen bits hold the address of the first SQ in the codeblock.

2.2.2 The Data Stack

Memory is allocated from a stack, initialized to null cfutures. A cfuture is a MDP data type on which most instructions fault. Thus, slots are pre-initialized to "empty". A heap would be a more efficient representation because memory could be freed and reused, but not enough time was available to implement one. The three run-time data structures allocated from the stack are frames, continuations, and I-structures, described in the following sections.

2.2.3 Frames

For a codeblock to execute, it needs a frame, a contiguous block of storage initialized to null cfutures (i.e. to empty). A pointer to the base of a frame is called a frame descriptor. Figure 2-3 shows a frame descriptor and a procedure frame. A user-defined tag value, FD, is used to indicate a pointer to a frame. The low sixteen bits of the descriptor hold the node number, and the high sixteen bits hold the local address, combining to provide a global address. Storing the node number in the low sixteen bits provides an efficiency bonus on the J-Machine as first described in [Horwat 1989, page 68].

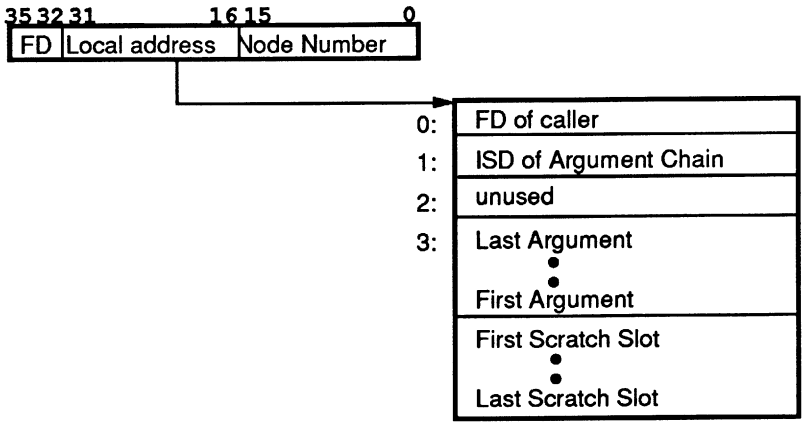


Figure 2-3: A Non-Loop Procedure Frame. A user-defined tag, FD, denotes a frame descriptor. It encodes the unique global address of a frame. The first slot of a frame holds a frame descriptor indicating where to send return values. The next slot holds the address of the I-structure chain of arguments. In some cases, the arguments can be passed directly in argument slots. The remaining slots are used for scratch values during the procedure's execution.

Slot 0 of the frame holds a frame descriptor telling where to send any return values. Some subtleties are involved in whether the arguments are passed in argument slots or as an I-structure chain. I retain Iannucci's conventions, and the interested reader is referred to [Iannucci 1988, pages 111-113]. The additional slots present in codeblocks with loops will be discussed in Section 2.3.3. Except for how I handle loops, my frames are identical to those used by Iannucci. The base of the frame currently executing is always kept in MDP address register A2. Taking all frame accesses relative to A2 allows multiple invocations of a procedure to run on the same processor.

2.2.4 Continuations

When an attempt is made to read an empty frame slot (i.e. a cfuture), a fault occurs whose handler does the following:

1. Stores a request to restart the SQ when the data arrives.

2. Suspends, in order to let another SQ execute.

In producing code, I ensure that at the time of a cfuture fault, the MDP register R0 holds a message indicating where execution should restart. I also take advantage of the MDP's always storing the absolute address of the last memory access in the MAR register. This allows the fault handler to determine which piece of data was missing. The handler allocates a triple (i.e. three words) from the stack and sets them to the following:

1. A message indicating where execution should restart (taken from R0).
2. The base of the current frame (taken from A2).
3. A pointer to the next continuation (if any) waiting on the faulted location. This is the old value of the slot.

The address of the triple is tagged as a cfuture and is written into the data location for which the fault occurred.[3] When the data arrives, the slot is checked just before the data is written. For every continuation present, the indicated message is sent and the continuation freed.[4] Because codeblocks execute within one processor, the message is sent from the processor to itself. J-Machine routing is done in such a manner that this is a cheap operation. Allocating and filling a continuation after a fault takes 18 cycles. Writing to a frame slot takes 7 cycles if no continuations are waiting and $8 + 6 * w$, if w continuations are waiting.

An Alternate Method for Continuations

I considered an alternate method of keeping track of suspended continuations. Instead of storing the continuation in a tuple allocated from the stack, the system could immediately send the message indicating where execution should restart, effectively putting it at the end of the local message queue. When the message reaches the head of the queue, it is tried again. If the data has arrived, it executes successfully (or at least until the next fault); otherwise, it will throw itself on the queue again.

This method has several advantages:

1. It seems to fit more elegantly on top of the J-Machine, taking advantage of the message queue provided.

[3] To be precise, a quadruple is sometimes needed instead of a triple, as will be explained in Section 2.3.3.
[4] Due to the primitive memory management of my system, the locations are freed in concept only.

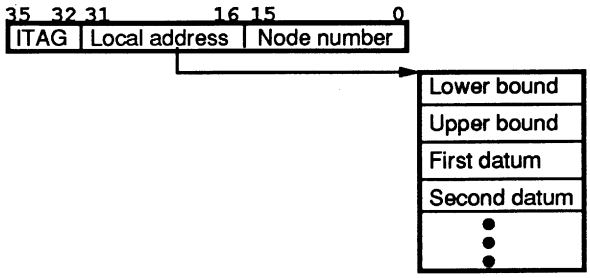


Figure 2-4: An I-Structure Descriptor and Storage. An I-structure descriptor includes its type and a global address that points to a block of storage, holding the bounds and the data.

2. Message suspension executes more quickly.

3. There is no need to check a frame location before writing a value to it.

The disadvantages, however, are major: A SQ could restart and fail many times, using an unbounded number of machine cycles. Additionally, the MDP message queue could overflow. For these reasons, I decided not to use this method.

2.2.5 I-Structures

I-structures are defined in Section 1.1.1. To review, they are array-like data structures whose entries can be written once. Reads before writes are silently deferred. (This shows one of the reasons high latency toleration is necessary.) I-structures are allocated explicitly by the user and implicitly for argument chains for procedure calls. Due to time constraints, I-structures are not handled by my compiler; however, I did develop and test the translation methods that would be used.

Figure 2-4 shows how I-structure descriptors and storage are implemented. I-structure descriptors are built analogously to frame descriptors, using the user-defined tag name ITAG. The low and high bounds of the I-structure are stored at the base of the region of storage, after which the data appear sequentially.

For a given cell of I-structure storage, there are three possible states, corresponding to the non-error states in Figure 1-1. The possibilities, and how they are indicated, are:

1. Empty, indicated by a null cfuture.

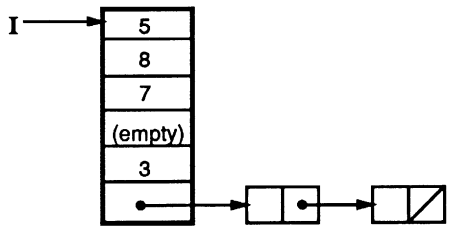


Figure 2-5: An I-Structure. The lower and upper bounds of this I-structure are 5 and 8, respectively. When a read or write request arrives, a run-time error occurs if the passed-in offset is out of bounds. If not, the lower bound is subtracted from the passed-in offset, and the corresponding cell is examined. In this example, data has been written to I[5] and I[7], there have been no attempts to read or write I[6], and there have been two reads to I[8] that will be satisfied when the data arrives. Writing to a slot more than once is a run-time error.

2. Waiting for data, indicated by a cfuture whose value points to a local linked list of continuations needing the data.

3. Full, indicated by a non-future (i.e. the data itself).

The continuations are of the same form as described in Section 2.2.4. An example of an I-structure is shown in Figure 2-5.

Writing an element of an I-structure takes $20 + 6 * r$ instructions, where r is the number of pending requests. The read handler takes 19 instructions if the data is present and 30 if it is not. These times include comparing against the bounds, subtracting off the lower bound, ensuring that no more than one write is done, and allocating any memory needed for continuations.

2.3 Control Structure

2.3.1 Execution Within a Codeblock

To see how execution proceeds within a codeblock, let us review the example block from Section 1.1.1. It is reproduced in Figure 2-6. Consider the possible orders of evaluation:

- If $x > 0$, $b \rightarrow bb \rightarrow a \rightarrow aa \rightarrow c$.
- If $x \leq 0$, $a \rightarrow aa \rightarrow b \rightarrow bb \rightarrow c$.

```
def abc x =
{ p = x > 0;
  a = if p then bb else 3;
  b = if p then 4 else aa;
  aa = a + 5;
  bb = b + 6;
  c = a + b;
in
  c};
```

Figure 2-6: A Statically Unschedulable Codeblock. It is impossible to determine the order in which a, b, aa, and bb must be computed without knowing whether $x > 0$.

Observe that in both cases, b precedes bb, a precedes aa, p is the first calculation, and c is the last. Using these static dependences, we partition the code into three scheduling quanta, as shown in Figure 2-7.[5]

Let us consider the case where $x > 0$. P is the first SQ to execute. As shown in Figure 2-8, it computes p then forks A, B, and C, in that order, and suspends. A begins, then suspends, because bb is needed but not available. B, next in the queue, begins and executes to completion. When it stores bb, it sees that A is waiting on the value and sends a message to restart A. C then begins executing and faults on a, suspending. The second attempt to execute A is now at the head of the message queue and completes, sending a request to restart C. C executes, performing the addition and whatever else follows (such as returning the resulting value).

The astute reader will have noticed that the sample procedure could be reduced to

```
def abc x =
  if x > 0 then
     14
  else
     11;
```

Despite this possible compile-time reduction, the example is still relevant for two reasons: First, the early stages of the compiler are not sophisticated enough to perform the reduction; second, examples exist for which no such reduction is possible. For example, if in the original

[5]Throughout the text, partitions are simplified to provide a more intuitive understanding than would be gained by going into the exact details on how a SQ is produced.

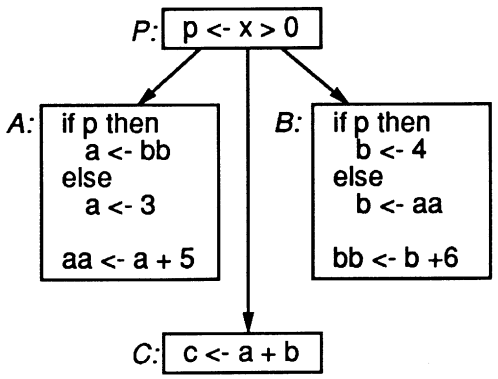


Figure 2-7: Scheduling Quanta for Unschedulable Example: The code in Figure 2-6 is divided into four scheduling quanta. The calculations for *b* and *bb* appear in the same quantum because *bb* depends only on *b*. It is impossible to determine statically whether SQ A or B executes first. Arrows indicate that one SQ forks another.

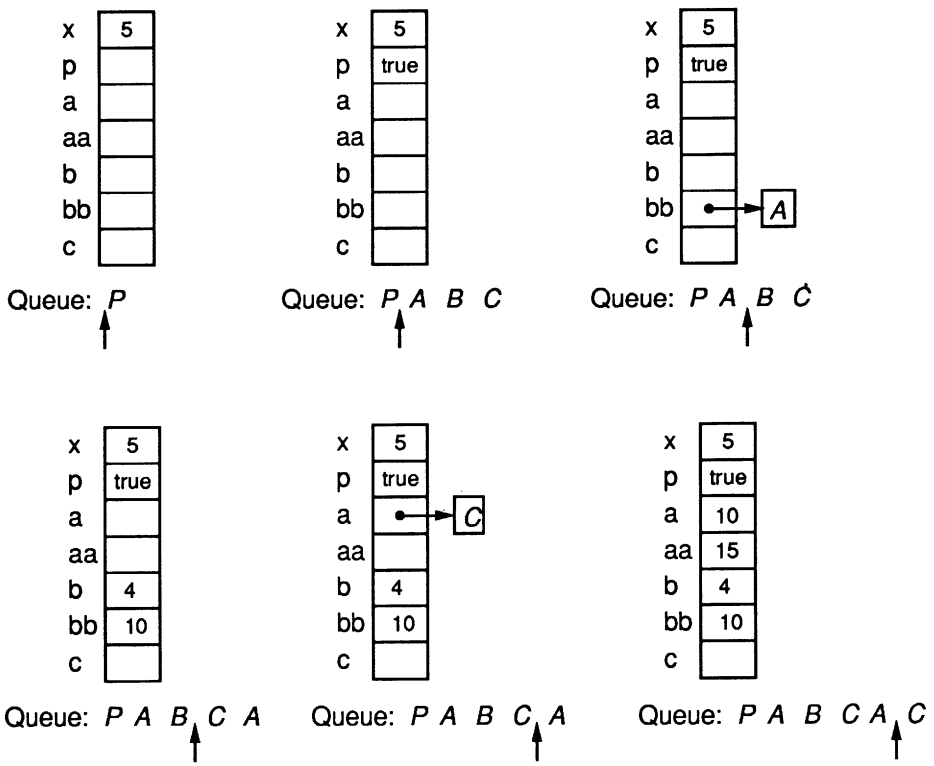


Figure 2-8: Snapshots for Codeblock Example. This shows snapshots of the message queue and frame before each SQ for the program in Figure 2-6.

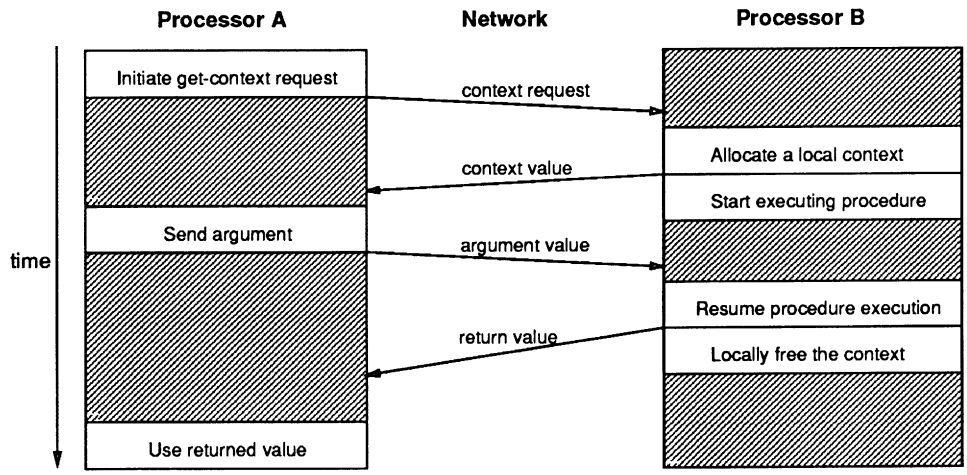


Figure 2-9: Procedure Linkage Example. Processor A requests a context on processor B. As soon as the frame is allocated, execution of the procedure call begins on B. When A receives the context value, it can send the argument(s), after which B can complete. Shaded rectangles indicate time that could be spent on other tasks. Note that those tasks are not interrupted when data arrives.

program (Figure 2-6), the bindings for a and b were changed to `a = f x bb` and `b = g x aa`, where f and g are passed in as parameters, no compile-time reductions would be possible [Traub 1989, page 2].

2.3.2 Procedure Calls

Figure 2-9 shows how procedure linkage is done without tying up either processor. When processor A wants to call a procedure on processor B, A must allocate a context (frame) on B for the codeblock's arguments and scratch area. Allocating a context has the side effect of starting execution of the first SQ in the procedure. After the address of the frame is returned to A, it sends the arguments to B, which will have faulted if the data was already needed. When the data arrives on B, suspended SQs are restarted. After B completes, it sends the return value (if any) and a signal to A, and it frees its frame. Note that other processes can execute while A and B are waiting for data.

While it would be more efficient in most cases for a caller to be able to send arguments at the same time as requesting the context, there was no clean way to do this. An interesting

effect of this policy is that (as in other Id implementations) a procedure can conceivably do substantial calculation or even return a value before receiving any arguments! This is necessary because procedure calls in Id are non-strict.

Currently, the system does not do any load-balancing, and it always spawns procedures to the same processor. The user must adjust the compiled code to provide a distribution appropriate to the problem.

2.3.3 Loops

As in all other implementations of Id, I provide a way for different iterations of a given loop to execute at a time. Because iterations of an outer loop execute on the same processor, they do not execute concurrently; instead, the SQs of up to K iterations of a loop are enabled at a time, where K is the loop-unfolding constant. When a calculation within one iteration is waiting for something, such as the result of a procedure call to another processor, instructions from other iterations may execute, subject to data dependences. Because up to K iterations may execute at once, there must be K places to store each intermediate value, so this method requires allocating K *iteration areas.* In [Iannucci 1988, Section 4.3.5], Iannucci presents and proves the correctness of a method for dynamically unfolding loops which guarantees the same results as sequential execution. I use his method, although I implement it differently.

Concepts

In Iannucci's method, an iteration includes the evaluation of the predicate and subsequent execution of either the loop body or the loop termination code. He observes that for iteration i to begin, three conditions must hold:

1. The predicate for iteration $(i-1)$ has been evaluated to "true".
2. The $(i-K)^{th}$ iteration has terminated, allowing us to reuse its iteration area.
3. The $(i+1-K)^{th}$ iteration must have already consumed its loop variables.

The third condition is the most subtle. It exists because iteration i will write the values of loop variables into the slots of iteration $i+1$. Hence, iteration i cannot execute until iteration $i+1-K$ is done with the values currently stored in these slots.

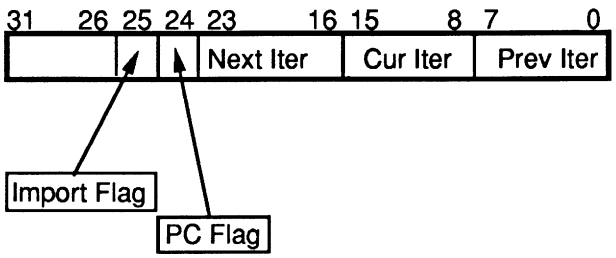


Figure 2-10: Possible Implementation of an Iteration Descriptor. The iteration fields hold the offsets from the frame base of the next, current, and previous iteration areas. The Import and PC flags tell whether this iteration may begin. Bits 26 through 31 are unused. This format was not used.

These rules are enforced with two flags, *PC* and *import*. Iteration i's PC flag is set when the first condition, that the predicate for iteration $i-1$ is true, has been established. The import flag is based on condition three; it is set when the next iteration area is ready to import new loop variables. In [Iannucci 1988, pages 129-131], Iannucci proves that the rules for the two flags cover all three conditions. When both of an iteration's flags are true, its first SQ (presumably to compute the predicate) may be enabled.

Implementation

Iannucci's hybrid architecture supports loops with several special-purpose instructions and hardware support. Specifically, *iteration descriptors*, containing the two flags and pointers to the previous, current, and next iteration areas, can be stored in one machine word. As Figure 2-10 shows, it was possible to store all these quantities into the MDP's shorter (32-bit) words, but, lacking hardware support for accessing these fields, shifting and masking were too slow. Additionally, in the small amount of space available for each iteration pointer, it was only possible to store offsets relative to the current frame, not absolute addresses, which would be more convenient. Hence, I decided not to mimic the hybrid architecture's implementation, and I developed my own data structures.

Figure 2-11 shows a frame for a procedure with a loop. In addition to the slots found in non-loop frames (see Figure 2-3), it has slots for the loop-unfolding constant, loop constants,

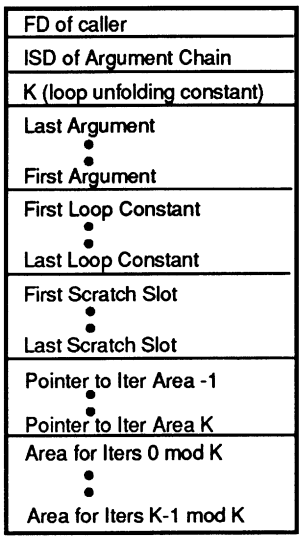


Figure 2-11: A Loop Procedure Frame. Loop procedure frames have several sets of slots in addition to those present in non-loop frames. Slot 2 holds K, the loop-unfolding constant. K specifies how many iterations may be unrolled. There is space for loop constants, values that could be hoisted out of the procedure's loop. Iteration areas are used for circulating variables and each iteration's temporaries. The pointers allow quick access to each iteration area.

iterations areas, and pointers to the iteration areas. Each iteration area's flags are stored within its pointer. The pointers to iteration areas can be viewed in a more conceptual way in Figure 2-12. In order to support iterations, an additional piece of data, an *iteration number* between 0 and $K - 1$ must be included in every continuation. When a loop SQ begins, the iteration number is used to find the pointer to the current iteration area. This pointer is stored in MDP address register A1. Slots relative to the current iteration area can then be indexed off A1. If it is necessary to access a slot in the previous or next iteration's area, the iteration number is decremented or incremented to find the appropriate pointer from the table of pointers within the frame. This is why there are $K + 2$ pointers to the K areas; i.e., if iteration 0 is active and wants to set the previous iteration's import flag, the pointer can be retrieved without providing a special check for the boundary condition. The import and PC flags are stored within the pointers.

As an example, consider the procedure in Figure 2-13 to sum the results of a function evaluated on the first n positive integers. The circulating loop variables are *count* and *total*.

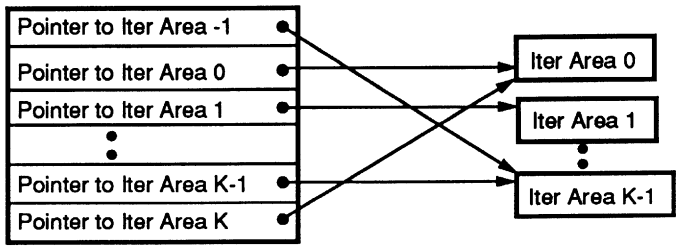


Figure 2-12: Iteration Areas and Pointers. Pointers to the iteration areas are stored contiguously from a known offset within the frame. Having $K + 2$ pointers to the K iteration areas is an optimization: If the current iteration number is 0 and the need arises to access the previous iteration area, the pointer can be found in a straightforward manner, i.e. by looking one slot earlier than the pointer to the current iteration area. This eliminates costly boundary condition checks. The PC and import flags, not shown, are packed into the high bits

```
def combine n f =
{ total = 0
in
  {  for count <- 1 to n do
       next total = (f count) + total
     finally total }}
```

Figure 2-13: Loop Program Example. Procedure `combine` applies function f to the first n positive integers, summing the results. For example, `(combine 10 square)` would return the sum of the squares of the numbers from 1 to 10.

1. Initialize the K iteration pointers.
2. Set the import flag of each iteration area.
3. Set *count* to 1 and *total* to 0 in iteration area zero and reset area $K-1$'s import flag to ensure that area zero gets to read *count* and *total* before they are written over.
4. Set area zero's PC flag, which will enable it, as the import flag is already set.
5. For each enabled iteration,
   (a) Compare *count* to n.
   (b) If *count* $\leq n$ then
      i. Write *count* $+1$ into the first slot of the next iteration area and set its PC flag.
      ii. Spawn *(f count)*.
      iii. Add the result of the previous step to *total*, writing the result to the *total* slot in the next iteration area.
      iv. Now done with all incoming circulating variables, set the previous iteration area's import flag.
   (c) If *count* $> n$ then write the current value of *total* to a frame slot outside the iteration areas.
6. Once the final result has been written to the outside frame slot designated for the *finally* value, pass it up to the caller.

Figure 2-14: Pseudo-Code Produced for Loop Example

Pseudo-code corresponding to the code that would be produced is shown in Figure 2-14. Figure 2-15 illustrates how this scheme reveals possible parallelism. Up to K invocations of f will execute at once. If f is slow, this is a big win.

The reader will observe that this scheme does not address nested loops. Those are pulled out of procedures at compile-time and form new codeblocks that will be called by the original procedure. Thus inner loops can execute in parallel on separate processors.

Because of a bug in the Id compiler's interaction with Iannucci's code, I was unable to have my compiler support loops. (The version of the Id compiler currently used is different from the one Iannucci wrote his system to interface with.) For my research, I hand-compiled loop procedures to explore the different methods of implementation.

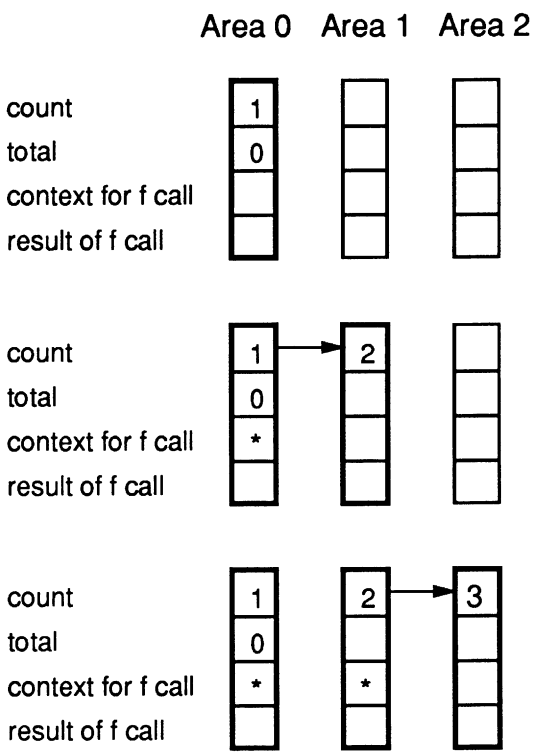


Figure 2-15: Snapshots for Loop Example. The snapshots show how the contents of the first three iteration areas for the program in Figure 2-13 change over time. The first snapshot shows the values in the iteration areas after they are initialized. The only non-empty locations are the initial values for *count* and *total* in iteration area 0, which has been enabled, as indicated by the darkened border. In the second snapshot, the first iteration has tested the predicate, written an incremented *count* into the next iteration area, and has made the function call. In the third snapshot, the second iteration does the same. Note that the function calls execute in parallel.

2.4 Conclusion

Conventions were found to allow Id code to run on the J-Machine in the same style used by Iannucci on the hybrid architecture. The benefits of this strategy are:

1. Frames allow dynamic dataflow, i.e. every invocation has its own data area.
2. SQs reduce the amount of necessary run-time scheduling.
3. Using multiple phases for instructions with unbounded latency frees the processor for useful work.
4. Loop unrolling exposes and exploits parallelism.

These powerful techniques are supported at run-time by special data structures, fault handlers, and library routines. The next chapter describes the compile-time work necessary to convert from hybrid format to MDP format.

Chapter 3

Compilation

I have heard of your paintings too, well enough;
God has given you one face,
and you make yourselves another.
You jig, you amble, you lisp...
— William Shakespeare, *Hamlet*, Act III, Scene i, line 150.

Because the MDP architecture is so different from the hybrid architecture, substantial work must be done to create MDP code from hybrid code. Keeping with the philosophy of the original ID compiler, described below, I perform my transformations in several stages. The intermediate forms my compiler recognizes or produces are:

- Hybrid code.
- Complex MDP code, machine instructions whose opcodes are the same as those on the MDP (with a few extensions) but whose addressing modes, etc., are not legal.
- Simple MDP code, s-expressions of legal MDP instructions.
- MDP assembly code.

My back-end converts from the first form to the last. The rest of the chapter describes this process.

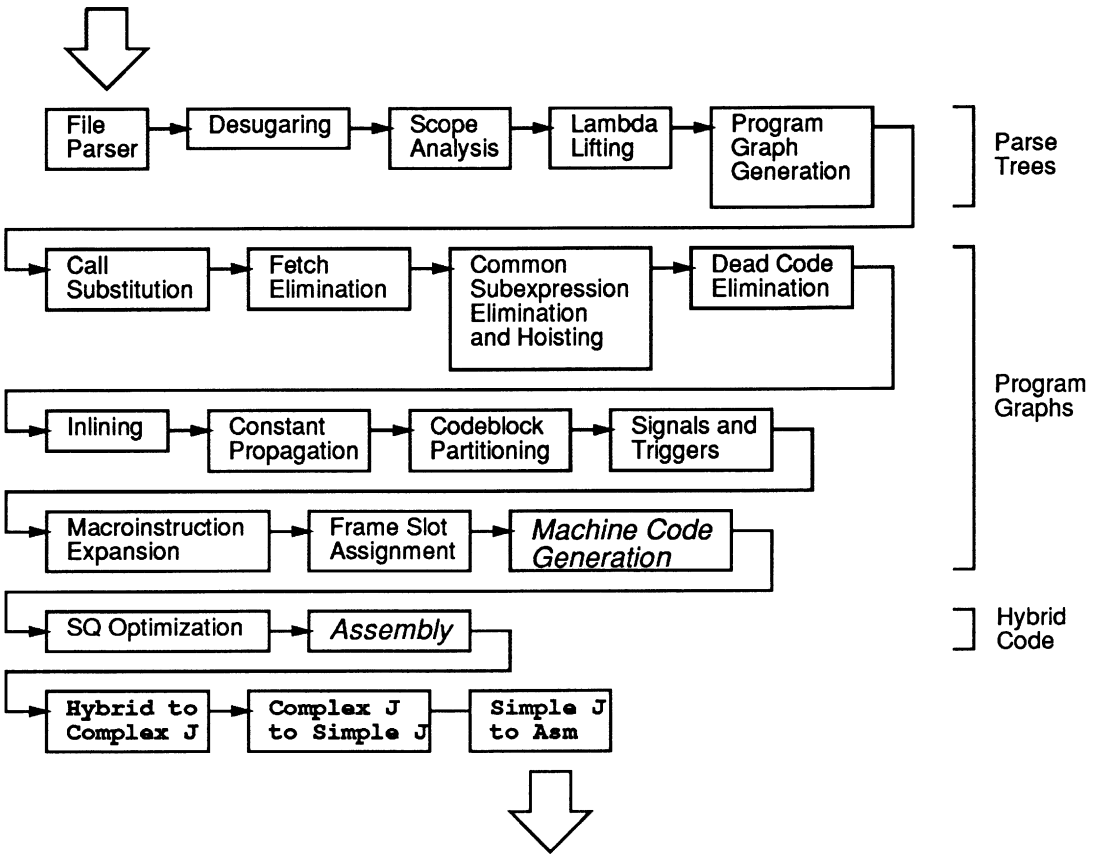


Figure 3-1: Structure of the Id-to-MDP Compiler: Plain roman text indicates modules of the original Id-to-hybrid compiler, italics indicate modules I changed, and bold indicates modules I added. Program graphs are a form of dataflow graph. This picture is modeled after one in [Iannucci 1988, page 97].

The original Id compiler is written in Common Lisp and is based on the Dataflow Compiler Substrate [Traub 1986b], a set of abstractions for building modular compilers. Each module inputs and outputs a stream of Lisp objects (except for the first and last modules which only emit or collect, respectively). Figure 3-1 shows how my modules fit on top of the Id compiler. Figure 3-2 shows the formats of instructions flowing through all of the new or changed stages. They will be explained in more detail below. The appendices contain complete listings of the files I created.

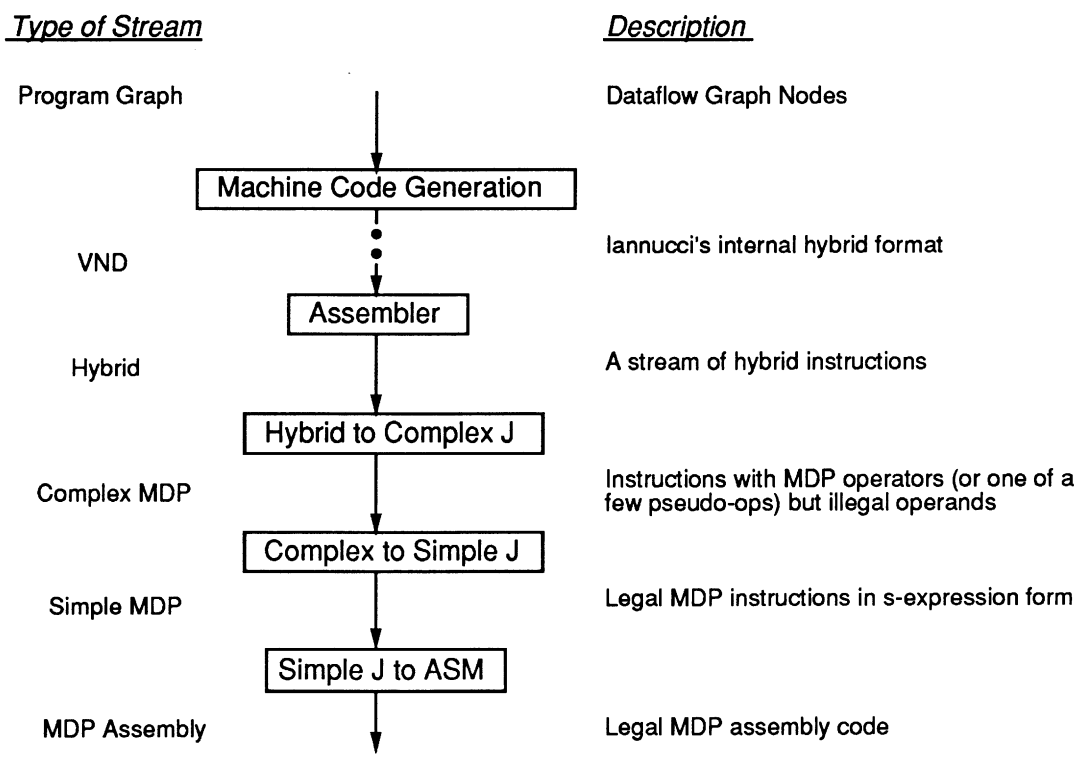


Figure 3-2: New and Modified Compiler Stages: Dataflow code flows through several stages in order to become MDP assembly code. The term "VND" is used to distinguish Iannucci's internal representation of code from my "hybrid" format. The ellipses between the first two stages indicate that other stages go between them.

3.1 Changes to Machine Code Generation

The machine code generation module, called *generate-vnd-instructions* and written by Iannucci, takes program graph instructions and converts them to hybrid instructions. In some cases, such as for arithmetic instructions, the transformation is trivial. For conditionals, loops, and procedure calls, however, a single program graph instruction expands into many hybrid instructions. Because my control structure transformations for loops and procedure linkage differ from Iannucci's, I wrote a file *changes.lisp* that replaced his templates for loops and procedure calls with my own.

3.1.1 Loops

Originally, for the loop program graph instruction, instructions were generated to support the hybrid architecture's implementation of loops. Section 2.3.3 describes how my implementation differs. I emit different hybrid instructions for the loop set-up instruction to initialize the iteration area pointers. Code within loop SQs is passed through unchanged, to be converted in later stages of the compiler, as only structural changes are made in this module.

3.1.2 Procedure Calls

Section 2.3.2 described my multi-phase convention for procedure linkage, but it glossed over a few details. Specifically, my implementation differs from the hybrid one in an important way: On the hybrid architecture, the *get-context* instruction calls a *local* manager that selects a frame on another processor where the procedure can be spawned [Iannucci 1988, page 174]. This requires a processor to know memory usage on other processors. When designing the system for the J-Machine, I decided each processor should know as little as possible about the other processors, particularly because the J-Machine is massively parallel. One consequence was that I rejected this scheme. Instead, I changed the protocol so that *get-context* is a two-phase instruction, where the calling node, A, asks the called node, B, for a frame address. The complete calling protocol is:

1. Execute a *get-context* instruction on A. This sends a request to processor B to allocate a frame and start execution of the appropriate procedure, and to send the frame descriptor

F back to processor A.

2. Compute the return location for the procedure call (an offset into the current frame) and send it to B, attached to F. Because F is the frame descriptor, B will know where to put the return location.

3. Send each of the arguments to B, attached to F.

If *get-context* were merely local, no data faults would occur during the first three steps; hence, the value for the return location could be written into a register instead of a more permanent place like a frame slot. In my strategy, a fault will occur during step 2 because F is not locally available yet. Hence, I must insert a suspensive check for F before the second step. This way, it will be safe to store the return location into a register. There will be no danger that a fault will occur on F between the time the register is written and when the register is accessed to send its value to B. (The values in registers are not guaranteed between suspensions, and it would have been too difficult for me to change the hybrid compiler's frame allocation.)

Even this is not the whole story. Consider a doubly-recursive procedure like a naive implementation of Fibonacci. Figure 3-3 shows the code that would be produced by the J-Machine strategy just described. The problem with this code is that the second *get-context* request would not be made until after the first one returns. This introduces unnecessary dependences, as it implies that steps 5–8 in the figure cannot occur until steps 1–4 are finished. This was not a problem on the hybrid architecture, where it was known that steps 1–4 would not suspend. Because step 2 will suspend, steps 5–8 will be delayed unnecessarily. This is illustrated in Figure 3-4. The arrow indicates the short-cut that exists: The second request can be started immediately after the first. Hence, before the suspensive check, we add an instruction to fork a continuation corresponding to whatever follows the procedure call — essentially splitting the SQ.

3.2 Assembling Hybrid Code

The last stage of Iannucci's compiler is an assembler that converts his internal representation of hybrid code into one suitable for his interpreter. I modified this stage to produce a stream of hybrid instructions suitable for my stages.

1. Execute *get-context* for the first recursive call. The value for the frame *F1* will be returned at some unknown time.

2. Make a suspensive reference to *F1*, so that we can't get to the next step unless it has arrived.

3. Compute the return location for the first procedure call and send it to $B1$ attached to *F1*.

4. Send the arguments to *B1* attached to *F1*.

5. Execute *get-context* for the second recursive call. The value for the frame *F2* will be returned at some unknown time.

6. Make a suspensive reference to *F2*, so that we can't get to the next step unless it has arrived.

7. Compute the return location for the second procedure call and send it to *B2* attached to *F2*.

8. Send the arguments to *B2* attached to *F2*.

Figure 3-3: A Non-Optimal J-Machine Calling Convention. *B1* and *B2* represent the two processors on which the subprocedures are spawned. The code is non-optimal, because *F2* would not be requested until after *F1* had been received.

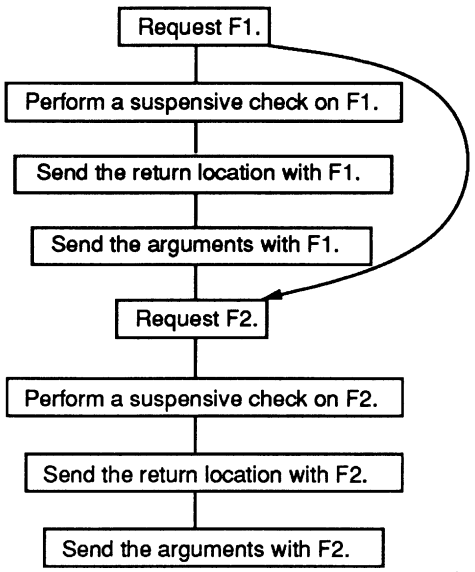


Figure 3-4: The Ordering Specified by Successive Function Calls. Unless the first instruction explicitly forks the second request, as show by the arrow, code will execute sequentially as indicated by the plain lines. This unnecessarily lessens the amount of exploited parallelism.

3.3 Convert Hybrid to Complex J

"Complex J" code is an intermediate format that is relatively easy to produce from hybrid code. The steps for converting an instruction are:

1. If any operand is suspensive,
   (a) Emit: `(suspensive-instruction)`
   (b) For every possibly-suspensive operand *s*, emit: `(suspensive-operand` *s*`)`
   (c) Emit: `(suspensive-check-done)`
2. Convert all references to hybrid general-purpose registers to references to temporary storage on the MDP.
3. Emit code specified by the template corresponding to the hybrid instruction.

Below, I describe the different templates for classes of hybrid instructions, in order to provide a deeper understanding of the hybrid instruction set as well as of the transformation process.

In this section, I go into considerable detail. Readers are prewarned, lest they fall off the bottom of this depth-first search. Casual readers may wish to read the first few templates and then skip to the conclusion of this section on page 39.

3.3.1 Label Instruction

The template for converting a *label* instruction is:

```
(defconversion label :label (label-name)
  '((label ,label-name)
    (move (:message (:base 1)) (:j-register A2))))
```

The first line generates a MDP label with the same name as the hybrid label. The second line says to move the value at offset one from the current message, i.e. the frame address, into MDP address register A2.[1] That line is there because execution can begin at any label, and A2 is always assumed to hold the base of frame pointer.

This example illustrates one of the differences between complex and simple MDP code: On the J-Machine, one of the operands of a *move* must be a general-purpose register. The above *move* will be broken into two moves in the next stage, *convert-cj-to-sj*. At this stage, we do not have to concern ourselves with such details.

3.3.2 Simple Arithmetic Instructions

The template for converting an arithmetic instruction such as *add* is:

```
(defconversion j-add :+ (s1 s2 d)
  (append (lookup-into d)
          '((add ,s1 ,s2 ,d))))
```

The *lookup-into* routine generates code to restart any continuations waiting for a value to be written to location *d*, as described in Section 2.2.4. First, the slot number is copied into R1, then the library routine *lookup-vector* is called.[2] Figure 3-5 shows the conversion of an addition instruction.

[1] In the hybrid and MDP assembly formats, **(move A B)** moves the contents of **A** into **B**, not vice versa.

[2] In retrospect, explicitly mentioning the register to pass the argument in at this stage is an unnecessary violation of abstraction.

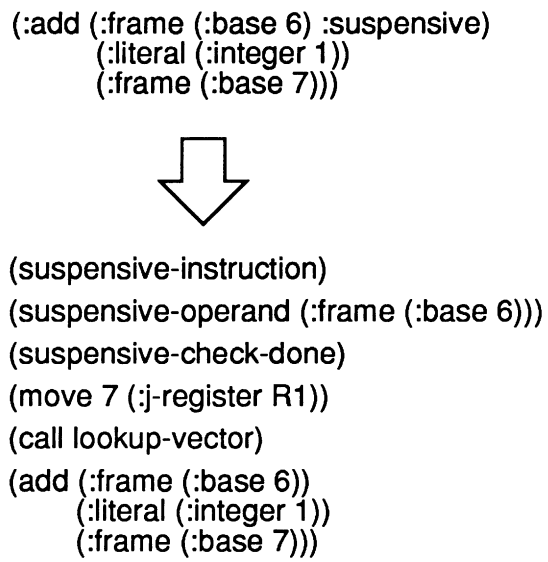


Figure 3-5: The Hybrid-to-Complex-J Conversion of an Addition. Execution will only get past the *suspensive-operand* virtual instruction if slot 6 of the current frame is present.

3.3.3 Complicated Arithmetic Instructions

Some arithmetic instructions are more complicated, such as *abs*, *min*, and *max*, because they are machine instructions on the hybrid architecture but not on the J-Machine. Thus they have larger templates that use temporary registers. Figure 3-6 shows the template for *abs*. The *reserve* and *free* pseudo-ops tell the next stage of the compiler where MDP registers should be allocated. Without this facility, the conversion of templates requiring temporary storage would be much less efficient. They will be discussed in more detail in the section on the next stage of the compiler.

3.3.4 Move Instructions

The template for converting a *move* instruction is:

```
(defconversion move :move (source dest)
  (append (lookup-into dest)
  '((move ,source ,dest))))
```

If the destination is a frame slot, this generates code to restart any continuations waiting on

```
(defconversion j-abs :abs (s d)
  (append (lookup-into d)
  '((reserve (:register scratch1))
    (reserve (:register scratch2))
    (ash ,s -31 (:register scratch1))
    (xor ,s (:register scratch1) (:register scratch2))
    (sub (:register scratch2) (:register scratch1) ,d)
    (free (:register scratch1))
    (free (:register scratch2)))))
```

Figure 3-6: The Template for Converting Absolute Value. Two scratch registers must be reserved for the optimal absolute value strategy. They are used for temporary values and are freed at the end of the template. The *reserve* and *free* are instructions to later stages of the compiler and do not directly produce any code.

the value and then performs the move.

The *move-remote* instruction moves a value into a slot of another frame. Its template is:

```
(defconversion movr :move-remote (frame-ptr offset value)
  '((send0 ,frame-ptr)          ; Node number
    (send0 (:ref local_movr))   ; MSG word
    (send0 ,frame-ptr)          ; First argument: frame descriptor
    (send0 ,offset)             ; Second argument: offset within frame
    (sende0 ,value)))           ; Third argument: value to write
```

On the J-Machine, the first word of a send sequence is a number specifying the destination node. The second word, the message header, specifies both how long the message is and the address of the handler to receive it. The meaning of subsequent words is determined by the handler.

To understand the above template, recall from Section 2.2.3 that the node number is stored in the low sixteen bits of the frame descriptor. Because the router only looks at the low sixteen bits, sending the frame descriptor specifies the correct destination node. When the message reaches that node, execution will begin at the *local_movr* library routine, which writes the passed value into the specified slot after checking if any continuations are waiting. The *move-remote* instruction is typically used for passing arguments and return values.

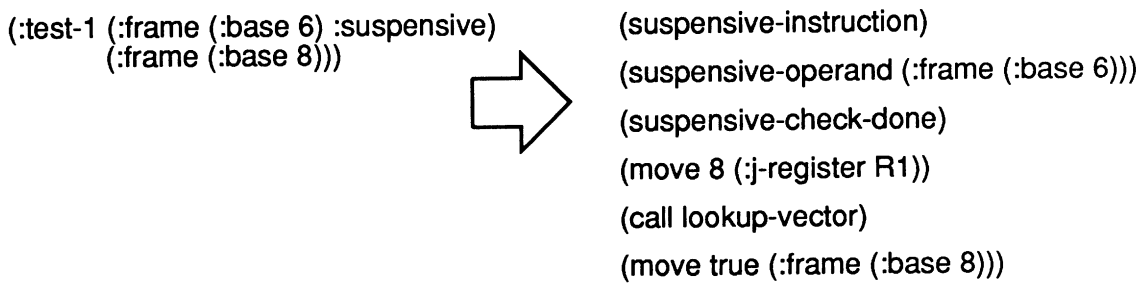


Figure 3-7: The Hybrid-to-Complex-J Conversion of a Test-1. Despite the template's apparently ignoring the source, the instruction is converted correctly. Before the template is even considered, code is emitted to check for the suspensive operand.

3.3.5 Test Instructions

The hybrid architecture includes the *test-1* and *test-2* instructions to write *true* into the destination if the source(s) are present. Execution should suspend if any source is unavailable.

The template for *test-1* is simply:

```
(defconversion tst1 :test-1 (s1 dest)
  (append (lookup-into dest)
  '((move (:tagged-literal ,boolean-tag 1) ,dest))))
```

The transformation for *test-2* is identical. The simplicity lies in how the converter handles suspensive arguments: Before the template stage is even reached, code will have been emitted to check suspensive operands and to suspend if they are not present. Figure 3-7 shows the conversion of a *test-1* instruction.

3.3.6 Continuation Instructions

Two hybrid instructions exist to fork continuations. They are used to start SQs within a codeblock. The template for the *continue* instruction is:

```
(defconversion cntn :continue (cont)
  '((send0 (:j-register NNR))
    ; Convert it from (:literal (:symbol :SQ-1)) to (:ref :SQ-1)
    (send0 (:ref ,(second (second cont))))
    (sende0 (:j-register A2))))
```

This sends a message from a processor to itself (the NNR register holds a processor's own

node number), along with the specified SQ base and the current frame pointer, kept in A2.

While the *continue* instruction is sufficient, it is non-optimal, in that the new continuation is likely to immediately suspend on the first value it checks for. With this observation, Iannucci designed the *continue-test* instruction which tests the first slot accessed by the new SQ. It the value is there, the continuation is forked as above; otherwise, a local continuation is immediately created and stored in the appropriate slot. This saves a message send in the worst — and most common — case. The conversion template is:

```
(defconversion cntt :continue-test (check-slot cont)
  ; Convert it from (:literal (:symbol :SQ-1)) to (:ref :SQ-1)
  `((move (:ref ,(second (second cont))) (:j-register R0))
    (move (:literal ,(frame-base-offset check-slot)) (:j-register R1))
    (call (:literal ,cntt-vector))))
```

This calls a local library routine, *cntt*, that does the check and, depending on whether or not the data is present, either sends the message or stores the continuation. The *cntt* routine expects R0 to hold the SQ address and R1 to hold the number of the needed slot.

3.3.7 Procedure Linkage Instructions

The procedure linkage convention was described in great detail in Sections 2.3.2 and 3.1.2. Briefly, there are three steps to spawning a procedure:

1. Initiate a *get-context* request, sending the codeblock descriptor and the address of where to write the new context pointer.

2. Use *index-current-context* to create a new global address for return values to be sent to. For example, if the first return value should be sent to slot 8, index the current context by 8.

3. Perform remote moves to transfer the indexed context and the arguments into the newly-allocated frame.

The third step uses the *move-remote* instruction described earlier. The transformations for *get-context* and *index-current-context* for the first two steps are described here.

Get-Context The transformation for the *get-context* instruction appears in Figure 3-8. Rather than try to explain it here, I have added detailed comments to the code. As mentioned earlier, no attempt at load balancing is made by the compiler. A library routine, *get-context*, resides on every processor to use the information sent and to perform the callee's half of the protocol.

Index-Current-Context The *Index-Current-Context* instruction is slightly more complicated. By convention, the n return values of a procedure are sent to the first n slots of the calling frame. Because we really never want the return values sent to the start of the current frame, we increment the current context and send that value to the callee instead. The template is shown in Figure 3-9.

3.3.8 Conclusion

In the *convert-hybrid-to-cj* stage of the compiler, hybrid instructions are transformed into complex J-Machine code. The transformations ignore the intricacies of MDP addressing modes, making the transformation process simpler and more conceptual. Several pseudo-operators for handling suspensive instructions and register allocation are used.

```
(defconversion getc :get-context (context-slot return-slot)
   ;; The first scratch register will be used to hold the global
   ;; frame descriptor of the calling frame, so that the callee
   ;; knows where to send the context value back to.  Recall that
   ;; the format of a FD is that the local address is in the high
   ;; sixteen bits, and the node number is in the low sixteen.
  `((reserve (:register scratch))
    ; Take the local address of the current frame from A2.
    (move (:j-register A2) (:register scratch))
    ; Tag it as an integer (instead of an address) so we can munge it.
    (wtag (:register scratch)
          (:literal ,int-tag)
          (:register scratch))
    ; Shift it over 16, to fit into FD format.
    (lsh (:register scratch)
         (:literal ,(- 16 *sys-len-bits*))
         (:register scratch))
    ; Add in the local node number (i.e. put it in low 16 bits).
    (add (:register scratch) (:j-register NNR) (:register scratch))
    ; Tag it as a FD.
    (wtag (:register scratch) (:literal ,fd-tag) (:register scratch))

    (send0 (:literal 1))          ; Send to node 1 always
    (send0 (:ref local_getc))     ; Handler is the local_getc lib routine
    (send0 ,context-slot)         ; Send the codeblock descriptor.
    (send0 (:register scratch)) ; Send the current FD, so it knows
    (free (:register scratch))  ;  where to send the context back to.
    (sende0 ,(frame-base-offset return-slot)))) ; Send the return offset.
```

Figure 3-8: Transformation the Get-Context Instruction to MDP Code. The purpose of the *get-context* instruction is to send off a request to allocate a context and return its value.

```
(defconversion ixcc :index-current-context (frame-base dest)
  (append (lookup-into dest)
  ; A scratch register is needed
  '((reserve (:register scratch))
    ; Move the local frame address into the scratch register
    (move (:j-register A2) (:register scratch))
    ; Tag it as an integer so we can adjust it
    (wtag (:register scratch)
          (:literal ,int-tag)
          (:register scratch))
    ; Add in the new base, shifted over into the address
    ; portion of the instruction
    (add (:register scratch)
         (:literal ,(* (literal-base-offset frame-base)
                       (expt 2 *sys-len-bits*)))
         (:register scratch))
    ; Shift the sum into the top half of the word
    (lsh (:register scratch)
         (:literal ,(- 16 *sys-len-bits*))
         (:register scratch))
    ; Add the local node number into the low half of the word
    (add (:register scratch)
         (:j-register NNR)
         (:register scratch))
    ; Tag it as a frame descriptor
    (wtag (:register scratch)
          (:literal ,fd-tag)
          (:register scratch))
    ; Move it into the specified destination.
    (move (:register scratch) ,dest)
    ; Free the scratch register.
    (free (:register scratch)))))
```

Figure 3-9: Transformation of Index-Current-Context. The purpose of *index-current-context* is to take the address of the current frame, conceptually add a constant offset to it, and convert it to file descriptor format. It can then be sent to a spawned procedure as the frame to return results to.

3.4 Convert Complex J to Simple J

This section is the most complex of the new modules. Its tasks include:

1. Converting literal operands into tagged literals.
2. Converting the *suspensive-instruction*, *suspensive-operand*, and *suspensive-check-done* pseudo-ops into MDP code.
3. Allocating and substituting MDP registers where they were requested with the *reserve* and *free* pseudo-ops.
4. Adjusting instructions to use legal MDP addressing modes.

We will examine each of these stages.

3.4.1 Converting Literals to Tagged Literals

Because all values on the MDP are tagged, references to literals must be changed to tagged literals. The integer literal operands from the addition example in Figure 3-5 would both be converted:

```
7 → (:tagged-literal int-tag 7)
(:literal (:integer 1)) → (:tagged-literal int-tag 1)
```

Booleans and labels are similarly transformed.

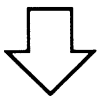
The other type of "literal" used is a reference — a constant whose value is determined at assemble-time [Horwat and Totty 1987, page 9]. References are used to denote codeblock pointer values, addresses of suspensive instructions, and branch destinations. These are denoted with the imaginary tag name, "special-tag". These operands are converted to MDP reference format in the last stage of the compiler.

3.4.2 Generating Suspensive Code

Before a suspensive instruction, several things must be done to ensure proper behavior:

1. Store the current instruction pointer location into R0, so if a fault occurs, the handler will know where execution should resume.

```
(suspensive-instruction)
(suspensive-operand (:frame (:base 6)))
(suspensive-check-done)
```



```
(label (:tagged-literal special-tag (:label suspensive19)))
(dc (:tagged-literal special-tag :suspensive19))
(move (:message (:base 1)) (:j-register A2))
(rtag (:frame (:base 6)) (:j-register R3))
```

Figure 3-10: Intermediate Code Produced for Suspensive Pseudo-Operands. The *DC* ("data constant") instruction loads its assemble-time constant operand into R0. If the *rtag* ("read tag") instruction faults, the handler can use the R0 value to know where execution should restart, as described in Section 2.2.4.

2. Because execution could be resumed here, SQ setup code must be emitted to load the base of frame address into MDP register A2, i.e. `(move (:message (:base 1)) (:j-register A2))`.

3. Check whether each suspensive operand is present, faulting if not.

For the rationale behind these rules, refer back to Section 2.2.4, where the continuation format was described. Figure 3-10 shows the conversion of the suspensive pseudo-ops in the *add* instruction introduced in Figure 3-5. First, a unique label, created with the Lisp procedure *gensym*, is emitted. A reference to it is loaded into R0 with the *DC* ("data constant") instruction. The frame base is loaded into A2, after which the tag of the suspensive operand is read. If it faults, the run-time handler described in Section 2.2.4 will set up a continuation.

Although it would be more efficient not to explicitly read the tags of the suspensive operands, it is necessary if the hybrid instruction has side effects. For example, a destination might be written or a message might be sent before a specific suspensive operand was accessed. A later version of this compiler would optimize out the "read tag" instructions in cases where the explicit check would suffice.

3.4.3 Allocating MDP Registers

MDP registers have two uses: passing arguments to system calls and holding temporary values within hybrid instructions. When used for system calls, they are explicitly referred to as in Figure 3-5 earlier. When they are used as temporaries, generally it does not matter which of the four MDP general-purpose registers is used. The *reserve* and *free* pseudo-ops generated by the templates in *convert-hybrid-to-cj* are used to create and destroy bindings of symbols to MDP registers. For example,

```
(reserve (:register scratch))
```

binds `scratch` to a free MDP register. Until a

```
(free (:register scratch))
```

is encountered, all occurrences of `(:register scratch)` are converted to `(:j-register R`n`)`, where n is the register bound to `scratch`. Because no more than four temporary registers are ever needed, no spilling needs to be done.

The only conflict arises because R0 is different from the other GPRs. The MDP instruction *DC* loads a 32-bit quantity into R0.[3] Except for a few special values, only 7-bit quantities can be specified as constants to move directly into the other registers. Thus there is an internal compiler routine, *request-appropriate-register* that takes an argument specifying what will go in the register and returns a binding to an appropriate register — i.e. R0 if the argument is a big value, another register otherwise. If R0 has already been allocated, an instruction to move the old contents of R0 into another register is generated, and the previous binding to R0 is changed. This process is illustrated in Figure 3-11.[4]

3.4.4 Converting to Legal MDP Operands

Instructions on the MDP are only 17 bits long. While this permits tight packing and quick loading, it limits the operand space. Specifically, general-purpose registers are required as

[3] *DC* is more accurately an assembler pseudo-op. It must have a constant value for its operand which is then put directly into the instruction stream. During execution, if the instruction pointer is at something that is not tagged instruction, it is loaded into R0. This allows 32-bit values to be directly loaded into a register, despite the normal 17-bit instruction length.

[4] Nate Osgood helped me develop this one-pass register allocation scheme.

Request	Bindings	Code Emitted
(request-appropriate-register 100) *reg91*	reg91 -> R0	
(request-appropriate-register 11) *reg92*	reg91 -> R0 reg92 -> R3	
(request-appropriate-register 500) *reg93*	reg91 -> R2 reg92 -> R3 reg93 -> R0	(move (:j-register R0) (:j-register R2))

Figure 3-11: Compiler Register Allocation. Requests for registers and the return values are shown in the leftmost column. The binding names are generated by the Lisp *gensym* procedure. The middle column shows the internal set of bindings after each instruction. A conflict arises on the third request where R0 is needed but is already part of another binding, *reg91*. The register allocator emits code to move whatever has been placed in R0 into a previously-free register, R2. The binding for *reg91* is then changed to R2, and the new request can get R0.

operands to certain instructions, and only very short constants can be encoded in instructions.

Consider the following hybrid instruction:

```
(:add (:frame (:base 6))
      (:literal (:integer 30))
      (:frame (:base 7)))
```

There are two reasons why it cannot be encoded into one MDP three-operand instruction:

1. The first and last operands must be general-purpose registers.

2. If the second operand is a constant, it must be in the range [15...–16].

The above *add* instruction would be translated into four MDP instructions:

```
(move (:frame (:base 6))
      (:j-register R3))
(move (:tagged-literal int 30)
      (:j-register R2))
(add (:j-register R3)
     (:j-register R2)
```

```
     (:j-register R3))
(move (:j-register R3)
      (:frame (:base 7)))
```

The astute reader will have observed that if the order of the source operands were changed, they could be encoded into one less MDP instruction. I did not have time to incorporate this optimization for commutative instructions.

As another example, consider a hybrid instruction to move an immediate into a frame slot:

```
(:move (:literal (:integer 500))
       (:frame (:base 20)))
```

Because 500 is more than seven bits long, it must be loaded into R0 through the *DC* instruction:

```
(dc (tagged-literal int 500))
(move (:j-register R0)
      (:frame (:base 20)))
```

Like immediates, offsets from the frame base can only be five bits in three-operand instructions and seven bits in two-operand instructions. If the destination of the above *move* had an offset of 100 instead of 20, the code would be:

```
; (:move (:literal (:integer 500)) (:frame (:base 100)))
(dc (tagged-literal int 500))
(move (:j-register R0)
      (:j-register R3))
(dc (tagged-literal int 100))
(move (:j-register R3)
      (:frame (:base (:j-register R0))))
```

This illustrates the R0 conflict maneuver described in Section 3.4.3.

3.5 Convert Simple to ASM

This last stage converts the code to a format suitable for the MDP assembler. This involves converting from s-expressions into plain text and translating the operands into a suitable format. Offsets are converted:

`(:register (:base X))` → `[X,A0]`

`(:frame (:base X))` → `[X,A2]`

`(:message (:base X))` → `[X,A3]`

The first transformation is to convert hybrid registers to temporary storage. On the J-Machine, accesses off of A0 are absolute addresses. The first twenty words of MDP memory are devoted to temporary storage, so hybrid register n is stored at absolute address n on the J-Machine. As on the hybrid architecture, the value is not guaranteed to be the same between suspensions.

Additionally, assemble-time references must be output properly. When a reference is encountered as an operand, it is converted:

`(:tagged-literal special-tag X)` → `{X_msg_ref}`

Additionally, X is added to a list of references. At the end of compilation, for each reference X in the list, the following is output:

```
ref X_msg_ref = MSG:(((X+N_loc)<<10))+2
```

where N is the name of the procedure. This creates a reference whose value includes the absolute address of the associated label (labels are usually relative addresses), as well as specifying that it will be used as a header of a message with two words. (The two words will be the message itself and the frame value.)

3.6 Conclusion

In order to convert hybrid code to MDP assembly code, I created intermediate formats and routines to convert from more complex formats to simpler ones. These are useful not only for this compiler but as a general-purpose J-Machine utility. A MDP assembly coder or compiler writer could produce complex J-Machine code and be spared the trouble of remembering how many bits of operand are available for each instruction. While my register allocation is still too primitive to give optimal results — for example, the same value could be stored in two different registers — it is still good enough to provide a new dialect of MDP assembly language that a programmer might choose for its greater abstraction and simplicity.

Chapter 4

Analysis

"A slow sort of country!" said the Queen. "Now, here, you see, it takes all the running you can do, to keep in the same place. If you want to go somewhere else, you must run at least twice as fast as that."

— Lewis Carroll, *Through the Looking-Glass.*

I am pleased with the system, in that it works and reasonable solutions were found for every problem. However, while some of the mechanisms worked out well, not all turned out to be as efficient as I would like. In this chapter, I provide a detailed example of code produced and executed for an Id routine, several benchmark results, and analysis of both my system and the J-Machine.

4.1 Detailed Benchmark: Factorial

In this section, I will go into great detail by providing listings and statistics for a sample Id procedure. Specifically, I will describe the composition and execution of the simple recursive factorial program shown in Figure 4-1.

4.1.1 The Dataflow Graph

First, the initial stages of the compiler convert the program into a dataflow graph, such as the one shown in Figure 4-2. I have abstracted away some of the details in order to highlight the essential parts of the graph. First, the input arrives at node 1. It is passed through unchanged

```
fact n =
  if n <= 1 then
    n
  else
    n * fact (n-1);
```

Figure 4-1: Id Code for Factorial

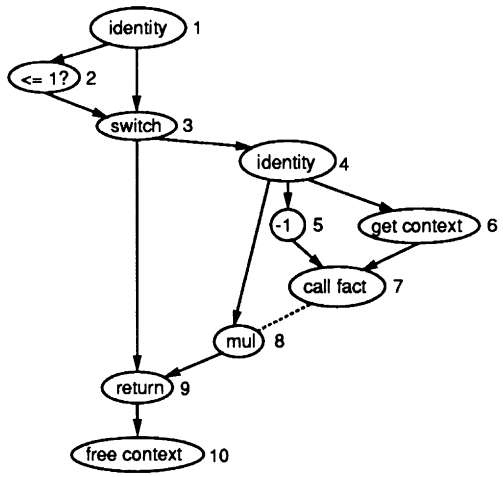


Figure 4-2: A Dataflow Graph for Factorial. If an integer n is input to the top *identity* node, $n!$ will be computed. The *switch* node uses its left input as a control signal and its right input as data. If the control signal is true, data goes to the left output arc; otherwise, to the right. The *identity* node copies its inputs to its output arcs. The dotted line from the *call* node to the *mul* node indicates that the connection is indirect. The numbers are for expository purpose only.

by the *identity* instruction to nodes 2 and 3. Node 2, the predicate, passes a boolean value to node 3, a *switch* instruction. The semantics of the *switch* instruction are such that it passes its data input to the left output arc if the control input is true, and to the right arc if the control input is false. Thus, if the predicate is true — i.e. if the argument is less than or equal to one — the argument itself will be sent to node 9 and returned. In the inductive case, the argument is sent to *identity* node 4. Node 7, *call fact*, makes the recursive call, specifying that the return value should be sent to node 8, *mul*. When it arrives, the multiplication is performed, and a value is sent to node 9 to be returned.

The purpose of showing and describing the graph is to give an idea of how the compiler looks at a procedure. Iannucci's stages of the compiler can only see the dataflow graph, not the source code.

4.1.2 The Hybrid Code

The hybrid code produced by the factorial example is shown in Figure 4-3. I have added comments, lines headed with semicolons, to describe the process. Readers uninterested in such technical detail should skip to Figure 4-4 which shows the SQs' composition at a higher level. Figure 4-5 shows frame usage.

```
;;; SQ-1 does initialization, forks local SQs, and tries to return
;;; the result.
((:LABEL (:LITERAL (:SYMBOL :SQ-1))))
;; Put the codeblock pointer to FACT into [12].
((:MOVE (:LITERAL (:CODE-BLOCK :FACT)) (:FRAME (:BASE 12))))
;; Fork SQ 2, immediately suspending it if [3], n, is empty.
((:CONTINUE-TEST (:FRAME (:BASE 3) :SUSPENSIVE) (:LITERAL (:SYMBOL :SQ-2))))
;; Fork SQ 11, immediately suspending it if [0], the return location,
;; is empty.
((:CONTINUE-TEST (:FRAME (:BASE 0) :SUSPENSIVE) (:LITERAL (:SYMBOL :SQ-11))))

((:LABEL (:LITERAL (:SYMBOL :SEND-RESULT-0))))
;; Pass [6], the return value, up to offset 1 from the calling frame.
((:MOVE-REMOTE (:FRAME (:BASE 0) :SUSPENSIVE)
               (:LITERAL (:INTEGER 1))
               (:FRAME (:BASE 6) :SUSPENSIVE)))
;; Pass [5], a signal ("true"), up to offset 0 from the calling frame.
((:MOVE-REMOTE (:FRAME (:BASE 0))
               (:LITERAL (:INTEGER 0))
               (:FRAME (:BASE 5) :SUSPENSIVE)))
((:TERMINATE))

;;; SQ-11 sets [5], the signal, when locations [0] and [7] have data.
((:LABEL (:LITERAL (:SYMBOL :SQ-11))))
((:TEST-2 (:FRAME (:BASE 0) :SUSPENSIVE)
          (:FRAME (:BASE 7) :SUSPENSIVE)
          (:FRAME (:BASE 5))))
((:TERMINATE))

;;; SQ-2 evaluates the predicate and runs appropriate code.
((:LABEL (:LITERAL (:SYMBOL :SQ-2))))
;; Put in [4] the result of checking if [3], the argument, is <= 1.
((:<= (:FRAME (:BASE 3) :SUSPENSIVE)
      (:LITERAL (:INTEGER 1))
      (:FRAME (:BASE 4))))
;; If not, branch to ELSE-4.
((:BRANCH-FALSE (:FRAME (:BASE 4)) (:LITERAL (:SYMBOL :ELSE-4))))
;; Copy [3], the argument, into [6], the slot for the result.
((:MOVE-IDENTITY (:FRAME (:BASE 3) :SUSPENSIVE) (:FRAME (:BASE 6))))
;; Copy [4], the predicate result ("true") into [7].
((:MOVE-IDENTITY (:FRAME (:BASE 4)) (:FRAME (:BASE 7))))
;; Branch past inductive case code.
((:BRANCH (:LITERAL (:SYMBOL :END-IF-4))))
```

;; Continued on next page.

;; Continuing from previous page.

```
;; This code gets executed for the inductive case.
((:LABEL (:LITERAL (:SYMBOL :ELSE-4))))
;; Subtract 1 from [3], the argument, and put the result in [14].
((:- (:FRAME (:BASE 3) :SUSPENSIVE)
     (:LITERAL (:INTEGER 1))
     (:FRAME (:BASE 14))))
;; Spawn the codeblock whose name is in [12] (fact), putting the context
;; value into [10].
((:GET-CONTEXT (:FRAME (:BASE 12)) (:FRAME (:BASE 10))))
;; Specify [8] as the base location for return values for the
;; spawned procedure.
((:INDEX-CURRENT-CONTEXT (:LITERAL (:BASE 8)) (:REGISTER 0)))
;; Send this adjusted context (i.e. the return location) to slot zero
;; in the spawned procedure.
((:MOVE-REMOTE (:FRAME (:BASE 10))
               (:LITERAL (:INTEGER 0))
               (:REGISTER 0)))
;; Send [10], the argument minus 1, to slot three in the spawned procedure.
((:MOVE-REMOTE (:FRAME (:BASE 10))
               (:LITERAL (:INTEGER 3))
               (:FRAME (:BASE 14))))
;; Fork SQ-5, immediately suspending it if [3], the argument, isn't here.
((:CONTINUE-TEST (:FRAME (:BASE 3) :SUSPENSIVE) (:LITERAL (:SYMBOL :SQ-5))))
;; Fork SQ-8, immediately suspending it if [8], the signal that the spawned
;; procedure is done, isn't here.
((:CONTINUE-TEST (:FRAME (:BASE 8) :SUSPENSIVE) (:LITERAL (:SYMBOL :SQ-8))))
((:LABEL (:LITERAL (:SYMBOL :END-IF-4))))
((:TERMINATE))

;; SQ-8 frees the context of the spawned procedure if it's not needed
;; any more.
((:LABEL (:LITERAL (:SYMBOL :SQ-8))))
;; Suspend if [8], the signal that the spawned procedure is done, is present.
((:TEST-1 (:FRAME (:BASE 8) :SUSPENSIVE) (:REGISTER 0)))
;; Return [10], the context of the spawned procedure, writing true into [11].
((:RETURN-CONTEXT (:FRAME (:BASE 10) :SUSPENSIVE) (:FRAME (:BASE 11))))
;; Copy [11], the true signal, into [7], a signal that all work is done.
((:MOVE-IDENTITY (:FRAME (:BASE 11)) (:FRAME (:BASE 7))))
((:TERMINATE))

;; SQ-5 is spawned only for the inductive case.
((:LABEL (:LITERAL (:SYMBOL :SQ-5))))
;; Multiply [3], the argument, by [9], the value returned by the recursive
;; call, putting the result into [13].
((:* (:FRAME (:BASE 3) :SUSPENSIVE) (:FRAME (:BASE 9) :SUSPENSIVE)
     (:FRAME (:BASE 13))))
;; Move this value into [6], the slot for the return value.
((:MOVE-IDENTITY (:FRAME (:BASE 13)) (:FRAME (:BASE 6))))
((:TERMINATE))
```

Figure 4-3: Hybrid Code for Factorial.

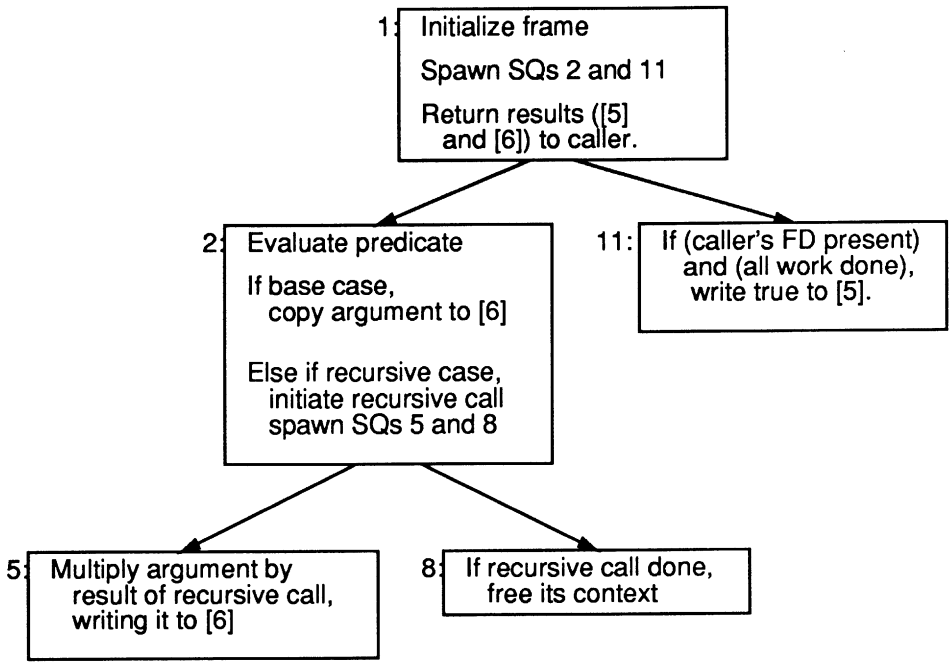


Figure 4-4: There are five scheduling quanta in *factorial*. The numbers in the SQ names have no significance, except that the first is always named SQ-1. Arrows indicate where SQs may be forked and can be thought of as a subset of data dependences. SQs 5 and 11 are only spawned in the recursive case. Observe that on its first execution, SQ-1 will fault midway through, because the return results will not be ready. Execution will restart in the middle of the SQ.

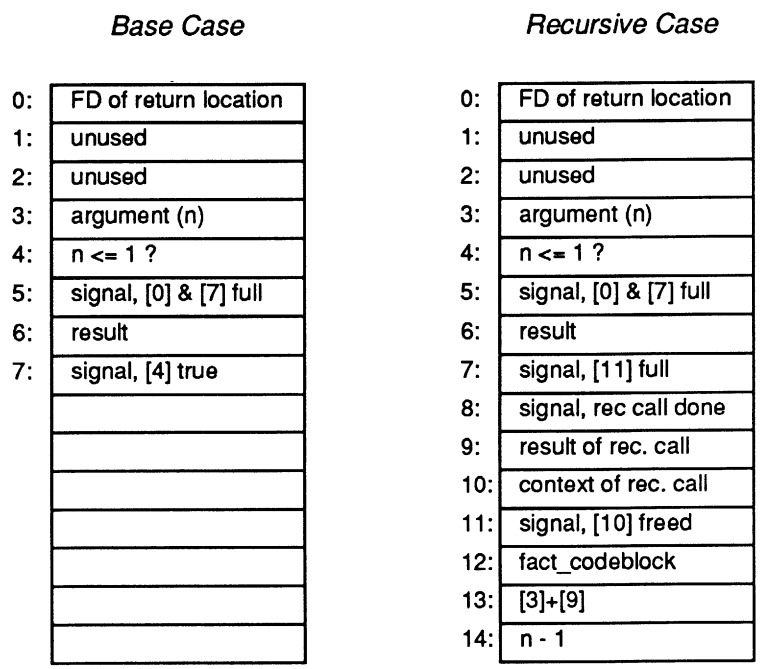


Figure 4-5: Frame Slots Used by Factorial Code. The left frame shows slot usage in the base case, and the right frame shows slot usage in the recursive case. Signals are flags that are set to indicate that the described condition has been met; i.e. [5] is explicitly set to *true* after values are written to [0] and [7]. Note that the same slot, such as [7], can have a different meaning for the two mutually exclusive cases.

4.1.3 The MDP Code

The MDP code is included in Appendix A.1. It has all of the same characteristics as the hybrid code, i.e. the same frame slot assignments and SQs (modulo my slightly different calling convention). The hybrid code had 28 instructions; the MDP code has 180, not counting code in library routines. Thus there are an average of 6.4 MDP instructions per hybrid instruction. This blow-up is not as bad as it seems because a MDP instruction word is roughly one-fourth the size of a hybrid instruction word.[1] Part of the growth is thus the accepted expansion factor between CISCy and RISCy architectures. As the reader will recall, there are two reasons that one hybrid word expands into many instruction words: First, hybrid instructions are more powerful and suited to the special purpose than the J-Machine; second, an expansion occurs to fit the code into the more restrictive MDP addressing modes.

4.1.4 Load Balancing

As mentioned in Section 2.3.2, my compiler does no load balancing. The user must modify the code produced by the *get-context* instruction to spawn procedures to an appropriate processor, usually a function of the argument(s); otherwise, all calls will go to the same node. Because factorial is singly recursive, it makes sense to spawn `(fact n)` onto processor n, because no task will already be running there. I changed one line of the compiled routine to implement this. If n were potentially larger than p, the number of processors, we would take its value modulo p. This would guarantee an even distribution.

4.1.5 Dynamic Counts

When I ran `(fact 4)` on the MDP simulator, it took 1263 ticks for the result to be written to the original calling frame. A tick is the time unit used by the simulator: One tick equals one instruction, even though not all instructions on the J-Machine will take the same time. The simulator also ignores network latency. Four processors were enabled, and utilization was 37% — i.e. on average, a processor did useful work a little over a third of the time. Fault

[1] I cannot give an exact length for hybrid words, because the compiler I used was for a paper version of the architecture where word lengths were essentially unlimited. According to Iannucci in private correspondence, the word size can be roughly thought of as 64 bits.

Handler Name	*Times Called*	*Ticks/Call*	*Total Ticks*
Lookup	25	5 or $6 + 6w$	212
CFUT	21	18	378
Move-Remote	16	$13 + 7w$	264
Continue-Test	14	7 or 20	189
Get-Context	3	24	72
Allocate	3	12	36
Total	136	n/a	1061

Table 4.1: System Calls for `(fact 4)`. Ranges are specified for the ticks/call column because the time may depend on the data. For *lookup* and *move-remote*, w specifies the number of waiting continuations. An estimated average number is used (with $w \approx \frac{1}{2}$) to approximate the total ticks.

Instruction Type	*Times Used*	*Percent*	*Comments*
Move	882	47.8	Both reg-reg and reg-frame
Field	247	13.3	Operations on tags
Network	237	12.8	Sending messages
DC	159	8.6	Loading constants into R0
Branch	125	6.7	Does not include busy-looping
Fault	87	4.7	Entering and leaving system calls
ALU	61	3.3	ALU ops for program and libraries
NOP	46	2.4	NOPs used as padding to align instructions

Table 4.2: Dynamic Instruction Usage for `(fact 4)`.

and library usage is shown in Table 4.1. As the totals show, 84% of the time was spent in the libraries. The routine that consumed the most time was the cfuture fault handler. It was called 21 times, and each time took 18 ticks. As described in Section 2.2.4, the cfuture handler must allocate space to store a continuation and fill in the necessary data. Dynamic instruction usage (not counting idle cycles) is shown by category in Table 4.2.

The average number of instructions executed per message is 92.6, which is larger than the 55 instructions per message empirically found by Horwat in [Horwat 1989, page 104]. His Concurrent Smalltalk version of the same factorial program takes only 315 ticks to complete [Horwat 1989, page 110], compared to my 1263.

Argument	*Ticks/Call*	*Ticks/Skewed Calls*		*Ticks/Nonskewed Calls*	
		1st Result	2nd Result	1st Result	2nd Result
4	1263	1864	2163	1992	2271
8	2691	4204	4590	4332	4611
12	4119	6544	6846	6672	6951

Table 4.3: Throughput for Factorial. This table compares the number of ticks required to compute one and two calls of factorial. For each case, the number of processors used is the same as the argument. The first data column shows how long it takes for one call executing alone to complete. The second set of columns shows how long it takes to complete two factorial calls made at the same time, skewed among the enabled processors. The last set of columns shows the completion times when the two calls are not skewed among the processors.

4.1.6 Throughput

One reason for the high latency is that, at every design decision, throughput was favored over latency. This is due to the decision to break apart any transaction of unbounded latency, which increased the latency of tasks but improved throughput. Table 4.3 shows that computing two invocations of factorial concurrently on the J-Machine takes significantly less than twice as much time as computing a single call. This is true for two reasons:

1. Each task suspends itself when it is waiting for a result from another processor.

2. The factorial calls can be skewed among the processors.

The table isolates these factors by including results for when the procedure calls are skewed and when they are not. Even when two factorial calls execute on the same processors, in the same order, throughput is increased over the single call case. This is because subtasks of the second factorial call can execute when no work can be done on a given processor toward the first factorial call.

4.1.7 Conclusion

Although I was pleased that the throughput of the system was better than the reciprocal of the latency, I was disappointed by the high latency, although it was predictable. One of my purposes in following the factorial program through each step was to show where all the

```
fib n =
  if n <= 1 then
    n
  else
    fib (n-1) + fib (n-2);
```

Figure 4-6: Id Code for Fibonacci

overhead was added. I should have expected the traditional costs of simulating one architecture on another. Almost half the time was spent on the "cfuture" and "lookup" handlers which store suspended continuations and revive them, respectively. It would be impossible to simulate these features with high efficiency. The problem can be summarized succinctly: Because almost all the synchronization is handled in software, it is impractical to synchronize on individual frame slots. While the costs incurred to synchronize on arguments and return values would be reasonable, synchronizing on temporary values is excessively expensive. This is exacerbated by the hybrid compiler's lavish creation of frame slots, which make sense on its architecture but not when synchronization is done in software.

While I was initially optimistic after the 6.4:1 code expansion because of the normal CISC/RISC trade-offs, we see now that this number is actually irrelevant, as the vast majority of time is spent in library routines that the hybrid architecture would have in hardware. Simulating the hybrid architecture was thus not an optimal choice for implementing dataflow on the J-Machine. A better choice is described in the next chapter.

4.2 Fibonacci

Another program I benchmarked was the doubly-recursive Fibonacci routine shown in Figure 4-6. The corresponding MDP code is in Appendix A.2. There are 46 lines of hybrid code, which translate to 271 lines of MDP code, yielding a ratio of 5.9:1. Because its transformation is so similar to factorial's, I will not go into detail, except to mention that I added a distribution function to load balance. Empirically, I found the function $(((p \text{ and } n)+(p \text{ or } n)) \text{ and } 31)$, where p is the current processor number and n the new argument. In runs with more than one processor, I used this function to map calls to processors.

Argument	*# Processors*	*Number of Ticks*
1	1	166
4	1	4353
4	6	2105
6	1	13760
6	13	3473
8	22	5628
10	32	9566
12	32	67641

Table 4.4: Timings for Fibonacci. Note that, until the argument gets very large, the growth in number of ticks is not exponential when many processors are used. Computing Fib(n) takes roughly $(\frac{1+\sqrt{5}}{2})^n$ procedure calls, which can be distributed among the processors.

The times and statistics for execution with different arguments is shown in Table 4.4. Note that for low arguments using multiple processors, growth is closer to linear than exponential. This is illustrated in Figure 4-7.[2]

While the number of ticks was higher than I would have liked for Fibonacci, the change in its order of growth was just the sort of thing one hopes to see on a parallel computer. I was only able to simulate 32 processors. The results should be fantastic when a 4096-node J-Machine comes on-line.

4.3 Loop Parallelization

In the Fibonacci example, the parallelism was due to a function distribution strategy that I added by hand, thus it cannot really be counted as part of the system. This is in contrast to loop parallelization, for which it is straightforward for the compiler to provide parallelism: If there are K iteration areas, each need only be assigned unique processors to send subcalls to; for example, iteration area i could spawn its subcalls to processors i, $i + K$, etc. Because K would be available at run-time (and optionally at compile-time), this could easily be computed.

Because the compiler did not handle loops, as explained in Section 2.3.3, I compiled simple loop programs by hand and did not have the time or compute-power for a large example. The

[2]Unfortunately, I was unable to get a Concurrent Smalltalk timing to compare it to.

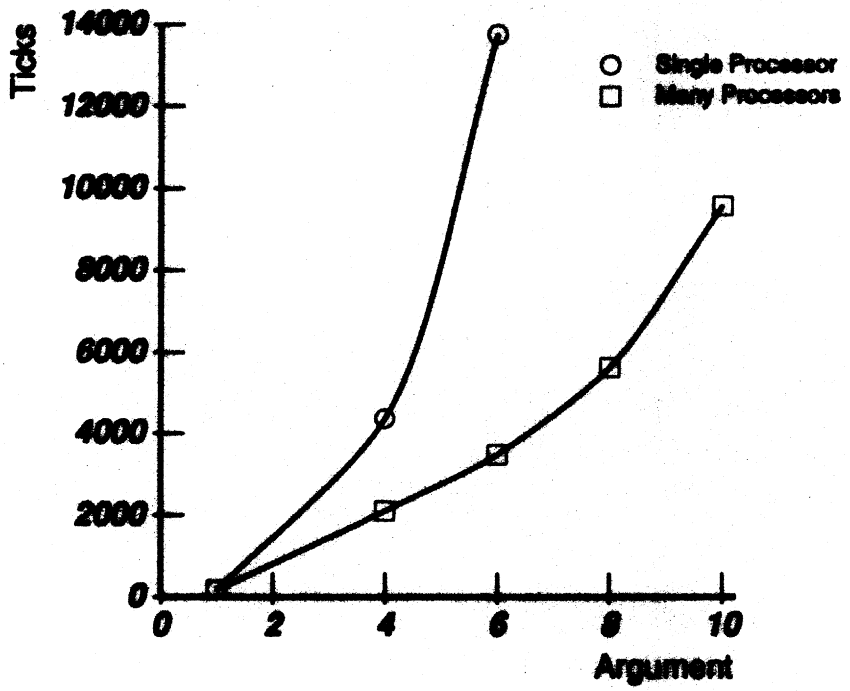


Figure 4-7: Plot of Ticks for Fibonacci

```
def loop n =
  { sum = 0
   in
  { for i <- 1 to n do
      sum_increment = sum + i;
      next sum = sum_increment
    finally sum }};
```

Figure 4-8: Id Code for Loop Example

loop program I used is shown in Figure 4-8. The program returns the sum of the first n integers.[3]

The produced code may be seen in Appendix A.3. There are 48 lines of hybrid code which were translated to 188 lines of MDP code, for a 3.9:1 ratio. The better instruction ratio is due to my hand-compiling the code rather than using my non-optimizing compiler. While I purposely did not generate top-quality code, I still used better register allocation than the compiler, saving reloads. Another factor was the reliance on additional library routines.

This program is a useful benchmark in that it shows the overhead to set up iteration areas and to launch iterations of a loop in parallel. The number of ticks, as a function of K, the number of iterations to unroll, and n, the argument, is $50 + 5 * K + 135 * n$. The three addends of the formula can be interpreted:

1. The constant term, 50, indicates that the additional cost for a procedure to use loop parallelization is low. There is thus no inhibition against parallelizing loops.

2. The $5 * K$ term is a pleasant surprise: Once the base cost for loop parallelization has been paid, it only costs 5 ticks to add and support each iteration area. This makes it reasonable to unroll many iterations of a loop.

3. The $135 * n$ term shows that each dynamic iteration of the loop is costly. However, this also can be thought of in terms of constant overhead: If each iteration of the loop spawns a long subroutine, as in the example in Figure 2-13, the only additional code that will

[3]The body of the loop could have been written more succinctly as `next sum = sum + i`. I work with this version because the hacked-up version of Iannucci's compiler could not compile new loop programs, and the hybrid code for this example was the only available.

execute on the loop processor is that to spawn a procedure call. This means that each iteration of the loop will use fewer than 200 ticks on its home processor, regardless of how big a computation it performs. As described above, it is trivial to distribute its procedure calls so that they do not interfere with those of other iterations.

I thus consider the loop parallelization strategy a success, although I am still dissatisfied with the overhead. A primary reason for the high overhead is the small number of registers on the MDP. There are only four general-purpose registers and four address registers. Two of the address registers are special-purpose and cannot be used by my system, and one, A2, is dedicated to holding the base of frame. This only leaves one address register, A1, to use as an iteration area pointer, which is inadequate. Because hybrid addressing modes exist to directly access slots of the previous, current, and next iteration areas, as well as offsets from the current frame, it would be useful to have spare address registers for each of these pointers. As things are now, the value in A1 keeps getting clobbered as references are made to the other iteration areas, requiring the addresses to be recomputed frequently. Because of the shortage of general-purpose registers, I cannot use them to cache frequently-needed values.

4.4 Conclusion

For simple programs like factorial and Fibonacci, the code performed several times worse than Concurrent Smalltalk code. While this is disappointing, it is to be expected, as one architecture is being used to simulate another. Loop parallelization provided very promising results, particularly because the semantics of Id and the state of its compiler are such the programmer need never be aware of possible parallelization. Any gain in parallelization and efficiency that occurs without any programmer effort is a big win.

Chapter 5

Conclusion

And oftentimes, to win us to our harm,
The instruments of darkness tell us truths,
Win us with honest trifles, to betray 's
In deepest consequence.
— William Shakespeare, *Macbeth*, Act I, Scene iii, line 123.

The current system has several strengths and weaknesses. I consider its primary strengths to be:

- It successfully simulates the hybrid architecture within an acceptable factor of code expansion.
- It includes a powerful loop parallelization strategy that shows the feasibility of concurrent execution of iterations of a loop.
- The observed throughput of the system implies that it succeeds to some extent at latency toleration — something more important in real systems and big programs than in toy benchmarks.
- By taking advantage of the Id language and compiler, it is possible for to write parallel programs for the J-Machine without explicit mention of parallelism.

The only disappointment is that the costs of going through the hybrid architecture may outweigh the benefits. There are three incremental approaches that can be taken in future

efforts: improving MDP code generation, improving hybrid code generation, and eliminating weaknesses of the J-Machine. I discuss each of these and then propose taking a different approach.

5.1 Improving MDP Code

As mentioned in appropriate sections throughout the document, the MDP code I produce is not optimal. Specifically, register assignment is primitive, and various peephole optimizations could be performed. In contrast, the libraries (see Appendix B) are tightly hand-coded, as I wrote them directly in MDP assembly language. Because roughly 80% of execution time is spent in the libraries, local optimizations of compiled code are unimportant. Even if I could double the speed of the compiled code produced, the total execution time would only increase by 10%. Therefore, it is not feasible to drastically improve the code through local optimization.

5.2 Improving Hybrid Code

One problem with my system is that the hybrid code I begin with is non-optimal, particularly in terms of the J-Machine, where cfuture faults, lookup calls, etc., are costly. I think optimizations to the hybrid compiler would go much further than ones in my back-end for the MDP. For several reasons, however, it seems that modifying the hybrid compiler would be a poor idea:

1. The hybrid compiler does not fit quite properly on top of the current version of the base Id compiler, and work would be required to bring them into synch.

2. Particularly because the code was written by someone else, writing new code might be easier than modifying it. This is meant not to criticize Iannucci's excellent and very readable coding style but as a general comment on the difficulty of one programmer's modifying another's code.

3. If extensive modification or a re-write is necessary, there is no reason for the extra costs added by going through an intermediate architecture.

Because the hybrid architecture is too different for the J-Machine to execute its code as efficiently as code generated specifically for the J-Machine, it makes little sense to put effort into generating hybrid code that would be better for the J-Machine.

5.3 Strengths and Weaknesses of the J-Machine

Several features of the J-Machine make it excellent for running dataflow code; it was designed to support fine-grained computation as described in [Dally 1988a]. The features not found on most computers that proved most beneficial were:

1. Hardware support for cfutures.

2. The low-latency network which gives the freedom to send frequent messages encouraging the division of tasks.

3. User-defined tag types, which aided debugging.

4. The large number of processors that will be available.

There were some things I did not like about the J-Machine. Suggestions for alleviating two of the worst problems are:

1. Increase the number of address and general-purpose registers. Four of each, particularly when some have special purposes, is inadequate, as described in Section 4.3.

2. Hardware support for cfuture suspensions would make frequent synchronization much more affordable. The 18 ticks for each call of the cfuture fault handler is too expensive.

At a recent Concurrent VLSI Architecture group meeting, I was pleased to find that others felt the same needs and that such changes might be made for the next version of the J-Machine.

In several instances, however, of imperfect fits between hybrid code and the J-Machine, it is impossible to blame either architecture. From this observation and the above descriptions of rejected ideas for incremental changes, I would like to propose a different approach that does not rely on trying to fit the two architectures together.

5.4 Synchronization on Tokens

After reading this document based on Traub and Iannucci's method of partially sequentializing dataflow programs, it is difficult to step back and imagine a different method that does not use frames and continuation lists. Such a method exists, based more directly on Greg Papadopoulos' *explicit token store* (ETS) [Papadopoulos 1988].[1] The basic idea, used on Monsoon, is that each cycle, a token is removed from the queue. Its context value, c, is added to the destination instruction offset, s, and that location is checked. If the location is empty, the value, v, is stored there. If the location is not empty, the value stored there must be the other argument, so the instruction is executed. It is not obvious why this method is better for the J-Machine, but empirical results suggest it is.

Last year, as a UROP, I designed a method to use the explicit token store on the MDP [Spertus 1989]. Its only similarity to my new system is that the message words are:

1. Instruction address, s.
2. Context value, c.
3. Data value, v.

Figures 5-1 and 5-2 show code for the *-1* and *multiply* nodes, respectively, from the dataflow graph in Figure 4-2 provide examples of monadic and dyadic nodes. The cfuture fault handler is only two lines long and is shown in Figure 5-3. Bill Dally played a major role in developing these templates. For further details, such as the calling convention, see [Spertus 1989].

The only benchmark for this system is factorial. It took 431 ticks to compute 4!, compared to the new system's 1263. The comparison is fair even though it is between hand-compiled and machine-generated code, as transforming dataflow nodes is straightforward. Actually, the comparison is unfair in the other direction, because so much intelligent effort has been put into the hybrid system. If I had spent the past year studying how to improve the ETS code, such as by discovering how to combine a few instructions with known orderings into a single macro-dataflow node, this technique would surely surpass the performance of the hybrid system, especially because it is already better.

[1] While Iannucci's method uses an explicit token store also, the schema I am presenting is more trivially based on Papadopoulos' ideas.

```
; Subtract 1 node
fact1_node5:
        move    [2,A3], R1
        sub     R1, 1, R1
        DC      MSG:(fact1_node7_left<<sys_len_bits)+3
        send2   3, R0, 0
        send2e  [1,A3], R1, 0
        suspend
```

Figure 5-1: A Monadic Node Using ETS. The data value is taken from offset two in the message, and the constant 1 is subtracted. The result is sent to the left input of node 7 on processor 3.

```
; Multiply node
fact1_node8:
        move    [1,A3], R0      ; Put data_addr in A2
        move    R0, A2
        move    [2,A3], R1      ; Get new argument

        ; This line may fault
        mul     R1, [0, A2], R1

        DC      MSG:(fact1_node9_right<<sys_len_bits)+3
        send2   6, R0, 0
        send2e  [1,A3], R1, 0

        ; Clean up
        wtag    R0, CFUT, R0
        move    R0, [0,A2]
        suspend
```

Figure 5-2: A Dyadic Node Using ETS. If the other argument has not arrived yet, a fault occurs instead of a multiply. The fault handler will write the new argument to the faulted slot. If the other argument is already there, the multiply precedes, and a token is sent to node 9, after which the slot must be emptied if the frame is to be reused.

```
; cfuture handler
fault_cfut_loc:
        move R1, [0,A2]
        suspend
```

Figure 5-3: The Cfuture Handler for ETS. It simply moves the new argument, guaranteed to be in R1, into the slot reserved for the argument.

5.5 Conclusion

Even though I do not think the system good enough to justify continuing dataflow research on the J-Machine by building on it, I consider the experiment with the hybrid architecture to be a success. In addition to the successful results described in the analysis, there were several other successful aspects to the project:

- By working with both the Computation Structures Group and the Concurrent VLSI Architecture group, I was able to help cross-fertilize two groups that have very different outlooks on the same problem, parallel computation. MIT has been criticized for not having enough communication between groups.

- By stretching on the J-Machine in ways its designers never imagined, I have found some of its limits. While this does not mean the J-Machine is flawed or necessarily should be changed, its architects should keep aware of what trade-offs they have made and reconsider them.

- In the process of building my compiler, I have built utilities that will convert among different formats of MDP code. This should aid other J-Machine programmers in future work.

- There have still been few enough MDP coders that I have significantly increased the number of hours spent MDP hacking. I have helped contribute to the set of known neat hacks for the J-Machine (such as with the code in Figure 3-6).

- By proving the feasibility of parallelizing iterations of a loop and presenting ideas on how "straight-line" Id code could be better converted, I have made a powerful case for

the possible efficiency of dataflow program execution on the J-Machine.

In conclusion, I consider the project very worthwhile and am optimistic about the results of further research on dataflow computation on the J-Machine.

Appendix A

MDP Program Examples

A.1 MDP Code for Factorial

```
module FACT

;((:LABEL (:LITERAL (:SYMBOL :SQ-1))))
SQ_1:
        MOVE [1,A3], R3
        MOVE R3, A2

;((:MOVE (:LITERAL (:CODE-BLOCK :FACT)) (:FRAME (:BASE 12))))
        MOVE 12, R1
        CALL LOOKUP_VECTOR
        DC {FACT_codeblock_ref}
        MOVE R0, [12,A2]

;((:CONTINUE-TEST (:FRAME (:BASE 3) :SUSPENSIVE) (:LITERAL (:SYMBOL :SQ-2))))
        DC {SQ_2_msg_ref}
        MOVE 3, R1
        CALL CNTT_VECTOR

;((:CONTINUE-TEST (:FRAME (:BASE 0) :SUSPENSIVE) (:LITERAL (:SYMBOL :SQ-11))))
        DC {SQ_11_msg_ref}
        MOVE 0, R1
        CALL CNTT_VECTOR

;((:LABEL (:LITERAL (:SYMBOL :SEND-RESULT-0))))
SEND_RESULT_0:
        MOVE [1,A3], R3
        MOVE R3, A2

;((:MOVE-REMOTE (:FRAME (:BASE 0) :SUSPENSIVE)
;               (:LITERAL (:INTEGER 1))
;               (:FRAME (:BASE 6) :SUSPENSIVE)))
SUSPENSIVE4683:
        MOVE [1,A3], R3
        MOVE R3, A2
        DC {SUSPENSIVE4683_msg_ref}
        RTAG [0,A2], R3
```

```
        RTAG [6,A2], R3
        SENDO [0,A2]
        DC {LOCAL_MOVR_msg_ref}
        SENDO R0
        SENDO [0,A2]
        SENDO 1
        SENDEO [6,A2]

;((:MOVE-REMOTE (:FRAME (:BASE 0)) (:LITERAL (:INTEGER 0))
;               (:FRAME (:BASE 5) :SUSPENSIVE)))
SUSPENSIVE4689:
        MOVE [1,A3], R3
        MOVE R3, A2
        DC {SUSPENSIVE4689_msg_ref}
        RTAG [5,A2], R3
        SENDO [0,A2]
        DC {LOCAL_MOVR_msg_ref}
        SENDO R0
        SENDO [0,A2]
        SENDO 0
        SENDEO [5,A2]

;((:TERMINATE))
        SUSPEND

;((:LABEL (:LITERAL (:SYMBOL :SQ-11))))
SQ_11:
        MOVE [1,A3], R3
        MOVE R3, A2

;((:TEST-2 (:FRAME (:BASE 0) :SUSPENSIVE) (:FRAME (:BASE 7) :SUSPENSIVE)
;          (:FRAME (:BASE 5))))
SUSPENSIVE4695:
        MOVE [1,A3], R3
        MOVE R3, A2
        DC {SUSPENSIVE4695_msg_ref}
        RTAG [0,A2], R3
        RTAG [7,A2], R3
        MOVE 5, R1
        CALL LOOKUP_VECTOR
        MOVE true, R3
        MOVE R3, [5,A2]

;((:TERMINATE))
        SUSPEND

;((:LABEL (:LITERAL (:SYMBOL :SQ-2))))
SQ_2:
        MOVE [1,A3], R3
        MOVE R3, A2

;((:<= (:FRAME (:BASE 3) :SUSPENSIVE) (:LITERAL (:INTEGER 1))
;      (:FRAME (:BASE 4))))
SUSPENSIVE4702:
        MOVE [1,A3], R3
        MOVE R3, A2
        DC {SUSPENSIVE4702_msg_ref}
        RTAG [3,A2], R3
        MOVE 4, R1
        CALL LOOKUP_VECTOR
        MOVE [3,A2], R3
        LE R3, 1, R2
        MOVE R2, [4,A2]
```

```
;((:BRANCH-FALSE (:FRAME (:BASE 4)) (:LITERAL (:SYMBOL :ELSE-4))))
        MOVE [4,A2], R3
        BT R3, 2
        DC {ELSE_4_ip_ref}
        MOVE RO, IP
:

;((:MOVE-IDENTITY (:FRAME (:BASE 3) :SUSPENSIVE) (:FRAME (:BASE 6))))
SUSPENSIVE4710:
        MOVE [1,A3], R3
        MOVE R3, A2
        DC {SUSPENSIVE4710_msg_ref}
        RTAG [3,A2], R3
        MOVE 6, R1
        CALL LOOKUP_VECTOR
        MOVE [3,A2], R3
        MOVE R3, [6,A2]

;((:MOVE-IDENTITY (:FRAME (:BASE 4)) (:FRAME (:BASE 7))))
        MOVE 7, R1
        CALL LOOKUP_VECTOR
        MOVE [4,A2], R3
        MOVE R3, [7,A2]

;((:BRANCH (:LITERAL (:SYMBOL :END-IF-4))))
        DC {END_IF_4_ip_ref}
        MOVE RO, IP

;((:LABEL (:LITERAL (:SYMBOL :ELSE-4))))
ELSE_4:
        MOVE [1,A3], R3
        MOVE R3, A2

;((:- (:FRAME (:BASE 3) :SUSPENSIVE) (:LITERAL (:INTEGER 1))
;     (:FRAME (:BASE 14))))
SUSPENSIVE4718:
        MOVE [1,A3], R3
        MOVE R3, A2
        DC {SUSPENSIVE4718_msg_ref}
        RTAG [3,A2], R3
        MOVE 14, R1
        CALL LOOKUP_VECTOR
        MOVE [3,A2], R3
        SUB R3, 1, R2
        MOVE R2, [14,A2]

;((:CONTINUE (:LITERAL (:SYMBOL #:SQ4674))))
        MOVE NNR, R3
        SENDO R3
        DC {SQ4674_msg_ref}
        SENDO RO
        SENDEO A2

;((:GET-CONTEXT (:FRAME (:BASE 12)) (:FRAME (:BASE 10))))
        MOVE A2, R3
        WTAG R3, INT, R3
        LSH R3, 6, R3
        MOVE NNR, R2
        ADD R3, R2, R3
        WTAG R3, FD, R3
        send0 [14,A2] ; argument
        DC {LOCAL_GETC_msg_ref}
        SENDO RO
        SENDO [12,A2]
```

```
        SENDO R3
        SENDEO 10

;((:SPECIAL-TEST-1 (:FRAME (:BASE 10))))
SUSPENSIVE4729:
        MOVE [1,A3], R3
        MOVE R3, A2
        DC {SUSPENSIVE4729_msg_ref}
        RTAG [10,A2], R3

;((:INDEX-CURRENT-CONTEXT (:LITERAL (:BASE 8)) (:REGISTER 0)))
        MOVE A2, R3
        WTAG R3, INT, R3
        MOVE R0, R2
        DC 8192
        ADD R3, R0, R3
        LSH R3, 6, R3
        MOVE NNR, R1
        ADD R3, R1, R3
        WTAG R3, FD, R3
        MOVE R3, [0,A0]

;((:MOVE-REMOTE (:FRAME (:BASE 10)) (:LITERAL (:INTEGER 0))
; (:REGISTER 0)))
        SENDO [10,A2]
        DC {LOCAL_MOVR_msg_ref}
        SENDO R0
        SENDO [10,A2]
        SENDO 0
        SENDEO [0,A0]

;((:MOVE-REMOTE (:FRAME (:BASE 10)) (:LITERAL (:INTEGER 3))
;               (:FRAME (:BASE 14))))
        SENDO [10,A2]
        DC {LOCAL_MOVR_msg_ref}
        SENDO R0
        SENDO [10,A2]
        SENDO 3
        SENDEO [14,A2]

;((:TERMINATE))
        SUSPEND

;((:LABEL (:LITERAL (:SYMBOL #:SQ4674))))
SQ4674:
        MOVE [1,A3], R3
        MOVE R3, A2

;((:CONTINUE-TEST (:FRAME (:BASE 3) :SUSPENSIVE) (:LITERAL (:SYMBOL :SQ-5))))
        DC {SQ_5_msg_ref}
        MOVE 3, R1
        CALL CNTT_VECTOR

;((:CONTINUE-TEST (:FRAME (:BASE 8) :SUSPENSIVE) (:LITERAL (:SYMBOL :SQ-8))))
        DC {SQ_8_msg_ref}
        MOVE 8, R1
        CALL CNTT_VECTOR

;((:LABEL (:LITERAL (:SYMBOL :END-IF-4))))
END_IF_4:
        MOVE [1,A3], R3
        MOVE R3, A2

;((:TERMINATE))
```

```
        SUSPEND

;((:LABEL (:LITERAL (:SYMBOL :SQ-8))))
SQ_8:
        MOVE [1,A3], R3
        MOVE R3, A2

;((:TEST-1 (:FRAME (:BASE 8) :SUSPENSIVE) (:REGISTER 0)))
SUSPENSIVE4741:
        MOVE [1,A3], R3
        MOVE R3, A2
        DC {SUSPENSIVE4741_msg_ref}
        RTAG [8,A2], R3
        MOVE true, R3
        MOVE R3, [0,A0]

;((:RETURN-CONTEXT (:FRAME (:BASE 10) :SUSPENSIVE) (:FRAME (:BASE 11))))
SUSPENSIVE4746:
        MOVE [1,A3], R3
        MOVE R3, A2
        DC {SUSPENSIVE4746_msg_ref}
        RTAG [10,A2], R3
        MOVE 11, R1
        CALL LOOKUP_VECTOR
        MOVE true, R3
        MOVE R3, [11,A2]

;((:MOVE-IDENTITY (:FRAME (:BASE 11)) (:FRAME (:BASE 7))))
        MOVE 7, R1
        CALL LOOKUP_VECTOR
        MOVE [11,A2], R3
        MOVE R3, [7,A2]

;((:TERMINATE))
        SUSPEND

;((:LABEL (:LITERAL (:SYMBOL :SQ-6))))
SQ_6:
        MOVE [1,A3], R3
        MOVE R3, A2

;((:* (:FRAME (:BASE 3) :SUSPENSIVE) (:FRAME (:BASE 9) :SUSPENSIVE)
;     (:FRAME (:BASE 13))))
SUSPENSIVE4753:
        MOVE [1,A3], R3
        MOVE R3, A2
        DC {SUSPENSIVE4753_msg_ref}
        RTAG [3,A2], R3
        RTAG [9,A2], R3
        MOVE 13, R1
        CALL LOOKUP_VECTOR
        MOVE [3,A2], R3
        MUL R3, [9,A2], R1
        MOVE R1, [13,A2]

;((:MOVE-IDENTITY (:FRAME (:BASE 13)) (:FRAME (:BASE 6))))
        MOVE 6, R1
        CALL LOOKUP_VECTOR
        MOVE [13,A2], R3
        MOVE R3, [6,A2]

;((:TERMINATE))
        SUSPEND
end
```

```
ref SUSPENSIVE4753_msg_ref = MSG:(((SUSPENSIVE4753+FACT_loc)<<10))+2
ref SUSPENSIVE4746_msg_ref = MSG:(((SUSPENSIVE4746+FACT_loc)<<10))+2
ref SUSPENSIVE4741_msg_ref = MSG:(((SUSPENSIVE4741+FACT_loc)<<10))+2
ref SQ_8_msg_ref = MSG:(((SQ_8+FACT_loc)<<10))+2
ref SQ_5_msg_ref = MSG:(((SQ_5+FACT_loc)<<10))+2
ref SUSPENSIVE4729_msg_ref = MSG:(((SUSPENSIVE4729+FACT_loc)<<10))+2
ref SQ4674_msg_ref = MSG:(((SQ4674+FACT_loc)<<10))+2
ref SUSPENSIVE4718_msg_ref = MSG:(((SUSPENSIVE4718+FACT_loc)<<10))+2
ref SUSPENSIVE4710_msg_ref = MSG:(((SUSPENSIVE4710+FACT_loc)<<10))+2
ref SUSPENSIVE4702_msg_ref = MSG:(((SUSPENSIVE4702+FACT_loc)<<10))+2
ref SUSPENSIVE4695_msg_ref = MSG:(((SUSPENSIVE4695+FACT_loc)<<10))+2
ref SUSPENSIVE4689_msg_ref = MSG:(((SUSPENSIVE4689+FACT_loc)<<10))+2
ref SUSPENSIVE4683_msg_ref = MSG:(((SUSPENSIVE4683+FACT_loc)<<10))+2
ref SQ_11_msg_ref = MSG:(((SQ_11+FACT_loc)<<10))+2
ref SQ_2_msg_ref = MSG:(((SQ_2+FACT_loc)<<10))+2
ref END_IF_4_ip_ref = IP:(((END_IF_4+FACT_loc)<<10))+ABSOLUTE
ref ELSE_4_ip_ref = IP:(((ELSE_4+FACT_loc)<<10))+ABSOLUTE
ref FACT_codeblock_ref = CB:(FACT_loc<<16)+15
```

A.2 MDP Code for Fibonacci

```
module FIB


;((:LABEL (:LITERAL (:SYMBOL :SQ-1))))
SQ_1:
        MOVE [1,A3], R3
        MOVE R3, A2

;((:MOVE (:LITERAL (:CODE-BLOCK :FIB)) (:FRAME (:BASE 18))))
        MOVE 18, R1
        CALL LOOKUP_VECTOR
        DC {FIB_codeblock_ref}
        MOVE R0, [18,A2]

;((:CONTINUE-TEST (:FRAME (:BASE 3) :SUSPENSIVE) (:LITERAL (:SYMBOL :SQ-2))))
        DC {SQ_2_msg_ref}
        MOVE 3, R1
        CALL CNTT_VECTOR

;((:CONTINUE-TEST (:FRAME (:BASE 0) :SUSPENSIVE) (:LITERAL (:SYMBOL :SQ-17))))
        DC {SQ_17_msg_ref}
        MOVE 0, R1
        CALL CNTT_VECTOR

;((:LABEL (:LITERAL (:SYMBOL :SEND-RESULT-0))))
SEND_RESULT_0:
        MOVE [1,A3], R3
        MOVE R3, A2

;((:MOVE-REMOTE (:FRAME (:BASE 0) :SUSPENSIVE)
;               (:LITERAL (:INTEGER 1))
;               (:FRAME (:BASE 6) :SUSPENSIVE)))
SUSPENSIVE2503:
        MOVE [1,A3], R3
        MOVE R3, A2
        DC {SUSPENSIVE2503_msg_ref}
        RTAG [0,A2], R3
        RTAG [6,A2], R3
        SENDO [0,A2]
        DC {LOCAL_MOVR_msg_ref}
        SENDO R0
        SENDO [0,A2]
        SENDO 1
        SENDEO [6,A2]

;((:MOVE-REMOTE (:FRAME (:BASE 0)) (:LITERAL (:INTEGER 0)) (:FRAME (:BASE 5) :SUSPENSIVE)))
SUSPENSIVE2509:
        MOVE [1,A3], R3
        MOVE R3, A2
        DC {SUSPENSIVE2509_msg_ref}
        RTAG [5,A2], R3
        SENDO [0,A2]
        DC {LOCAL_MOVR_msg_ref}
        SENDO R0
        SENDO [0,A2]
        SENDO 0
        SENDEO [5,A2]

;((:TERMINATE))
```

```
        SUSPEND

;((:LABEL (:LITERAL (:SYMBOL :SQ-17))))
SQ_17:
        MOVE [1,A3], R3
        MOVE R3, A2

;((:TEST-2 (:FRAME (:BASE 0) :SUSPENSIVE) (:FRAME (:BASE 7) :SUSPENSIVE) (:FRAME (:BASE 5))))
SUSPENSIVE2515:
        MOVE [1,A3], R3
        MOVE R3, A2
        DC {SUSPENSIVE2515_msg_ref}
        RTAG [0,A2], R3
        RTAG [7,A2], R3
        MOVE 5, R1
        CALL LOOKUP_VECTOR
        MOVE true, R3
        MOVE R3, [5,A2]

;((:TERMINATE))
        SUSPEND

;((:LABEL (:LITERAL (:SYMBOL :SQ-2))))
SQ_2:
        MOVE [1,A3], R3
        MOVE R3, A2

;((:<= (:FRAME (:BASE 3) :SUSPENSIVE) (:LITERAL (:INTEGER 1)) (:FRAME (:BASE 4))))
SUSPENSIVE2522:
        MOVE [1,A3], R3
        MOVE R3, A2
        DC {SUSPENSIVE2522_msg_ref}
        RTAG [3,A2], R3
        MOVE 4, R1
        CALL LOOKUP_VECTOR
        MOVE [3,A2], R3
        LE R3, 1, R2
        MOVE R2, [4,A2]

;((:BRANCH-FALSE (:FRAME (:BASE 4)) (:LITERAL (:SYMBOL :ELSE-4))))
        MOVE [4,A2], R3
        BT R3, 2
        DC {ELSE_4_ip_ref}
        MOVE R0, IP
:

;((:MOVE-IDENTITY (:FRAME (:BASE 3) :SUSPENSIVE) (:FRAME (:BASE 6))))
SUSPENSIVE2530:
        MOVE [1,A3], R3
        MOVE R3, A2
        DC {SUSPENSIVE2530_msg_ref}
        RTAG [3,A2], R3
        MOVE 6, R1
        CALL LOOKUP_VECTOR
        MOVE [3,A2], R3
        MOVE R3, [6,A2]

;((:MOVE-IDENTITY (:FRAME (:BASE 4)) (:FRAME (:BASE 7))))
        MOVE 7, R1
        CALL LOOKUP_VECTOR
        MOVE [4,A2], R3
        MOVE R3, [7,A2]

;((:BRANCH (:LITERAL (:SYMBOL :END-IF-4))))
```

```
        DC {END_IF_4_ip_ref}
        MOVE R0, IP

;((:LABEL (:LITERAL (:SYMBOL :ELSE-4))))
ELSE_4:
        MOVE [1,A3], R3
        MOVE R3, A2

;((:- (:FRAME (:BASE 3) :SUSPENSIVE) (:LITERAL (:INTEGER 1)) (:FRAME (:BASE 20))))
SUSPENSIVE2538:
        MOVE [1,A3], R3
        MOVE R3, A2
        DC {SUSPENSIVE2538_msg_ref}
        RTAG [3,A2], R3
        MOVE 20, R1
        CALL LOOKUP_VECTOR
        MOVE [3,A2], R3
        SUB R3, 1, R2
        MOVE R2, [20,A2]

;((:- (:FRAME (:BASE 3) :SUSPENSIVE) (:LITERAL (:INTEGER 2)) (:FRAME (:BASE 19))))
SUSPENSIVE2544:
        MOVE [1,A3], R3
        MOVE R3, A2
        DC {SUSPENSIVE2544_msg_ref}
        RTAG [3,A2], R3
        MOVE 19, R1
        CALL LOOKUP_VECTOR
        MOVE [3,A2], R3
        SUB R3, 2, R2
        MOVE R2, [19,A2]

;((:CONTINUE (:LITERAL (:SYMBOL #:SQ2490))))
        MOVE NNR, R3
        SENDO R3
        DC {SQ2490_msg_ref}
        SENDO R0
        SENDEO A2

;((:GET-CONTEXT (:FRAME (:BASE 18)) (:FRAME (:BASE 12))))
        MOVE A2, R3
        WTAG R3, INT, R3
        LSH R3, 6, R3
        MOVE NNR, R2
        ADD R3, R2, R3
        WTAG R3, FD, R3
        ; make up destination
        move [19,A2],R1
        or R1, R2, R0
        and R1, R2, R2
        add R0, R2, R1
        move 31, R0
        and R1, R0, R1
        send0 R1
; send0 1
        DC {LOCAL_GETC_msg_ref}
        SENDO R0
        SENDO [18,A2]
        SENDO R3
        SENDEO 12

;((:SPECIAL-TEST-1 (:FRAME (:BASE 12))))
SUSPENSIVE2555:
        MOVE [1,A3], R3
```

```
        MOVE R3, A2
        DC {SUSPENSIVE2555_msg_ref}
        RTAG [12,A2], R3

;((:INDEX-CURRENT-CONTEXT (:LITERAL (:BASE 10)) (:REGISTER 0)))
        MOVE A2, R3
        WTAG R3, INT, R3
        MOVE R0, R2
        DC 10240
        ADD R3, R0, R3
        LSH R3, 6, R3
        MOVE NNR, R1
        ADD R3, R1, R3
        WTAG R3, FD, R3
        MOVE R3, [0,A0]

;((:MOVE-REMOTE (:FRAME (:BASE 12)) (:LITERAL (:INTEGER 0)) (:REGISTER 0)))
        SENDO [12,A2]
        DC {LOCAL_MOVR_msg_ref}
        SENDO R0
        SENDO [12,A2]
        SENDO 0
        SENDEO [0,A0]

;((:MOVE-REMOTE (:FRAME (:BASE 12)) (:LITERAL (:INTEGER 3)) (:FRAME (:BASE 19))))
        SENDO [12,A2]
        DC {LOCAL_MOVR_msg_ref}
        SENDO R0
        SENDO [12,A2]
        SENDO 3
        SENDEO [19,A2]

;((:TERMINATE))
        SUSPEND

;((:LABEL (:LITERAL (:SYMBOL #:SQ2490))))
SQ2490:
        MOVE [1,A3], R3
        MOVE R3, A2

;((:CONTINUE-TEST (:FRAME (:BASE 10) :SUSPENSIVE) (:LITERAL (:SYMBOL :SQ-8))))
        DC {SQ_8_msg_ref}
        MOVE 10, R1
        CALL CNTT_VECTOR

;((:CONTINUE (:LITERAL (:SYMBOL #:SQ2494))))
        MOVE NNR, R3
        SENDO R3
        DC {SQ2494_msg_ref}
        SENDO R0
        SENDEO A2

;((:GET-CONTEXT (:FRAME (:BASE 18)) (:FRAME (:BASE 15))))
        MOVE A2, R3
        WTAG R3, INT, R3
        LSH R3, 6, R3
        MOVE NNR, R2
        ADD R3, R2, R3
        WTAG R3, FD, R3
        ; make up destination
        move [20,A2],R1
        or R1, R2, R0
        and R1, R2, R2
        add R0, R2, R1
```

```
        move 31, R0
        and R1, R0, R1
        send0 R1
; SENDO 1
        DC {LOCAL_GETC_msg_ref}
        SENDO R0
        SENDO [18,A2]
        SENDO R3
        SENDEO 15

;((:SPECIAL-TEST-1 (:FRAME (:BASE 15))))
SUSPENSIVE2570:
        MOVE [1,A3], R3
        MOVE R3, A2
        DC {SUSPENSIVE2570_msg_ref}
        RTAG [15,A2], R3

;((:INDEX-CURRENT-CONTEXT (:LITERAL (:BASE 13)) (:REGISTER 0)))
        MOVE A2, R3
        WTAG R3, INT, R3
        MOVE R0, R1
        DC 13312
        ADD R3, R0, R3
        LSH R3, 6, R3
        MOVE NNR, R0
        ADD R3, R0, R3
        WTAG R3, FD, R3
        MOVE R3, [0,A0]

;((:MOVE-REMOTE (:FRAME (:BASE 15)) (:LITERAL (:INTEGER 0)) (:REGISTER 0)))
        SENDO [15,A2]
        DC {LOCAL_MOVR_msg_ref}
        SENDO R0
        SENDO [15,A2]
        SENDO 0
        SENDEO [0,A0]

;((:MOVE-REMOTE (:FRAME (:BASE 15)) (:LITERAL (:INTEGER 3)) (:FRAME (:BASE 20))))
        SENDO [15,A2]
        DC {LOCAL_MOVR_msg_ref}
        SENDO R0
        SENDO [15,A2]
        SENDO 3
        SENDEO [20,A2]

;((:TERMINATE))
        SUSPEND

;((:LABEL (:LITERAL (:SYMBOL #:SQ2494))))
SQ2494:
        MOVE [1,A3], R3
        MOVE R3, A2

;((:CONTINUE-TEST (:FRAME (:BASE 14) :SUSPENSIVE) (:LITERAL (:SYMBOL :SQ-12))))
        DC {SQ_12_msg_ref}
        MOVE 14, R1
        CALL CNTT_VECTOR

;((:CONTINUE-TEST (:FRAME (:BASE 13) :SUSPENSIVE) (:LITERAL (:SYMBOL :SQ-13))))
        DC {SQ_13_msg_ref}
        MOVE 13, R1
        CALL CNTT_VECTOR

;((:CONTINUE-TEST (:FRAME (:BASE 8) :SUSPENSIVE) (:LITERAL (:SYMBOL :SQ-14))))
```

```
        DC {SQ_14_msg_ref}
        MOVE 8, R1
        CALL CNTT_VECTOR

;((:LABEL (:LITERAL (:SYMBOL :END-IF-4))))
END_IF_4:
        MOVE [1,A3], R3
        MOVE R3, A2

;((:TERMINATE))
        SUSPEND

;((:LABEL (:LITERAL (:SYMBOL :SQ-14))))
SQ_14:
        MOVE [1,A3], R3
        MOVE R3, A2

;((:TEST-2 (:FRAME (:BASE 8) :SUSPENSIVE) (:FRAME (:BASE 9) :SUSPENSIVE) (:FRAME (:BASE 16))))
SUSPENSIVE2582:
        MOVE [1,A3], R3
        MOVE R3, A2
        DC {SUSPENSIVE2582_msg_ref}
        RTAG [8,A2], R3
        RTAG [9,A2], R3
        MOVE 16, R1
        CALL LOOKUP_VECTOR
        MOVE true, R3
        MOVE R3, [16,A2]

;((:MOVE-IDENTITY (:FRAME (:BASE 16)) (:FRAME (:BASE 7))))
        MOVE 7, R1
        CALL LOOKUP_VECTOR
        MOVE [16,A2], R3
        MOVE R3, [7,A2]

;((:TERMINATE))
        SUSPEND

;((:LABEL (:LITERAL (:SYMBOL :SQ-13))))
SQ_13:
        MOVE [1,A3], R3
        MOVE R3, A2

;((:TEST-1 (:FRAME (:BASE 13) :SUSPENSIVE) (:REGISTER 0)))
SUSPENSIVE2590:
        MOVE [1,A3], R3
        MOVE R3, A2
        DC {SUSPENSIVE2590_msg_ref}
        RTAG [13,A2], R3
        MOVE true, R3
        MOVE R3, [0,A0]

;((:RETURN-CONTEXT (:FRAME (:BASE 15) :SUSPENSIVE) (:FRAME (:BASE 8))))
SUSPENSIVE2595:
        MOVE [1,A3], R3
        MOVE R3, A2
        DC {SUSPENSIVE2595_msg_ref}
        RTAG [15,A2], R3
        MOVE 8, R1
        CALL LOOKUP_VECTOR
        MOVE true, R3
        MOVE R3, [8,A2]

;((:TERMINATE))
```

```
        SUSPEND

;((:LABEL (:LITERAL (:SYMBOL :SQ-12))))
SQ_12:
        MOVE [1,A3], R3
        MOVE R3, A2

;((:+ (:FRAME (:BASE 14) :SUSPENSIVE) (:FRAME (:BASE 11) :SUSPENSIVE) (:FRAME (:BASE 17))))
SUSPENSIVE2601:
        MOVE [1,A3], R3
        MOVE R3, A2
        DC {SUSPENSIVE2601_msg_ref}
        RTAG [14,A2], R3
        RTAG [11,A2], R3
        MOVE 17, R1
        CALL LOOKUP_VECTOR
        MOVE [14,A2], R3
        ADD R3, [11,A2], R0
        MOVE R0, [17,A2]

;((:MOVE-IDENTITY (:FRAME (:BASE 17)) (:FRAME (:BASE 6))))
        MOVE 6, R1
        CALL LOOKUP_VECTOR
        MOVE [17,A2], R3
        MOVE R3, [6,A2]

;((:TERMINATE))
        SUSPEND

;((:LABEL (:LITERAL (:SYMBOL :SQ-8))))
SQ_8:
        MOVE [1,A3], R3
        MOVE R3, A2

;((:TEST-1 (:FRAME (:BASE 10) :SUSPENSIVE) (:REGISTER 0)))
SUSPENSIVE2610:
        MOVE [1,A3], R3
        MOVE R3, A2
        DC {SUSPENSIVE2610_msg_ref}
        RTAG [10,A2], R3
        MOVE true, R3
        MOVE R3, [0,A0]

;((:RETURN-CONTEXT (:FRAME (:BASE 12) :SUSPENSIVE) (:FRAME (:BASE 9))))
SUSPENSIVE2615:
        MOVE [1,A3], R3
        MOVE R3, A2
        DC {SUSPENSIVE2615_msg_ref}
        RTAG [12,A2], R3
        MOVE 9, R1
        CALL LOOKUP_VECTOR
        MOVE true, R3
        MOVE R3, [9,A2]

;((:TERMINATE))
        SUSPEND
end
ref SUSPENSIVE2615_msg_ref = MSG:(((SUSPENSIVE2615+FIB_loc)<<10))+2
ref SUSPENSIVE2610_msg_ref = MSG:(((SUSPENSIVE2610+FIB_loc)<<10))+2
ref SUSPENSIVE2601_msg_ref = MSG:(((SUSPENSIVE2601+FIB_loc)<<10))+2
ref SUSPENSIVE2595_msg_ref = MSG:(((SUSPENSIVE2595+FIB_loc)<<10))+2
ref SUSPENSIVE2590_msg_ref = MSG:(((SUSPENSIVE2590+FIB_loc)<<10))+2
ref SUSPENSIVE2582_msg_ref = MSG:(((SUSPENSIVE2582+FIB_loc)<<10))+2
ref SQ_14_msg_ref = MSG:(((SQ_14+FIB_loc)<<10))+2
```

```
ref SQ_13_msg_ref = MSG:(((SQ_13+FIB_loc)<<10))+2
ref SQ_12_msg_ref = MSG:(((SQ_12+FIB_loc)<<10))+2
ref SUSPENSIVE2570_msg_ref = MSG:(((SUSPENSIVE2570+FIB_loc)<<10))+2
ref SQ2494_msg_ref = MSG:(((SQ2494+FIB_loc)<<10))+2
ref SQ_8_msg_ref = MSG:(((SQ_8+FIB_loc)<<10))+2
ref SUSPENSIVE2555_msg_ref = MSG:(((SUSPENSIVE2555+FIB_loc)<<10))+2
ref SQ2490_msg_ref = MSG:(((SQ2490+FIB_loc)<<10))+2
ref SUSPENSIVE2544_msg_ref = MSG:(((SUSPENSIVE2544+FIB_loc)<<10))+2
ref SUSPENSIVE2538_msg_ref = MSG:(((SUSPENSIVE2538+FIB_loc)<<10))+2
ref SUSPENSIVE2530_msg_ref = MSG:(((SUSPENSIVE2530+FIB_loc)<<10))+2
ref SUSPENSIVE2522_msg_ref = MSG:(((SUSPENSIVE2522+FIB_loc)<<10))+2
ref SUSPENSIVE2515_msg_ref = MSG:(((SUSPENSIVE2515+FIB_loc)<<10))+2
ref SUSPENSIVE2509_msg_ref = MSG:(((SUSPENSIVE2509+FIB_loc)<<10))+2
ref SUSPENSIVE2503_msg_ref = MSG:(((SUSPENSIVE2503+FIB_loc)<<10))+2
ref SQ_17_msg_ref = MSG:(((SQ_17+FIB_loc)<<10))+2
ref SQ_2_msg_ref = MSG:(((SQ_2+FIB_loc)<<10))+2
ref END_IF_4_ip_ref = IP:(((END_IF_4+FIB_loc)<<10))+ABSOLUTE
ref ELSE_4_ip_ref = IP:(((ELSE_4+FIB_loc)<<10))+ABSOLUTE
ref FIB_codeblock_ref = CB:(FIB_loc<<16)+21
```

A.3 MDP Code for Loop Example

```
; This is a rewrite of the loop program with non-Iannucci structures.
; Instead of having:
;       -----   -----   -----
;    /->|   |<->|   |<->|   |<----\
;    |  -----   -----   -----     |
;    \---------------------------/
; use a setup where different iterations iteration pointers are
; contiguous:
;        --------------
;       | normal frame |
;       |              |
;       |--------------|
;       |ptr to it. K-1|        (Think of as "-1")
;       |ptr to it. 0  |
;       |ptr to it. 1  |
;       |      ...     |
;       |ptr to it. K-1|
;       |ptr to it. 0  |        (Think of as "K")
;       |--------------|
;       | subframe 0   |
;       |              |
;       |--------------|
;       | subframe 1   |
;       |              |
;       |--------------|
;       |     ...      |
;       |--------------|
;       | subframe K-1 |
;        --------------
;
; The message format will be:
;       MSG:location of code
;       ADDR:frame base
;       ADDR:location of iteration pointer
; A2 will be loaded with the frame base.
; A1 will be loaded with the base of the subframe

; Times (including proc call overhead)

;       arg     k       time
;       ---     -       ----
;       0       2       325
;       1       2       460
;       10      2       1675
;       A       2       325+135*A
;
;       0       3       330
;
;       10      5       1690
;
;       A       K       325+135*A+5*K


label LIBRARY_PLACE=$180

label frame_size = 9
label frame_n_iteration_slots = 6
label argument = 10
label k = 5
```

```
label total_frame_size = frame_size + k * (frame_n_iteration_slots + 1) + 2
label slotID = total_frame_size - 1
label slotK = 2

include "lib3.mdp"

;; Program code

label CODE_PLACE=$400

module program_code

;       ((:LABEL (:LITERAL (:SYMBOL :SQ-1)))
sq_1:
        move    [1,A3], R0
        move    R0, A2

; Altered order is a temporary kludge

;        (:CNTT (:FRAME (:BASE 3)
;                       :SUSPENSIVE)
;               (:LITERAL (:SYMBOL :SQ-4)))
        DC      {sq_4_msg_ref}
        move    3, R1
        call    CNTT

;        (:CNTT (:FRAME (:BASE 0)
;                       :SUSPENSIVE)
;               (:LITERAL (:SYMBOL :SQ-3)))
        DC      {sq_3_msg_ref}
        move    0, R1
        call    CNTT

:       move    ip, R0
        move    R0, [0,A3]

;        (:LABEL (:LITERAL (:SYMBOL :SEND-RESULT-0)))
send_result_0:
        move    [1,A3], R0
        move    R0, A2

;        (:MOVR (:FRAME (:BASE 0)
;                       :SUSPENSIVE)
;               (:LITERAL (:INTEGER 1))
;               (:FRAME (:BASE 7)
;                       :SUSPENSIVE))
        DC      {local_movr_msg_ref}
        move    [7,A2], R1
        send2   [0,A2], R0, 0
        send    [0,A2], 0
        send2e  1, R1, 0

:       move    ip, r0
        move    r0, [0,A3]

        move    [1,A3], R0
        move    R0, A2

;        (:MOVR (:FRAME (:BASE 0))
;               (:LITERAL (:INTEGER 0))
;               (:FRAME (:BASE 5)
;                       :SUSPENSIVE))
        DC      {local_movr_msg_ref}
        move    [5,A2], R1
```

```
        send2   [0,A2], R0, 0
        send    [0,A2], 0
        send2e  0, R1, 0

;        (:TERMINATE)
        suspend

;        (:LABEL (:LITERAL (:SYMBOL :SQ-4)))
sq_4:
        move    [1,A3], R0
        move    R0, A2

;        (:MOVE (:FRAME (:BASE 3)
;                       :SUSPENSIVE)
;               (:FRAME (:BASE 4)))
        move    4, R1
        call    LOOKUP
        move    [3,A2], R1
        move    R1, [4,A2]

;;       (:SUB (:FRAME (:BASE 2))
;;             (:LITERAL (:INTEGER 2))
;;             (:REGISTER 0))

;;       (:MOVE (:LITERAL (:INTEGER 9))
;;              (:REGISTER 1))

;;       (:SUB (:REGISTER 1)
;;             (:LITERAL (:INTEGER 6))
;;             (:REGISTER 2))

;;       (:ADD (:REGISTER 1)
;;             (:LITERAL (:INTEGER 6))
;;             (:REGISTER 3))

;;       (:STPR (:REGISTER 2))

;;       (:STCR (:REGISTER 1))

;;       (:STNX (:REGISTER 3))

        ; Put base of ID memory into A1
        DC      INT:frame_size<<sys_len_bits
        move    A2, R2
        wtag    R2, INT, R2
        add     R2, R0, R2
        wtag    R2, ADDR, R2
        move    R2, A1

        ; Put k into R1
        move    [slotK,A2], R1
        sub     R1, 1, R1

        ; Put base offset into R2
        move    2+frame_size, R2
        add     R2, [2,A2], R2

        ; In this setup, 0 through k-2 get IM
        ;                -1, k-1, k do not.

        ; Loop through
        DC      INT:maskIM
        or      R2, R0, R2
        move    1, R3
```

```
loop_setup_loop:
        move    R2, [R3,A1]
        lt      R3, R1, R0
        add     R2, frame_n_iteration_slots, R2
        add     R3, 1, R3
        bt      R0, ^loop_setup_loop

        ; Store k-1 slot, etc.
        DC      INT:~(maskIM | maskPC)
        and     R2, R0, R2

        move    R2, [R3,A1]     ; k-1 slot
        move    R2, [0,A1]

        move    [1,A1], R2
        and     R2, R0, R2
        add     R3, 1, R3
        move    R2, [R3,A1]


;;      (:LABEL (:LITERAL (:SYMBOL :SETUP-LOOP-5)))
setup_loop_5:

;;      (:BRZ (:REGISTER 0)
;;            (:LITERAL (:SYMBOL :END-SETUP-LOOP-5)))

;;      (:IXID (:LITERAL (:INTEGER 6))
;;             (:FRAME (:NEXT-ITERATION 0)))

;;      (:SUB (:REGISTER 0)
;;            (:LITERAL (:INTEGER 1))
;;            (:REGISTER 0))

;;      (:BR (:LITERAL (:SYMBOL :SETUP-LOOP-5)))

;;      (:LABEL (:LITERAL (:SYMBOL :END-SETUP-LOOP-5)))
end_setup_loop_5:

;;      (:STNX (:REGISTER 2))

;;      (:IXID (:LITERAL (:INTEGER 6))
;;             (:REGISTER 4))

;;      (:IXID (:LITERAL (:INTEGER 0))
;;             (:FRAME (:ITERATION 0)))

;       (:STIM (:FRAME (:ITERATION 0))
;              (:LITERAL (:BOOLEAN :FALSE))
;              (:FRAME (:ITERATION 0)))
;      In new scheme, this means set -1's to zero.  Done above.

;;      (:IXID (:LITERAL (:INTEGER 6))
;;             (:REGISTER 4))

;;      (:IXID (:LITERAL (:INTEGER 0))
;;             (:FRAME (:ITERATION 0)))

;;      (:STNX (:REGISTER 1))

;       (:STPC (:FRAME (:NEXT-ITERATION 0))
;              (:LITERAL (:SYMBOL :ITERATE-5))
;              (:FRAME (:NEXT-ITERATION 0)))
        DC      INT:maskPC
```

```
        move    1, R1
        call    CHECK_ITER

;        (:TERMINATE)
        suspend

;        (:LABEL (:LITERAL (:SYMBOL :ITERATE-5)))
iterate_5:
        move    [1,A3], R1
        move    R1, A2
        wtag    R1, INT, R1

        move    [2,A3], R2
        add     R2, frame_size+1, R2
        move    [R2,A2], R2     ; Offset to base of cur loop subframe now in R2
        lsh     R2, sys_len_bits, R2
        add     R1, R2, R1
        wtag    R1, ADDR, R1
        move    R1, A1          ; Base of cur loop subframe in A1

;        (:MOVE (:FRAME (:ITERATION 1)
;                       :NONSTICKY :SUSPENSIVE)
;               (:FRAME (:ITERATION 5)))
        ; offset 5 is internal to loop and need not be checked
        move    [1,A1], R0
        move    R0, [5,A1]

        DC      CFUT:$0
        move    R0, [1,A1]

;        (:LE (:FRAME (:ITERATION 5))
;             (:FRAME (:BASE 4))
;             (:FRAME (:ITERATION 4)))
        move    [5,A1], R0
        le      R0, [4,A2], R0
        move    R0, [4,A1]

;        (:BRF (:FRAME (:ITERATION 4))
;              (:LITERAL (:SYMBOL :END-LOOP-5)))
        move    [4,A1], R0
        bf      R0, ^end_loop_5

;        (:STPC (:FRAME (:NEXT-ITERATION 0))
;               (:LITERAL (:SYMBOL :ITERATE-5))
;               (:FRAME (:NEXT-ITERATION 0)))
        DC      INT:maskPC
        move    1, R1
        call    CHECK_ITER

;        (:ADD (:FRAME (:ITERATION 5))
;              (:LITERAL (:INTEGER 1))
;              (:FRAME (:NEXT-ITERATION 1)))
        move    [2,A3], R1
        add     R1, frame_size+1+1, R1  ; next
        move    [R1,A2], R1
        add     R1, 1, R1       ; offset 1
        DC      INT:~(maskIM | maskPC)
        and     R1, R0, R1
        call    LOOKUP

        move    [5,A1], R0
        add     R0, 1, R0       ; integer 1
        move    R0, [R1,A2]
```

```
;        (:ADD (:FRAME (:ITERATION 2)
;                      :NONSTICKY :SUSPENSIVE)
;              (:FRAME (:ITERATION 5))
;              (:FRAME (:NEXT-ITERATION 2)))
        rtag    [2,A1], R0

        add     R1, 2-1, R1
        call    LOOKUP

        move    [2,A1], R0
        add     R0, [5,A1], R0
        move    R0, [R1,A2]
        DC      CFUT:$0
        move    R0, [2,A1]

:
        move    ip, R0
        move    R0, [0,A3]

        move    [1,A3], R1
        move    R1, A2
        wtag    R1, INT, R1

        move    [2,A3], R2
        add     R2, frame_size+1, R2
        move    [R2,A2], R2     ; Offset to base of cur loop subframe now in R2
        lsh     R2, sys_len_bits, R2
        add     R1, R2, R1
        wtag    R1, ADDR, R1
        move    R1, A1          ; Base of cur loop subframe in A1

;        (:TST1 (:FRAME (:ITERATION 3)
;                       :NONSTICKY :SUSPENSIVE)
;               (:FRAME (:NEXT-ITERATION 3)))
        rtag    [3,A1], R0
        DC      CFUT:$0
        move    R0, [3,A1]

        move    [2,A3], R1
        add     R1, frame_size+1+1, R1  ; next
        move    [R1,A2], R1
        add     R1, 3, R1       ; offset 3
        DC      INT:~(maskIM | maskPC)
        and     R1, R0, R1
        call    LOOKUP

        move    true, R0
        move    R0, [R1,A2]


;        (:STIM (:FRAME (:PREVIOUS-ITERATION 0))
;               (:LITERAL (:BOOLEAN :TRUE))
;               (:FRAME (:PREVIOUS-ITERATION 0)))
        DC      INT:maskIM
        move    -1, R1
        call    CHECK_ITER

;        (:TERMINATE)
        suspend

;        (:LABEL (:LITERAL (:SYMBOL :END-LOOP-5)))
end_loop_5:
        move    [1,A3], R1
        move    R1, A2
```

```
        wtag    R1, INT, R1

        move    [2,A3], R2
        add     R2, frame_size+1, R2
        move    [R2,A2], R2     ; Offset to base of cur loop subframe now in R2
        lsh     R2, sys_len_bits, R2
        add     R1, R2, R1
        wtag    R1, ADDR, R1
        move    R1, A1          ; Base of cur loop subframe in A1

;        (:CNTT (:FRAME (:ITERATION 2)
;                       :SUSPENSIVE)
;               (:LITERAL (:SYMBOL :COPY-LOOP-VARIABLE-1)))
        DC      {copy_loop_variable_1_msg_ref}
        move    2, R1
        call    CNTT_LOOP

;        (:MOVE (:FRAME (:ITERATION 5))
;               (:FRAME (:BASE 6)))
        move    6, R1
        call    LOOKUP

        move    [5,A1], R0
        move    R0, [6,A2]

;        (:TERMINATE)
        suspend

;        (:LABEL (:LITERAL (:SYMBOL :COPY-LOOP-VARIABLE-1)))
copy_loop_variable_1:
        move    [1,A3], R1
        move    R1, A2
        wtag    R1, INT, R1

        move    [2,A3], R2
        add     R2, frame_size+1, R2
        move    [R2,A2], R2     ; Offset to base of cur loop subframe now in R2
        lsh     R2, sys_len_bits, R2
        add     R1, R2, R1
        wtag    R1, ADDR, R1
        move    R1, A1          ; Base of cur loop subframe in A1

;        (:CNTT (:FRAME (:ITERATION 3)
;                       :SUSPENSIVE)
;               (:LITERAL (:SYMBOL :COPY-LOOP-VARIABLE-2)))
        DC      {copy_loop_variable_2_msg_ref}
        move    3, R1
        call    CNTT_LOOP

:
        move    ip, R0
        move    R0, [0,A3]

        move    [1,A3], R1
        move    R1, A2
        wtag    R1, INT, R1

        move    [2,A3], R2
        add     R2, frame_size+1, R2
        move    [R2,A2], R2     ; Offset to base of cur loop subframe now in R2
        lsh     R2, sys_len_bits, R2
        add     R1, R2, R1
        wtag    R1, ADDR, R1
        move    R1, A1          ; Base of cur loop subframe in A1
```

```
;        (:MOVE (:FRAME (:ITERATION 2)
;                       :NONSTICKY :SUSPENSIVE)
;               (:FRAME (:BASE 7)))
        rtag    [2,A1], R0

        move    7, R1
        call    LOOKUP

        move    [2,A1], R0
        move    R0, [7,A2]
        DC      CFUT:$0
        move    R0, [2,A1]

;        (:TERMINATE)
        suspend

;        (:LABEL (:LITERAL (:SYMBOL :COPY-LOOP-VARIABLE-2)))
copy_loop_variable_2:
        move    [1,A3], R1
        move    R1, A2
        wtag    R1, INT, R1

        move    [2,A3], R2
        add     R2, frame_size+1, R2
        move    [R2,A2], R2     ; Offset to base of cur loop subframe now in R2
        lsh     R2, sys_len_bits, R2
        add     R1, R2, R1
        wtag    R1, ADDR, R1
        move    R1, A1          ; Base of cur loop subframe in A1

;        (:MOVE (:FRAME (:ITERATION 3)
;                       :NONSTICKY :SUSPENSIVE)
;               (:FRAME (:BASE 8)))
        rtag    [3,A2], R0

        move    8, R1
        call    LOOKUP

        move    [3,A1], R0
        move    R0, [8,A2]
        DC      CFUT:$0
        move    R0, [3,A1]


;        (:MOVE (:LITERAL (:SYMBOL :SIGNAL))
;               (:FRAME (:BASE 5)))
        move    true, R0
        move    R0, [5,A2]

;        (:TERMINATE)
        suspend

;        (:LABEL (:LITERAL (:SYMBOL :SQ-3)))
sq_3:
        move    [1,A3], R0
        move    R0, A2

;        (:MOVE (:FRAME (:BASE 0)
;                       :SUSPENSIVE)
;               (:FRAME (:NEXT-ITERATION 3)))
        rtag    [0,A2], R0      ; Check if value there

        move    [2,A3], R2
```

```
        add     R2, frame_size+1+1, R2
        move    [R2,A2], R2
        add     R2, 3, R1
        DC      INT:~(maskIM | maskPC)
        and     R1, R0, R1

        call    LOOKUP

        move    [0,A2], R2
        move    R2, [R1,A2]

;        (:MOVE (:LITERAL (:INTEGER 1))
;               (:FRAME (:NEXT-ITERATION 1)))
        sub     R1, 3-1, R1
        call    LOOKUP

        move    1, R0
        move    R0, [R1,A2]

;        (:MOVE (:LITERAL (:INTEGER 0))
;               (:FRAME (:NEXT-ITERATION 2)))
        add     R1, 2-1, R1
        call    LOOKUP

        move    0, R0
        move    R0, [R1,A2]
;        (:TERMINATE))
        suspend
end

ref sq_3_msg_ref = MSG: ((sq_3+CODE_PLACE) << sys_len_bits) + 3
ref sq_4_msg_ref = MSG: ((sq_4+CODE_PLACE) << sys_len_bits) + 3
ref copy_loop_variable_1_msg_ref=MSG:((copy_loop_variable_1+CODE_PLACE) << sys_len_bits) + 3
ref copy_loop_variable_2_msg_ref=MSG:((copy_loop_variable_2+CODE_PLACE) << sys_len_bits) + 3
ref iterate_5_msg_ref = MSG:((iterate_5+CODE_PLACE) << sys_len_bits) + 3
ref loop_msg_ref = MSG:((iterate_5+CODE_PLACE) << sys_len_bits) + 3
ref LOOP_CB = CB:((sq_1+CODE_PLACE)<<16) + total_frame_size

place program_code, CODE_PLACE

label TOP_PLACE = $500

@0

;; Top level code
module top_code

        ;; Create the frame
        ; First we must allocate a frame
        DC      {top1_ref}
        move    R0, R3
        DC      {loop_cb}
        move    R0, R1
        call    ALLOCATE
top_1:
        move    R2, A2
        DC      FD:$600<<16             ; Where to put result
        move    R0, [0,A2]
        move    argument, R1            ; Argument
        move    R1, [3,A2]
        move    k, R1
        move    R1, [slotK, A2]         ; K

        DC      MSG:((SQ_1+CODE_PLACE)<<sys_len_bits)+3
```

```
        move    -1, R2
        send2   0, R0, 0
        send2c  A2, R2, 0
        suspend

end_of_code:
        br      ^end_of_code

end

ref top1_ref = IP:((top_1+TOP_PLACE)<<sys_len_bits) + ABSOLUTE

place library_code, LIBRARY_PLACE
place top_code, TOP_PLACE

01..2
module
        org TOP_PLACE
busy_loop:
        br ^busy_loop
end
i = 0

00
ip = ip:(TOP_PLACE<<sys_len_bits) + ABSOLUTE
;watch fetch all
;watch read write r0..r3
;watch read write a0..a3
;watch write @100..@11f          ; queue
;watch fault all
break write @001
;separate on
break fault fault_type
break fault fault_limit
break fault 8 ; drumerr
```

Appendix B

MDP Library Code

B.1 General Library

```
; This file holds the library for VMDI program execution on the J-machine.
; It puts it all in a module library_code.
; It includes and defines as necessary and loads the system call vector
; with the following (i for input, o for output):
;       ALLOCATE (0) - Allocate a frame on the current node given a codeblock
;                      R1 (i) holds CB.  Addr result will be in R2 (o).
;       LOOKUP (1) - Check a location in a frame before writing to it
;                      in order to start any waiting processes.
;                      [R1,A2] (i) holds data.
;       CNTT (2) - Same as VMDI CNTT.  R1 (i) holds test location,
;                      R0 (i) holds MSG to be spawned.
;       CALLOC (3) - Allocate the number of words in R1 (i) and
;                      return the result as an ADDR in R2 (o).
;       LOOKUP_ITER (4) - Like LOOKUP, but takes its offset from A1;
;                      thus it takes a base in A1 (i) and an offset in R1 (i).
;       CHECK_ITER (5) - A new value for an ID in R1 (i) is put in the
;                      first spot of A1 (i) and starts up the loop if
;                      the import and PC flags are set.  For now, no
;                      PC field included in ID.  This must be fixed up.
;
; Various fault handlers are also defined:
;       CFUT - Replace accessed location with info about current continuation
;              then suspend.
;       SEND - Continue after unavoidable delay.
; Additionally, some methods accessed by non-local MSGs are supplied:
;       LOCAL_MOVR - Take a MSG of the form:
;                                FD
;                                INT:offset1
;                                ANY:value1
;                      Eventually, one will be able to send any number
;                      of INT,ANY pairs.  This stores the values into
;                      the offsets of the specified frame.
;       LOCAL_GETC - Takes a MSG of the form:
;                                CB to allocate frame for
;                                FD to send new FD to
;                                INT offset in desination FD
;                      Locally allocate a frame and send it back
```

```
;                       to requesting node.  Also start up code block
;                       on current node.

label FREE_PTR = $A00
label STACK_BASE = $A01

; stack space will be from $a01 up.  $a00
; will hold the first free location (not last used).

; IXCC creates (from an ADDR) a frame descriptor FD:
;       31 ... 16 15 .. 0
;         <addr>     MBR
; where <addr> should be right-shifted four to be properly placed.
;
; Similarly, a codeblock is typed CB and is:
;       31 ... 16  15   ...    0
;         <addr>    <frame size>
;
; To summarize:
;       GETC: CB -> ADDR (allocate_loc)
;       IXCC: ADDR -> FD
;       MOVR: FD x (INT x ANY)*
;       CNTT: ADDR -> ADDR (because it stays on some processor)
;

;; J-machine constants
include "/home/gn/ellens/Id/hw.mdp"
include "/home/gn/ellens/Id/newq.mdp"
label sys_len_bits = 10
label ABSOLUTE = (1<<8)
label UNCHECKED = (1<<31)

;; Constants for loops
label posPrevious = 0
label maskPrevious = $0000ff
label posCurrent = 8
label maskCurrent = $ff00
label posNext = 16
label maskNext = $ff0000
label posIM = 24
label maskIM = 1<<posIM
label posPC = 25
label maskPC = 1<<posPC

;; User-defined tags
tagname 8 "CB"
tagname 9 "FD"
tagname 10 "ISA"

; System calls
label ALLOCATE = 0
label ALLOCATE_VECTOR = 0
label LOOKUP = 1
label LOOKUP_VECTOR = 1
label CNTT = 2
label CNTT_VECTOR = 2
label CALLOC = 3
label CALLOC_VECTOR = 3
label LOOKUP_ITER = 4
label LOOKUP_ITER_VECTOR = 4
label CHECK_ITER = 5
label CHECK_ITER_VECTOR = 5

namevector ALLOCATE+32, "Allocate"
```

```
namevector LOOKUP+32, "Lookup"
namevector CNTT+32, "CNTT"
namevector CALLOC+32, "Calloc"

; Constants for calloc
; For best efficiency, (ISTRUCT_Q_SIZE - 1) % ISTRUCT_Q_ENTRY_SIZE = 0
label ISTRUCT_Q_SIZE = 8
label ISTRUCT_Q_ENTRY_SIZE = 2

module library_code

terrible:
        halt    0
        br      ^terrible

; In case of cfut fault, replace CFUT with continuation info.
; Type checking is *turned off* when this interrupt is entered!
; When we get here, [0,A3] either contains a valid MSG, or
; it contains a IP with p=1, a=1
fault_cfut_loc:
        move    R0, IDO         ; *****
        move    MAR, R1

; At this point, R1 holds address to store pointer in
fault_cfut_none_allocated:
        ; allocate a triple from stack
        DC      addr:FREE_PTR<<sys_len_bits
        move    R0, A1
        move    [0,A1], R2

        DC      INT:3<<sys_len_bits
        add     R2, R0, R3
        move    R3, [0,A1]
        move    R2, A1

        ; R2 and A1 now point to empty triple
        move    IDO, R3         ; *****

fault_cfut_msg_okay:
        move    R3, [0,A1]
        move    A2, R0
        move    R0, [1,A1]
        move    [R1,A0], R0
        move    R0, [2,A1]

        wtag    R2, CFUT, R2    ; Write the triple to where R1 points
        move    R2, [R1,A0]

        suspend

; R1 is a CB with input info.
; result will be an ADDR in R2.  Clobbers registers (except A2).
allocate_loc:
        check   R1, CB, R2
        bf      R2, ^terrible
        wtag    R1, INT, R1             ; Not strictly needed
        and     R1, $ffff, R1           ; Get size

        DC      ADDR:FREE_PTR<<sys_len_bits
        move    R0, A1
        move    [0,A1], R2

        lsh     R1, sys_len_bits, R1    ; Shift size count into place
        add     R2, R1, R1
```

```
        move    R1, [0,A1]

        wtag    R2, ADDR, R2
        move    fip, ip

; We need support for MOVR.  The format of the message should be:
;       FD
;       INT:offset1
;       ANY:value1
; The number of items can be determined from the message header.
; It must be 1 (for now)  This also runs in unchecked mode.
local_movr:
        move    [1,A3], R1      ; Put frame descriptor into R1
        check   R1, FD, R0
        bf      R0, ^terrible
        wtag    R1, INT, R1
        lsh     R1, -16, R1             ; Shift out node number
        lsh     R1, sys_len_bits, R1    ; Shift it into address position
;       wtag    R1, ADDR, R1
        move    R1, A1

        ; First (and only) word
        move    [2,A3], R1
        move    [3,A3], R3
        move    [R1,A1], R0             ; Save to see if anything waiting
        move    R3, [R1,A1]
        bz      R0, ^local_movr_done

        ; We must restart a continuation because R0 <> 0.

local_movr_next_triple:
        move    R0, A1
        move    NNR, R1
        send    R1, 0
        send    [0,A1], 0
        sende   [1,A1], 0
        ; We would deallocate the triple around here
        move    [2,A1], R0
        bnz     R0, ^local_movr_next_triple

local_movr_done:
        suspend

; When a getc is done, it initiates a split-phase transaction
; (according to Iannucci's injuction).  It sends a message to the
; desired node of the form:
;       <header>                        [0,A3]
;       CB to allocate frame for        [1,A3]
;       FD to send result new FD to     [2,A3]
;       Offset within FD                [3,A3]
; The job of local_getc, after allocating space, is to notify the
; caller and to set the frame in motion.  For obvious reasons, it
; does the two subtasks in that order.

local_getc:
        ; set up for ALLOCATE call
        DC      IP:(local_getc_1<<sys_len_bits)+ABSOLUTE
        move    R0, R3
        move    [1,A3], R1
        call    ALLOCATE

local_getc_1:
        ; Built up the FD and send it back
        DC      {local_movr_msg_ref}
```

```
        wtag    R2, INT, R3
        lsh     R3, 16-sys_len_bits, R3
        move    NNR, R1
        add     R3, R1, R3
        wtag    R3, FD, R3
        send2   [2,A3], R0, 0
        send    [2,A3], 0
        send2e  [3,A3], R3, 0

        ; Set up for method specified by code block
        move    [1,A3], R1
        wtag    R1, INT, R1
        lsh     R1, -16, R1     ; Shift off low bits
        lsh     R1, sys_len_bits, R1
        add     R1, 2, R1       ; Put in length bits
        wtag    R1, MSG, R1

        ; The ADDR is still in R2
        move    NNR, R0
        send    R0, 0
        send2e  R1, R2, 0
        suspend

; fault_send_loc is used to wait, when we send messages too fast.
; This routine is lifted, verbatim, from Waldemar's MS thesis.
; It requires type checking to be disabled.
fault_send_loc:
        move    fip, R0
        rot     R0, -9, R0
        sub     R0, 1, R0
        rot     R0,  9, R0
        move    R0, fip
        move    fop0, R0
        move    fip, ip

; This expects R1 to hold the offset from A2.
; Only R1, A2, and A3 are guaranteed.  Checking must be off.
lookup_loc:
        move    [R1,A2], R0
        check   R0, CFUT, R2

        bf      R2, ^terrible   ; Double write
        bz      R0, ^lookup_done

        move    NNR, R3
lookup_next:
        move    R0, A1

        send    R3, 0
        send    [0,A1], 0
;       sende   [1,A1], 0
        sende   A2, 0
        ; Deallocate triple
        move    [2,A1], R0
        bnz     R0, ^lookup_next

lookup_done:
        move    fip, ip

; For CNTT, R1 should hold the offset of the test location,
; and R0 should hold the message name.  At least for now,
; the continuation will be spawned to the same node.
; Checking should be off (to avoid CFUT faults).
cntt_loc:
```

```
        move    [R1,A2], R2             ; Check test location
        check   R2, CFUT, R3            ; Is it a CFUT?
        bf      R3, ^cntt_send_it       ; If not, we can send
        ; Instead, key it on [R1,A2]
        move    R0, IDO                 ; Save MSG

        ; allocate a triple from stack
        DC      addr:FREE_PTR<<sys_len_bits
        move    R0, A1
        move    [0,A1], R2

        DC      INT:3<<sys_len_bits
        add     R2, R0, R3
        move    R3, [0,A1]

        ; R2 holds base of triple
        move    R2, A1

        ; Fill in triple
        move    IDO, R0 ; Restore it
        move    R0, [0,A1]
        move    A2, R0
        move    R0, [1,A1]
        move    [R1,A2], R0             ; Take old pointer
        move    R0, [2,A1]              ; Put it at end of triple

        ; Store pointer to new triple
        wtag    R2, CFUT, R2    ; <------------------
        move    R2, [R1,A2]

        move    fip, ip

cntt_send_it:
        move    NNR, R1
        send2   R1, R0, 0
        sende   A2, 0
        move    fip, ip

; R1 holds the number of words requested.
; Result will be an ADDR in R2.
; Preserves A1 through A3.
calloc_loc:
        DC      ADDR:FREE_PTR<<sys_len_bits
        move    A1, R3
        move    R0, A1
        move    [0,A1], R2

        lsh     R1, sys_len_bits, R1    ; Shift size count into place
        add     R2, R1, R1
        move    R1, [0,A1]

        wtag    R2, ADDR, R2
        move    R3, A1

        move    fip, ip

; This expects R1 to hold the offset from A1.
; Only R1, A1, A2, and A3 are guaranteed.  Checking must be off.
lookup_iter_loc:
        move    [R1,A1], R0
;       check   R0, CFUT, R2
;       bf      R2, ^terrible
        bz      R0, ^lookup_iter_done
```

```
        move    A1, R3
        move    R3, A0          ; Save it
        move    NNR, R3
lookup_iter_next:
        move    R0, A1

        send    R3, 0
        send    [0,A1], 0
        sende   [1,A1], 0
        ; Deallocate triple
        move    [2,A1], R0
        bnz     R0, ^lookup_iter_next
        move    A0, R3
        move    R3, A1

lookup_iter_done:
        move    fip, ip

; Check_iter_loc expects R1 to have the value to put into ID [0,A1].
; It moves it there and starts the loop if both flags are set.
; It saves the address registers.
check_iter_loc:
        DC      INT:maskIM + maskPC
        move    R1, [0,A1]
        and     R1, R0, R1
        eq      R1, R0, R2
        bt      R2, ^check_iter_start
        move    fip, ip

check_iter_start:
        DC      {loop_msg_ref}
        move    NNR, R1
        send2   R1, R0, 0
        send    A1, 0
        sende   [slotID,A2], 0
        move    fip, ip


end

fault_vec_addr_p0 + fault_cfut = IP:((LIBRARY_PLACE+fault_cfut_loc)<<sys_len_bits) + ABSOLUTE+UNCHECKED
fault_vec_addr_p0 + fault_send = IP:((LIBRARY_PLACE+fault_send_loc)<<sys_len_bits) + ABSOLUTE+UNCHECKED
syscall_vec_addr + ALLOCATE = IP:((LIBRARY_PLACE+allocate_loc)<<sys_len_bits) + ABSOLUTE
syscall_vec_addr + LOOKUP = IP:((LIBRARY_PLACE+lookup_loc)<<sys_len_bits) + ABSOLUTE+UNCHECKED
syscall_vec_addr + CNTT = IP:((LIBRARY_PLACE+cntt_loc)<<sys_len_bits) + ABSOLUTE+UNCHECKED
syscall_vec_addr + CALLOC = IP:((LIBRARY_PLACE+calloc_loc)<<sys_len_bits) + ABSOLUTE
ref local_movr_msg_ref = MSG:((LIBRARY_PLACE+local_movr)<<sys_len_bits)+UNCHECKED+4
ref local_getc_msg_ref = MSG:((LIBRARY_PLACE+local_getc)<<sys_len_bits)+4
ref local_fetch_msg_ref = MSG:((LIBRARY_PLACE+local_fetch)<<sys_len_bits)+UNCHECKED+4
ref local_store_msg_ref = MSG:((LIBRARY_PLACE+local_store)<<sys_len_bits)+UNCHECKED+3
syscall_vec_addr + LOOKUP_ITER = IP:((LIBRARY_PLACE+lookup_iter_loc)<<sys_len_bits) + ABSOLUTE+UNCHECKED
syscall_vec_addr + CHECK_ITER = IP:((LIBRARY_PLACE+check_iter_loc)<<sys_len_bits) + ABSOLUTE


include "lotsozots.mdp" ; CFUTUREs for stack
FREE_PTR=INT:STACK_BASE<<sys_len_bits
```

B.2 I-Structure Routines

```
; This is a changed version of istruct2.mdp that uses different
; representations:
;       EMPTY - null CFUT
;       WAITING - non-null CFUT
;       DATA - non-CFUT
; It also goes through local_movr.
;
; The format of i-structure addresses are:
;       The low 16 bits hold the node number
;       The high 16 bits hold the address on that node
; I-structure addresses are typed TAG8, which will be defined
; to ITAG.

;; This is what code to fetch an I-structure cell looks like:
; With the pointer (tagged int) in R1 and the I-struct offset in R2,
; and the frame offset in R3.
i_fetch_code:
        dc      {system_fetch_msg_ref}
        send20  R1, R0          ; Send node number, header
        send20  R1, R3          ; Send ISA
        send20e A2, R3          ; Send frame, offset
        suspend

; System fetch gets:
;       [0,A3]: MSG:<system-fetch>
;       [1,A3]: INT:<i-structure address>
;       [2,A3]: INT:<offset from i-structure>
;       [3,A3]: FD:<frame of dest>
;       [4,A3]: INT:<offset from frame>

;;; WARNING: SENSITIVE TO BIT CHANGES:
;;;          Specifically, assumes SYS_LEN_BITS = 10,
;;;                                MAX_NODES = 2^16
system_fetch:
        move    [1,A3], R1      ; Put ISA in R1
        lsh     R1, -16, R1     ; Slide over address portion to del node #
        lsh     R1, 10, R1      ; Slide into address position
        move    R1, A1
        move    [2,A3], R2      ; Put offset into R2
        gt      R2, [1,A1], R3  ; If it's greater than upper bound...
        bt      R3, ^i_err      ; ...it's an error.
        move    [0,A1], R0      ; Put lower bound in R0
        sub     R2, R0, R2      ; Subtract off base
        lt      R2, 0, R3       ; If it's lower than base...
        bt      R3, ^i_err      ; ...then it's an error
        add     R2, 2, R2       ; Point past two bounds words

        move    [R2,A1], R1     ; Take item in i-structure spot

        check   R1, CFUT, R0
        bt      R0, ^data_not_present

        ; If we get here, we have the data and can return it.
        send0   [3,A3]          ; Node number of destination
        send0   [4,A3]          ; MSG header of destination
        send20e [1,A3], R0      ; context, value
        suspend


;;; This case handles both a first and subsequent store.
```

```
;;; It allocates a triple for a linked-list.

data_not_present:
        ; If we get here, [R2,A1], the reference, R1, holds a cfuture.

        ; Get triple
        DC      ADDR:FREE_PTR<<sys_len_bits
        move    R0, A2
        DC      3<<sys_len_bits
        move    [0,A2], R3      ; Put start location in R2
        move    R3, A2          ; A2 now holds a ptr to a new triple
        add     R3, R0, R3      ; Put next free location in R3...
        move    R3, [0,A2]      ; ...and then back into free ptr

        ; Store A2 into i-structure location
        move    A2, R0
        move    R0, [R2,A1]

        ; Store in the following order:
        ;       Frame number of destination
        ;       Offset w/in frame
        ;       Next ptr
        move    [3,A3], R0      ; Frame number
        move    R0, [0,A2]
        move    [4,A3], R0      ; Frame offset
        move    R0, [1,A2]
        move    R1, [2,A2]      ; Next ptr

        suspend

; System-store gets:
;       [0,A3]: MSG:<system-store>
;       [1,A3]: INT:<i-structure address>
;       [2,A3]: INT:<offset>
;       [3,A3]: <data>

system_store:
        move    [1,A3], R1      ; Put ISA in R1
        lsh     R1, -16, R1     ; Slide over address portion to del node #
        lsh     R1, 10, R1      ; Slide into address position
        move    R1, A1          ; A1 now holds abs address of base
        move    [2,A3], R2      ; Put offset into R2
        gt      R2, [1,A1], R3  ; If it's greater than upper bound...
        bt      R3, ^i_err      ; ...it's an error.
        move    [0,A1], R0      ; Put lower bound in R0
        sub     R2, R0, R2      ; Subtract off base
        lt      R2, 0, R3       ; If it's lower than base...
        bt      R3, ^i_err      ; ...then it's an error
        add     R2, 2, R2       ; Point past two bounds words

        move    [R2,A1], R1     ; Take item in i-structure spot
        check   R1, CFUT, R0    ; It had better be a cfuture.
        bf      R0, ^i_err      ; If not, it's a write-twice error.

        move    [3,A3], R3      ; Put data value into R3
        move    R3, [R2,A1]     ; Store it into i-structure

        DC      {local_movr_msg_ref}
        bz      R1, ^sends_done

; At this point, R1 holds base of next linked-list entry.
; R0 holds the local_movr_msg_ref.

send_loop:
```

```
        move    R1, A1
        [illegible] [0,A1], R0      ; [illegible]
        [illegible] [0,A1]          ; [illegible]
        [illegible] [1,A1], R0      ; [illegible]

        move    [2,A1], R1
        bne     *send_loop

[illegible]:
        suspend

;; Out of bounds or double-write error.
i_err:
        halt    1
end
```

B.3 Loop Support

```
;; Constants for loops
label posPrevious = 0
label maskPrevious = $0000ff
label posCurrent = 8
label maskCurrent = $ff00
label posNext = 16
label maskNext = $ff0000
label posIM = 24
label maskIM = 1<<posIM
label posPC = 25
label maskPC = 1<<posPC

; System calls
label CHECK_ITER = 5

; Expects R0 to have the value (maskIM or maskPC) to be or'd into
; the ID R1 (+/-1) off from the current iteration.
; It moves it there and starts the loop if both flags are set.
; It saves the address registers.
; For now only, ignore wraparound
check_iter_loc:
        move    [2,A3], R2
        add     R2, R1, R2

        ; This sequence converts a value of k to 0.  Trust me.
;       ge      R2, [2,A2], R3
;       wtag    R3, INT, R3
;       neg     R3, R3
;       and     R3, [2,A2], R3
;       sub     R2, R3, R2

        ; Yo!  I can do even better:
;       lt      R2, [2,A2], R3
;       wtag    R3, INT, R3
;       neg     R3, R3
;       and     R2, R3, R2

        ; Whoops, must also convert -1 to k-1
;       sub     R2, [slotK,A2], R1
;       ge      R1, -1, R1
;       wtag    R1, INT, R1
;       neg     R1, R1
;       and     R1, [slotK,A2], R1
;       sub     R2, R1, R2

        ; Oops: above converted k-1 to 1, not v.v.
        lt      R2, [slotK,A2], R3
        wtag    R3, INT, R3
        neg     R3, R3
        and     R2, R3, R2

        lt      R2, 0, R3
        wtag    R3, INT, R3
        neg     R3, R3
        and     R3, [slotK,A2], R3
        add     R2, R3, R2

        add     R2, frame_size+1, R2
        move    [R2,A2], R1
        or      R1, R0, R1
```

```
        DC      INT:mackPH + mackPC
        and     R1, R0, R3
        eq      R3, R0, R3
        bt      R3, ^check_iter_start
        move    R1, [R2,A2]
        move    fip, ip

check_iter_start:
        not     R0, R0          ; Turn off flags
        and     R1, R0, R1
        move    R1, [R2,A2]
        sub     R2, frame_size+1, R2

        DC      {loop_msg_ref}
        move    ENR, R2
        send2   R0, R0, 0
        sendle  A2, R2, 0
        move    fip, ip
```

Appendix C

Source Code

C.1 Convert Hybrid to Complex J

```
;;; -*- Mode:Common-Lisp; Package:ID-COMPILER; Base:10 -*-
;;; hybrid-to-cj converts hybrid code to complex J-machine code.
;;; The next step is to send it through cj-to-sj to change it to
;;; J-machine s-expressions.

(in-package 'id-compiler)

(defcompiler-module convert-hybrid-to-complex-j id-compiler
  (:input vnd-instructions code-block)
  (:function convert-hybrid-to-cj)
  (:output vnd-instructions code-block)         ; This is a lie
;  (:before-function procedure file-asm-before-def)
;  (:after-function procedure asm-after-def)
;  (:wrapper-macro vnd-file-assembler-wrapper)
;  (:options input-file vnd-output-file vnd-output-file-format)
  )


;;; J-machine constants
;;; Originally, these were numbers.  They are more readable as symbols and
;;; can be replaced by MDPSim.  The constants are needed to know if they're okay literals.
(defconstant *sys-len-bits* 10)

(defconstant sym-tag 'sym)
(defconstant sym 0)

(defconstant int-tag 'int)
(defconstant int 1)

(defconstant fd-tag 'fd)
(defconstant fd 9)

(defconstant boolean-tag 'bool)
(defconstant bool 2)

(defconstant addr-tag 'addr)
(defconstant addr 3)
```

```
; For REFs and SYMBOLs (?)
(defconstant special-tag 'special_tag)
(defconstant special_tag 33)

(defconstant allocate-vector 'allocate_vector)
(defconstant allocate_vector 0)

(defconstant lookup-vector 'lookup_vector)
(defconstant lookup_vector 1)

(defconstant cntt-vector 'cntt_vector)
(defconstant cntt_vector 2)

(defconstant cntt-loop-vector 'cntt_loop-vector)
(defconstant cntt_loop-vector 3)

(defconstant calloc-vector 'calloc_vector)
(defconstant calloc_vector 4)

(defconstant check-iter-vector 'check_iter-vector)
(defconstant check_iter-vector 5)

(defconstant *posIM* 24)
(defconstant *posPC* 25)
(defconstant *maskIM* (expt 1 *posIM*))
(defconstant *maskPC* (expt 1 *posPC*))

(defun convert-hybrid-to-cj (cb)
  (let* ((cj-instructions (convert-hybrid-to-cj-inner (dataflow-graph-root-set cb)
                                                      (dataflow-graph-get cb :frame-descriptor))))
    (setf (dataflow-graph-root-set cb) cj-instructions))
  cb)

(defun convert-hybrid-to-cj-inner (instructions frame-desc)
  (if (null instructions)
      nil
      (let* ((instruction (car instructions))
             (opcode (car instruction))
             ; Get rid of hybrid register references -- ouch
             (operands (mapcar #'transform-hybrid-register (copy-list (cdr instruction))))
             (suspensive-code (mutate-suspensive-operands opcode operands))
             (fn (convert-opcode-to-fn opcode)))
        (if (null fn)
            (my-error :fatal nil (format nil "No opcode for function ~S" opcode)))
        (append
          `((hybrid-instruction ,instruction))
          suspensive-code
          (apply fn frame-desc operands)
          (convert-hybrid-to-cj-inner (cdr instructions) frame-desc)))))

(defvar *conversion-list*)

(defun convert-opcode-to-fn (op)
  (cdr (assoc op *conversion-list*)))

;; Very inefficient
(defun transform-hybrid-register (op)
  (if (and (listp op)
           (eq (car op) :register)
           (numberp (second op)))
      `(:temporary (:base ,(second op)))
      op))
```

```
(defmacro suspensivep (operand)
  `(member :suspensive ,operand))

;;; A few hours with this section could yield some major optimizations,
;;; not to mention what could be done with register allocation.

(defun mutate-suspensive-operands (opcode operands)
  (let ((suspensive-code (mutate-suspensive-operands-inner operands)))
    ; special
    (if (not (eq opcode :continue-test))
        (if suspensive-code
            (cons '(suspensive-instruction)
                  ; remove-duplicates to ensure only one
                  ; check for (:add (:suspensive X) (:suspensive X) Y)
                  (append (remove-duplicates suspensive-code :test #'equal)
                          '((suspensive-check-done))))))))

(defun mutate-suspensive-operands-inner (operands)
  (if (null operands)
      nil
      (append
        (if (suspensivep (car operands))
            (progn
              (setf (car operands) (remove :suspensive (car operands)))
              `((suspensive-operand ,(car operands)))))
        (mutate-suspensive-operands-inner (cdr operands)))))

(defvar *conversion-list*)
(setq *conversion-list* nil)

(defmacro defconversion (hybrid-name hybrid-symbol operands body)
  (progn
    (setq *conversion-list*
          (cons (cons hybrid-symbol hybrid-name)
                *conversion-list*))
    (let ((full-op-list (cons 'frame-desc operands)))
      `(defun ,hybrid-name ,full-op-list
         'frame-desc
         ,body))))

(defun frame-base-offset (operand)
  (if (eq (car operand) :frame)
      (base-offset operand)
      (error :fatal nil "Illegal operand supplied when frame-base value expected.")))

;; Used by cj-to-sj
(defun message-base-offset (operand)
  (if (eq (car operand) :message)
      (base-offset operand)
      (error :fatal nil "Illegal operand supplied when message-base value expected.")))

(defun base-offset (operand)
  (if (eq (car (second operand)) :base)
      (second (second operand))
      (error :fatal nil "Illegal operand supplied when base-offset value expected.")))

(defun literal-base-offset (operand)
  (if (and (eq (car operand) :literal)
           (eq (car (second operand)) :base))
      (second (second operand))
      (error :fatal nil "Illegal operand supplied when literal-base value expected.")))

(defconversion getc :get-context (context-slot return-slot)
  `((reserve (:register scratch))
```

```
    (move (:j-register A2) (:register scratch))
    (wtag (:register scratch) (:literal ,int-tag) (:register scratch))
    (lsh (:register scratch) (:literal ,(- 16 *sys-len-bits*)) (:register scratch))
    (reserve (:register scratch2))
    (move (:j-register NNR) (:register scratch2))
    (add (:register scratch)
         (:register scratch2)
         (:register scratch))
    (free (:register scratch2))
    (wtag (:register scratch) (:literal ,fd-tag) (:register scratch))

    (send0 (:literal 1))
    (send0 (:ref local_getc))
    (send0 ,context-slot)
    (send0 (:register scratch))
    (free (:register scratch))
    (sende0 ,(frame-base-offset return-slot))))

;;; Something should be done to handle falling into a loop
(defconversion label :label (label-name)
  `((label ,label-name)
    (move (:message (:base 1)) (:j-register A2))))

(defun lookup-into (dest)
  (if (eq (car dest) :frame)
      `((move (:literal ,(frame-base-offset dest)) (:j-register R1))
        (call (:literal ,lookup-vector)))))

;; For now, no loops
(defconversion move :move (source dest)
  (append (lookup-into dest)
          `((move ,source ,dest))))

(defconversion move-identity :move-identity (source dest)
  (append (lookup-into dest)
          `((move ,source ,dest))))

(defconversion cntt :continue-test (check-slot cont)
  ; Convert it from (:literal (:symbol :SQ-1)) to (:ref :SQ-1)
  `((move (:ref ,(second (second cont))) (:j-register R0))
    (move (:literal ,(frame-base-offset check-slot)) (:j-register R1))
    (call (:literal ,cntt-vector))))

(defconversion cntn :continue (cont)
  `((send0 (:j-register NNR))
    ; Convert it from (:literal (:symbol :SQ-1)) to (:ref :SQ-1)
    (send0 (:ref ,(second (second cont))))
    (sende0 (:j-register A2))))

(defconversion movr :move-remote (frame-ptr offset value)
  `((send0 ,frame-ptr)
    (send0 (:ref local_movr))
    (send0 ,frame-ptr)
    (send0 ,offset)
    (sende0 ,value)))

;;; This should set a flag
(defconversion terminate :terminate ()
  `((suspend)))

(defconversion le :<= (s1 s2 d)
  (append (lookup-into d)
          `((le ,s1 ,s2 ,d))))
```

```
(defconversion lt :< (s1 s2 d)
  (append (lookup-into d)
          `((lt ,s1 ,s2 ,d))))

(defconversion gt :> (s1 s2 d)
  (append (lookup-into d)
          `((gt ,s1 ,s2 ,d))))

(defconversion ge :>= (s1 s2 d)
  (append (lookup-into d)
          `((ge ,s1 ,s2 ,d))))

(defconversion j-eq := (s1 s2 d)
  (append (lookup-into d)
          `((eq ,s1 ,s2 ,d))))

(defconversion j-neq :<> (s1 s2 d)
  (append (lookup-into d)
          `((neq ,s1 ,s2 ,d))))

(defconversion j-neq2 :/= (s1 s2 d)
  (append (lookup-into d)
          `((neq ,s1 ,s2 ,d))))

(defconversion j-and :and (s1 s2 d)
  (append (lookup-into d)
          `((and ,s1 ,s2 ,d))))

(defconversion j-or :or (s1 s2 d)
  (append (lookup-into d)
          `((or ,s1 ,s2 ,d))))

(defconversion j-sub :- (s1 s2 d)
  (append (lookup-into d)
          `((sub ,s1 ,s2 ,d))))

(defconversion j-add :+ (s1 s2 d)
  (append (lookup-into d)
          `((add ,s1 ,s2 ,d))))

(defconversion j-mul :* (s1 s2 d)
  (append (lookup-into d)
          `((mul ,s1 ,s2 ,d))))

(defconversion j-not :not (s d)
  (append (lookup-into d)
          `((not ,s ,d))))

(defconversion j-abs :abs (s d)
  (append (lookup-into d)
          `((reserve (:register scratch1))
            (reserve (:register scratch2))
            (ash ,s -31 (:register scratch1))
            (xor ,s (:register scratch1) (:register scratch2))
            (sub (:register scratch2) (:register scratch1) ,d)
            (free (:register scratch1))
            (free (:register scratch2)))))

(defconversion j-max :max (a b d)
  `((reserve (:register scratch1))
    (append (lookup-into d)
            (reserve (:register scratch2))                              ; a >= b  |  a < b
            (ge ,a ,b (:register scratch1))                             ; R1: T   |  R1: F
            (wtag (:register scratch1) ,int-tag (:register scratch1))   ; R1: 1   |  R1: 0
```

```
            (neg (:register scratch1) (:register scratch1))            ; R1: -1  |  R1: 0
            (and (:register scratch1) ,a (:register scratch2))         ; R2: a   |  R2: 0
            (not (:register scratch1) (:register scratch1))            ; R1: 0   |  R1: -1
            (and (:register scratch1) ,b (:register scratch1))         ; R1: 0   |  R2: b
            (or (:register scratch2) (:register scratch1) ,d)          ; a       |  b
            (free (:register scratch1))
            (free (:register scratch2)))))

(defconversion j-min :min (a b d)
  (append (lookup-into d)
          `((reserve (:register scratch1))
            (reserve (:register scratch2))                             ; a >= b  |  a < b
            (ge ,a ,b (:register scratch1))                            ; R1: T   |  R1: F
            (wtag (:register scratch1) ,int-tag (:register scratch1))  ; R1: 1   |  R1: 0
            (neg (:register scratch1) (:register scratch1))            ; R1: -1  |  R1: 0
            (and (:register scratch1) ,b (:register scratch2))         ; R2: b   |  R2: 0
            (not (:register scratch1) (:register scratch1))            ; R1: 0   |  R1: -1
            (and (:register scratch1) ,a (:register scratch1))         ; R1: 0   |  R2: a
            (or (:register scratch2) (:register scratch1) ,d)          ; b       |  a
            (free (:register scratch1))
            (free (:register scratch2)))))

;; Not used
(defconversion loop-setup :loop-setup (label-name)
  `(let (frame-size (frame-descriptor-frame-size frame-desc)
         (k-slot (compute-slot-offset t :maximum-iterations))
         (slots-per-iteration (frame-descriptor-next-available-iteration-slot frame-desc))
         (loop-setup-label (gensym 'loop-loop)))
     `((dc (:literal ,(* frame-size (expt 2 *sys-len-bits*))))
       (move (:j-register A2) (:j-register R2))
       (wtag (:j-register R2) ,int-tag (:j-register R2))
       (add (:j-register R2) (:j-register R0) (:j-register R2))
       (wtag (:j-register R2) ,addr-tag (:j-register R2))
       (move (:j-register R2) (:j-register A1))

       (move (:frame (:base ,k-slot)) (:j-register R1))
       (sub (:j-register R1) (:literal 1) (:j-register R1))

       (move ,(+ 2 frame-size) (:j-register R2))
       (add (:j-register R2) (:frame (:base ,k-slot)) (:j-register R2))

       (dc (:literal *maskIM*))
       (or (:j-register R2) (:j-register R0) (:j-register R2))
       (move (:literal 1) (:j-register R3))

       (label ,loop-setup-label)
       (move (:j-register R2) (:frame (:loop A1)))
       (lt (:j-register R3) (:j-register R1) (:j-register R0))
       (add (:j-register R2) (:literal ,frame-n-iterations) (:j-register R2))
       (add (:j-register R3) 1 (:j-register R3))
       (bt (:j-register R0) ,loop-setup-label)

       (dc ,(lognot (logior *maskIM* *maskPC*)))
       (and (:j-register R2) (:j-register R0) (:j-register R2))

       (move (:j-register R2) (:frame (:loop (:j-register R3))))
       (move (:j-register R2) (frame (:loop 0)))

       (move (:frame (:loop 1)) (:j-register R2))
       (and (:j-register R2) (:j-register R0) (:j-register R2))
       (add (:j-register R3) 1 (:j-register R3))
       (move (:j-register R2) (:frame (:loop (:j-register R3)))))))

;; from (:literal (:symbol :SQ-1)) to (:label :SQ-1)
```

```
(defun convert-label (l)
  `(:tagged-literal ,special-tag (:label ,(second (second l)))))

(defconversion brf :branch-false (s1 s2)
  `((bf ,s1 ,(convert-label s2))))

(defconversion brt :branch-true (s1 s2)
  `((bt ,s1 ,(convert-label s2))))

(defconversion brz :branch-zero (s1 s2)
  `((bz ,s1 ,(convert-label s2))))

(defconversion brnz :branch-not-zero (s1 s2)
  `((bnz ,s1 ,(convert-label s2))))

(defconversion br :branch (s1)
  `((br ,(convert-label s1))))

(defconversion ixcc :index-current-context (frame-base dest)
  (append (lookup-into dest)
          `((reserve (:register scratch))
            (move (:j-register A2) (:register scratch))
            (wtag (:register scratch) (:literal ,int-tag) (:register scratch))
            (add (:register scratch)
                 (:literal ,(* (literal-base-offset frame-base)
                               (expt 2 *sys-len-bits*)))
                 (:register scratch))
            (lsh (:register scratch) (:literal ,(- 16 *sys-len-bits*)) (:register scratch))
            (add (:register scratch) (:j-register NNR) (:register scratch))
            (wtag (:register scratch) (:literal ,fd-tag) (:register scratch))
            (move (:register scratch) ,dest)
            (free (:register scratch)))))

;; These are okay because the operands will be suspensive
;; and caught by mutate-suspensive-operand.
(defconversion tst2 :test-2 (s1 s2 dest)
  (append (lookup-into dest)
          `((move (:tagged-literal ,boolean-tag 1) ,dest))))

(defconversion tst1 :test-1 (s1 dest)
  (append (lookup-into dest)
          `((move (:tagged-literal ,boolean-tag 1) ,dest))))

(defconversion stst1 :special-test-1 (s1)
  `((suspensive-instruction)
    (suspensive-operand ,s1)))

(defconversion retc :return-context (source dest)
  (append (lookup-into dest)
          `((move (:tagged-literal ,boolean-tag 1) ,dest))))
```

C.2 Convert Complex J to Simple J

```
;;; -*- Mode:Common-Lisp; Package:ID-COMPILER; Base:10 -*-

;;; cj-to-sj converts complex J-machine code (as produced by hybrid-to-cj)
;;; into J-machine s-expressions.  The s-expressions will correspond on an
;;; exact one-to-one basis with J-machine instructions.  The final step is
;;; to send it through sj-to-j, in the file of that name.

;;; Complex J-machine code differs from J-machine code in several ways:
;;;
;;; X At the beginning of every possibly suspensive instruction,
;;;       (:suspensive-instruction)
;;;   appears.  For each possibly suspensive operand,
;;;       (:suspensive-operand <operand>)
;;;   These must be converted to appropriate code.
;;;
;;; X (:reserve <symbol>) and (:free <symbol>) are used to bind the value
;;;   of the symbol so that (:register <symbol>) is meaningful.  The
;;;   usage is of the form:
;;;        (:reserve (:register scratch))
;;;          .
;;;        (:move (:j-register A2) (:register scratch))
;;;          .
;;;        (:free (:register scratch))
;;;   The usage is purposely verbose, to allow a change of representation,
;;;   as well as error-checking.  (Reserving a second register of the same
;;;   name, using a nonreserved register, and freeing a nonreserved register
;;;   are all errors.)
;;;
;;; X Specific register names are denoted with :j-register, i.e. (:j-register 'A2).
;;;   The only time specific GPRs are used is to set up for CALLs.  This is
;;;   almost certainly a violation of abstraction.  This is a source of potential
;;;   bugs as well if this module trashes those registers.
;;;
;;; X No consideration is made whether the operation can fit in one J-instruction.
;;;   In many cases, it cannot.  For example, this is a legal cj instruction:
;;;        (:add (:frame (:base 6))
;;;              (:literal 82932)
;;;              (:frame (:base 9)))
;;;
;;; X There are both :literal and :tagged-literal operands.

;;; The register allocation is correct and stable, to the best of my knowledge.
;;; It is non-optimal but acceptable.

(in-package 'id-compiler)

;;; For some reason that I can't figure out, I'm having trouble getting neq.

(defmacro neq (a b)
  `(not (eq ,a ,b)))

(defcompiler-module convert-complex-j-to-simple-j id-compiler
  (:input vnd-instructions code-block)          ; A lie
  (:before-function procedure reset-cj-to-sj-system)
  (:function convert-cj-to-sj)
  (:output vnd-instructions code-block))        ; Yuck!  I've got to fix these abstractions

(defun reset-cj-to-sj-system ()
  (setq *j-instructions* nil)
  (setq *virtual-registers* nil)
```

```
  (setq *free-register-list* genl-purpose-regs))

(defun my-error (a b c)
  (print c)
  (break)
  (error a b c))

;;; These are functions to specify basic J-machine characteristics.
(defun make-tagged-literal (l)
  (cond ((numberp l) `(:tagged-literal ,int-tag ,l))
        ((referencep l) `(:tagged-literal ,special-tag ,l))
        ((eq (car l) :tagged-literal) l)
        ; Converts from (:label (:literal (:symbol :foobar))) to
        ; (:tagged-literal special-tag (:label (:symbol :foobar)))
        ((eq (car l) :label)
         (list :tagged-literal special-tag (list :label l)))
;        (list :tagged-literal special-tag (list :label (second (second (second l))))))
        ((eq (car l) :literal)
         (if (listp (second l))
             (if (eq (car (second l)) :integer)
                 `(:tagged-literal ,int-tag ,(second (second l)))
                 (list :tagged-literal special-tag (second l)))
             (list :tagged-literal int-tag
                   (if (listp (second l))
                       (if (eq (car (second l)) :integer)
                           (second (second l))
                           (my-error :fatal nil "Illegal format of literal"))
                       (second l)))))
        (t nil)))



;; Only converts if appropriate
(defun make-tagged-literal-if-appropriate (l)
  (let ((result (make-tagged-literal l)))
    (if result
        result
        l)))

(defun hex-value (h)
  (cond ((and (>= h #\0) (<= h #\9)) (- h #\0))
        ((and (>= h #\A) (<= h #\F)) (+ 10 (- h #\A)))
        ((and (>= h #\a) (<= h #\f)) (+ 10 (- h #\a)))))

(defmacro hex-to-dec (h-string)
  (do ((count (- (length h-string) 1) (- count 1))
       (value 0 (+ (* value 16)
                   (hex-value (char h-string count)))))
      ((< count 0)
       value)))

(defconstant op0-literals (list
                           (cons sym-tag 0)                    ; nil
                           (cons boolean-tag 0)        ; false
                           (cons boolean-tag 1)        ; true
                           (cons int-tag (hex-to-dec "80000000"))
                           (cons int-tag (hex-to-dec "ff"))
                           (cons int-tag (hex-to-dec "3ff"))
                           (cons int-tag (hex-to-dec "ffff"))
                           (cons int-tag (hex-to-dec "fffff"))))

(defun op0-literal-p (l)
  (op0-literal-p-inner l nil))
```

```
(defun op0-extended-literal-p (l)
  (op0-literal-p-inner l t))

(defun tagged-literal-p (op)
  (eq (car op) :tagged-literal))

(defun j-register-p (op)
  (or (eq (car op) :j-register)
      (eq (car (translate-operand op)) :j-register)))

;; To distinguish it from dfcs frames.
(defun j-framep (op)
  (eq (car op) :frame))

(defun j-offset-p (op)
  (or (j-framep op)
      (j-messagep op)
      (j-temporary op)))

(defun j-temporary (op)
  (eq (car op) :temporary))

(defun j-messagep (op)
  (eq (car op) :message))

(defun labelp (op)
  (eq (car op) :label))

(defun j-symbolp ()
  (eq (car op) :symbol))

(defun referencep (op)
  (eq (car op) :ref))

(defun bindingp (op)
  (eq (car op) :binding))

;(print (output-tagged-literal (make-tagged-literal '(:literal int))))

(defun op0-literal-p-inner (l extendedp)
  (if (tagged-literal-p l)
      (let* ((tag (second l))
             (value (if (eq tag int-tag)
                        (eval (third l))                ; To allow us to use symbols instead of ints
                        (third l))))
        (cond ((numberp value)
               (if (member (cons tag value) op0-literals :test #'equal)
                   t
                   (if extendedp
                       (and (eq tag int-tag)
                            (>= value -64)
                            (<= value 63))
                       (and (eq tag int-tag)
                            (>= value -16)
                            (<= value 15)))))
              ((labelp value) nil)      ; Safe assumption
              (t nil)))))

(defun op0-operand-p (op)
  (op0-operand-p-inner op nil))

(defun op0-extended-operand-p (op)
  (op0-operand-p-inner op t))
```

```
(defun op0-operand-p-inner (op extendedp)
  (cond ((j-register-p op)
         (let* ((actual (translate-operand op))
                (value (second actual)))
           (or (genl-purpose-reg-p op)
               (eq value 'A0) (eq value 'A1) (eq value 'A2) (eq value 'A3))))
        ((tagged-literal-p op)
         (op0-literal-p-inner op extendedp))
        ((j-offset-p op)
         (let ((offset (base-offset op)))
           (cond ((numberp offset)
                  (if extendedp
                      (and (< offset 63) (>= offset 0))
                      (and (< offset 16) (>= offset 0))))
                 ((genl-purpose-reg-p offset)
                  t))))))

;; Register-oriented op0 mode
(defun rop0-operand-p (op)
  (let* ((operand (translate-operand op))
         (value (second operand)))
    (and (j-register-p operand)
         (or (genl-purpose-reg-p op)
             (member value '(A0 A1 A2 A3 NNR IP))))))   ; More exist, but these only ones used

(defconstant genl-purpose-regs '(R3 R2 R1 R0))

(defun genl-purpose-reg-p (operand)
  (let ((op (translate-operand operand)))
    (or (bindingp op)
        (and (j-register-p op)
             (member (second op) genl-purpose-regs)))))

(defun basic-add (arg1 &rest args)
  (+ (if arg1 1 0)
     (count t args)))

;;; Current register scheme due in part to Nate.

;;; This system is still primitive.  Some notable omissions:
;;;     - It might reload a register with a value already in it.

(defvar *free-register-list*)
(setq *free-register-list* genl-purpose-regs)

(defvar symbols-bound-to-regs)
(setq symbols-bound-to-regs nil)

(defun request-register-inner ()
  (if (null *free-register-list*)
      (my-error :fatal nil "No registers available in request-register-inner")
      (let* ((temp (remove 'R0 *free-register-list*))
             (reg (if (null temp) 'R0 (car temp))))
        (setq *free-register-list* (remove reg *free-register-list*))
        reg)))

(defun request-appropriate-register (item)
  (if (genl-purpose-reg-p item)
      (my-error :fatal nil "Reg-reg move requested!"))
  (if (and (tagged-literal-p item)
           (not (op0-extended-literal-p item)))
      (if (member 'R0 *free-register-list*)
          (progn (setf *free-register-list* (remove 'R0 *free-register-list*))
                 (setf symbols-bound-to-regs (cons (cons (gensym 'reg) 'R0) symbols-bound-to-regs))
```

```
            `(:binding ,(caar symbols-bound-to-regs)))
        ;; If we get here, we need to slide R0 into another register
        (let* ((r0-pair (rassoc 'R0 symbols-bound-to-regs))
               (new-reg (request-register-inner))     ; Get another register
               (cur-name (gensym)))           ; Name to return with new register
          ; Emit the move -- to a global??
          (if (null r0-pair)
              (my-error :fatal nil "R0 invariant violated"))
          (emit-j-instruction `(move (:j-register R0) (:j-register ,new-reg)))
          (setf (cdr r0-pair) new-reg)
          (setq symbols-bound-to-regs
                (cons (cons cur-name 'R0)
                      symbols-bound-to-regs))
          `(:binding ,cur-name)))
      (request-any-register)))

(defun request-any-register ()
  (let ((reg (request-register-inner)))
    (setq symbols-bound-to-regs (cons (cons (gensym 'reg) reg) symbols-bound-to-regs))
    `(:binding ,(caar symbols-bound-to-regs))))

(defun return-register (reg)
  (if (eq (car reg) :binding)
      (let ((pair (assoc (second reg) symbols-bound-to-regs)))
        (if (null pair)
            (my-error :fatal nil "Illegal binding freed in return-register")
            (let ((actual (cdr pair)))
              (setf *free-register-list* (cons actual *free-register-list*))
              (setq symbols-bound-to-regs (remove pair symbols-bound-to-regs)))))
      (my-error :fatal nil "Illegal register return")))

(defun binding-to-register (symbol)
  (if (and (listp symbol)
           (eq (car symbol) :binding))
      (let ((pair (assoc (second symbol) symbols-bound-to-regs)))
        (if (null pair)
            (my-error :fatal nil "Binding not found")
            `(:j-register ,(cdr pair))))))


;;; Emit commands

;; This "forces" register assigments when the code is emitted.
(defun translate-operand (op)
  (if (listp op)
      (cond ((null op) nil)
            ((eq (car op) :binding) (binding-to-register op))
            ((eq (car op) :register) (translate-virtual-register op))
            (t (cons (translate-operand (car op))
                     (translate-operand (cdr op)))))
      op))

(defvar *j-instructions* nil)
(setq *j-instructions* nil)

(defun emit-j-instruction (inst &key (pass-through nil))
  (let* ((opcode (car inst))
         (operands (if pass-through
                       (cdr inst)
                       (mapcar #'translate-operand (cdr inst))))
         (instruction (cons opcode operands)))
    (setq *j-instructions* (append *j-instructions*
                                   (list instruction)))
    ; For trace purposes, just return latest new instruction
```

```
    instruction))

(defun emit-j-instructions (ilist)
  (mapcar #'emit-j-instruction ilist))


;;; Movement routines

(defun make-legal-move (source dest)
  (if (or (genl-purpose-reg-p source)
          (genl-purpose-reg-p dest))
      ; At least one is a register
      (make-legal-move-with-register source dest)
      (let ((register (request-appropriate-register source)))
        (make-legal-move source register)
        (make-legal-move register dest)
        (return-register register))))

;;; Possible operands include:
;;;     (:register ..)
;;;     (:j-register ..)
;;;     (:binding ..)
;;;     (:frame (:base #))
;;;     (:tagged-literal # #)

(defun make-legal-move-with-register (source dest)
  (if (genl-purpose-reg-p source)
      (if (or (rop0-operand-p dest)
              (op0-extended-operand-p dest))
          (emit-j-instruction `(move ,source ,dest))
          ;; If we get here, source is a register, but dest is too big
          (make-legal-big-move source dest))
      (if (genl-purpose-reg-p dest)
          (if (or (rop0-operand-p source)
                  (op0-extended-operand-p source))
              (emit-j-instruction `(move ,source ,dest))
              (make-legal-big-move source dest)))))

;;; make-legal-big-move called when one operand is a register and the
;;; other is something that can't be represented in op0 or register-oriented
;;; op0 mode, such as a big literal or a frame value with a large offset.
(defun make-legal-big-move (source dest)
  (if (genl-purpose-reg-p source)
      ;; destination must be frame (or equiv.) (i.e. can't be literal)
      (let* ((offset (base-offset dest))
             (tagged-offset (make-tagged-literal offset))
             (reg (request-appropriate-register tagged-offset))
             (new-operand (replace-offset reg dest)))
        (make-legal-move tagged-offset reg)
        (make-legal-move source new-operand)
        (return-register reg))
      ;; If we get here, dest must be a gpr
      (cond ((tagged-literal-p source)
             (if (op0-literal-p source)
                 (emit-j-instruction (list 'move source dest))
                 (let ((actual-dest (translate-operand dest)))
                   (if (equal actual-dest '(:j-register R0))
                       (emit-j-instruction `(dc ,source))
                       (message :fatal nil "R0 not reserved when required")))))
            (t (my-error :fatal nil "Unhandled case in make-legal-big-move")))))


(defstruct code-bundle
  operand-list
```

```
  regs-to-be-freed)

(defun bundle-return-registers (bundle)
  (mapcan #'return-register (code-bundle-regs-to-be-freed bundle))
  (setf (code-bundle-regs-to-be-freed bundle) nil)
  bundle)

(defun convert-cj-to-sj (cb)
  (mapc #'make-legal (dataflow-graph-root-set cb))
  (setf (dataflow-graph-root-set cb) *j-instructions*)
  cb)

(defun make-legal (instruction)
  (let* ((opcode (car instruction))
         (operands (if (eq opcode 'hybrid-instruction)
                       (cdr instruction)
                       (mapcar #'make-tagged-literal-if-appropriate (cdr instruction))))
         (num-ops (length operands))
         (instruction (cons opcode operands)))
    (if (pseudo-op-p opcode)
        (process-pseudo-op opcode operands)
        (cond ((= num-ops 0)                    ; Typically, suspend
               (emit-j-instruction instruction))
              ((= num-ops 1)                    ; Typically, send or branch
               (if (eq opcode 'br)
                   (make-branch opcode operands)
                   (make-into-form opcode
                                   operands
                                   (cons #'ext-op0 'source))))
              ;; Shouldn't something for branches be here?
              ((= num-ops 2)                    ; Typically move, unary op, or bcc
               (cond ((equal opcode 'move)
                      (make-legal-move (first operands) (second operands)))
                     ((or (equal opcode 'neg) (equal opcode 'not) (equal opcode 'rtag))
                      (make-into-form opcode
                                      operands
                                      (cons #'ext-op0 'source)
                                      (cons #'gpr 'dest)))
                     ((member opcode '(bf bt bz bnz))
                      (make-branch opcode operands))
                     (t
                      (message :fatal nil "Illegal opcode in make-legal"))))
              ((= num-ops 3)                    ; Typically binary op (all have same format)
               ;; It should try exchanging the first two operands to execute more cheaply
               (make-into-form opcode
                               operands
                               (cons #'gpr 'source)
                               (cons 'op0 'source)
                               (cons 'gpr 'dest)))))))

;; Some conditional branches can't be encoded into one instruction; additionally, in
;; my simple one-pass assembler, I can't determine displacements, etc.  Hence, all
;; jumps will be converted in a pessimistic way, e.g.
;;     bz R1, label1
;;     bnz R1, new_label
;;     br label1
;;   new_label:
;; The types of branches are: bf, bt, bz, bnz, bnil, bnnil.
;; (The last two aren't used by hybrid stuff but are in for completeness.)

(defvar branch-opposites '((bf . bt) (bz . bnz) (bnnil . bnil)
                           (bt . bf) (bnz . bz) (bnil . bnnil)))

(defun make-branch (opcode operands)
```

```
  (if (eq opcode 'br)                          ; Absolute branch
      (make-legal `(move ,(car operands) (:j-register ip)))
      (let (;(new-label (gensym 'jog))
            (opposite-opcode (cdr (assoc opcode branch-opposites)))
            (condition (first operands))
            (original-label (second operands)))
; The following line would give us an infinite loop!
;       (make-legal `(,opposite-opcode ,condition (:label ,new-label)))
; Instead, do a violation of abstraction:
        (make-into-form opposite-opcode `(,condition (:tagged-literal ,int-tag 2))
                        (cons #'gpr 'source) (cons #'ext-op0 'source))
        (make-legal `(br ,original-label))
        (make-legal `(align))
;       (make-legal `(label (:literal (:symbol ,new-label))))
        )))

(defun replace-offset (reg operand)
  (list (car operand)
        (list
          (caadr operand)
          reg)))

(defun gpr (arg dir bundle)
  (if (genl-purpose-reg-p arg)
      (make-code-bundle
        :operand-list (append (code-bundle-operand-list bundle) (list arg))
        :regs-to-be-freed (code-bundle-regs-to-be-freed bundle))
      (make-move-with-register arg dir bundle)))

(defun make-move-with-register (arg dir bundle)
  (let ((reg (request-appropriate-register arg)))
    (if (eq dir 'source)
        (progn
          (make-legal-move arg reg)
          (make-code-bundle
            :operand-list (append (code-bundle-operand-list bundle) (list reg))
            :regs-to-be-freed (append (code-bundle-regs-to-be-freed bundle) (list reg))))
        ;; dest
        (progn
          (make-legal-move reg arg)
          (make-code-bundle
            :operand-list (append (code-bundle-operand-list bundle) (list reg))
            :regs-to-be-freed (append (code-bundle-regs-to-be-freed bundle) (list reg)))))))

(defmacro base-tagged-offset (a)
  `(make-tagged-literal (base-offset ,a)))

(defun op0 (arg dir bundle)
  (if (op0-operand-p arg)
      (make-code-bundle
        :operand-list (append (code-bundle-operand-list bundle) (list arg))
        :regs-to-be-freed (code-bundle-regs-to-be-freed bundle))
      (make-big-item-into-gpr arg dir bundle)))

(defun make-big-item-into-gpr (arg dir bundle)
  ;; There are two possibilities:
  ;; (1) it is a frame reference that we could convert (in which case direction is irrelevant)
  (if (eq (car arg) :frame)
      (let* ((value (base-tagged-offset arg))
             (reg (request-appropriate-register value)))
        (make-legal-move value reg)
        (make-code-bundle
          :operand-list (append (code-bundle-operand-list bundle) (list (replace-offset reg arg)))
          :regs-to-be-freed (append (code-bundle-regs-to-be-freed bundle) (list reg))))
```

```
      ;; (2) it must be stored into a separate register
      (make-move-with-register arg dir bundle)))

(defun ext-op0 (arg dir bundle)
  (if (op0-extended-operand-p arg)
      (make-code-bundle
        :operand-list (append (code-bundle-operand-list bundle) (list arg))
        :regs-to-be-freed (code-bundle-regs-to-be-freed bundle))
      (make-big-item-into-gpr arg dir bundle)))

(defun guaranteed-ok (arg dir bundle)
  bundle)

(defun process-operand-if-source (operands patterns count bundle)
  (if (>= count (length operands))
      bundle
      (let ((op (nth count operands))
            (pat (nth count patterns)))
        (if (eq (cdr pat) 'source)
            (apply (car pat) (list op 'source bundle))
            bundle))))

(defun symbol> (x y)
  (string> (string x) (string y)))

(defun process-operand-if-dest (op pat bundle)
  ;; Process only if destination
  (if (neq (cdr pat) 'dest)
      nil
      (list
        (if (genl-purpose-reg-p op)
            op
            (let ((reg (request-any-register)))
              (setf (code-bundle-regs-to-be-freed bundle)
                    (cons reg (code-bundle-regs-to-be-freed bundle)))
              reg)))))

;; Unfortunately, it seems we have to code in some specifics to keep
;; the code from being too complex.  The assumptions are:
;;
;; - An instruction has up to two sources.
;;
;; - The last operand is the only one that can be a destination.
;;   If it is a destination, it is also a gpr (except for moves,
;;   which are handled specially).

(defun make-into-form (opcode operands &rest pattern)
  ; First, check that same # of operands as patterns
  (if (/= (length operands)
          (length pattern))
      (my-error :fatal nil "Not enough operands for pattern")
      ; Generate the code for up to two sources and up to one dest
      (let* ((step-one (process-operand-if-source operands pattern 0 (make-code-bundle)))
             (bundle (process-operand-if-source operands pattern 1 step-one))
             (dest-reg (process-operand-if-dest (car (last operands))
                                                (car (last pattern))
                                                bundle)))       ;! Bundle mutated !
        ; Emit instruction
        (emit-j-instruction (cons opcode (append (code-bundle-operand-list bundle)
                                                 dest-reg)))
        ; Emit the code (if any) to put result into destination
        (if (and dest-reg
                 (not (equal dest-reg (last operands))))        ; eq and eql too strong for lists
            (make-legal-move (car dest-reg) (car (last operands))))
```

```
        ; Free registers
        (mapcar #'return-register (code-bundle-regs-to-be-freed bundle)))))


;;; Pseudo-op functions, for :reserve and :free, :suspensive*, and :label

;;; - (:reserve <symbol>) and (:free <symbol>) are used to bind the value
;;;   of the symbol so that (:register <symbol>) is meaningful.  The
;;;   usage is of the form:
;;;        (:reserve (:register scratch))
;;;          .
;;;        (:move (:j-register A2) (:register scratch))
;;;          .
;;;        (:free (:register scratch))
;;;   The usage is purposely verbose, to allow a change of representation,
;;;   as well as error-checking.  (Reserving a second register of the same
;;;   name, using a nonreserved register, and freeing a nonreserved register
;;;   are all errors.)

(defvar *virtual-registers*)
(defvar *pseudo-op-list*)

(defun pseudo-op-p (op)
  (assoc op *pseudo-op-list*))

(defun process-pseudo-op (opcode operands)
  (apply (cdr (assoc opcode *pseudo-op-list*)) (list operands)))

(defun reserve-virtual-register (operands)
  (let* ((operand (first (car operands)))
         (name (second (car operands))))
    (if (neq operand :register)
        (my-error :fatal nil "Illegal :reserve syntax")
        ; Check if it's already allocated
        (if (assoc name *virtual-registers*)
            (my-error :fatal nil "An attempt was made to re-allocate a virtual register")
            (let ((reg (request-any-register))) ; This is a TEMPORARY measure -- it might need RO
              (setq *virtual-registers*
                    (cons (cons name reg)
                          *virtual-registers*))))))
  nil)

(defun free-virtual-register (operands)
  (let ((operand (first (car operands)))
        (name (second (car operands))))
    (if (neq operand :register)
        (my-error :fatal nil "Illegal :free syntax")
        ; Check if it's already allocated
        (if (assoc name *virtual-registers*)
            (progn
              (return-register (cdr (assoc name *virtual-registers*)))
              (setq *virtual-registers*
                    (remove (assoc name *virtual-registers*)
                            *virtual-registers*)))
            (my-error :fatal nil "An attempt was made to free an unallocated virtual register"))))
  nil)

;; There are two things this could be called for:
;;     (:register <name>)
;; or
;;     (:register #)
;; The meanings are very different.  The first was a temporary assigned by (my) hybrid-to-cj.
;; The latter was a temporary assigned by Iannucci's generate-vnd-instruction.  Both map to
;; the same thing however.  For now, use [#,A0] for the latter.  Inefficient, but correct.
```

```
;; I implement this in hybrid-to-cj but describe it here, because the real fix should be here.
;; A solution I considered but which is NOT implemented:
;; Because reserve & free are emitted for the first type and *omitted*
;; for the second, we have to assume a little: A first access is an implicit reserve, and a
;; second is an implicit free.  This matches how Iannucci uses registers (I think!) for
;; non-loop-setup.

(defun translate-virtual-register (ref)
  (let ((name (second ref)))
    (translate-operand (cdr (assoc name *virtual-registers*)))))

;; Originating J-machine code here might be something of a violation of abstraction.

(defvar suspensive-binding)

(defun make-suspensive-instruction-code (dummy)
  (let* ((name (gensym 'suspensive))
         (label `(:literal (:symbol ,name)))
         (ref (make-tagged-literal `(:ref ,name))))
  (make-legal `(label ,label))
  (setq suspensive-binding (request-appropriate-register ref))  ; R0
  (make-legal `(move (:message (:base 1)) (:j-register A2)))
  (make-legal-move ref suspensive-binding)))
;  (align)
;  (make-legal `(move (:j-register ip) (:message (:base 0))))
;  (make-legal `(move (:message (:base 1)) (:j-register A2))))

;; This is inefficient.
(defun make-presence-check (operands)
  (let ((op (car operands))
        (reg (request-any-register)))
    (make-legal `(rtag ,op ,reg))
    (return-register reg)))

(defun end-suspensive-part (dummy)
  (return-register suspensive-binding)
  (setq suspensive-binding nil))

(defun handle-label (operands)
  (emit-j-instruction (list 'label (car operands))))

(defun pass-through-hybrid-instruction (operands)
  (emit-j-instruction `(hybrid-instruction ,@operands) :pass-through t))

(setq *pseudo-op-list* (list (cons 'reserve #'reserve-virtual-register)
                             (cons 'free #'free-virtual-register)
                             (cons 'suspensive-instruction #'make-suspensive-instruction-code)
                             (cons 'suspensive-operand #'make-presence-check)
                             (cons 'suspensive-check-done #'end-suspensive-part)
                             (cons 'label #'handle-label)
                             (cons 'hybrid-instruction #'pass-through-hybrid-instruction)))
```

C.3 Convert Simple J to Assembly

```
;;; -*- Mode:Common-Lisp; Package:ID-COMPILER; Base:10 -*-

;;;; sj-to-asm.lisp converts MDP code from s-expressions into format suitable for MDPSim.
;;;;     - Convert from s-expressions to strings,
;;;;       which includes putting in commas and newlines
;;;;     - Replacing characters like ":" and "-" with "_" to
;;;;       make legal MDP identifiers.
;;;;     - Handles references, labels, and symbols.

(in-package 'id-compiler)

(defcompiler-module convert-sexp-j-to-asm id-compiler
  (:input vnd-instructions code-block)          ; A lie
  (:before-function procedure reset-sj-to-asm-system)
;  (:options vnd-output-file)
  (:function convert-sj-to-asm))

(defmacro cat (&rest args)
  `(concatenate 'string ,@args))

(defvar *output-string*)
(defvar *msg-ref-list*)
(defvar *ip-ref-list*)
(defvar operand-list)

(defun reset-sj-to-asm-system ()
  (setq *ip-ref-list* nil)
  (setq *msg-ref-list* nil)
  (setq *output-string* ""))

(defun make-j-string (sym)
  (let ((s (copy-seq (my-string sym))))
    (make-j-string-inner s 0)
    s))

(defun make-j-string-inner (s index)
  (if (< index (length s))
      (let ((c (char s index)))
        (if (or (eql c #\:)
                (eql c #\-))
            (setf (char s index) #\_))
        (make-j-string-inner s (1+ index)))))

(defun asm-output-opcode (opcode)
  (setq operand-list nil)
  (if opcode
      (setq *output-string* (cat *output-string* (format nil "~%~T~A~T" (string opcode))))
      (setq *output-string* (cat *output-string* (format nil "~%")))))

(defun asm-output-label (l)
  (asm-output-opcode nil)
  (asm-output-operand (cat (make-j-string (second (third l))) ":"))
  (asm-output-end-line))

(defun asm-output-align ()
  (asm-output-opcode nil)
  (asm-output-operand ":")
  (asm-output-end-line))

(defun asm-output-comment (text)
```

```
  (setq *output-string* (cat *output-string* (format nil "~%~%;~S" text))))

(defun asm-output-operand (operand)
  (setq operand-list (nconc operand-list (list operand))))

(defun asm-output-end-line ()
  (asm-output-end-line-inner (length operand-list) operand-list))

(defun asm-output-end-line-inner (len ops)
  (if (> len 0)
      (progn
        (setq *output-string* (cat *output-string* (first ops)))
        (if (> len 1)
            (setq *output-string* (cat *output-string* ", ")))
        (asm-output-end-line-inner (- len 1) (cdr ops)))))

(defvar *current-frame-descriptor*)

(defun convert-sj-to-asm (cb)
  (let ((name (dataflow-graph-get cb :procedure-name))
        (instructions (dataflow-graph-root-set cb)))
    ; Yuck: Do this right.  On second thought, don't bother.
    (setq *current-frame-descriptor* (dataflow-graph-get cb :frame-descriptor))
    (mapc #'convert-sj-instruction-to-asm instructions)
    (let ((filename (open (make-pathname :type "MDP"
                                         :defaults (cat "o:>ellens>" (string name)))
                          :direction :output)))
      (princ *output-string*)
      ; Output the module
      (format filename "module ~a~%" name)
      (princ *output-string* filename)
      (format filename "~%end~%")
      ; Output the references
      (loop for ref in (set-difference *msg-ref-list* '(local_movr local_getc))
            doing (format filename "ref ~a_msg_ref = MSG:(((~a+~a_loc)<<~d))+2~%"
                          (make-j-string ref)
                          (make-j-string ref)
                          name
                          *sys-len-bits*))
      (loop for label in *ip-ref-list*
            doing (format filename "ref ~a_ip_ref = IP:(((~a+~a_loc)<<~d))+ABSOLUTE~%"
                          (make-j-string label)
                          (make-j-string label)
                          name
                          *sys-len-bits*))
      ; Bogus for loops
      (format filename "ref ~a_codeblock_ref = CB:(~a_loc<<16)+~D~%"
              (dataflow-graph-get cb :procedure-name)
              (dataflow-graph-get cb :procedure-name)
              (frame-descriptor-next-available-scratch-slot *current-frame-descriptor*))
      (close filename))))

(defun convert-sj-instruction-to-asm (instruction)
  (let ((operator (car instruction)))
    (cond ((eq operator 'label)                ; special cases
           (asm-output-label (cadr instruction)))
          ((eq operator 'align)
           (asm-output-align))
          ((eq operator 'hybrid-instruction)
           (begin-hybrid-instruction-conversion (cdr instruction)))
          (t
           (asm-output-opcode operator)
           (mapc #'convert-sj-operand-to-asm (cdr instruction))
           (asm-output-end-line)))))
```

```
(defun begin-hybrid-instruction-conversion (text)
  (asm-output-comment text))

(defun convert-sj-operand-to-asm (operand)
  (asm-output-operand
    (case (car operand)
      ((:tagged-literal) (output-tagged-literal operand))
      ((:j-register) (my-string (second operand)))
      ((:frame) (format nil "[~S,A2]" (cadadr operand)))
      ((:message) (format nil "[~S,A3]" (cadadr operand)))
      ((:temporary) (format nil "[~S,A0]" (cadadr operand)))
     )))

(defun output-tagged-literal (operand)
  (let ((tag (second operand)))
    (if (eq tag special-tag)
        ; Everything as REFs not labels (labels would be more appropriate for branches)
        (cond ((eq (car (third operand)) :code-block)
               ; It goes without saying that the code-block ref will be output
               (cat "{" (make-j-string (second (third operand))) "_codeblock_ref}"))
              ((eq (car (third operand)) :ref)
               (setq *msg-ref-list* (remove-duplicates (cons (second (third operand))
                                                             *msg-ref-list*)))
               (cat "{" (make-j-string (second (third operand))) "_msg_ref}"))
              ((eq (car (third operand)) :label)
               (setq *ip-ref-list* (remove-duplicates (cons (second (third operand)) *ip-ref-list*)))
               (cat "{" (make-j-string (second (third operand))) "_ip_ref}"))
              (t
               (break)))
        (cond ((eq tag int-tag)
               (format nil "~D" (third operand)))
              ((and (eq tag boolean-tag) (numberp (third operand)))
               (if (= 0 (third operand))
                   "false"
                   "true"))
              (t
               (format nil "~A:~D" (string tag) (third operand)))))))

(defun my-string (x)
  (if (numberp x)
      (format nil "~D" x)
      (string x)))
```

Bibliography

[Arvind and Nikhil 1988] Arvind and Rishiyur S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. Computation Structures Group Memo 271. MIT Laboratory for Computer Science, Cambridge, MA, 1988.

[Dally 1988a] Dally, William J. The J-Machine: System Support for Actors. In Hewitt, Carl, and Agha Gul, editors, *Concurrent Object Programming for Knowledge Processing: An Actor Perspective*, MIT Press, Cambridge, MA, 1989.

[Dally et al 1988b] Dally, William, et al. Message-Driven Processor Architecture. MIT Artificial Intelligence Lab Memo 1069, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1988.

[Gaudiot and Bic 1989] Gaudiot, Jean-Luc and Lubomir Bic. Data-Flow: A Status Report. *Computer Architecture News*, December 1989, pages 111-118.

[Horwat 1989] Horwat, Waldemar. Concurrent Smalltalk on the Message-Driven Processor. Master's Thesis, Department of EECS, MIT, 1989.

[Horwat and Totty 1987] Horwat, Waldemar and Brian Totty. Message-Driven Processor Simulator. MIT Concurrent VLSI Architecture Memo 5, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1987.

[Iannucci 1988] Iannucci, Robert Alan, A Dataflow / von Neumann Hybrid Architecture. Technical Report MIT/LCS/TR-228, MIT Laboratory for Computer Science, Cambridge, MA, 1988. (PhD Thesis, Department of EECS, MIT.)

[Nikhil 1988] Nikhil, Rishiyur S., ID Version 88.1 Reference Manual. Technical Report Computation Structures Group Memo 284, MIT Laboratory for Computer Science, Cambridge, MA, 1988.

[Papadopoulos 1988] Papadopoulos, Gregory Michael. Implementation of a General Purpose Dataflow Multiprocessor. Technical Report MIT/LCS/TR-432, MIT Laboratory for Computer Science, Cambridge, MA, 1989. (PhD Thesis, Department of EECS, MIT.)

[Papadopoulos and Culler 1990] Papadopoulos, Gregory M., and David E. Culler, Monsoon: An Explicit Token Store Architecture. In *Proceedings of the* 17th *International Symposium on Computer Architecture*, Seattle, Washington, May 1990.

[Sakai et al 1989] Sakai, Shuichi; Yamaguchi, Yoshinori; Hiraki, Kei; Kodama, Yuetsu; and Yuba, Toshitsugu. An Architecture of a Dataflow Single Chip Processor. *Proceedings of the* 16th *Annual International Symposium on Computer Architecture*, Jerusalem, Israel, 1989, pages 46–53.

[Spertus 1989] Spertus, Ellen. Preliminary Dataflow on the MDP. MIT Concurrent VLSI Archtiecture Memo 21, MIT Artificial Intelligence Laborary, Cambridge, MA, 1989.

[Traub 1986a] Traub, Kenneth R. A Compiler for the MIT Tagged-token Dataflow Architecture. Technical Report MIT/LCS/TR-370, MIT Laboratory for Computer Science, Cambridge, MA, 1986.

[Traub 1989] Traub, Kenneth R., Compilation at Partitioning: A New Approach to Compiling Non-Strict Functional Languages. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, London, September 1989.

[Traub 1986b] Traub, Kenneth R. A Dataflow Compiler Substrate. Computation Structures Group Memo 261, MIT Laboratory for Computer Science, Cambridge, MA, 1986. (Master's Thesis, Department of EECS, MIT.)

[Traub 1988] Traub, Kenneth R., Sequential Implementation of Lenient Programming Languages. Technical Report MIT/LCS/TR-417. MIT Laboratory for Computer Science, Cambridge, MA, September 1988. (PhD Thesis, Department of EECS, MIT.)

This blank page was inserted to preserve pagination.

CS-TR Scanning Project
Document Control Form

Date : 6/22/95

Report # AI-TR-1233

Each of the following should be identified by a checkmark:
Originating Department:

☒ Artificial Intellegence Laboratory (AI)
☐ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 137(145-IMAGES)
Not to include DOD forms, printer instructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☒ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:

☒ DOD Form(2) ☐ Funding Agent Form ☒ Cover Page
☒ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description : Page Number:
IMAGE MAP (1-8) PAGES #'ED i-viii (INCLUDING TITLE PAGE)
(9-137) PAGES #'ED 1-129
(138-142) SCANCONTROL, COVER, SPINE, DOD (2)
(143-145) TRGT'S (3)

Scanning Agent Signoff:
Date Received: 6/22/95 Date Scanned: 6/28/95 Date Returned: 6/29/95

Scanning Agent Signature: Michael W. Cook

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AI-TR 1233	2. GOVT ACCESSION NO. AD-A228612	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Dataflow Computation for the J-Machine		5. TYPE OF REPORT & PERIOD COVERED technical report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Ellen Spertus		8. CONTRACT OR GRANT NUMBER(s) N00014-87-K-0825 N00014-88-K-0738 N00014-85-K-0124 MIP-8657531
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE June 1990 13. NUMBER OF PAGES 129
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

parallel processing
compilation
dataflow

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The dataflow model of computation exposes and exploits parallelism in programs without requiring programmer annotation; however, instruction-level dataflow is too fine-grained to be efficient on general-purpose processors. A popular solution is to develop a "hybrid" model of computation where regions of dataflow graphs are combined into sequential blocks of code. I

(continued on back)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Block 20 continued:

have implemented such a system to allow [illegible] programs, leaving exposed a high amount of parallelism — such as among loop iterations. I describe this system and provide an analysis of its strengths and weaknesses and [illegible], along with ideas for improvement.

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 6/28/1995

M.I.T. Libraries
Document Services