Execution of Dataflow Programs on General-Purpose Hardware

by

Ellen Spertus

S.B., Massachusetts Institute of Technology

(1990)

Submitted to the Department of Electrical Engineering and Computer Science

In Partial Fulfillment of the Requirements for the Degree of

Master of Science

in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

August 1992

Signature of Author ____

Department of Electrical Engineering and Computer Science

August 7, 1992

Certified by ____

Dr. William J. Dally

Associate Professor of Electrical Engineering and Computer Science

Thesis Supervisor

Accepted by ____

Prof. Campbell L. Searle

Chairman, Departmental Committee on Graduate Students

Execution of Dataflow Programs on General-Purpose Hardware

by

Ellen Spertus

Submitted to the
Department of Electrical Engineering and Computer Science
on August 7, 1992, in partial fulfillment of
the requirements for the Degree of Master of Science in
Electrical Engineering and Computer Science

Abstract

Fine-grained programming is important to parallel computing because it covers up the unavoidable latency occuring in large systems. Dataflow computing, where small groups of instructions can be sequentialized into threads, is an extreme of the fine-grained or multi-threading approaches. Originally, dataflow programs were run on specialized hardware. The research described in this document unites advances in compiling dataflow programs to run on non-specialized hardware with the architectural advances of the J-Machine, a general-purpose massively-parallel computer.

Thesis Supervisor: Dr. William J. Dally
Title: Associate Professor of Electrical Engineering and Computer Science

Keywords: Compiling, Multi-threading, Dataflow, Fine-Grained Computation, Parallelism.

Acknowledgments

The Berkeley TAM group was tremendously helpful. David Culler, Seth Copen Goldstein, Klaus Erik Schauser, and Thorsten von Eicken not only gave me complete access to their software and showed me how to use and modify it but made modifications for me and took the time to critique my work.

I also received help from several members of the MIT Computation Structures Group: George Andy Boughton, Alexandro Caro, Mike Flaster, R. Paul Johnson, and Yuli Zhou. They too shared their compiler.

As always, members of the MIT Concurrent VLSI Architecture group were wonderful. Mike Noakes helped me with the idiosyncrasies of the hardware. Scott Furman was also helpful and maintained the simulator, monitor, and floating point libraries. Brennan Gaunce was always willing to discuss compiler optimizations, Debby Wallach helped me with LaTeXand Unix problems, and Lisa Sardegna always helped me with whatever random thing I stopped by her office for. My officemates Todd Dampier, Waldemar Horwat, and Kathy Knobe put up with my quirks and provided random help, and Kathy was like a second advisor to me. Anne McCarthy of the Computer Architecture Group was also helpful.

Eric Brewer of the MIT Parallel Software Research Group shared with me the graph generation tools he created with Chris Dellarocas. The graphs I produced were useful not just as illustrations for this document but in helping me understand where cycles were spent. I used the Alewife group's computer to run these programs, with the help of Kirk Johnson, John Kubiatowicz, Beng-Hong Lim, and Chris Metcalf.

Other MIT students who gave me useful feedback were Nate Osgood, Fred Chong, and Bob Gruber.

I am grateful to my advisor, Bill Dally, who managed to keep my work swapped into his mind despite all of his other projects. His encouragement and assistance made this work possible. My graduate counselor Bill Weihl also gave me general guidance and support, beyond the call of duty.

Finally, I want to thank my parents for always believing in me and encouraging me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This document describes a system I built to execute dataflow programs on the J-Machine, a massively-parallel general-purpose parallel computer. I designed and implemented two different methods of mapping programs compiled for the Berkeley Threaded Abstract Machine (TAM) [7] to the J-Machine. The first approach used the same data and control structures as TAM, while the second approach restructured TAM programs to take advantage of the J-Machine's support for fine-grained computing, specifically its hardware support for buffering incoming messages.

The two implementations are functional and support almost the entire Id language, compiling programs to run on a multi-processor J-Machine (although most of the runs were performed on the simulator for ease of profiling). It was found that restructuring TAM programs significantly improved their performance, implying that the J-Machine's mechanisms for supporting fine-grained parallelism are successful. While the compiler was for a dataflow language, the results apply equally well to any fine-grained programming system.

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, run-time overhead is unnecessarily high. On the other hand, dataflow architectures, with their per instruction synchronization, are more tolerant of latency than von Neumann machines: While one task is waiting for data, tasks for which data is ready can run, keeping the processor busy. The motivation for executing dataflow programs on non-dataflow processors 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 predetermined, 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 [18], a feature that is retained by implementations for non-dataflow machines.

1.1.1 Id

Id is a primarily-functional lenient language developed in the Computation Structures Group of the MIT Laboratory for Computer Science for programming dataflow and other parallel computers. [14] is a reference for the latest version. A quick overview of pertinent features of the language is presented here.

[1] See [12] for criticism of dataflow processors and [4, 13] for further justification of incorporating dataflow ideas into von Neumann computing.

Types

Id is a statically-typed language, although users need not always explicitly provide type information. Additionally, Id allows polymorphism. Primitive types include characters, booleans, integers, floats, strings, and symbols. Type constructors can be used to build array, list, tuple, function, algebraic, and record types.

Function Application

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

$$f\ a_1 ... a_n$$

Id supports currying: If function f "expects" two arguments, fa 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. Currying causes additional overhead in run-time procedure linkage.

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 [27, 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.[2]

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, either on the same or different processors, i.e., statements in the i^{th} iteration may execute before or at the same time as statements in the j^{th} iteration, as long as data dependences are respected. Inner loops are put in separate codeblocks and can be spawned to separate processors.

[2] The reader may have observed that the sample procedure could be totally statically scheduled by observing that it returns 14 if $x > 0$ and 11 otherwise. It is easy to extend the example, however, so that no static scheduling is possible [27, page 2].

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.

If the programmer wishes to limit the extent to which a loop can be parallelized, loops can also be specified by the following schema:

```
{for x <- eIndex bound eBound do ... }
```

This would restrict dynamic unfolding to *eBound* iterations.

Non-Functional Constructs

I-structures are used to avoid the inefficiency of arrays in functional languages while still guaranteeing deterministic programs. I-structures are arrays with elements that can only be written to once. After a value has been written, 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 and space. 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, mentioned above, are implemented as I-structures.

In addition to supporting user types, I-structures are used to create closures for currying procedure calls. Whenever an argument is applied to a procedure, a check is made whether

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

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. I-structures are also used to implement thunks, which are objects whose evaluation is delayed.

For cases where it would be useful to modify an array cell, *M-structures* are provided [3]. When an M-structure cell is read, the value is taken from the cell, which becomes empty. As with I-structures, reads before writes are silently deferred.

1.1.2 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 and are used to indicate values that are not yet present. The MDPs communicate with each other through a low-latency network by sending messages. When a message arrives at its destination, 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 MDP has three priority levels: background, priority 0, and priority 1. Each level has its own set of registers, and priorities 0 and 1 have separate message queues. Background execution is interrupted by a priority 0 message, which in turn will be interrupted by any priority 1 messages.

Several J-Machines have been built, including one with 128 processors. See [9, 11] for a complete description of the MDP and the J-Machine.

1.2 Previous Experiments in Executing Dataflow Programs on the J-Machine

1.2.1 Dataflow Graphs

Dataflow compilers convert programs into dataflow graphs, where the nodes of the graph represent operators, and the arcs represent dependencies. Figure 1-2 shows an Id procedure

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

Figure 1-2: Id Code for Factorial

to recursively compute factorial, and Figure 1-3 shows the corresponding dataflow graph.

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 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.

These graphs can be directly executed by dataflow machines, where the machine language is dataflow graphs, and data is passed in *tokens*, objects that encode their data value and destination.

Static Implementation of Dataflow Graphs on the MDP

Our first experiments with dataflow on the J-Machine involved *static* dataflow, in which dataflow graphs must be acyclic and nonreentrant. Because there is no way to distinguish different invocations of a procedure under static dataflow, only one instantiation of a dataflow graph can be active at a time.

Figure 1-3: 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.

For a J-Machine implementation of static dataflow, a token can be represented by two words:

1. A header with the address of the code that will operate on the token.

2. The data value.

When a token is sent to a dataflow node, this two-word message is sent to the processor on which it should run. The code for a *plus* node, for example, written symbolically, is:

```
[Initialization code]
R0 <- CFUT:0
R1 <- MSG.VALUE + R0
[Code to send result to destination]
```

The first time this code fragment runs, the first line loads a cfuture into register *R0*, signifying that the needed data is not present. When the second line tries adding the cfuture in *R0* to the new argument, a *cfuture fault* occurs because no arithmetic operations can be performed on cfutures. The cfuture fault handler, which is not built into the hardware, encodes the arriving token's value into the location of the first instruction and then suspends.[4] After this happens, the code has been changed to:

```
[Initialization code]
R0 <- [value of first token]
R1 <- MSG.VALUE + R0
[Code to send result and clean up]
```

When this is executed, the value of the first argument to arrive will be loaded into *R0*, and the new argument will be added to it by the second instruction. The sum is then sent elsewhere to be processed by one or more other nodes. [8] goes into more detail about implementing static dataflow on the J-Machine.

Dynamic Implementation of Dataflow Graphs on the MDP

Because static dataflow is too restrictive for most purposes, not allowing code reentrance, we extended the system to support dynamic dataflow, where multiple invocations of the code corresponding to a dataflow graph can be active at once. To implement dynamic dataflow, we added an additional word to each token to hold the *context* pointer. Every instantiation of a dataflow graph, corresponding to a procedure application, for example, has its own context. In order to allow the code corresponding to a node to be reentrant (by keeping tokens from different invocations separate), unmatched values are not stored within the code but in a separate area of memory where the locations are determined as a function not only of the destination address but also of the context. This way, two left tokens waiting for their partners at the same node will be stored in different locations. This format is modeled after Papadopoulos' explicit token store (ETS) [17].

Sample code for a *plus* node in the dynamic system is:

```
A1 <- MSG.CONTEXT
```

[4]This approach would be less efficient if the encoding of the instruction to load a constant into *R0* on the J-Machine were not so simple. To load the constant X into *R0*, one simply places X in the instruction stream. Because it is not tagged as an instruction, the decoder knows to interpret it as a constant to load into *R0*. Thus, creating the instruction to load X into *R0* is trivial.

```
R1 <- MSG.VALUE
R2 <- 1
R1 <- [A1+R2] + R1    ; This line may fault
[Code to send result and clean up]
```

First, the context pointer is loaded into address register *A1*. Next, the value of the just-arrived token is loaded into general-purpose register *R1*. The addition will cause a cfuture fault if the previous token has not been stored into the location pointed to by *A1* with a hard-coded constant offset. If the other token has already arrived, it can be found in this location, and the addition can take place. The purpose of the constant offset is to keep tokens from different nodes of a graph in different locations when their contexts are the same.

The cfuture fault handler is simply:

```
[A1+R2] <- R1
suspend
```

This stores the value of the newly-arrived token into the location pointed to by the context pointer and then suspends. See [20] for further details.

1.2.2 Simulation of Iannucci's Hybrid Architecture on the MDP

The first system to use partially sequentialized code, as opposed to dataflow graphs, simulated Iannucci's hybrid dataflow/von Neumann architecture [13]. When Id programs are compiled to hybrid code, 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.

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

Figure 1-4: 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.

processor, with execution of the invocations interleaved. After a codeblock starts running, 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 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 1-4.

MDP support for Iannucci's hybrid architecture is described in [21]. A quantitative comparison of the three systems described here appears in [23, 22].

1.3 Overview

The next chapter provides a high-level description of the TAM programming model and two MDP implementations of TAM. Chapter 3 describes the implementations in greater detail.

Chapter 4 describes the compilation process. Analysis appears in Chapter 5, and conclusions and areas for future research are stated in Chapter 6. Library routines and sample code appear in the appendices.

A more complete version of this report will appear as [24].

Chapter 2

The TAM Model and Two Mappings to the MDP

In this chapter, the TAM model is described, and high-level descriptions of the two MDP implementations are given. Lower-level details on the implementations appear in the next chapter.

2.1 TAM

In order to support the fine-grained parallelism of dataflow programs with minimal hardware support, Culler et al. have designed a threaded abstract machine (TAM) in which synchronization, scheduling, and storage management are specified explicitly. For TAM, an Id codeblock is compiled into a set of *inlets* and *threads* made up of TL0 (Threaded Language 0) instructions. Inlets are short message handlers that receive arguments from outside of the codeblock, and threads are sequences of code corresponding to the body of the codeblock. Instruction i can only appear before instruction j in the same thread if the following conditions are met:

1. It can be determined at compile-time that i never depends on a result computed by j. (See Section 1.1.1 for a brief discussion of instructions that cannot be sequentialized.)

2. If j depends on i, the dependence must not involve an operation of unbounded latency; for example, an instruction that initiates an I-structure read must not occur in the same

thread as an instruction that uses the returned value.

For details on how an Id codeblock is correctly sequentialized, see [19, 26].

When a codeblock is invoked, a *frame* is allocated for storage of arguments, local variables, *entry counts*, and a *remote continuation vector* (RCV). Each entry count indicates the number of times the associated thread must be *posted* or *forked* before it may run, representing the number of control and data dependencies the thread has. The RCV contains a list of threads that are ready to run. When an inlet is executed, it typically writes the incoming value(s) into the frame and *posts* a dependent thread. If the thread is *non-synchronizing* — that is, if it does not have an explicit entry count — a pointer to the thread is placed in the RCV, indicating that the thread may run. If the thread is *synchronizing*, the associated entry count is decremented, and a pointer to the thread is placed in the RCV only if the count has reached zero. Observe that a "non-synchronizing" thread has an implicit entry count of one, because it is placed in a continuation vector on the first attempt.

When the frame is activated, the frame's continuation vector is copied into the *local continuation vector* (LCV),[1] and a specially-designated entry thread is executed. Any threads that fork other threads within the codeblock do so by placing them in the LCV, or, if the fork appears at the end of the thread, branching directly. Threads are executed from the LCV until none remain, after which the exit thread is executed. The set of threads executed in a single frame activation is called a *quantum*.[2]

Entry and exit threads exist to take advantage of that most scheduling quanta contain multiple threads, i.e., if one thread within a codeblock runs, another thread within the same codeblock is likely to immediately follow. Time can thus be saved by having an entry thread load commonly-used frame slots into registers where they will remain during the entire scheduling quantum, at the end of which the exit thread will store them into frame memory. For a fuller description of TAM, see [7]. For a partial specification of TL0, see [30]. For details on compiling Id programs to run on TAM, see [19].

---

[1] The reasons for copying the LCV into the RCV are two-fold: First, the LCV's maximum size (equal to the number of inlets, each of which may post just one thread) is typically smaller than that of the RCV (equal to the number of threads), so separating the two saves memory. Second, the LCV can be stored in limited fast memory which can be reclaimed easily for the next process when the activation completes.

[2] Note that the meanings of the terms *thread* and *quantum* in TAM are the reverse of how the terms are used in [13, 21].

2.2 Implementing TAM on the MDP

I developed two different methods for compiling TAM programs to run on the MDP: a direct implementation that closely models TAM and a more efficient implementation that flattens the TAM scheduling hierarchy.

2.2.1 A Direct MDP Implementation

In the direct implementation, ss described above, an argument to a codeblock is sent to an inlet, which runs at priority 0. The inlet places the argument into the frame and posts a thread. If the thread is non-synchronizing, or if decrementing its entry count yields zero, a pointer to the thread is placed in the RCV, which is located in the frame. A queue is maintained of frames whose RCVs are non-empty. When the processor is otherwise idle, a background process chooses a frame from this queue, copies its RCV into an on-chip LCV, and branches to its entry thread. Whenever a thread finishes, the address of the next thread is popped from the LCV. When the fork instruction is encountered in a thread, the appropriate entry count is decremented. If the entry count has reached zero or if the thread is non-synchronizing, a pointer to the thread is placed on the LCV. The last item in the LCV is the address of the code to swap in a new frame. An optimization carried over from other implementations of TAM is that a fork at the end of a thread is converted into a branch when possible. Support for entry and exit threads was not implemented because the Id-to-TL0 compiler does not yet generate them. System routines, to manipulate shared data structures and allocate frames, run at priority 0, interrupting thread code.

A difficulty arises due to the ability of inlets to interrupt threads. Specifically, consider the case where a "fork t" instruction is interrupted by an inlet that includes the instruction "post t". If the thread and the inlet belong to the same codeblock activation, the synchronization counter for thread t may be improperly decremented. Similarly, the LCV and the pointer to the top of the LCV are places of contention. This is described in greater detail in Section 3.3.3. In order to ensure atomic access of data strucures accessible from both threads and inlets, interrupts are disabled during the bodies of threads.

TAM Mechanism	Direct Implementation	Flattened Implementation
inlet	priority 0 message handler	priority 0 message handler
post from inlet	placement of thread in RCV	jump directly to thread
activation of frame	background library routine	n/a
entry thread	background priority code, which jumps to threads within procedure	not used
threads	background priority code	priority 0 code
exit thread	sequence of code run at background priority	not used
fork from thread	jump or push onto LCV	jump or push onto CV
system routines	priority 0 message handlers	priority 1 message handlers

Table 2.1: Mapping of TAM Constructs to the J-Machine

2.2.2 A Flattened MDP Implementation

A more efficient implementation flattens TAM's two-level scheduling hierarchy.

Under this model, inlets contain branches directly to threads, eliminating the need for the RCV. Because control can transfer directly from an inlet to a thread, both run at priority 0. Entry and exit codeblocks are not used. If a thread forks multiple threads, because control can only be transferred to one, the others are pushed onto a LCV. When a thread that does not end with a fork completes, the next thread address is popped from the LCV. The only code that runs at priority 1 is that to service system calls, such as allocating frames or accessing I-structures.

Unlike in the straightforward implementation, atomicity is not a problem, because inlets run at the same priority as threads and do not interrupt them.

The MDP mechanisms for the two implementations are summarized in Table 2.1.

2.2.3 Comparison of Implementation Methods

While the direct model suggested above is trivially a correct implementation of TAM, the flattened model deserves further discussion. The flattened model fits the TAM specification that once a frame is activated, all enabled threads are executed (i.e., the LCV is emptied); however, executing inlets at the same priority as threads changes the dynamic behavior of the system. To illustrate, if two inlet messages arrive at about the same time, under other TAM implementations, one will run, then the other, followed by any threads they forked. Under

	Direct	Flattened
Post from inlet		
non-synchronizing	T_{push}	1
synchronizing (successful)	$3 + T_{push}$	3
synchronizing (unsuccessful)	4	4
Enqueue frame in global list	$\sim$6	not needed
Begin activation		
select frame from global list	$\sim$10	not needed
copy RCV into LCV	?	not needed
pop LCV	T_{pop}	not needed
Fork from thread		
non-synchronizing	1 or T_{push}	1 or T_{push}
synchronizing (successful)	3 or $3 + T_{push}$	3 or $3 + T_{push}$
synchronizing (unsuccessful)	4	4
Pop LCV at end of quantum	T_{pop}	T_{pop}

Table 2.2: Comparison of Two Methods of Implementing TAM. T_{push} and T_{pop} represent the time to push or to pop a continuation onto the LCV, about 3 instructions each. When two times are listed for forking, the first value is for when a branch can be performed (if the fork is the last instruction in the thread) and the second value is for when the continuation must be pushed onto the LCV.

the flattened system, the first inlet will run, followed by any thread it posts, with the second inlet only running after all the enabled threads within the frame complete.

There are two possibly negative consequences of the flattened design. First, because messages invoking inlets are not executed at high priority, the message queue is unloaded less quickly and has a greater likelihood of overflowing. Second, because of the shorter quanta, entry and exit threads would run more often. Recall that entry and exit threads are used to load frame slots into registers and to restore them back to frame memory. Because the J-Machine has so few general-purpose registers (4), none can be spared for cacheing frame slots, so entry and exit threads would be of little use anyway. Additionally, the J-Machine does not have a hardware-managed cache, so there are no automatic memory benefits obtained by scheduling threads associated with the same frame together. We thus decided it would not be worthwhile to exploit the locality of threads within a quantum. In our analysis, we quantify how much the flattened implementation reduces locality.

Some of the benefits of the flattened scheme can be seen in the back-of-the-envelope comparisons in Table 2.2.3. More detailed counts appear in Section 5.1. Observe how much less time it takes to post from an inlet, enqueue a frame into the global list, and to begin an activation. An additional benefit is that, because inlets jump directly to threads instead of placing them into a continuation vector, a bigger region of code is open to conventional optimization. For example, consider the following inlet and thread, as they might be implemented on the MDP:

```
inlet 0:
(I1)    gpr0 <- message.argument
(I2)    frame.5 <- gpr0
(I3)    post thread 1

thread 1:
(T1)    gpr0 <- frame.5
        [and so forth]
(Tn)    stop
```

If thread 1 is non-synchronizing and if only inlet 0 posts or forks thread 1, the reload of the register in line T1 can be eliminated. Additionally, the code for the thread can be placed immediately after the inlet, eliminating the need for line I3. If no other threads use frame slot 5, line I2 can be removed. The stop statement in line Tn usually must be implemented as a pop of the LCV into the instruction register. If thread 1 contains no pushes onto the LCV, then it is known that the LCV is empty, and the stop statement can be converted into a suspend instruction. Even if only some of these conditions are met, a subset of these optimizations can be performed.

In the next two chapters, I will describe the execution and compilation of TAM programs in greater detail.

Chapter 3

Details of Implementations

The last chapter provided an overview of the two J-Machine implementations of TAM. This chapter describes the data structures, memory usage, and control structures in greater detail.

3.1 Data Structures

3.1.1 Frames

As described briefly in the last chapter, a frame is allocated when a codeblock is invoked. In the direct implementation, the frame holds arguments, local variables, intermediate computations, synchronization counters, and the RCV. Additionally, the following three pieces of information are stored at the base of every frame:

- A pointer to the next ready frame. If the frame is not ready, 0 is stored in this location. If it is the last ready frame, nil is stored.
- The offset of the top valid entry in the RCV.
- The size of the frame.

The RCV is stored at the end of the frame and is as large as the number of inlets, because each inlet can post up to one thread. The size of the frame is used to determine how many entries are in the RCV. Figure 3-1 shows three frames, two of which are ready and one of which is not.

Figure 3-1: Sample Frames in Direct Implementation. The first slot in a frame holds a link in the queue of ready frames, the second slot holds the offset of the top of the RCV, and the third holds the frame size. Instruction pointers to the ready threads are stored at the end of these frames. Frame (b) is after frame (a) in the queue of ready frames and is at the end of the queue, as indicated by the nil in its queue-link slot. Frame (c) is not in the queue, specified by the zero in its queue-link slot. Its RCV index is the same as the stored size of the frame, indicating that the RCV is empty and no threads are ready to run.

Frames are the same in the flattened implementation, except there is no RCV or queue of ready frames, which eliminates the need for all the data words at the base of the frame except the pointer to the next ready frame.

3.1.2 Codeblocks

A codeblock is represented as three words of data followed by code for the codeblock's inlets and threads and by the inlet pointer table. The inlets and threads are ordered by the compiler with the aim of minimizing branches. The first data word of a codeblock holds the base-two logarithm of the size needed for a frame (which will be used as an argument to the memory manager). The second data word holds a pointer to a linked list of frames that have previously been allocated and freed for the codeblock. The third data word of a codeblock holds the address of inlet 0. This is typically branched to when a codeblock is allocated.

The inlet pointer table holds the addresses of the codeblock's inlets. The i^{th} entry in the table holds a pointer to inlet i in the form of a MDP message header. Codeblocks are

Data Type	Size (in words)
Boolean	1
Integer	1
Synchronization counter	1
Float	1
Codeblock pointer	2
Frame pointer	2
Structure pointer	2
Inlet pointer	1
General	2

Table 3.1: Sizes of Data Objects Stored in Frames. Pointers in general are two words long in order to specify both the processor number and the address within the processor. The exception is inlet pointers, which specify no processor number, because code is placed in the same position on every processor. Generals are two words long in order to hold any type of object.

replicated in the same position on each processor, so the inlet table for a codeblock on one processor can be used to find the message header for specifying an inlet of the same codeblock on another processor.[1]

.

3.1.3 The Processor Table

To support multiprocessing, each processor has a table used to map an integer counter to a processor node number. The table is accessed to choose the target node whenever a new frame is requested. To support pseudo-random task distribution, each processor table contains a different ordering of the processors.

3.1.4 TAM Data Types

This section describes TAM data types of items that can be stored in frames. Table 3.1 shows how many words each type requires on the J-Machine. Booleans and integers are both supported directly by the J-Machine. Synchronization counters are implemented as integers. Floating point numbers are implemented in a single word using the library routines written

[1] Like several ideas in this chapter, this originated with the Berkeley team.

by Todd Dampier and Scott Furman. Pointers to codeblocks, frames, and structures are two words long. The first word holds the processor number, and the second holds the address on the processor.

Inlet pointers are used to specify the inlet to which a data value should be returned. An inlet pointer could not be implemented as the address of the inlet, because inlet pointers may be incremented or decremented by TL0 code during procedure linkage, as described below in Section 3.3.2. An inlet pointer is represented as an integer specifying a location within the inlet pointer table described in Section 3.1.2. Pointer arithmetic is trivial, and a message header can be obtained from an inlet pointer by simply dereferencing it locally. Taken together, a frame pointer and an inlet pointer specify a *continuation*, the logical destination of a message.

Generals are the data type used for quantities whose type cannot be determined at compile-time. Hence, the size of a general is equal to that of the largest data type, 2 words.

3.1.5 I-structures and M-structures

When a TL0 statement requesting a new n-element I-structure is executed on processor p, a high-priority request is sent from processor p to the I-structure allocation routine on processor p (i.e., I-structures are allocated locally). The library routine does the following:

1. Request $2 * n + 1$ words from the memory manager. The coefficient is 2 because 2 is the size of a general.

2. Store the number $2 * n + 1$ in the first word of the newly-allocated block of memory. This way, when memory is deallocated, the size of the block is known and can be given to the memory manager.

3. Build a two-word pointer to offset 1 of the block of memory and return it to the target continuation.[2]

4. Set the first word of each element (i.e., every other word, starting at offset 1) to have the cfuture tag and the value -1. This indicates that each element is empty.

[2] Because I-structures are allocated locally, we could have omitted the first word of the pointer in the return message, but writing the code for the more general case seemed preferable.

A more sophisticated scheme would partition I-structures across processors.

A given memory location in an I-structure has one of the following three states in its first word:

- Empty, indicated by the cfuture -1.
- Waiting, indicated by a cfuture whose data value is the address of a linked list of waiting continuations.
- Full, indicated by a non-cfuture.

Checks are not performed to ensure that locations are written only once and that accesses are in bounds. These could easily be added. While the length field in the MDP address type could be used for bounds checks, some I-structures are more than $2^{10} - 1$ words long, and their length could not fit in the field.

When a request is made to read the word at offset n in an I-structure i, a request is sent to the processor on which i is located (specified by the first word). The message handler attempts to read the value at offset $2 * n$ from the base of the I-structure. If the value is present, it is returned to the continuation specified in the request. If it is not, the cfuture fault handler is executed, which does the following:

1. Check that the fault was due to the read of an I-structure and not a thunk. (Thunks are described below.)
2. Request four words from the memory manager.
3. Store the continuation into the first three words of the block, and store the former contents of the faulted memory location into the fourth word.
4. Tag the address of the block as a cfuture, and store it in the faulted memory location.

Following this scheme, each unfulfilled request is attached in a linked-list rooted at the location of the awaited value.

When a request is made to write a word to an I-structure, the target location is first checked for waiting continuations. If there are any, they are sent the data, and the storage

registers	low priority	high priority
R0–R2	two as temporaries, one as TL0 register	temporaries
R3	index to top of LCV	temporary
A0	always 0	always 0
A1	frame pointer	inlet frame pointer
A2	processor pointer	processor pointer
A3	message pointer	message pointer
ID0–ID3	spill registers	temporaries

Table 3.2: Usage of MDP Registers. R0–R3 are the general-purpose registers, and A0–A3 are the address registers. Which of the low priority registers R0–R2 is used to hold a TL0 register value varies. In the processor mode used, A0 is always read as 0. A3 is also a special register, which points to the head of the message queue. ID0–ID3 are used to hold live registers during calls to the floating-point libraries. In the direct implementation, the column headings "low priority" and "high priority" correspond to background priority and priority zero, respectively. In the flattened implementation, they correspond to priorities zero and one, respectively.

space for the continuations is deallocated. The data is stored in the location for any future use.

M-structures are implemented similarly, except, on a read, the location is cleared, and, on a write, only one continuation is served.

3.2 Register and Memory Usage

3.2.1 Register Usage

Table 3.2 shows how MDP registers are used. Because the MDP only has four general-purpose registers, only one could be used to hold a TL0 register value. Which MDP register is used for this purpose varies over time. The registers ID0–ID3 are used to hold live registers during calls to the floating-point libraries.

3.2.2 Memory Map

Table 3.3 shows how memory is used in the direct implementation. Locations 0 through \$E hold any TL0 registers beyond the one that can be placed in a MDP register. (Recall that these locations can be easily accessed offset from A0.) The location of the fault and system

Location	Contents
\$0..\$E	TL0 overflow registers
\$F	Number of processors minus one ($P-1$)
\$10..\$25	Memory allocator storage
\$3F	processor allocation counter
\$40..\$40 + P	node number table
\$80..\$D2	fault vectors and system call vectors
\$100..\$1C4	floating-point library
\$200..\$201	priority 1 message queue
\$202..\$203	final result
\$204	pointer to first frame in queue
\$205..\$23f	local continuation vector
\$240..~\$357	library routines
\$400..\$480	initialization code
\$400..\$4FF	priority 0 message queue
\$500..\$500 + $U-1$	user code
\$500 + U..\$600 + $U-1$	floating-point data table
\$600 + U..\$73D + $U-1$	top-level code and default fault handlers
\$73D + U..	heap

Table 3.3: Placement of Data in Direct Implementation. P represents the number of processors being used and U represents the size of user code. All numbers are specified in hexadecimal (as indicated by the \$). The first \$1000 words are on-chip. The priority 0 message queue overwrites the initialization code after it has been executed.

Location	Contents
\$0..\$E	TL0 overflow registers
\$F	Number of processors minus one ($P-1$)
\$3F	processor allocation counter
\$64..\$79	memory allocator storage
\$80..\$D2	fault vectors and system call vectors
\$100..\$100 + P	node number table
\$180..\$1FF	local continuation vector
\$200..~\$2E3	library routines
\$300..~\$3C4	floating-point library
\$3FE..\$3FF	final result
\$400..\$477	initialization code
\$400..\$5FF	priority 0 message queue
\$600..\$6FF	priority 1 message queue
\$700..\$700 + $U-1$	user code
\$700+U..\$800 + $U-1$	floating-point table
\$800 + U..~\$940 + $U-1$	start-up code (which gets overwritten)
\$940 + U..	heap

Table 3.4: Placement of Data in Flat Implementation. P represents the number of processors being used and U represents the size of user code.

call vectors is as specified by the hardware. System calls are used to access the floating-point library and the routines to post threads. The initialization code is placed in the same location as the priority 0 message queue, because the code only needs to run once, after which it can safely be overwritten. Observe that the libraries are stored on chip, i.e., within the low \$1000 words, as are the first \$B00 words of user code. A table used for floating-point division is stored after user code.

The memory lay-out in the flat implementation, shown in Table 3.4 is similar, except the library routines are smaller and the message queues are larger, because more buffering is required due to the delay in processing inlets.

3.2.3 Heap Memory

Both implementations used memory management routines written by Bill Dally and Ellen Spertus in a combination of Brennan Gaunce's J language and MDP assembly language. A list of bins is maintained, with the n^{th} bin containing a linked list of pointers to free memory regions of size 2^n. In both systems, the memory manager is only called by high priority code (i.e., priority 0 for the direct implementation and priority 1 for the flat implementation).

3.3 Control Structure

This section describes the control flow constructs used in system initialization, codeblock invocation, and execution within a codeblock.

3.3.1 System Initialization

To begin execution, code is loaded onto processors, and a few data values are downloaded, such as the number of processors in the machine and each processor's node number. The start-up code does the following:

1. Initialize the processor, as described in [16].
2. Set the LCV to empty, and make R3 point to it.
3. Make A2 point to the processor table.
4. Divide up heap memory, and place it in bins for the memory manager.
5. On processor 0 only, issue a read of the thunk named "top".

The read of "top" will cause evaluation of the associated codeblock, as described in Section 3.3.2. "Top" is a reserved word in my system.

3.3.2 Codeblock Invocation

When a thread or fault handler requests a frame, it sends a high priority request to the allocation routine, passing the name of the codeblock and a continuation indicating where to send the new frame pointer. The signal indicating the child codeblock has completed will be sent to the following inlet, and the child codeblock's returns value, if it has one, will be sent to the subsequent inlet. A processor is selected by incrementing a counter which is used as an offset in the processor table described in Section 3.1.3. The resultant node number is made the target of the allocation request.

Figure 3-2: Procedure Linkage. A codeblock running on processor A requests a frame on processor B. Shaded areas represent unrelated computation. While illustrating the general process, this figure glosses over some details, such as whether delays occur between an inlet and any thread it posts.

On the target processor, the base of the codeblock is checked to see if any frames are available that had been allocated for the codeblock and subsequently freed. If so, one of these frames is selected; otherwise, a frame is obtained from the memory manager. The frame pointer is sent back to the caller (except in the case of thunks) and is used to send any arguments. This process is illustrated at a high level in Figure 3-2. The rest of this section describes differences between the two implementations and varieties of frame allocation requests.

Ordinary Procedure Linkage

In the direct implementation, if a new frame is allocated, its codeblock and RCV index are initialized. (Refer back to Figure 3-1.) If a frame is being recycled, those locations are already set. Finally, a branch is made to inlet 0 of the codeblock. Inlet 0 stores the continuation that the caller passed to the allocation routine, initializes the frame's synchronization counter, and returns the new frame pointer to the caller.

Behavior in the flattened implementation is different in two ways. First, there are no special frame slots to initialize. The second issue is more subtle, namely at which priority to run the code to allocate a frame and return the pointer. In an ordinary frame allocation call, the new frame pointer is returned by inlet 0 of the called codeblock. The difficulty is that inlets run at low priority, but it is desirable to return the address of the new frame pointer as soon as possible, so arguments can be sent. Three possibilities were considered and dismissed:

- Running the allocation routine at priority 1, then branching to inlet 0, which leads to inlet and thread code running at priority 1.
- Running the allocation routine at priority 0.
- Running the allocation routine at priority 1, and having it send a priority 0 message to the first inlet.

The first option was considered unacceptable, because inlet 0 can post threads and thus cause priority 1 execution for an unbounded amount of time. While the second option has the advantage of leading to priority 0 inlet and thread execution, it delays the allocation and return of the frame address. The third option also caused inlets and threads to be executed at the correct priority level, but it too delayed return of the frame pointer to the calling procedure.

Instead, inlet 0 is split into two parts, namely instructions that should run at high priority (most notably the send of the new frame pointer) and those that should run at low priority (the post instruction). The post is placed in a new inlet, called inlet 1, and inlet 0 ends by sending a low priority message to execute inlet 1. This is illustrated in Figure 3-3. This allows us to run the system allocator at high priority, which then branches directly to the inlet code that will return the frame, but we still avoid running the body of the codeblock at high priority. Inlet 0 is not transformed in the cases where it does not post a thread.

Thunks

Thunks are used in lenient languages to represent computations that could be performed but haven't yet been executed. Their requirements are similar to I-structures: A read of a thunk is silently deferred until the data is available, after which the data can be read henceforth.

Figure 3-3: The Transformation of Inlet 0 in the Flattened Implementation. Subfigure (a) shows the original inlet. Subfigure (b) shows the two parts it is broken into, the first of which will run at high priority, the second at low priority. Inserted statements are indicated in bold type.

The difference between a thunk and an I-structure is that a thunk contains a pointer to a codeblock that can compute the value. When the first attempt is made to read the thunk, a frame is allocated so the codeblock may run. When the result of the computation is performed and has been written to the thunk slot, waiting continuations are notified and deallocated, just as for I-structures. Figure 3-4 illustrates this process. Note that the frame allocation routine executed for thunks does not return a frame pointer.

Loops

The TL0 compiler uses k-bounded loops [6] to extract parallelism from loops. This involves pulling a loop into a separate codeblock which is called by the procedure in which the loop appears. The arguments to the codeblock are the loop constants and circulating variables. This new codeblock calls itself to begin the next iteration of the loop. The processor for a call is chosen in the same manner as for ordinary procedure calls; thus, the loops get distributed and can run in parallel.

The exact details of k-bounded loops are outside of the scope of this thesis, as the conversion is performed before TL0 code is produced. The only addition to my system needed to support k-bounded loops was a separate version of the frame allocation routine that returns

Figure 3-4: The Lifetime of a Thunk. In subfigure (a), there have been no requests for the data in the thunk. When a request arrives, the codeblock “foo” is started, and the waiting continuation is stored, as shown by subfigure (b). Additional requests arriving before the value arrived would be queued up as a linked list. When the data arrives, it is sent to the waiting continuations, the storage for which is deallocated. The data value is written to the thunk, as shown in subfigure (c).

Name	sys-alloc	sys-alloc-raw	sys-alloc-thunk
Use	normal codeblocks	loop codeblocks	thunks
Returns frame pointer?	no	yes	no
Inlet 0 returns frame pointer?	yes	no	no

Table 3.5: Comparison of Procedure Linkage Library Routines

the new frame pointer and does not pass control to inlet 0. As soon as it receives the new frame pointer, the caller sends a message to inlet 0 containing all of the arguments.

All three varieties of library routines to allocate frames have now been described. They are summarized in Table 3.5.

Example

Figure 3-5 shows an Id program to compute three factorial (3!). Figure 3-6 graphs exe-

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

def top = fact 3;
```

Figure 3-5: Id Program to Compute 3!

cution on four processors in the direct implementation.[3] First, all processors run the same initialization sequence. Next, processor 0 initiates a read of the thunk, top. The high-priority sequence beginning at tick 844 is the attempted service of the fetch. Since the value is not available, space is allocated for the continuation and for a frame in which to execute top. After these have been set up, top begins running at low priority. It requests a frame on processor 1 in which to evaluate 3!. The system allocation routine runs at high priority, then branches to inlet 0, which returns the frame pointer for factorial to a top inlet on processor 0, at tick 1048. The inlet runs and places a thread in the RCV. When the inlet completes, the frame is swapped in and runs at low priority, sending the argument to factorial at tick 1126. The argument is received by a factorial inlet, which places the thread that requires the argument into the RCV. The factorial frame is scheduled, the thread executed, and the process continues.

The base case is reached on processor 3. It returns the value 1 to its caller on processor 2 on tick 1603. It also sends a signal to its caller indicating completion. These allow the recursive cases to follow suit, finally returning the result, 6, to the invocation of top on processor 0. When it receives the value, it writes it to the I-structure that represents the thunk. This occurs at high priority, starting at tick 1941. The I-store routine passes the result to the waiting continuation, which begins execution at tick 2065. The final sequence of code executed on processor 0 frees the frame used for top.

The same information is shown for the flat implementation in Figure 3-7. The most notable difference is that inlets run at low priority. Hence, when a value (argument, frame pointer, or trigger) is received by an inlet, the RCV is bypassed, and control is transferred to child threads directly. This is visible in the graphs both by the fewer high priority sequences and by the time savings achieved by bypassing the RCV.

[3] I am grateful to Eric Brewer and Chris Dellarocas for the use of their graph program, stats.

Figure 3-6: Execution of 3! in Direct Implementation. Inlets and most system routines run at high priority, while threads and RCV-swapping run at low priority. These timings were taken on the instruction simulator.

Figure 3-7: Execution of 3! in Flat Implementation. System routines run at high priority, while inlets and threads run at low priority. These timings were taken on the instruction simulator.

3.3.3 Execution Within a Codeblock

Forking a Thread

Both systems implement forking of threads in the same way. Each fork instruction can be either to a synchronizing or non-synchronizing thread and can appear either in the body or at the end of a thread. The earlier stages of the compiler always try to place forks and switches at the end of a thread but cannot when more than one such instruction appears in a thread. The cases are handled as follows:

1. Non-synchronizing fork at end of thread — In this case, the compiler will try to place the child thread immediately after the parent thread and allow execution to fall through. If the thread cannot be so placed, a branch is emitted.

2. Non-synchonizing fork in the body of a thread — Push the address of the thread into the LCV.

3. Synchronizing fork at end of thread — Decrement the synchronization counter. If it reaches zero, branch to the child thread; otherwise, write the decremented value back into the frame.

4. Synchronizing fork in the body of a thread — Decrement the synchronization counter. If it reaches zero, push the address of the child thread into the LCV.

Whenever the stop instruction is encountered at the end of a thread, the LCV is popped into the instruction pointer. In order to avoid explicit boundary checks, the last item in the LCV always points to the same library routine [7]. In the direct implementation, it points to the code to swap in a new frame. In the flattened implementation, it points to code that simply suspends. Code for switch statements, which conditionally forks one of two threads, is built out of these templates.

Posting a Thread

In the flattened implementation, posts are treated exactly the same as forks. Because an inlet can post only one thread, this post can always be the final instruction. Thus, the cheaper end-of-thread cases are used.

Thread-Level Code	Inlet-Level Code
R1 ← [SSLOT1,A1]	
R1 ← R1 − 1	
	R1 ← [SSLOT1,A1]
	R1 ← R1 − 1
	branch-on-zero R1, POST_IT
	[SSLOT1,A1] ← R1
	suspend
branch-on-zero R1, THREAD1	
[SSLOT1,A1] ← R1	

Figure 3-8: Race Condition when a Fork is Interrupted by a Post of the Same Thread and Frame. Even though the registers of the two priority levels do not interfere, the code is unsafe because the value in the synchronization slot will be decremented once instead of twice. The child thread will thus never be enabled.

Posting a thread in the direct implementation is more complicated. Recall that, in the direct implementation, threads run at background priority and are interrupted by inlets at priority 0. If the inlet's frame pointer (IFP) and the thread's frame pointer (FP) are different, the posted thread is written to the RCV of the frame specified by IFP. If the RCV had previously been empty, the frame is placed in the linked list of frames that are ready to run.

If IFP and FP are the same, the thread pointer is placed on the current LCV. Care had to be taken to avoid two different sets of race conditions. The first, shown in Figure 3-8 occurs when a fork is interrupted by a post of the same child thread. The danger is that the synchronization counter will not be properly decremented. A race condition exists even if different threads are being posted and forked, as both low priority and high priority code could modify the pointer to the top of the LCV. This case is shown in Figure 3-9.

In order to avoid these situations, interrupts are only briefly enabled at the start of each thread, after which they are disabled. While this prevents race conditions, it does not remove the need for separate code depending on whether the post is to a frame that is currently running. The code to properly post a thread is so long (22–35 instructions) that it is in the library instead of being placed inline. A post instruction is compiled to load the offset of the synchronization counter (in the case of a synchronizing thread) into one register and the address of the target thread in the second and then to call this library routine. The post routines can be found in Figure 3-10.

Thread-Level Code	Inlet-Level Code
R3 ← R3 + 1	
	R3 ← R3B
	R3 ← R3 + 1
	R3B ← R3
	R0 ← THREAD0
	[R3,A0] ← R0
R0 ← THREAD1	
[R3,A0] ← R0	

Figure 3-9: Race Condition when a Fork is Interrupted by a Post of the Same Frame. "R3B" is how the inlet-level code refers to the background priority level's R3, the LCV pointer. Observe that it gets incremented twice, but the two threads get written to the same location in the LCV.

Scheduling a Frame

Explicitly scheduling a frame is not necessary in the flattened implementation, as control passes directly from inlets to threads. In the direct implementation, the code to swap in a frame is executed when the LCV has been emptied. If the queue of ready frames is empty, control passes to a loop which keeps checking the queue pointer. When a frame becomes available, it removes it from the queue, copies the RCV into the LCV, resets the pointer to the top of the RCV, and begins executing frames from the LCV.

3.4 Summary

In this chapter, the details of TAM execution on the J-Machine were presented. The following chapter describes the inner workings of the compiler. It contains information similar to that in this chapter, from the perspective of how the code is generated.

```
maybe_post_it:
        ;; R0 holds target
        ;; R1 holds offset
        ;; A1 holds frame pointer
        ;; First check if we should post
        move [R1,A1], R2
        sub R2, 1, R2
        bnz R2, ^post_fails
        ;; Check now what type of post
        move A1B, R2
        eq R2, A1, R2
        bt R2, ^post_into_special_rcv
        ;; Avoid branch by duplicating post_normal_rcv
;       br ^post_normal_rcv
        ;; Write into normal RCV
        move [cv_top,A1], R3
        sub R3, 1, R3
        move R0, [R3,A1]
        move R3, [cv_top,A1]
        move [next_pointer,A1], R3
        check R3, int, R3
        bt R3, ^post_into_queue
        suspend

post_fails:
        move R2, [R1,A1]
        suspend

post_it:
        ; check if frame is currently running
        move A1b, R1
        eq R1, A1, R1
        bt R1, ^post_into_special_rcv
post_normal_rcv:
        ;; Write into normal RCV
        move [cv_top,A1], R3
        sub R3, 1, R3
        move R0, [R3,A1]
        move R3, [cv_top,A1]
        move [next_pointer,A1], R3
        check R3, int, R3
        bt R3, ^post_into_queue
        suspend

post_into_queue:
        dc queue
        move [R0, A0], R2
        move R2, [next_pointer,A1]
        move A1, R1
        move R1, [R0,A0]
        suspend

post_into_special_rcv:
        move R3B, R3
        add R3, 1, R3
        move R3, R3B
        move R0, [R3,A0]
        suspend
```

Figure 3-10: Library Code for Post Routine in Direct Implementation

Chapter 4

The Compiler

The languages and intermediate forms my compiler recognizes or produces are:

- TL0 code, the TAM machine language.
- Complex MDP code, s-expressions whose opcodes are the same as those on the MDP (with a few extensions) but whose addressing modes, etc., are not necessarily legal.
- Simple MDP code, s-expressions using legal addressing modes.
- MDP assembly code.

Figure 4-1 shows how the same expression is represented in each of these formats. A more extensive example appears in Appendix A, where the factorial program is shown in each code

TL0	ADD islot2.i = islot0.i + islot1.i
Complex MDP	(add (:frame islot0) (:frame islot1) (:frame islot2))
Simple MDP	(move (:frame (:tagged-literal int (islot1 . 5))) (:j-register R2)) (add (:j-register R2) (:frame (:tagged-literal int (islot2 . 6))) (:j-register R1)) (move (:j-register R1) (:frame (:tagged-literal int (islot3 . 7))))

MDP	move [islot1,A1], R2 ; islot1=5
	add R2, [islot2,A2], R1 ; islot2=6
	move R1, [islot3,A1] ; islot3=7

Figure 4-1: Examples of Source, Intermediate, and Target Languages

representation. TL0 code is described in [30], while MDP assembly language is described in [9, 16]. A complex- or simple-MDP instruction is a list of an opcode followed by zero or more operands. Operands can be literals or offsets with the frame, LCV, or temporary storage.

The Id-to-TL0 compiler was created by Klaus Erik Schauser at Berkeley [19] and includes code from the MIT Computation Structures Group Id compiler [28, 1]. I modified Thorsten von Eicken's TL0-to-nCube compiler to make it create complex MDP code. The final stages of conversion were performed by a modified version of a translator that I first described in [21].

The remainder of this chapter is primarily of use to people who will be reading or modifying the source code to my compiler.

4.1 Converting TL0 to Complex MDP

The TL0-to-Complex-MDP compiler does the following:

1. Parses the TL0 source file.
2. Allocates frame slots and registers for variables.
3. Collects statistics.
4. Generates and outputs code.

I inherited all but the last step from von Eicken's TL0-to-nCube compiler, modifying only one file of the back end to use different code templates, as described later in this chapter.

To take advantage of the frame and register allocation routines written by von Eicken, all I needed to do was initialize a global structure to specify constants for the MDP: For register allocation, I specified that the J-Machine has one general-purpose register (the rest being needed as short-term temporaries, as will be explained later in this chapter). The other constant values needed were the size of TL0 data types on the MDP, shown earlier in Table 3.1.

I rewrote the back end to produce complex MDP code.

4.1.1 Templates

The following routines are used to help generate code in the back end:

Data Class	Inputs	Output String
integer	the value N	N
floating-point number	the value F	F
inlet pointer	the codeblock name C and the inlet number N	(:ref C_inlet_N_pointer_ref)
codeblock	the codeblock name C and which word (N) of the codeblock pointer	(:ref C_N)
local thread	the codeblock name C and the thread number N	(:label C_threadN)
true	—	(:tagged-literal bool 1)
false	—	(:tagged-literal bool 0)
nil	—	(:tagged-literal sym 0)
minint	—	(:tagged-literal int 0x80000000)
maxint	—	(:tagged-literal int 0x7FFFFFFF)
empty	—	(:tagged-literal cfut −1)

Table 4.1: Constant Formats Produced by genConst in the TL0-to-MDP Stage. Codeblock pointers are the only two-word constant. An argument to genConst specifies which word is requested.

Data Type	Inputs	Output String
frame slot	the name of the variable S	(:frame S)
message word	the message word offset N	(:message N)
TL0 register	a unique integer N and the register's type T	(:register tempNT)
overflow register	a unique integer N	(:temporary N)

Table 4.2: Reference Formats Produced by genOpref in the TL0-to-MDP Stage.

Instruction	Purpose	Types
move	move a data object	Any
receive	transfer an object from the message queue into the frame.	Any

Table 4.3: TL0 Movement Instructions

- genConst — Convert a constant TL0 operand into a complex MDP operand in one of the formats shown in Table 4.1.
- genOpref — Convert the specified one-word TL0 operand into a complex MDP operand in one of the formats shown in Table 4.2. If genOpref's argument is a constant, it calls genConst.
- genOprefOff — Convert the specified word of the multi-world TL0 operand argument into a complex MDP operand. The format is similar to that shown for genOpref, except "_N" is appended to the reference to indicate that the N^{th} word is desired.
- genBinary — Combine the passed operation, type, two sources, and destination into a complex MDP instruction; e.g.:

  ```
  (add (:frame islot6) (:frame islot7) (:frame islot8))
  ```

Move Instructions

Movement instructions are shown in Table 4.3. If the source and destination of a *move* instruction are the same size, the following statement is generated for every word of the data:

```
(move S D)
```

where S and D represent the source and destination, respectively, and are produced by GenOpref or genOprefoff as appropriate. If the destination is larger than the source, the first word is copied into both words of the destination. If the destination is smaller than the source, an error occurs.

The *receive* instruction is converted similarly, except the source is always the message queue. The offset within the message queue is increased after each data transfer.

Class	Instruction	Purpose	Types
arithmetic	add	add	I, F, J
	sub	subtract	I, F, J
	mul	multiply	I, F
	neg	negate	I, F
	div	divide	*I*, F
	quo	quotient	*I*
	rem	remainder	*I*
	mod	modulus	*I*
	abs	absolute value	I, F
	min	minimum	I, F
	max	maximum	I, F
logical	not	logical not	I
	and	logical and	I
	or	logical or	I
	xor	exclusive or	I
bitwise	bitnot	bitwise not	B
	bitand	bitwise and	B
	bitor	bitwise or	B
	bitxor	logical xor	B
comparison	eq	equal	Any
	ne	not equal	Any
	lt	less than	I, F, *C*
	le	less than or equal to	I, F, *C*
	gt	greater than	I, F, *C*
	ge	greater than or equal to	I, F, *C*

Table 4.4: TL0 Arithmetic, Logical, and Bitwise Instructions. I, F, B, C, and J represent integer, floating-point, boolean, character, and inlet, respectively. The italicized instructions are not yet implemented. These conventions are used in tables throughout this chapter.

Arithmetic, Logical, and Bitwise Instructions

Arithmetic, logical, and bitwise operations are listed in Table 4.4. The following instructions are converted in a straightforward manner with the genBinary routine: *add*, *sub*, *mul*, *div*, *min*, *max*, *and*, *bitand*, *or*, *bitor*, *xor*, and *bitxor*. Similar conversions are done for the unary operations *neg*, *abs*, *not*, and *bitnot*. The *abs*, *min*, and *max* instructions are expanded at later stages, while the other instructions are MDP primitives. In the case when the constant 0 is added to or subtracted from a value, the instruction is converted to a move.

When floating-point numbers are compared with the *lt*, *le*, *gt*, or *ge* instructions, calls

Class	Instruction	Purpose	Types
Type Conversion	itof	convert an integer to a floating-point number	I
	ftoi	convert a floating-point number to an integer	F
	btoi	convert a boolean to an integer	B
	itob	convert an integer to a boolean	I
	itoc	convert an integer to a character	*I*
	itos	convert an integer to a string	*I*
	ctoi	convert a character to an integer	*C*
	stoi	convert a string to an integer	*S*
Abstract Type Support	isptr	check if an abstract-type object is a recursive (true) or base (false) case	PS
	ptoi	convert a pointer to an integer	PS
	itop	convert an integer to a pointer	*I*

Table 4.5: TL0 Type Instructions

are generated to the floating-point libraries. When floating-point numbers are compared with *eq* or *neq*, or when any comparison is performed on any other pair of one-byte objects, genBinary is used, as MDP primitives will be used. When comparing two-word quantities, code is generated to check only as many words as necessary (e.g., if the first words of an *eq* comparison are unequal, there is no need to compare the second words).

Type Instructions

Instructions involving type are shown in Table 4.5. They can be divided into two classes, those to support type conversion among scalar types and those that support user-defined abstract types.

Conventional Type Conversion

The *itof* and *ftoi* operations are compiled by generating calls to the appropriate routines in the floating point libraries. The *itob* instruction produces false if its input is equal to zero, true otherwise. The *btoi* instruction produces the integer 0 or 1, depending on whether its input is false or true, respectively. If a compiler flag to generate type-correct code is set, the data is retagged.

Class	Instruction	Purpose
All structures	ialloc	Allocate a structure
	ifree	Free a structure
	iclear	Clear the elements of a structure
	global_struct	Declare a global structure
I-structures	ifetch	fetch a copy of an element (or store a request)
	istore	write an element, serving all waiting requests
M-structures	itake	remove an element (or store a request)
	iput	write an element, serving up to one waiting request
Debugging only	iread	read an element (possibly empty)
	iwrite	write an element (perhaps overwriting)

Table 4.6: TL0 Structure Instructions. The Id-to-TL0 compiler appears not to produce ifree, iclear, iread, and iwrite instructions.

Abstract Type Support

The following line of Id code, taken from paraffins.id [15], defines an abstract type, *radical*:

```
type radical = H | C radical radical radical;
```

The meaning is that *radical* is the type of an object than can either be a base case **H** (representing hydrogen), or a **C** (carbon) together with three more objects of type *radical*. If r is of type *radical*, *isptr* distinguishes between the case where it holds the integer corresponding to **H** or a four-element list which holds the integer corresponding to **C** in its first slot and objects of type *radical* in the subsequent slots.

The *isptr* instruction simply checks whether its argument is a small integer (less than 256). If not, it is a pointer. The *ptoi* instruction is performed only after *isptr* has returned false for an object. It simply moves the integer value into the specified destination.

Structure Access Instructions

Table 4.6 shows TL0 instructions to manipulate structures. The semantics of I-structures and M-structures can be found in Section 1.1.1. The *global_struct* instruction is used to statically declare thunks and structures that can be accessed globally. The other instructions are converted into message sends to library routines. For *ialloc*, the message is sent to the current processor. For the other routines, the message is sent to the processor on which the structure resides.

Instruction	Purpose
postq	Try spawning a thread from an inlet
fork	Try spawning a thread from a thread
switch	Fork one of two threads depending on the value of a boolean argument
case	Fork one of many threads
ctrap	Call a library procedure

Table 4.7: TL0 Control Flow Instructions

	Non-synchronizing	Synchronizing
genJump	; fall through *or* (br T)	(sub S 1 (:register genjump)) (bz (:register genjump) T) (move (:register genjump) S) (free (:register genjump)) (suspend-string)
genPushLcv	(add (:j-register R3) 1 (:j-register R3)) (move S (:lcv (:j-register R3)))	(sub S 1 (:register genjump)) (bnz (:register genjump) L1) (add (:j-register R3) 1 (:j-register R3)) (move S (:lcv (:j-register R3))) (br L2) (label L1) (move (:register genjump) S) (label L2)

Table 4.8: Code Generated for Fork Instruction. S represents the synchronization counter and T the target thread. $L1$ and $L2$ are labels.

Control Flow Instructions

The semantics of the *fork* and *postq* (post) instructions are described above in Section 2.1, and their implementation is discussed in Section 3.3.3. The *fork* instruction is expanded using the routine genPushLcv for *fork*s in the body of a thread and genJump for a *fork* at the end of a thread. The routines generate correct code for both synchronizing and non-synchronizing threads. The templates they use are shown in Table 4.8.

The translation of the *postq* instruction depends which system is being used. In the flattened implementation, the same routines are used as for *fork*.In the direct implementation, the synchronization slot offset (for synchronizing posts) and the target thread address are loaded into R1 and R0, respectively, and a call is generated to the library routine “post_thread”

Instruction	Purpose
falloc	Allocate a new frame and start its execution
ffree	Free the current frame
swap	Swap in the next frame
stop	Pop the LCV
finit	Initialize the current frame
set_enter	Set the frame's entry thread
set_leave	Set the frame's exit thread
send	Send a data value (or a trigger) to another frame

Table 4.9: TL0 Frame Management Instructions

or "maybe_post_thread".

Code generation for the *switch* statement is performed by making multiple calls to genPushLcv and genJump and by generating conditional branches.

The *ctrap* instruction is used to call a library routine. The supported routines in my implementation are floor, ceiling, truncate, and round, which are performed on floating-point numbers. In this stage of the compiler, a *ctrap* instruction is simply renamed to a "call" instruction, which gets further expanded in later stages.

Frame Management Instructions

The *falloc* instruction is converted by emitting the complex-MDP pseudo-instruction *mp-falloc-setup*, followed by instructions to send a message to the appropriate sys-alloc routine (described in Section 3.3.2) with the name of the codeblock to start, as well as the continuation in the currently-executing codeblock and frame to which to return the new frame pointer. Later stages will expand *mp-falloc* to produce code to choose a target processor as described in Section 3.1.3.

The *ffree* instruction is passed through as "(ffree)" to be dealt with by later stages of the compiler. The *swap* instruction, which follows *ffree* is ignored, because the library routine "ffree" does not return and contains a branch directly to the "swap" code.

The instruction *stop* is often eliminated if immediately preceded by a *postq* or *fork*. Otherwise, it is translated into the pseudo-instruction, "(suspend-string)," which will later be translated into code that pops the LCV into the instruction pointer.

The instruction *finit* is ignored, because frame initialization is performed in the library routines. The instructions *set_enter* and *set_leave* are also ignored, because the current Id-to-TL0 compiler does not use them (and the flattened implementation would not support them).

The *send* instruction is expanded into a message send. In the flattened implementation, where threads and inlets run at the same priority level, interrupts are explicitly suspended for the message send.

4.1.2 Additional Information Produced

A directive is produced to indicate each of the following elements of the source file: a source line number (which becomes a "comment"), codeblock start ("codeblock"), codeblock end ("end"), inlet end ("end-inlet"), and labels ("label"). The bindings of frame variables to offsets are expressed through the assembler-label pseudo-opcode.

Additionally, when the codeblock has been processed, a special pseudo-instruction, "inlet-handler" is produced whose operands are the number of message words expected for each inlet. This will allow a later stage of the compiler to produce the inlet table.

4.1.3 Example

Figure 4-2 shows a portion of inlet 0 from factorial as it passes through the TL0-to-Complex MDP compiler stage. See Appendix A for the translation of the entire procedure.

4.1.4 Flat Optimizations

In order to implement the special optimizations for the flattened implementation described in Section 2.2.3, the compiler to produce TL0 code was modified to generate the following pragmas:

- **@USE** i f $t_1 \ldots t_n$ indicates that frame slot f, which is received by inlet i, is used only by threads $t_1, \ldots t_n$.

Input	Output
INLET 0	(comment “line 57”)
	(start inlet 0)
	(label (:label fact_inlet0))
RECEIVE pfslot0.pf jslot0.j	(move (:message 1) (:j-register A1))
	(move (:message 2) (:frame pfslot0_0))
	(move (:message 3) (:frame pfslot0_1))
	(move (:message 4) (:frame jslot0))
FINIT	(comment “line 59”)
MOVE sslot2.s = 2.s	(comment “line 60”)
	(move 2 (:frame sslot2))
MOVE sslot1.s = 2.s	(comment “line 61”)
	(move 2 (:frame sslot1))
MOVE sslot0.s = 3.s	(comment “line 62”)
	(move 3 (:frame sslot0))
SET_ENTER 11.t	(comment “line 63”)
SET_LEAVE 12.t	(comment “line 64”)
SEND pfslot0.pf[jslot0.j+-1.i] <- fp.pf	(comment “line 65”)
	(send0 (:frame pfslot0_0))
	(reserve (:register add-inlet))
	(add (:frame jslot0) -1 (:register add-inlet))
	(send0 (:temporary (:register add-inlet)))
	(free (:register add-inlet))
	(send0 (:frame pfslot0_1))
	(send0 (:j-register NNR))
	(sende0 (:j-register A1))

Figure 4-2: Sample Inlet as it Passes Through TL0-to-Complex MDP. The output is the code for the direct implementation. The code for the flattened implementation is similar.

- **@ONLY_USE** i f t indicates that frame slot f, which is received by inlet i, is only used by thread t.[1]

- **@ONLY_POST** i t indicates that the only post of thread t occurs in inlet i and that there are no forks of thread t.

As a prerequisite for the flat optimizations, all threads are replicated (i.e., inlined) at the tail of each inlet that forks them. Not only does this eliminate the time and space required for a branch but it paves the way for better register usage by merging the basic blocks. For each frame slot, a list is constructed of threads that use it. The inlet statement to receive the frame slot value is translated differently for each of the following cases:

- If a frame slot is never used, the transfer of the data from the network to the frame is eliminated. (This situation does occur, because TL0 programs also run on machines such as the CM-5 where incoming data must be explicitly removed from the network even if it is unneeded.)

- If the frame slot is not used by the posted thread but is used by other threads, transfer it from the message queue into the frame. (This is the case of no optimization.)

- If the frame slot is used by the thread posted by the inlet, mark the variable so that, when it is used, it is read from the message queue instead of from the frame. Additionally, if other threads also use the frame slot, annotate the variable to indicate that when it is first read, it should be written to the frame.

The final case is complicated by the possibility that the inlet only conditionally posts a thread that uses it. In this case, a copy of the thread is inlined that looks for the value in the message queue. Additionally, for the case of an unsuccessful post, code is generated in the inlet to move the values from the message queue to the frame; afterward, the variable is marked so that when the copy of the thread that is posted or forked from elsewhere is generated, it looks in the frame for the data (as do other threads that use the value). A final optimization is that the stop at the end of a successfully-posted thread can be converted to a suspend instead of to the pop-lcv sequence, if the thread does not push anything onto the LCV.

[1] Clearly, this pragma is not strictly necessary, as it is a special case of USE.

Figure 4-3 illustrates this process. Observe that the receive statement is translated into code that loads the frame pointer but does not perform the actual data transfer. If the post is successful, the version of thread 1 located at label L2 will be reached, which transfers the value from the message queue to the frame immediately before use, so the value will be in a register when needed (because a register is an intermediate step in a memory-memory move). If the frame variable islot3 were not used anywhere besides thread 1, the store would not be generated. Because control transfers directly from the inlet to the thread, and the thread does not push anything onto the LCV, no pop of the LCV is needed at the end of the thread. Observe that if the post is unsuccessful, islot3 is immediately moved from the message queue into the frame. Another version of thread 1 that will be the target of fork instructions will be generated that reads islot3 from the frame instead of from the message queue. Because less is known about the history of the LCV, the stop instruction in the duplicate thread is translated into the pseudo-instruction that pops the LCV into the instruction pointer.

4.2 Converting Complex MDP to Simple MDP

This section of the compiler was based on the a compiler stage first described in [21]. Its tasks include:

1. Massaging literal operands into tagged literals.
2. Converting pseudo-instructions into MDP code.
3. Adjusting instructions to use legal MDP addressing modes.
4. Allocating MDP registers.
5. Inserting code to save and restore registers around system calls.

The non-trivial phases are described in this section.

4.2.1 Converting Pseudo-Instructions into MDP Code

There are three classes of actions that can be performed on pseudo-instructions generated by the TL0-to-MDP compiler. A pseudo-instruction can be:

Input	Output
INLET 12	(label (:label partitions_64_inlet12))
RECEIVE islot3.i.g	(move (:message 1) (:j-register A1))
@USE 12.j islot3.i.g 8.t 3.t 1.t	
POSTQ 1.t "default	(reserve (:register genjump))
	(sub (:frame sslot0) 1 (:register genjump))
	(bz (:register genjump) (:tagged-literal rel (:label L2)))
	(move (:register genjump) (:frame sslot0))
	(free (:register genjump))
	(move (:message 2) (:frame islot3))
STOP	
	(suspend)
	(label (:label L2))
THREAD 1	
ITOF freg0.f = islot3.i	(move (:message 2) (:frame islot3))
	(itof float (:frame islot3) (:temporary 0))
STOP	
	(suspend)
THREAD 1	(label (:label partitions_64_thread1))
ITOF freg0.f = islot3.i	(itof float (:frame islot3) (:temporary 0))
STOP	(suspend-string)

Figure 4-3: Optimizations of Message Storage in Flat Implementation. Two versions of thread 1 are created, one that is successfully posted by inlet 12, and one that is reached by other means. This code is a modification of a portion of the 4_PARTITIONS procedure in paraffins.id.

1. translated into MDP instructions,
2. passed untouched to the final stage of the compiler, or
3. translated into MDP assembler pseudo instructions.

Additionally, compiler data structures may be modified.

The translation of the *abs*, *min*, and *max* pseudo-ops depends on the type of the arguments. If the arguments are integers, code is produced to perform the computation. If the arguments are floating-point numbers, a call is made to the appropriate routine in the floating-point library.

Instructions with the following pseudo-opcodes are passed through to the final stage of the compiler: *codeblock*, *comment*, *fall-through*, *suspend-string*, *end-codeblock*, *global-struct*, and *ffree*.

Instructions with the *label* pseudo-opcode modify internal compiler data structures and are passed through. The various *assembler-label* instructions that express the numerical offsets of each frame variable are collected into a single *assembler-label-list* instruction that is passed through. The *mp-falloc* pseudo-instruction is passed through after reserving a register with which to work.

4.2.2 Converting to Legal MDP Operands

For three-address instructions, the MDP requires that the first and last operands be general-purpose registers. The second operand must be representable in *op0 mode*, which encompasses:

- a general purpose register,
- an address register,
- a constant between -16 and $+15$, inclusive,
- one of the following constants: nil (sym:0), false (bool:0), true (bool:0), $80000000, $FF, $3FF, $FFFF, or $FFFFF,
- a general-purpose register plus an address register, or
- a constant between 0 and 15, inclusive, plus an address register.

For most two-address instructions, one operand must be a general-purpose register, while the other must be representable in *extended op0 mode.* Extended op0 encodings are the same as op0 encodings except an extra two bits are available for representing constants. A constant value can thus range from -64 to 63, and a constant offset can range from 0 to 63. The final encoding format, *register-oriented op0 mode* allows access to the MDP's other registers and is fully described in [16].

To understand the conversion process, consider the following complex MDP instruction:

```
(add (:frame islot0) (:tagged-literal int 30) (:frame islot1))
```

Even if "islot0" and "islot1" are small, there are two reasons why the instruction cannot be encoded into one MDP three-operand instruction:

1. The first and last operands of three-address instructions are not general-purpose registers.

2. The second operand is not a legal op0 constant.

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

```
(move (:frame (:tagged-literal int (islot0 . 5))) (: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 (:tagged-literal int (islot1 . 6))))
```

This illustrates the use of intra-instruction temporary registers. 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. This would be a future optimization for commutative instructions.

As another example, consider the complex MDP instruction to move an immediate into a frame slot:

```
(:move (:tagged-literal int 500) (:frame islot3))
```

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

```
(dc (:tagged-literal int 500))
(move (:j-register R0) (:frame (:tagged-literal int (islot3 . 8))))
```

4.2.3 Register Allocation

The system uses a one-pass register allocator scheme that is an extension of the scheme described in [21]. A data structure maps virtual register names to MDP registers. If a reference is made to a virtual register not in the data structure, an appropriate register is allocated. Recall that the first use of a register is always as a destination and that the MDP register R0 can be loaded with large quantities through the DC pseudo-opcode.

1. If the source is one of the values that can be loaded without the DC pseudo-opcode, try allocating R1 or R2. Only allocate R0 if it is the only register left.

2. If the source is big enough to require the DC pseudo-opcode, try allocating R0. If R0 is already reserved:

   - If the virtual register already mapped to R0 is a compiler-generated intra-instruction temporary, an instruction is emitted to move the old contents of R0 into another register, and the binding of the original virtual register mapped to R0 is changed, as illustrated in Figure 4-4.
   - If the virtual register mapped to R0 is a TL0 register, generate an error. Sliding the register as just described could generate incorrect code if the statement is only conditionally executed because later code that uses the register originally in R0 would not know whether to expect it in R0 or in the new register. This scheme could be extended to account for this situation, but the need has not arisen.

A binding is removed by an explicit free instruction (in the case of compiler-generated virtual registers) or by the end of an thread (in the case of TL0 registers).

The R0-conflict maneuver is illustrated by the translation of the following instruction which is the same as the previous example, except we will assume that islot3 = 100:

```
(:move (:tagged-literal int 500) (:frame islot3))
```

It is translated into:

```
(dc (:tagged-literal int 500))
```

Figure 4-4: 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.

```
(move (:j-register R0) (:j-register R3))
(dc (:tagged-literal int (islot3 . 100)))
(move (:j-register R3) (:frame (:j-register R0)))
```

Because only one TL0 register is allowed and because at most two MDP registers will be needed as intra-instruction temporaries, there will never be a need to spill registers.

The register allocator also keeps tracks of what values are in MDP registers and uses this information to generate better code. For example, if an instruction causes 100 to be loaded into R1, and the constant 100 is again needed in an instruction that occurs before R1 has been overwritten, R1 will be used, and the 100 will not be reloaded. This optimization is even more valuable for minimizing memory accesses, as shown in Figure 4-5.

4.2.4 Register Management Around System Calls

Code to perform several steps must be generated to correctly make a system call:

1. Save registers that are in use into the ID registers.

Input Code	Output Code
(move (:message 3) (:frame islot0))	(move (:message 3) (:j-register R1))
	(move (:j-register R1) (:frame (:tagged-literal int (islot0 . 7))))
(add (:frame islot0) 1 (:register reg91))	(add (:j-register R1) (:tagged-literal int 1) (:j-register R2))

Figure 4-5: Register Optimization Example. The second input line can be translated into one instruction instead of two because the register allocator knows that R1 holds the value at offset 7 within the current frame. This optimization eliminates both an instruction and a data access.

2. Load the arguments into the required registers, assigning R0 last in case it is needed as a temporary. (Subtleties arise if the value had been in a register that has been saved away.)

3. Make the appropriate call.

4. Store the return value to the destination operand.

5. Restore the registers that had been in use before the call. (One has to take care not to overwrite the return value.)

4.3 Converting Simple MDP to MDP Assembly

The final stage of the compiler is the simplest and does the following:

- Produces code needed to work around chip bugs.
- Determines whether each branch should be long or short.
- Combines "send" instructions into "send2" instructions where possible.
- Converts the remaining pseudo-instructions into MDP instructions.
- Emits assembler directives indicating where to place code.
- Rewrites instructions into the format expected by the compiler.

The section describes these steps.

Number	Versions	Problem	Workaround
198	A	First instruction after a branch to external memory skipped.	Insert a NOPs after each branch target in external memory.
199	A	First instruction in a message-handler in external memory skipped.	Insert a NOP at the start of each inlet.
200	A & B	Memory operands in first or last send instruction damage M bit.	Send memory operands from registers.

Table 4.10: Chip Bugs Worked Around. Version indicates whether a bug occurs in A-step or B-step chips.

4.3.1 Bug Workarounds

Table 4.10 shows the chip bugs that are worked around. While almost all of the bug workarounds are performed in this module, a bug 200 workaround occurs in the previous stage.[2] Most chip bugs were eliminated in the second (B-step) version of the chip, and future versions of the compiler will not support A-step chips. The fix employed for bug 200 is overkill and could be made more discriminating.

4.3.2 Fixing Up Branches

On the MDP, branches must be within 63 words in either the forward or backward directions. To branch a further distance, the target instruction pointer value must be loaded into the instruction register. This requires two instructions: a DC to load R0 followed by copying R0 into the IP register.

This phase of the compiler passes over the code, counting instruction words to determine the offset of each label. A second pass is performed to determine which branches are more than 63 words long. These branches are converted to the two-instruction sequence, and the label list is modified. The process is repeated until a pass finds no out-of-bounds branches.

[2]The numbering system for chip bugs does not start at zero, so it should not be inferred there are 200 chip bugs!

4.3.3 Combining Send Instructions

The earlier stages of the compiler only generate the "send" instruction, which transfers a single word to the network interface. In this stage of the compiler, a pair of "send" instructions is combined into a "send2" where possible, i.e., when the first operand is encodable in op0 mode and the second is a general-purpose register.

4.3.4 Templates for Converting Instructions

The basic way in which an instruction is converted is by outputting its opcode directly and translating each operand according to the following rules:

`(:lcv (:base X))` → `[X,A0]`

`(:temporary (:base X))` → `[X,A0]`

`(:frame (:base X))` → `[X,A1]`

`(:message (:base X))` → `[X,A3]`

More work is required for pseudo-instructions, shown in Table 4.11.

4.4 Conclusion

This chapter described each stage of the compiler. Except for branch-length determination, each module only takes one pass. While this limits the optimizations that can be performed, peephole optimizations at various stages improve code quality (such as treating a TL0 *fork* followed by a *stop* as a special case and combining MDP *send* instructions), as do the register allocation hacks. Perusal of the code in Appendix A should show that code quality is good.

Pseudo-Opcode	Action
(codeblock-start)	Emit directives to declare a global label indicating the start of a codeblock.
(codeblock-end)	Output the inlet table in a manner such that it will be globally accessible. Output the codeblock's global structures.
(ffree)	Output code to load the base of a codeblock into R0 and to call the "ffree" library routine.

(mp-falloc Rn)	Output the following sequence:
	move [_ALLOC_LOC,A0], Rn
	add Rn, 1, Rn
	and Rn, [_NPROC_MINUS_1_LOC,A0], Rn
	move Rn, [_ALLOC_LOC,A0]
	send1 [Rn,A2]
	where Rn is the register reserved for computing
	a target processor. A2 is always set to the base
	of the processor table.

(suspend-string)	Output the following sequence:
	move [R3,A0], R0
	sub R3, 1, R3
	move R0, IP
	This pops the LCV.

(label L)	Output: L:
(assembler-label-list ...)	For each frame slot name S and offset N, emit the assembler pseudo-instruction: set S = N This allows later instructions to refer to [S,A1].
(comment S)	Output: ; S where S is the comment string.

Table 4.11: Final Expansion of Pseudo-Instructions

Chapter 5

Analysis

Two forms of analysis are described in this chapter. First, I give static instruction counts for sequences to implement TL0 primitives, contrasting the differences in the two implementations. Second, I use dynamic counts of instructions executed, memory accesses, and estimated cycles spent to further highlight differences between the implementations.

5.1 Static Timings

Most MDP instructions take one cycle if they operate on registers, assuming they are executed from internal memory. Estimated cycle penalties are shown in Table 5.1. These penalties are used to calculate the costs in the rest of this section. Table 5.2 shows the costs of library routines common to both the direct and flattened implementations. Table 5.3 shows the costs of functions that differ between the implementations.

5.2 Benefits of Flattened Implementation

To review briefly, there are two major advantages of the flattened implementation over the direct:

- The RCV and its associated overhead are obviated.
- Allowing control to pass directly from inlets to threads opens the code to additional optimizations. (See Section 2.2.3.)

Instruction Type	Cycle Penalty
DC	1
NOP for alignment	1
dispatch	3
branch taken	1
fault/call	2
on-chip memory access	1
off-chip memory access	6

Table 5.1: Penalty Cycles for MDP Instructions. NOPs are inserted to align branches on word boundaries.

Routine	Condition	Cycles	Instructions
myMalloc	average	53	33
sys-alloc	frame available	64	23
	frame unavailable	88+53	30+33
sys-alloc-raw	frame available	57	18
	frame unavailable	81+53	25+33
ffree	library routine	43	12
	getting there	8.5	4
	total	51.5	16
i-fetch	data available	32	12
	data not written	86+53	35+33
	thunk not written	146+53	42+33
i-store1	constant cost	35	11
	each waiting cont.	58	15
i-store2	constant cost	36	11
	each waiting cont.	59	15
nfork		0–6	0–3
sfork	successful	10–19	3–8
	unsucessful	16–22	4–6

Table 5.2: Instruction Counts of Implementation-Independent Sequences. The functions “i-store1” and “i-store2” store one and two words, respectively. “Nfork” and “sfork” represent non-synchronizing and synchronizing forks, respectively. The second term of timings listed as a sum indicates the time spent in myMalloc.

Function	Condition	Direct		Flat	
		Cycles	Instructions	Cycles	Instructions
swap	constant cost	67	26	—	—
	each thread	14	6	—	—
npost	getting to library	5–6	2	—	—
	idle	50	16	—	—
	ready	35	11	—	—
	running	10	8	0	0
spost	getting to library	6–9	3–4	—	—
	succ. idle	59	19	—	—
	succ. ready	44	14	—	—
	succ. running	19	11	10–13	3–4
	unsucc. idle	18	5	—	—
	unsucc. ready	18	5	—	—
	unsucc. running	18	5	16–19	4–5

Table 5.3: Comparison of Costs Between the Two Systems. The functions "npost" and "spost" are for non-synchronizing and synchronizing posts respectively.

The primary disadvantages of the flattened implementation are that it reduces locality, which would harm performance on a machine with code and data caches, and it places more demand on the message queues, which could be a significant problem as larger programs are run. In this section, we quantify the advantages. In future research, we will quantify the negative effects.

The program used for evaluation was paraffins, a 175-line Id program described in [15]. It is included in Appendix B. Paraffins was selected because it uses most features of Id: lists, I-structures, algebraic types, bounded loops, array and list comprehensions, and floating-point math [15]. Larger progams, such as gamteb and simple could not be compiled because they used transcendental functions not yet supported by our floating-point libraries.

5.2.1 Comparison of Direct and Flattened Implementations

Table 5.4 shows instruction counts and memory access for paraffins running with an argument of 10 on a single processor. The flattened implementation required only 76% the instruction counts of the direct implementation. For analysis purposes, all user code was placed off chip and library code on chip. As expected, the direct implementation had substantially

	direct	flattened	ratio
cfuture faults	347	612	1.76
on-chip reads	141665	126965	.90
on-chip writes	131822	118481	.90
on-chip fetches	516716	340406	.66
off-chip reads	124113	97932	.79
off-chip writes	102395	81699	.80
off-chip fetches	658676	548950	.83
estimated CPI	4.15	4.39	1.06
instruction count	1195498	909893	.76
counts×CPI	4961317	3994430	.81

Table 5.4: Paraffins Dynamic Instruction Counts and Memory Accesses for Direct and Flattened Implementations. It is assumed that library routines are in on-chip memory and user code is stored off-chip. Data in on-chip memory consists of operating system variables and tables, message queues, and, in the direct implementation, the RCV. The instruction count is greater than the sum of on-chip and off-chip fetches due to cycles spent doing hardware dispatch.

more on-chip fetches than the flattened implementation. This presumably corresponds to the time spent in the library routines to post threads, to manage the ready frame queue, and to swap in the RCV. (Visually-oriented readers will want to refer back to the graphs on page 33.) The difference in off-chip reads and writes appears to be due to the special flat optimizations that reduce access of frame memory. These optimizations will be discussed more below. Because the direct implemenation makes better use of internal memory than the flattened implementation, it has a lower CPI (4.15 vs. 4.39). After scaling instruction counts by CPI, the flattened implementation requires 81% the time of the direct implementation, a substantial improvement.

An unexpected difference is that 76% more cfuture faults occurred under the flattened implemenation. To review, cfuture faults are caused by reads before writes of I-structure, M-structure, and thunks. It is not seen why more such faults would occur in the flattened implementation. There are two possible responses, either:

1. It is bad luck and an artifact of this one program that more cfuture faults occurred under the flat implementation, and its performance should thus look even better over a larger benchmark set.

	without opts	with opts	ratio
cfuture faults	612	612	1
on-chip reads	126930	126965	1.00
on-chip writes	118476	118481	1.00
on-chip fetches	342915	340406	.99
off-chip reads	106239	97932	.92
off-chip writes	87594	81699	.93
off-chip fetches	578718	548950	.95
estimated CPI	4.45	4.39	.99
instruction count	942170	909893	.97
counts×CPI	4192656	3994430	.95

Table 5.5: Paraffins Dynamic Instruction Counts and Memory Accesses for Flattened Implementations With and Without Flat Optimizations

2. There is something about the flat implementation that leads to more reads before writes.

Further research is needed to determine which is the case. I recalculated the statistics in Table 5.4 as though both implementations had had 612 cfuture faults and discovered that the effect on the bottom line was negligible.

5.2.2 Benefits of Special Flat Optimizations

To get a better understanding of what gains of the flattened implementation were due to the special flat optimizations, I computed the same statistics with the optimizations turned of. The results are shown in Table 5.5.

The optimizations lowered the instruction count (slightly) and the number of off-chip reads and writes significantly. In improved the bottom line by 5%. Thus, the majority of the benefit of using the flattened implementation is eliminating the RCV, although the flat optimizations also contribute.

5.2.3 Effect on Locality

By not consolidating threads corresponding to the same frame, the flattened implementation damages locality of reference. This is not a factor for the J-Machine, but it impacts how relevant the benefits of the flattened implementation are to machines that have caches. In

processors	Threads per Quantum direct	flat	Quanta per Activation direct	flat
1	8.56	4.25	6.89	13.86
4	7.37	4.27	8.00	13.82
16	7.16	4.29	8.23	13.74

Table 5.6: Threads per Quantum and Quanta per Activation as a Function of Processors and Implementation Method. There were a total of 430 activations and 25355 threads for each run.

this section, I compare threads per quantum (TPQ) and quanta per activation (QPA) in the two implementations. TPQ is related to how long a frame remains swapped in, while QPA indicates how many times a frame is swapped in during its lifetime.

The method for determining quanta in the direct implementation is straightforward: Each thread increments a counter that is recorded and reset in the swap routine. In the flat implementation, the accounting is slightly more complex. The code pointed to by the underflow entry in the LCV records the TPQ counter (along with the frame pointer) and resets it.[1] If an inlet posts unsuccessfully, no TPQ count is emitted. In a second stage, adjacent TPQ values are added if they correspond to the same frame. Thus, the sequence inlet-thread followed by inlet-thread with the same frame pointer would be counted as one quantum of size two. The justification for this accounting method is that it better predicts cache performance.

The lower value for TPQ (ranging from 50% to 60%) Table 5.6 suggests that locality is significantly worse in the flattened implementation, as one would predict. Not enough information is available from the numbers, however, to determine exactly how much cache performance is degraded by using the flat implementation instead of the direct. This will be an area of future research.

5.2.4 Multiprocessor Performance

For both systems, I ran paraffins on 1, 4, and 16 processors. Table 5.7 shows the speedups. The diminishing returns of additional processors is not unique to my implementation. Accord-

[1] Observe that measuring TPQ and QPA is intrusive and that the suspend optimization described in Section 2.2.3 must be disabled.

Processors	Direct		Flat		Ratio	
	Count	Speed-up	Count	Speed-up	Count	Speed-up
1	1195498	1	909893	1	.76	1
4	436260	2.74	320966	2.83	.74	1.03
16	260738	4.59	181004	5.02	.69	1.09

Table 5.7: Parallel Speed-Up of Paraffins. Speed-up for each system is relative to 1 processor running the same implementation.

ing to a study performed on Monsoon [5], paraffins appears to lack parallelism, although they did get better speed-up than we did (3.52 for four processors and 5.52 for eight processors). It is not clear why better speed-up occured for the flattened implementation than the direct. This is an area for future research.

To visualize the processor usage in paraffins, I graphed how busy each processor was during the execution of the program. The graph for the flat implementation running with 4 processors is shown in Figure 5-1 and the graph for 16 processors is shown in Figure 5-2. It is clear that, especially for 16 processors, the load-balancing was poor.

5.3 Summary

While the data is not yet conclusive, the flattened implementation appears to substantially outperform the direct implementation. As expected, the direct implementation spent more instruction counts and a larger percentage of its instruction counts in library routines. An area of concern is the flattened implementation's substantially lower average TPQ. The concluding chapter describes further experiments for a more thorough analysis.

Figure 5-1: Profile of Paraffins Running on 4 Processors. The darkness of a line indicates how busy the processor was during the 1000-instruction interval represented by the line.

Figure 5-2: Profile of Paraffins Running on 16 Processors. The darkness of a line indicates how busy the processor was during the 1000-instruction interval represented by the line.

Chapter 6

Conclusions

Several achievements were described in this document:

- A parallelizing TAM-based compiler was built for the J-Machine that supports almost the entire Id language and runs on the actual hardware.
- Two different back ends and supporting library routines were successfully built that implement interesting different compiling approaches.
- Preliminary comparisons of the two implementations were made.

The rest of this chapter describes further research that should be performed and what conclusions can be drawn from the current research.

6.1 Areas for Further Study

There are severals ways in which this work could be expanded to provide more useful information. These directions are described in this section.

6.1.1 Supporting the Entire Id Language

Only a few TL0 instructions are not supported. These can be easily added. Additionally, a more complete and accurate floating-point library is needed to support the entire Id language. Such a library is being built at Caltech and can be incorporated into this system.

The benefit of supporting the entire language is that it will allow other large programs besides paraffins to be compiled and run. A larger program suite is needed to evaluate performance.

6.1.2 Analyzing Cache Effects

While the flattened implementation outperforms the direct implementation on the J-Machine, it is not clear how much better it would be on a system with a cache. It would be easy to make runs produce address traces that could be processed by a cache simulator, enabling one to quantify how much the lower TPQ of the flattened implementation hurts cache performance (or, alternately, how much larger a cache would be required for the flattened implementation to result in the same cache miss ratio as for the direct implementation).

6.1.3 Performing Timings on the Hardware

Measurements taken on the actual hardware, particularly as it grows larger, would be valuable. The current research does not address whether network latency, for example, affects the two implementations differently. Additionally, using the hardware would allow larger programs and problem sizes to be computed.

6.1.4 Generating More Statistical Information

The compiler could be easily modified to generate more statistics about the code produced. (Indeed, work has begun in this area.) This would be useful in evaluating the J-Machine, particularly how much performance is harmed by having so few registers and limited addressing modes.

6.1.5 Making Comparisons to Other Hardware Running the Same Software

The work described in this document has the potential for being a useful basis for comparing the J-Machine with other computers that run Id, both dataflow and general-purpose. A comparison to Monsoon [18] would help evaluate whether one needs special-purpose dataflow machines or if general-purpose machines are equally or more efficient. A comparison to the Connection Machine 5 (CM-5) would allow one to evaluate the effectiveness of the J-Machine's mechanisms for parallelism, such as the multiple priority levels, hardware-managed message queues, and fast network.

The comparisons would be especially compelling because there are Id compilers for both Monsoon and the CM-5. Both use the MIT Computation Structures Group Id compiler, and the CM-5 compiler is also TAM-based and uses the same software as my implementations except for the back end. This would allow comparisons among the platforms to be a controlled experiment.

6.1.6 Adding Software-Controlled Caching

Because it is so expensive to fetch instructions and to access data in external memory, it might be valuable to implement software-controlled caching. This could bring the MDP's performance more in line with other processors that have caches.

6.1.7 Better Load Balancing

As described in the previous chapter, load balancing has room for improvement. An obvious first step is distributing large I-structures and M-structures across multiple processors. If that does not improve load balancing enough, the method for determining which processor on which to allocate a frame will have to be changed.

6.1.8 Research Plans

My plans are to implement all of the ideas mentioned above with the exception of software-controlled caching. While adding software-controlled caching could improve the performance of the system, it would not contribute greatly to drawing widely-applicable lessons, as it is accepted that processors should have hardware-controlled caches.

6.2 Conclusions

A major purpose of this research is to evaluate the features of the J-Machine, which was designed to support fine-grained parallel programs. The J-Machine features that I found most useful were:

- the ease and speed with which messages are sent. On most machines, the expansion of the TL0 *send* instruction would be much longer and more complicated.

- the multiple priority levels. Both implementations used two priority levels in order to quickly process certain classes of requests.

- the hardware support for message reception. On the MDP, user or operating system code does not have to poll the network. Dispatch on messages occurs automatically. Additionally, automatic buffering of message words enables easy access to incoming data.

- fast context switch. The MDP's extremely quick context switching time is necessary for fine-grained computing.

J-Machine features that were of mixed success were:

- cfuture tag support. The only place cfuture tags were used was to support I-structures, M-structures, and thunks. I could have managed easily without the cfuture fault capability by explicitly checking the tag. Alternately, it would not have been very costly to use separate words to represent tags if tags were not supported on the MDP.

- programmer-resizable message queues. It was useful that I could specify the size of each of the message queues, and large message queues were necessary for the flattened implementation with its delayed processing of inlets; however, the 1-Kword size limit may prove oppressive, especially for the flattened implementation.

Other J-Machine features impaired performance:

- Four general-purpose registers is simply too few.

- The lack of a hardware-controlled cache delayed access to user code and data.

In addition to these narrow conclusions, we can also draw a broader one: The minimal overhead of the code, particularly in the flattened implementation, supports the position that dataflow machines are not necessary to efficiently execute dataflow programs.

Appendix A

Representations of Factorial

A.1 Id

```
typeof fact = I -> I;

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

A.2 TL0

```
% TL0 Version 2.1
% CURRENT ID2TL COMPILER OPTIONS
% VERSION-NUMBER: 017
% K-BOUNDED-LOOPS: T
% BASIC-PARTITIONING: DEPENDENCE-SETS-PARTITIONING
% MERGE-PARTITIONS: T
% USE-REGISTERS: IN-THREADS
% RE-USE-REGISTERS: T
% RE-USE-FRAME-SLOTS: NIL
% MOVE-INSERTION: ALWAYS
% COLLECT-STATISTICS: NIL
% COLLECT-GMG-STATISTICS: NIL
% ONLY-POST-PRAGMA: T
% ONLY-USE-PRAGMA: T

% TYPE-SIGNATURE: (I -> I)

GLOBAL_STRUCT gc_FACT.ps = gc_0_FACT.ps.g END_STRUCT
GLOBAL_STRUCT gc_0_FACT.ps = FACT.pc.g  1.i.g EMPTY.g NIL.ps.g END_STRUCT


CBLOCK FACT.pc
TL0_VERSION 2.1
% External references
EXTERN
% Named constants
CONSTANT
% Activation frame header layout
FRAME_HEADER
% Activation frame body contents and continuation vector sizes
FRAME_BODY RCV=6 LCV=11
        islot0.i
        islot1.i
        islot2.i
        islot3.i
        islot4.i
        pfslot0.pf
        pfslot1.pf
        psslot0.ps
        jslot0.j
        sslot0.s
        sslot1.s
```

```
        sslot2.s
        gslot0.g

@PARENT pfslot0.pf
% Register variables used in inlets
INLET_REG
% Register variables used in threads
REGISTER
        breg0.b
%
% Inlets
%
INLET 0
        RECEIVE pfslot0.pf jslot0.j
        @USE 0.j pfslot0.pf 0.t 6.t
        @USE 0.j jslot0.j 0.t 6.t
        FINIT
        MOVE sslot2.s = 2.s
        MOVE sslot1.s = 2.s
        MOVE sslot0.s = 3.s
        SET_ENTER 11.t % set enter-activation
        SET_LEAVE 12.t % set leave-activation
        SEND pfslot0.pf[jslot0.j+-1.i] <- fp.pf  % send frame pointer back
        POSTQ 0.t "default
        STOP
INLET 11
        RECEIVE psslot0.ps
        @USE 11.j psslot0.ps
        POSTQ 1.t "default
        @ONLY_POST 11.j 1.t
        STOP
INLET 12
        RECEIVE islot1.i.g
        @USE 12.j islot1.i.g 5.t 7.t 9.t 2.t
        POSTQ 2.t "default
        @ONLY_POST 12.j 2.t
        STOP
INLET 14
        RECEIVE pfslot1.pf
        @USE 14.j pfslot1.pf 8.t
        @ONLY_USE 14.j pfslot1.pf 8.t
        POSTQ 8.t "default
        STOP
INLET 15
        RECEIVE
        POSTQ 10.t "default
        @ONLY_POST 15.j 10.t
        STOP
INLET 16
        RECEIVE islot4.i.g
        @USE 16.j islot4.i.g 9.t
        @ONLY_USE 16.j islot4.i.g 9.t
        POSTQ 9.t "default
        STOP
%
% threads
%
THREAD 0
        SYNC sslot0.s

% INLET inlet0 = pfslot0.pf jslot0.j thr0
        SEND pfslot0.pf[jslot0.j+0.i] <-
        FFREE fp.pf "default
        SWAP "default     % swap to next activation
        STOP
THREAD 1
% INLET inlet11 = psslot0.ps thr1
        STOP
THREAD 2
% INLET inlet12 = islot1.i thr2
        LE breg0.b = islot1.i 1.i
        SWITCH breg0.b 5.t 7.t
        SWITCH breg0.b 3.t 4.t
        STOP
THREAD 3
        FORK 0.t
        STOP
THREAD 4
        FALLOC 14.j = FACT.pc "default
        STOP
THREAD 5
        MOVE islot0.i = islot1.i
        FORK 6.t
        STOP
THREAD 6
        SEND pfslot0.pf[jslot0.j+1.i] <- islot0.i.g
        FORK 0.t
        STOP
THREAD 7
        SUB islot2.i = islot1.i 1.i
        FORK 8.t
        FORK 9.t
        STOP
THREAD 8
        SYNC sslot2.s

% INLET inlet14 = pfslot1.pf thr8
        SEND pfslot1.pf/FACT.pc[0.i/FACT.pc+12.i] <- islot2.i.g
```

```
        STOP
THREAD 9
        SYNC sslot1.s

% INLET inlet16 = islot4.i thr9
        MUL islot3.i = islot1.i islot4.i
        MOVE islot0.i = islot3.i
        FORK 6.t
        STOP
THREAD 10
% INLET inlet15 = thr10
        FORK 0.t
        STOP
THREAD 11 % enter activation
        STOP % no registers to restore...
THREAD 12 % leave activation
        SWAP "default        % swap to next activation
        STOP   % no registers to save...
END_CBLOCK
```

A.3 Direct Implementation

A.3.1 Complex MDP

```
(global-struct (:label gc_fact) (:ref gc_0_fact_0) (:ref gc_0_fact_1))
(global-struct (:label gc_0_fact) (:ref fact_0) (:ref fact_1) 1
              (:tagged-literal sym 0) (:tagged-literal cfut -1)
              (:tagged-literal cfut -1) (:tagged-literal sym 0)
              (:tagged-literal sym 0))
(codeblock "fact")
(comment "file FACT.tl0")
(comment "line 22")
(comment "FRAME SLOTS")
(comment "Frame header")
(comment "Frame body")
(assembler-label islot0 4)
(assembler-label islot1 5)
(assembler-label islot2 6)
(assembler-label islot3 7)
(assembler-label islot4 8)
(assembler-label pfslot0_0 10)
(assembler-label pfslot0_1 11)
(assembler-label pfslot1_0 12)
(assembler-label pfslot1_1 13)
(assembler-label psslot0_0 14)
(assembler-label psslot0_1 15)
(assembler-label jslot0 9)
(assembler-label sslot0 16)
(assembler-label sslot1 17)
(assembler-label sslot2 18)
(assembler-label gslot0_0 20)
(assembler-label gslot0_1 21)
(comment "Frame size = 33")
(dc 33)
(dc 6)
(dc (:tagged-literal sym 0))
(dc (:tagged-literal local-ip (:label fact_inlet0)))
(comment "line 55")
(start inlet 0)
(label (:label fact_inlet0))
(move (:message 1) (:j-register A1))
(move (:message 2) (:frame pfslot0_0))
(move (:message 3) (:frame pfslot0_1))
(move (:message 4) (:frame jslot0))
(comment "line 57")
(comment "line 58")
(comment "line 59")
(comment "line 60")
(move 2 (:frame sslot2))
(comment "line 61")
(move 2 (:frame sslot1))
(comment "line 62")
(move 3 (:frame sslot0))
(comment "line 63")
(comment "line 64")
(comment "line 65")
(send0 (:frame pfslot0_0))
(reserve (:register add-inlet))
(add (:frame jslot0) -1 (:register add-inlet))
(send0 (:temporary (:register add-inlet)))
(free (:register add-inlet))
(send0 (:frame pfslot0_1))
(send0 (:j-register NNR))
(sende0 (:j-register A1))
(comment "line 66")
(move 16 (:j-register R1))
(move (:tagged-literal local-ip (:label fact_thread0)) (:j-register r0))
(call maybe_post_thread)
(end-inlet)
(end inlet 0)
(comment "line 68")
```

```
(start inlet 11)
(label (:label fact_inlet11))
(move (:message 1) (:j-register A1))
(move (:message 2) (:frame psslot0_0))
(move (:message 3) (:frame psslot0_1))
(comment "line 70")
(comment "line 71")
(move (:tagged-literal local-ip (:label fact_thread1)) (:j-register r0))
(call post_thread)
(end-inlet)
(end inlet 11)
(comment "line 74")
(start inlet 12)
(label (:label fact_inlet12))
(move (:message 1) (:j-register A1))
(move (:message 2) (:frame islot1))
(comment "line 76")
(comment "line 77")
(move (:tagged-literal local-ip (:label fact_thread2)) (:j-register r0))
(call post_thread)
(end-inlet)
(end inlet 12)
(comment "line 80")
(start inlet 14)
(label (:label fact_inlet14))
(move (:message 1) (:j-register A1))
(move (:message 2) (:frame pfslot1_0))
(move (:message 3) (:frame pfslot1_1))
(comment "line 82")
(comment "line 83")
(comment "line 84")
(move 18 (:j-register R1))
(move (:tagged-literal local-ip (:label fact_thread8)) (:j-register r0))
(call maybe_post_thread)
(end-inlet)
(end inlet 14)
(comment "line 86")
(start inlet 15)
(label (:label fact_inlet15))
(move (:message 1) (:j-register A1))
(comment "line 88")
(move (:tagged-literal local-ip (:label fact_thread10)) (:j-register r0))
(call post_thread)
(end-inlet)
(end inlet 15)
(comment "line 91")
(start inlet 16)
(label (:label fact_inlet16))
(move (:message 1) (:j-register A1))
(move (:message 2) (:frame islot4))
(comment "line 93")
(comment "line 94")
(comment "line 95")
(move 17 (:j-register R1))
(move (:tagged-literal local-ip (:label fact_thread9)) (:j-register r0))
(call maybe_post_thread)
(end-inlet)
(end inlet 16)
(comment "line 100")
(start thread 0)
(label (:label fact_thread0))
(move 0 (:j-register r0))
(move (:j-register R0) (:j-register i))
(nop)
(move 1 (:j-register r0))
(move (:j-register R0) (:j-register i))
(comment "line 104")
(send0 (:frame pfslot0_0))
(reserve (:register add-inlet))
(add (:frame jslot0) 0 (:register add-inlet))
(send0 (:temporary (:register add-inlet)))
(free (:register add-inlet))
(sende0 (:frame pfslot0_1))
(comment "line 105")
(ffree)
(end thread 0)
(comment "line 108")
(start thread 1)
(label (:label fact_thread1))
(move 0 (:j-register r0))
(move (:j-register R0) (:j-register i))
(nop)
(move 1 (:j-register r0))
(move (:j-register R0) (:j-register i))
(suspend-string)
(end thread 1)
(comment "line 111")
(start thread 2)
(label (:label fact_thread2))
(move 0 (:j-register r0))
(move (:j-register R0) (:j-register i))
(nop)
(move 1 (:j-register r0))
(move (:j-register R0) (:j-register i))
(comment "line 113")
(le (:frame islot1) 1 (:register temp0b))
(comment "line 114")
(tl0 start nswitch)
```

```
(bf (:register temp0b) (:tagged-literal rel (:label L0)))
(add (:j-register R3) 1 (:j-register R3))
(move (:tagged-literal local-ip (:label fact_thread5)) (:lcv (:j-register R3)))
(br (:tagged-literal rel (:label L1)))
(label (:label L0))
(add (:j-register R3) 1 (:j-register R3))
(move (:tagged-literal local-ip (:label fact_thread7)) (:lcv (:j-register R3)))
(label (:label L1))
(tl0 end nswitch)
(comment "line 115")
(tl0 start sswitch)
(bt (:register temp0b) (:tagged-literal rel (:label fact_thread3)))
(comment "   fall through")
(tl0 end sswitch)
(end thread 2)
(comment "line 120")
(start thread 4)
(label (:label fact_thread4))
(move 0 (:j-register r0))
(move (:j-register R0) (:j-register i))
(nop)
(move 1 (:j-register r0))
(move (:j-register R0) (:j-register i))
(comment "line 121")
(mp-falloc-setup)
(send1 (:ref sys_alloc_msg_ref))
(send1 (:ref fact_1))
(send1 (:j-register nnr))
(send1 (:j-register A1))
(sende1 (:ref fact_inlet_14_pointer_ref))
(suspend-string)
(end thread 4)
(comment "line 117")
(start thread 3)
(label (:label fact_thread3))
(move 0 (:j-register r0))
(move (:j-register R0) (:j-register i))
(nop)
(move 1 (:j-register r0))
(move (:j-register R0) (:j-register i))
(comment "line 118")
(tl0 start sfall)
(reserve (:register genjump))
(sub (:frame sslot0) 1 (:register genjump))
(bz (:register genjump) (:tagged-literal rel (:label fact_thread0)))
(move (:register genjump) (:frame sslot0))
(free (:register genjump))
(suspend-string)
(tl0 end sfall)
(end thread 3)
(comment "line 123")
(start thread 5)
(label (:label fact_thread5))
(move 0 (:j-register r0))
(move (:j-register R0) (:j-register i))
(nop)
(move 1 (:j-register r0))
(move (:j-register R0) (:j-register i))
(comment "line 124")
(move (:frame islot1) (:frame islot0))
(comment "line 125")
(tl0 start nfall)
(comment "   fall through")
(tl0 end nfall)
(end thread 5)
(comment "line 127")
(start thread 6)
(label (:label fact_thread6))
(move 0 (:j-register r0))
(move (:j-register R0) (:j-register i))
(nop)
(move 1 (:j-register r0))
(move (:j-register R0) (:j-register i))
(comment "line 128")
(send0 (:frame pfslot0_0))
(reserve (:register add-inlet))
(add (:frame jslot0) 1 (:register add-inlet))
(send0 (:temporary (:register add-inlet)))
(free (:register add-inlet))
(send0 (:frame pfslot0_1))
(send0 (:frame islot0))
(sende0 (:tagged-literal sym 0))
(comment "line 129")
(tl0 start sfall)
(reserve (:register genjump))
(sub (:frame sslot0) 1 (:register genjump))
(bz (:register genjump) (:tagged-literal rel (:label fact_thread0)))
(move (:register genjump) (:frame sslot0))
(free (:register genjump))
(suspend-string)
(tl0 end sfall)
(end thread 6)
(comment "line 131")
(start thread 7)
(label (:label fact_thread7))
(move 0 (:j-register r0))
(move (:j-register R0) (:j-register i))
(nop)
```

```
(move 1 (:j-register r0))
(move (:j-register R0) (:j-register i))
(comment "line 132")
(sub (:frame islot1) 1 (:frame islot2))
(comment "line 133")
(tl0 start sfork)
(reserve (:register genjump))
(sub (:frame sslot2) 1 (:register genjump))
(bnz (:register genjump) (:tagged-literal rel (:label label4)))
(add (:j-register R3) 1 (:j-register R3))
(move (:tagged-literal local-ip (:label fact_thread8)) (:lcv (:j-register R3)))
(br (:tagged-literal rel (:label label5)))
(label (:label label4))
(move (:register genjump) (:frame sslot2))
(label (:label label5))
(free (:register genjump))
(tl0 end sfork)
(comment "line 134")
(tl0 start sfall)
(reserve (:register genjump))
(sub (:frame sslot1) 1 (:register genjump))
(bz (:register genjump) (:tagged-literal rel (:label fact_thread9)))
(move (:register genjump) (:frame sslot1))
(free (:register genjump))
(suspend-string)
(tl0 end sfall)
(end thread 7)
(comment "line 142")
(start thread 9)
(label (:label fact_thread9))
(move 0 (:j-register r0))
(move (:j-register R0) (:j-register i))
(nop)
(move 1 (:j-register r0))
(move (:j-register R0) (:j-register i))
(comment "line 146")
(mul (:frame islot1) (:frame islot4) (:frame islot3))
(comment "line 147")
(move (:frame islot3) (:frame islot0))
(comment "line 148")
(tl0 start nfall)
(br (:tagged-literal rel (:label fact_thread6)))
(tl0 end nfall)
(end thread 9)
(comment "line 136")
(start thread 8)
(label (:label fact_thread8))
(move 0 (:j-register r0))
(move (:j-register R0) (:j-register i))
```

```
(nop)
(move 1 (:j-register r0))
(move (:j-register R0) (:j-register i))
(comment "line 140")
(send0 (:frame pfslot1_0))
(send0 (:ref fact_inlet_12_ref))
(send0 (:frame pfslot1_1))
(send0 (:frame islot2))
(sende0 (:tagged-literal sym 0))
(suspend-string)
(end thread 8)
(comment "line 150")
(start thread 10)
(label (:label fact_thread10))
(move 0 (:j-register r0))
(move (:j-register R0) (:j-register i))
(nop)
(move 1 (:j-register r0))
(move (:j-register R0) (:j-register i))
(comment "line 152")
(tl0 start sfall)
(reserve (:register genjump))
(sub (:frame sslot0) 1 (:register genjump))
(bz (:register genjump) (:tagged-literal rel (:label fact_thread0)))
(move (:register genjump) (:frame sslot0))
(free (:register genjump))
(suspend-string)
(tl0 end sfall)
(end thread 10)
(comment "line 154")
(start thread 11)
(label (:label fact_thread11))
(move 0 (:j-register r0))
(move (:j-register R0) (:j-register i))
(nop)
(move 1 (:j-register r0))
(move (:j-register R0) (:j-register i))
(suspend-string)
(end thread 11)
(comment "line 156")
(start thread 12)
(label (:label fact_thread12))
(move 0 (:j-register r0))
(move (:j-register R0) (:j-register i))
(nop)
(move 1 (:j-register r0))
(move (:j-register R0) (:j-register i))
(comment "line 157")
(suspend-string)
```

```
(end thread 12)
(inlet-handler fact  (0 5) (11 4) (12 4) (14 4) (15 2) (16 4))
(end-codeblock fact  (0 5) (11 4) (12 4) (14 4) (15 2) (16 4))
```

A.3.2 Simple MDP

```
((GSLOT0_1 . 21)
 (GSLOT0_0 . 20)
 (SSLOT2 . 18)
 (SSLOT1 . 17)
 (SSLOT0 . 16)
 (JSLOT0 . 9)
 (PSSLOT0_1 . 15)
 (PSSLOT0_0 . 14)
 (PFSLOT1_1 . 13)
 (PFSLOT1_0 . 12)
 (PFSLOT0_1 . 11)
 (PFSLOT0_0 . 10)
 (ISLOT4 . 8)
 (ISLOT3 . 7)
 (ISLOT2 . 6)
 (ISLOT1 . 5)
 (ISLOT0 . 4))
((GLOBAL-STRUCT (:LABEL GC_FACT) (:TAGGED-LITERAL SPECIAL_TAG (:REF GC_0_FACT_0))
                (:TAGGED-LITERAL SPECIAL_TAG (:REF GC_0_FACT_1)))
 (GLOBAL-STRUCT (:LABEL GC_0_FACT) (:TAGGED-LITERAL SPECIAL_TAG (:REF FACT_0))
                (:TAGGED-LITERAL SPECIAL_TAG (:REF FACT_1)) (:TAGGED-LITERAL INT 1)
                (:TAGGED-LITERAL SYM 0) (:TAGGED-LITERAL CFUT -1)
                (:TAGGED-LITERAL CFUT -1) (:TAGGED-LITERAL SYM 0)
                (:TAGGED-LITERAL SYM 0))
 (CODEBLOCK "fact")
 (COMMENT "file FACT.tl0")
 (COMMENT "line 22")
 (COMMENT "FRAME SLOTS")
 (COMMENT "Frame header")
 (COMMENT "Frame body")
 (COMMENT "Frame size = 33")
 (DC (:TAGGED-LITERAL INT 33))
 (DC (:TAGGED-LITERAL INT 6))
 (DC (:TAGGED-LITERAL SYM 0))
 (DC (:TAGGED-LITERAL LOCAL-IP (:LABEL FACT_INLET0)))
 (COMMENT "line 55")
 (START INLET 0)
 (LABEL (:TAGGED-LITERAL LABEL FACT_INLET0))
 (MOVE (:MESSAGE (:TAGGED-LITERAL INT 1)) (:J-REGISTER R2))
 (MOVE (:J-REGISTER R2) (:J-REGISTER A1))
 (MOVE (:MESSAGE (:TAGGED-LITERAL INT 2)) (:J-REGISTER R1))
 (MOVE (:J-REGISTER R1) (:FRAME (:TAGGED-LITERAL INT (PFSLOT0_0 . 10))))
 (MOVE (:MESSAGE (:TAGGED-LITERAL INT 3)) (:J-REGISTER R2))
 (MOVE (:J-REGISTER R2) (:FRAME (:TAGGED-LITERAL INT (PFSLOT0_1 . 11))))
 (MOVE (:MESSAGE (:TAGGED-LITERAL INT 4)) (:J-REGISTER R1))
 (MOVE (:J-REGISTER R1) (:FRAME (:TAGGED-LITERAL INT (JSLOT0 . 9))))
 (COMMENT "line 57")
 (COMMENT "line 58")
 (COMMENT "line 59")
 (COMMENT "line 60")
 (MOVE (:TAGGED-LITERAL INT 2) (:J-REGISTER R2))
 (MOVE (:J-REGISTER R2) (:FRAME (:TAGGED-LITERAL INT (SSLOT2 . 18))))
 (COMMENT "line 61")
 (MOVE (:J-REGISTER R2) (:FRAME (:TAGGED-LITERAL INT (SSLOT1 . 17))))
 (COMMENT "line 62")
 (MOVE (:TAGGED-LITERAL INT 3) (:J-REGISTER R1))
 (MOVE (:J-REGISTER R1) (:FRAME (:TAGGED-LITERAL INT (SSLOT0 . 16))))
 (COMMENT "line 63")
 (COMMENT "line 64")
 (COMMENT "line 65")
 (SEND0 (:FRAME (:TAGGED-LITERAL INT (PFSLOT0_0 . 10))))
 (MOVE (:FRAME (:TAGGED-LITERAL INT (JSLOT0 . 9))) (:J-REGISTER R2))
 (ADD (:J-REGISTER R2) (:TAGGED-LITERAL INT -1) (:J-REGISTER R1))
 (SEND0 (:TEMPORARY (:J-REGISTER R1)))
 (SEND0 (:FRAME (:TAGGED-LITERAL INT (PFSLOT0_1 . 11))))
 (MOVE (:J-REGISTER NNR) (:J-REGISTER R2))
 (SEND0 (:J-REGISTER R2))
 (SENDE0 (:J-REGISTER A1))
 (COMMENT "line 66")
 (MOVE (:TAGGED-LITERAL INT 16) (:J-REGISTER R1))
 (DC (:TAGGED-LITERAL LOCAL-IP (:LABEL FACT_THREAD0)))
 (CALL "maybe_post_it_call")
 (END 8 INLET 0)
 (COMMENT "line 68")
 (START INLET 11)
 (LABEL (:TAGGED-LITERAL LABEL FACT_INLET11))
 (MOVE (:MESSAGE (:TAGGED-LITERAL INT 1)) (:J-REGISTER R1))
 (MOVE (:J-REGISTER R1) (:J-REGISTER A1))
 (MOVE (:MESSAGE (:TAGGED-LITERAL INT 2)) (:J-REGISTER R2))
 (MOVE (:J-REGISTER R2) (:FRAME (:TAGGED-LITERAL INT (PSSLOT0_0 . 14))))
 (MOVE (:MESSAGE (:TAGGED-LITERAL INT 3)) (:J-REGISTER R1))
 (MOVE (:J-REGISTER R1) (:FRAME (:TAGGED-LITERAL INT (PSSLOT0_1 . 15))))
 (COMMENT "line 70")
 (COMMENT "line 71")
 (DC (:TAGGED-LITERAL LOCAL-IP (:LABEL FACT_THREAD1)))
 (CALL "post_it_call")
 (END 3 INLET 11)
```

```
(COMMENT "line 74")
(START INLET 12)
(LABEL (:TAGGED-LITERAL LABEL FACT_INLET12))
(MOVE (:MESSAGE (:TAGGED-LITERAL INT 1)) (:J-REGISTER R2))
(MOVE (:J-REGISTER R2) (:J-REGISTER A1))
(MOVE (:MESSAGE (:TAGGED-LITERAL INT 2)) (:J-REGISTER R1))
(MOVE (:J-REGISTER R1) (:FRAME (:TAGGED-LITERAL INT (ISLOT1 . 5))))
(COMMENT "line 76")
(COMMENT "line 77")
(DC (:TAGGED-LITERAL LOCAL-IP (:LABEL FACT_THREAD2)))
(CALL "post_it_call")
(END 2 INLET 12)
(COMMENT "line 80")
(START INLET 14)
(LABEL (:TAGGED-LITERAL LABEL FACT_INLET14))
(MOVE (:MESSAGE (:TAGGED-LITERAL INT 1)) (:J-REGISTER R2))
(MOVE (:J-REGISTER R2) (:J-REGISTER A1))
(MOVE (:MESSAGE (:TAGGED-LITERAL INT 2)) (:J-REGISTER R1))
(MOVE (:J-REGISTER R1) (:FRAME (:TAGGED-LITERAL INT (PFSLOT1_0 . 12))))
(MOVE (:MESSAGE (:TAGGED-LITERAL INT 3)) (:J-REGISTER R2))
(MOVE (:J-REGISTER R2) (:FRAME (:TAGGED-LITERAL INT (PFSLOT1_1 . 13))))
(COMMENT "line 82")
(COMMENT "line 83")
(COMMENT "line 84")
(MOVE (:TAGGED-LITERAL INT 18) (:J-REGISTER R1))
(DC (:TAGGED-LITERAL LOCAL-IP (:LABEL FACT_THREAD8)))
(CALL "maybe_post_it_call")
(END 3 INLET 14)
(COMMENT "line 86")
(START INLET 15)
(LABEL (:TAGGED-LITERAL LABEL FACT_INLET15))
(MOVE (:MESSAGE (:TAGGED-LITERAL INT 1)) (:J-REGISTER R1))
(MOVE (:J-REGISTER R1) (:J-REGISTER A1))
(COMMENT "line 88")
(DC (:TAGGED-LITERAL LOCAL-IP (:LABEL FACT_THREAD10)))
(CALL "post_it_call")
(END 1 INLET 15)
(COMMENT "line 91")
(START INLET 16)
(LABEL (:TAGGED-LITERAL LABEL FACT_INLET16))
(MOVE (:MESSAGE (:TAGGED-LITERAL INT 1)) (:J-REGISTER R2))
(MOVE (:J-REGISTER R2) (:J-REGISTER A1))
(MOVE (:MESSAGE (:TAGGED-LITERAL INT 2)) (:J-REGISTER R1))
(MOVE (:J-REGISTER R1) (:FRAME (:TAGGED-LITERAL INT (ISLOT4 . 8))))
(COMMENT "line 93")
(COMMENT "line 94")
(COMMENT "line 95")
(MOVE (:TAGGED-LITERAL INT 17) (:J-REGISTER R1))
(DC (:TAGGED-LITERAL LOCAL-IP (:LABEL FACT_THREAD9)))
(CALL "maybe_post_it_call")
(END 2 INLET 16)
(COMMENT "line 100")
(START THREAD 0)
(LABEL (:TAGGED-LITERAL LABEL FACT_THREAD0))
(MOVE (:TAGGED-LITERAL INT 0) (:J-REGISTER R0))
(MOVE (:J-REGISTER R0) (:J-REGISTER I))
(NOP)
(MOVE (:TAGGED-LITERAL INT 1) (:J-REGISTER R0))
(MOVE (:J-REGISTER R0) (:J-REGISTER I))
(COMMENT "line 104")
(SEND0 (:FRAME (:TAGGED-LITERAL INT (PFSLOT0_0 . 10))))
(MOVE (:FRAME (:TAGGED-LITERAL INT (JSLOT0 . 9))) (:J-REGISTER R2))
(ADD (:J-REGISTER R2) (:TAGGED-LITERAL INT 0) (:J-REGISTER R1))
(SEND0 (:TEMPORARY (:J-REGISTER R1)))
(SENDE0 (:FRAME (:TAGGED-LITERAL INT (PFSLOT0_1 . 11))))
(COMMENT "line 105")
(FFREE)
(END 1 THREAD 0)
(COMMENT "line 108")
(START THREAD 1)
(LABEL (:TAGGED-LITERAL LABEL FACT_THREAD1))
(MOVE (:TAGGED-LITERAL INT 0) (:J-REGISTER R0))
(MOVE (:J-REGISTER R0) (:J-REGISTER I))
(NOP)
(MOVE (:TAGGED-LITERAL INT 1) (:J-REGISTER R0))
(MOVE (:J-REGISTER R0) (:J-REGISTER I))
(SUSPEND-STRING)
(END 0 THREAD 1)
(COMMENT "line 111")
(START THREAD 2)
(LABEL (:TAGGED-LITERAL LABEL FACT_THREAD2))
(MOVE (:TAGGED-LITERAL INT 0) (:J-REGISTER R0))
(MOVE (:J-REGISTER R0) (:J-REGISTER I))
(NOP)
(MOVE (:TAGGED-LITERAL INT 1) (:J-REGISTER R0))
(MOVE (:J-REGISTER R0) (:J-REGISTER I))
(COMMENT "line 113")
(MOVE (:FRAME (:TAGGED-LITERAL INT (ISLOT1 . 5))) (:J-REGISTER R2))
(LE (:J-REGISTER R2) (:TAGGED-LITERAL INT 1) (:J-REGISTER R1))
(COMMENT "line 114")
(TLO START NSWITCH)
(BF (:J-REGISTER R1) (:TAGGED-LITERAL REL (:LABEL L0)))
(ADD (:J-REGISTER R3) (:TAGGED-LITERAL INT 1) (:J-REGISTER R3))
(DC (:TAGGED-LITERAL LOCAL-IP (:LABEL FACT_THREAD5)))
(MOVE (:J-REGISTER R0) (:LCV (:J-REGISTER R3)))
(BR (:TAGGED-LITERAL REL (:LABEL L1)))
(LABEL (:TAGGED-LITERAL LABEL L0))
(ADD (:J-REGISTER R3) (:TAGGED-LITERAL INT 1) (:J-REGISTER R3))
```

```
(DC (:TAGGED-LITERAL LOCAL-IP (:LABEL FACT_THREAD7)))
(MOVE (:J-REGISTER R0) (:LCV (:J-REGISTER R3)))
(LABEL (:TAGGED-LITERAL LABEL L1))
(TLO END NSWITCH)
(COMMENT "line 115")
(TLO START SSWITCH)
(BT (:J-REGISTER R1) (:TAGGED-LITERAL REL (:LABEL FACT_THREAD3)))
(COMMENT "   fall through")
(TLO END SSWITCH)
(END 1.0 THREAD 2)
(COMMENT "line 120")
(START THREAD 4)
(LABEL (:TAGGED-LITERAL LABEL FACT_THREAD4))
(MOVE (:TAGGED-LITERAL INT 0) (:J-REGISTER R0))
(MOVE (:J-REGISTER R0) (:J-REGISTER I))
(NOP)
(MOVE (:TAGGED-LITERAL INT 1) (:J-REGISTER R0))
(MOVE (:J-REGISTER R0) (:J-REGISTER I))
(COMMENT "line 121")
(MP-FALLOC (:J-REGISTER R2))
(DC (:TAGGED-LITERAL SPECIAL_TAG (:REF SYS_ALLOC_MSG_REF)))
(SEND1 (:J-REGISTER R0))
(DC (:TAGGED-LITERAL SPECIAL_TAG (:REF FACT_1)))
(SEND1 (:J-REGISTER R0))
(MOVE (:J-REGISTER NNR) (:J-REGISTER R2))
(SEND1 (:J-REGISTER R2))
(SEND1 (:J-REGISTER A1))
(DC (:TAGGED-LITERAL SPECIAL_TAG (:REF FACT_INLET_14_POINTER_REF)))
(SENDE1 (:J-REGISTER R0))
(SUSPEND-STRING)
(END 4 THREAD 4)
(COMMENT "line 117")
(START THREAD 3)
(LABEL (:TAGGED-LITERAL LABEL FACT_THREAD3))
(MOVE (:TAGGED-LITERAL INT 0) (:J-REGISTER R0))
(MOVE (:J-REGISTER R0) (:J-REGISTER I))
(NOP)
(MOVE (:TAGGED-LITERAL INT 1) (:J-REGISTER R0))
(MOVE (:J-REGISTER R0) (:J-REGISTER I))
(COMMENT "line 118")
(TLO START SFALL)
(MOVE (:FRAME (:TAGGED-LITERAL INT (SSLOT0 . 16))) (:J-REGISTER R2))
(SUB (:J-REGISTER R2) (:TAGGED-LITERAL INT 1) (:J-REGISTER R1))
(BZ (:J-REGISTER R1) (:TAGGED-LITERAL REL (:LABEL FACT_THREAD0)))
(MOVE (:J-REGISTER R1) (:FRAME (:TAGGED-LITERAL INT (SSLOT0 . 16))))
(SUSPEND-STRING)
(TLO END SFALL)
(END 0 THREAD 3)
(COMMENT "line 123")
(START THREAD 5)
(LABEL (:TAGGED-LITERAL LABEL FACT_THREAD5))
(MOVE (:TAGGED-LITERAL INT 0) (:J-REGISTER R0))
(MOVE (:J-REGISTER R0) (:J-REGISTER I))
(NOP)
(MOVE (:TAGGED-LITERAL INT 1) (:J-REGISTER R0))
(MOVE (:J-REGISTER R0) (:J-REGISTER I))
(COMMENT "line 124")
(MOVE (:FRAME (:TAGGED-LITERAL INT (ISLOT1 . 5))) (:J-REGISTER R2))
(MOVE (:J-REGISTER R2) (:FRAME (:TAGGED-LITERAL INT (ISLOT0 . 4))))
(COMMENT "line 125")
(TLO START NFALL)
(COMMENT "   fall through")
(TLO END NFALL)
(END 1 THREAD 5)
(COMMENT "line 127")
(START THREAD 6)
(LABEL (:TAGGED-LITERAL LABEL FACT_THREAD6))
(MOVE (:TAGGED-LITERAL INT 0) (:J-REGISTER R0))
(MOVE (:J-REGISTER R0) (:J-REGISTER I))
(NOP)
(MOVE (:TAGGED-LITERAL INT 1) (:J-REGISTER R0))
(MOVE (:J-REGISTER R0) (:J-REGISTER I))
(COMMENT "line 128")
(SEND0 (:FRAME (:TAGGED-LITERAL INT (PFSLOT0_0 . 10))))
(MOVE (:FRAME (:TAGGED-LITERAL INT (JSLOT0 . 9))) (:J-REGISTER R1))
(ADD (:J-REGISTER R1) (:TAGGED-LITERAL INT 1) (:J-REGISTER R2))
(SEND0 (:TEMPORARY (:J-REGISTER R2)))
(SEND0 (:FRAME (:TAGGED-LITERAL INT (PFSLOT0_1 . 11))))
(SEND0 (:FRAME (:TAGGED-LITERAL INT (ISLOT0 . 4))))
(SENDE0 (:TAGGED-LITERAL SYM 0))
(COMMENT "line 129")
(TLO START SFALL)
(MOVE (:FRAME (:TAGGED-LITERAL INT (SSLOT0 . 16))) (:J-REGISTER R1))
(SUB (:J-REGISTER R1) (:TAGGED-LITERAL INT 1) (:J-REGISTER R2))
(BZ (:J-REGISTER R2) (:TAGGED-LITERAL REL (:LABEL FACT_THREAD0)))
(MOVE (:J-REGISTER R2) (:FRAME (:TAGGED-LITERAL INT (SSLOT0 . 16))))
(SUSPEND-STRING)
(TLO END SFALL)
(END 1 THREAD 6)
(COMMENT "line 131")
(START THREAD 7)
(LABEL (:TAGGED-LITERAL LABEL FACT_THREAD7))
(MOVE (:TAGGED-LITERAL INT 0) (:J-REGISTER R0))
(MOVE (:J-REGISTER R0) (:J-REGISTER I))
(NOP)
(MOVE (:TAGGED-LITERAL INT 1) (:J-REGISTER R0))
(MOVE (:J-REGISTER R0) (:J-REGISTER I))
(COMMENT "line 132")
```

```
(MOVE (:FRAME (:TAGGED-LITERAL INT (ISLOT1 . 5))) (:J-REGISTER R1))
(SUB (:J-REGISTER R1) (:TAGGED-LITERAL INT 1) (:J-REGISTER R2))
(MOVE (:J-REGISTER R2) (:FRAME (:TAGGED-LITERAL INT (ISLOT2 . 6))))
(COMMENT "line 133")
(TLO START SFORK)
(MOVE (:FRAME (:TAGGED-LITERAL INT (SSLOT2 . 18))) (:J-REGISTER R1))
(SUB (:J-REGISTER R1) (:TAGGED-LITERAL INT 1) (:J-REGISTER R2))
(BNZ (:J-REGISTER R2) (:TAGGED-LITERAL REL (:LABEL LABEL4)))
(ADD (:J-REGISTER R3) (:TAGGED-LITERAL INT 1) (:J-REGISTER R3))
(DC (:TAGGED-LITERAL LOCAL-IP (:LABEL FACT_THREAD8)))
(MOVE (:J-REGISTER R0) (:LCV (:J-REGISTER R3)))
(BR (:TAGGED-LITERAL REL (:LABEL LABEL5)))
(LABEL (:TAGGED-LITERAL LABEL LABEL4))
(MOVE (:J-REGISTER R2) (:FRAME (:TAGGED-LITERAL INT (SSLOT2 . 18))))
(LABEL (:TAGGED-LITERAL LABEL LABEL5))
(TLO END SFORK)
(COMMENT "line 134")
(TLO START SFALL)
(MOVE (:FRAME (:TAGGED-LITERAL INT (SSLOT1 . 17))) (:J-REGISTER R1))
(SUB (:J-REGISTER R1) (:TAGGED-LITERAL INT 1) (:J-REGISTER R2))
(BZ (:J-REGISTER R2) (:TAGGED-LITERAL REL (:LABEL FACT_THREAD9)))
(MOVE (:J-REGISTER R2) (:FRAME (:TAGGED-LITERAL INT (SSLOT1 . 17))))
(SUSPEND-STRING)
(TLO END SFALL)
(END 2.0 THREAD 7)
(COMMENT "line 142")
(START THREAD 9)
(LABEL (:TAGGED-LITERAL LABEL FACT_THREAD9))
(MOVE (:TAGGED-LITERAL INT 0) (:J-REGISTER R0))
(MOVE (:J-REGISTER R0) (:J-REGISTER I))
(NOP)
(MOVE (:TAGGED-LITERAL INT 1) (:J-REGISTER R0))
(MOVE (:J-REGISTER R0) (:J-REGISTER I))
(COMMENT "line 146")
(MOVE (:FRAME (:TAGGED-LITERAL INT (ISLOT1 . 5))) (:J-REGISTER R1))
(MUL (:J-REGISTER R1) (:FRAME (:TAGGED-LITERAL INT (ISLOT4 . 8))) (:J-REGISTER R2))
(MOVE (:J-REGISTER R2) (:FRAME (:TAGGED-LITERAL INT (ISLOT3 . 7))))
(COMMENT "line 147")
(MOVE (:J-REGISTER R2) (:FRAME (:TAGGED-LITERAL INT (ISLOT0 . 4))))
(COMMENT "line 148")
(TLO START NFALL)
(BR (:TAGGED-LITERAL REL (:LABEL FACT_THREAD6)))
(TLO END NFALL)
(END 2 THREAD 9)
(COMMENT "line 136")
(START THREAD 8)
(LABEL (:TAGGED-LITERAL LABEL FACT_THREAD8))
(MOVE (:TAGGED-LITERAL INT 0) (:J-REGISTER R0))
(MOVE (:J-REGISTER R0) (:J-REGISTER I))
(NOP)
(MOVE (:TAGGED-LITERAL INT 1) (:J-REGISTER R0))
(MOVE (:J-REGISTER R0) (:J-REGISTER I))
(COMMENT "line 140")
(SEND0 (:FRAME (:TAGGED-LITERAL INT (PFSLOT1_0 . 12))))
(DC (:TAGGED-LITERAL SPECIAL_TAG (:REF FACT_INLET_12_REF)))
(SEND0 (:J-REGISTER R0))
(SEND0 (:FRAME (:TAGGED-LITERAL INT (PFSLOT1_1 . 13))))
(SEND0 (:FRAME (:TAGGED-LITERAL INT (ISLOT2 . 6))))
(SENDE0 (:TAGGED-LITERAL SYM 0))
(SUSPEND-STRING)
(END 1 THREAD 8)
(COMMENT "line 150")
(START THREAD 10)
(LABEL (:TAGGED-LITERAL LABEL FACT_THREAD10))
(MOVE (:TAGGED-LITERAL INT 0) (:J-REGISTER R0))
(MOVE (:J-REGISTER R0) (:J-REGISTER I))
(NOP)
(MOVE (:TAGGED-LITERAL INT 1) (:J-REGISTER R0))
(MOVE (:J-REGISTER R0) (:J-REGISTER I))
(COMMENT "line 152")
(TLO START SFALL)
(MOVE (:FRAME (:TAGGED-LITERAL INT (SSLOT0 . 16))) (:J-REGISTER R1))
(SUB (:J-REGISTER R1) (:TAGGED-LITERAL INT 1) (:J-REGISTER R2))
(BZ (:J-REGISTER R2) (:TAGGED-LITERAL REL (:LABEL FACT_THREAD0)))
(MOVE (:J-REGISTER R2) (:FRAME (:TAGGED-LITERAL INT (SSLOT0 . 16))))
(SUSPEND-STRING)
(TLO END SFALL)
(END 0 THREAD 10)
(COMMENT "line 154")
(START THREAD 11)
(LABEL (:TAGGED-LITERAL LABEL FACT_THREAD11))
(MOVE (:TAGGED-LITERAL INT 0) (:J-REGISTER R0))
(MOVE (:J-REGISTER R0) (:J-REGISTER I))
(NOP)
(MOVE (:TAGGED-LITERAL INT 1) (:J-REGISTER R0))
(MOVE (:J-REGISTER R0) (:J-REGISTER I))
(SUSPEND-STRING)
(END 0 THREAD 11)
(COMMENT "line 156")
(START THREAD 12)
(LABEL (:TAGGED-LITERAL LABEL FACT_THREAD12))
(MOVE (:TAGGED-LITERAL INT 0) (:J-REGISTER R0))
(MOVE (:J-REGISTER R0) (:J-REGISTER I))
(NOP)
(MOVE (:TAGGED-LITERAL INT 1) (:J-REGISTER R0))
(MOVE (:J-REGISTER R0) (:J-REGISTER I))
(COMMENT "line 157")
(SUSPEND-STRING)
```

```
(END 0 THREAD 12)
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_list") :NO-BRANCH)
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_0_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL MSG "(FACT_inlet0+FACT_place<<ip_offset_pos)|msg_u|5"))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_1_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL SYM 0))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_2_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL SYM 0))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_3_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL SYM 0))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_4_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL SYM 0))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_5_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL SYM 0))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_6_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL SYM 0))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_7_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL SYM 0))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_8_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL SYM 0))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_9_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL SYM 0))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_10_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL SYM 0))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_11_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL MSG "(FACT_inlet11+FACT_place<<ip_offset_pos)|msg_u|4"))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_12_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL MSG "(FACT_inlet12+FACT_place<<ip_offset_pos)|msg_u|4"))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_13_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL SYM 0))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_14_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL MSG "(FACT_inlet14+FACT_place<<ip_offset_pos)|msg_u|4"))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_15_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL MSG "(FACT_inlet15+FACT_place<<ip_offset_pos)|msg_u|2"))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_16_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL MSG "(FACT_inlet16+FACT_place<<ip_offset_pos)|msg_u|4"))
(END-CODEBLOCK FACT (0 5) (11 4) (12 4) (14 4) (15 2) (16 4)))
```

A.3.3 MDP Assembly

```
set GSLOT0_1 = 21
set GSLOT0_0 = 20
set SSLOT2 = 18
set SSLOT1 = 17
set SSLOT0 = 16
set JSLOT0 = 9
set PSSLOT0_1 = 15
set PSSLOT0_0 = 14
set PFSLOT1_1 = 13
set PFSLOT1_0 = 12
set PFSLOT0_1 = 11
set PFSLOT0_0 = 10
set ISLOT4 = 8
set ISLOT3 = 7
set ISLOT2 = 6
set ISLOT1 = 5
set ISLOT0 = 4

module fact ; size = 167
entry fact_codeblock_start
entry fact_inlet_list
entry GC_FACT

GC_FACT:
dc {GC_0_FACT_0}
dc {GC_0_FACT_1}
entry GC_0_FACT

GC_0_FACT:
dc {FACT_0}
dc {FACT_1}
dc 1
dc nil
dc cfut:-1
dc cfut:-1
dc nil
dc nil

fact_codeblock_start:

;file FACT.t10
;line 22
;FRAME SLOTS
;Frame header
;Frame body
;Frame size = 33
 DC 33
 DC 6
 DC nil
 DC ip:((FACT_INLET0+fact_place)<<ip_offset_pos)|ip_u|ip_a0_absolute
;line 55
FACT_INLET0:
 MOVE [1,A3], R2
 MOVE R2, A1
```

```
 MOVE [2,A3], R1
 MOVE R1, [PFSLOT0_0,A1]  ; PFSLOT0_0 = 10
 MOVE [3,A3], R2
 MOVE R2, [PFSLOT0_1,A1]  ; PFSLOT0_1 = 11
 MOVE [4,A3], R1
 MOVE R1, [JSLOT0,A1]  ; JSLOT0 = 9
;line 57
;line 58
;line 59
;line 60
 MOVE 2, R2
 MOVE R2, [SSLOT2,A1]  ; SSLOT2 = 18
;line 61
 MOVE R2, [SSLOT1,A1]  ; SSLOT1 = 17
;line 62
 MOVE 3, R1
 MOVE R1, [SSLOT0,A1]  ; SSLOT0 = 16
;line 63
;line 64
;line 65
 MOVE [JSLOT0,A1], R2  ; JSLOT0 = 9
 ADD R2, -1, R1
 SEND0 [PFSLOT0_0,A1]  ; PFSLOT0_0 = 10
 SEND0 [R1,A0]
 MOVE NNR, R2
 SEND20 [PFSLOT0_1,A1], R2  ; PFSLOT0_1 = 11
 SENDE0 A1
;line 66
 MOVE 16, R1
 DC ip:((FACT_THREAD0+fact_place)<<ip_offset_pos)|ip_u|ip_a0_absolute
 CALL maybe_post_it_call
;line 68
FACT_INLET11:
 MOVE [1,A3], R1
 MOVE R1, A1
 MOVE [2,A3], R2
 MOVE R2, [PSSLOT0_0,A1]  ; PSSLOT0_0 = 14
 MOVE [3,A3], R1
 MOVE R1, [PSSLOT0_1,A1]  ; PSSLOT0_1 = 15
;line 70
;line 71
 DC ip:((FACT_THREAD1+fact_place)<<ip_offset_pos)|ip_u|ip_a0_absolute
 CALL post_it_call
;line 74
FACT_INLET12:
 MOVE [1,A3], R2
 MOVE R2, A1
 MOVE [2,A3], R1
 MOVE R1, [ISLOT1,A1]  ; ISLOT1 = 5
;line 76
;line 77
 DC ip:((FACT_THREAD2+fact_place)<<ip_offset_pos)|ip_u|ip_a0_absolute
 CALL post_it_call
;line 80
FACT_INLET14:
 MOVE [1,A3], R2
 MOVE R2, A1
 MOVE [2,A3], R1
 MOVE R1, [PFSLOT1_0,A1]  ; PFSLOT1_0 = 12
 MOVE [3,A3], R2
 MOVE R2, [PFSLOT1_1,A1]  ; PFSLOT1_1 = 13
;line 82
;line 83
;line 84
 MOVE 18, R1
 DC ip:((FACT_THREAD8+fact_place)<<ip_offset_pos)|ip_u|ip_a0_absolute
 CALL maybe_post_it_call
;line 86
FACT_INLET15:
 MOVE [1,A3], R1
 MOVE R1, A1
;line 88
 DC ip:((FACT_THREAD10+fact_place)<<ip_offset_pos)|ip_u|ip_a0_absolute
 CALL post_it_call
;line 91
FACT_INLET16:
 MOVE [1,A3], R2
 MOVE R2, A1
 MOVE [2,A3], R1
 MOVE R1, [ISLOT4,A1]  ; ISLOT4 = 8
;line 93
;line 94
;line 95
 MOVE 17, R1
 DC ip:((FACT_THREAD9+fact_place)<<ip_offset_pos)|ip_u|ip_a0_absolute
 CALL maybe_post_it_call
;line 100
FACT_THREAD0:
 MOVE 0, R0
 MOVE R0, I
 NOP
 MOVE 1, R0
 MOVE R0, I
;line 104
 MOVE [JSLOT0,A1], R2  ; JSLOT0 = 9
 ADD R2, 0, R1
 SEND0 [PFSLOT0_0,A1]  ; PFSLOT0_0 = 10
 SEND0 [R1,A0]
```

```
 SENDEO [PFSLOTO_1,A1]  ; PFSLOTO_1 = 11
;line 105
dc {fact_1}
move R0, R1
dc ip:(ffree+library_place<<ip_offset_pos)|ip_a0_absolute
move R0, ip
;line 108
FACT_THREAD1:
 MOVE 0, R0
 MOVE R0, I
 NOP
 MOVE 1, R0
 MOVE R0, I
MOVE [R3,A0], R0
SUB R3, 1, R3
MOVE R0, IP

;line 111
FACT_THREAD2:
 MOVE 0, R0
 MOVE R0, I
 NOP
 MOVE 1, R0
 MOVE R0, I
;line 113
 MOVE [ISLOT1,A1], R2  ; ISLOT1 = 5
 LE R2, 1, R1
;line 114
 BF R1, ^L0
 ADD R3, 1, R3
 DC ip:((FACT_THREAD5+fact_place)<<ip_offset_pos)|ip_u|ip_a0_absolute
 MOVE R0, [R3,A0]
 BR ^L1
L0:
 ADD R3, 1, R3
 DC ip:((FACT_THREAD7+fact_place)<<ip_offset_pos)|ip_u|ip_a0_absolute
 MOVE R0, [R3,A0]
L1:
;line 115
 BT R1, ^FACT_THREAD3
;   fall through
;line 120
FACT_THREAD4:
 MOVE 0, R0
 MOVE R0, I
 NOP
 MOVE 1, R0
 MOVE R0, I
;line 121
 MOVE [_ALLOC_LOC,A0], R2
 ADD R2, 1, R2
 AND R2, [_NPROC_MINUS_1_LOC,A0], R2
 MOVE R2, [_ALLOC_LOC,A0]
 MOVE [R2,A2], R2
 DC {SYS_ALLOC_MSG_REF}
 SEND20 R2, R0
 DC {FACT_1}
 MOVE NNR, R2
 SEND20 R0, R2
 DC {FACT_INLET_14_POINTER_REF}
 SEND2E0 A1, R0
MOVE [R3,A0], R0
SUB R3, 1, R3
MOVE R0, IP

;line 117
FACT_THREAD3:
 MOVE 0, R0
 MOVE R0, I
 NOP
 MOVE 1, R0
 MOVE R0, I
;line 118
 MOVE [SSLOT0,A1], R2  ; SSLOT0 = 16
 SUB R2, 1, R1
 BZ R1, ^FACT_THREAD0
 MOVE R1, [SSLOT0,A1]  ; SSLOT0 = 16
MOVE [R3,A0], R0
SUB R3, 1, R3
MOVE R0, IP

;line 123
FACT_THREAD5:
 MOVE 0, R0
 MOVE R0, I
 NOP
 MOVE 1, R0
 MOVE R0, I
;line 124
 MOVE [ISLOT1,A1], R2  ; ISLOT1 = 5
 MOVE R2, [ISLOT0,A1]  ; ISLOT0 = 4
;line 125
;   fall through
;line 127
FACT_THREAD6:
 MOVE 0, R0
 MOVE R0, I
 NOP
```

```
 MOVE 1, R0
 MOVE R0, I
;line 128
 MOVE [JSLOT0,A1], R1  ; JSLOT0 = 9
 ADD R1, 1, R2
 SEND0 [PFSLOT0_0,A1]  ; PFSLOT0_0 = 10
 SEND0 [R2,A0]
 SEND0 [PFSLOT0_1,A1]  ; PFSLOT0_1 = 11
 SEND0 [ISLOT0,A1]
 SENDE0 nil  ; ISLOT0 = 4
;line 129
 MOVE [SSLOT0,A1], R1  ; SSLOT0 = 16
 SUB R1, 1, R2
 BZ R2, ^FACT_THREAD0
 MOVE R2, [SSLOT0,A1]  ; SSLOT0 = 16
MOVE [R3,A0], R0
SUB R3, 1, R3
MOVE R0, IP

;line 131
FACT_THREAD7:
 MOVE 0, R0
 MOVE R0, I
 NOP
 MOVE 1, R0
 MOVE R0, I
;line 132
 MOVE [ISLOT1,A1], R1  ; ISLOT1 = 5
 SUB R1, 1, R2
 MOVE R2, [ISLOT2,A1]  ; ISLOT2 = 6
;line 133
 MOVE [SSLOT2,A1], R1  ; SSLOT2 = 18
 SUB R1, 1, R2
 BNZ R2, ^LABEL4
 ADD R3, 1, R3
 DC ip:((FACT_THREAD8+fact_place)<<ip_offset_pos)|ip_u|ip_a0_absolute
 MOVE R0, [R3,A0]
 BR ^LABEL5
LABEL4:
 MOVE R2, [SSLOT2,A1]  ; SSLOT2 = 18
LABEL5:
;line 134
 MOVE [SSLOT1,A1], R1  ; SSLOT1 = 17
 SUB R1, 1, R2
 BZ R2, ^FACT_THREAD9
 MOVE R2, [SSLOT1,A1]  ; SSLOT1 = 17
MOVE [R3,A0], R0
SUB R3, 1, R3
MOVE R0, IP

;line 142
FACT_THREAD9:
 MOVE 0, R0
 MOVE R0, I
 NOP
 MOVE 1, R0
 MOVE R0, I
;line 146
 MOVE [ISLOT1,A1], R1  ; ISLOT1 = 5
 MUL R1, [ISLOT4,A1], R2  ; ISLOT4 = 8
 MOVE R2, [ISLOT3,A1]  ; ISLOT3 = 7
;line 147
 MOVE R2, [ISLOT0,A1]  ; ISLOT0 = 4
;line 148
 BR ^FACT_THREAD6
;line 136
FACT_THREAD8:
 MOVE 0, R0
 MOVE R0, I
 NOP
 MOVE 1, R0
 MOVE R0, I
;line 140
 DC {FACT_INLET_12_REF}
 SEND20 [PFSLOT1_0,A1], R0  ; PFSLOT1_0 = 12
 SEND0 [PFSLOT1_1,A1]  ; PFSLOT1_1 = 13
 SEND0 [ISLOT2,A1]
 SENDE0 nil  ; ISLOT2 = 6
MOVE [R3,A0], R0
SUB R3, 1, R3
MOVE R0, IP

;line 150
FACT_THREAD10:
 MOVE 0, R0
 MOVE R0, I
 NOP
 MOVE 1, R0
 MOVE R0, I
;line 152
 MOVE [SSLOT0,A1], R1  ; SSLOT0 = 16
 SUB R1, 1, R2
 BNZ R2, ^jog5071
 DC ip:((FACT_THREAD0+fact_place)<<ip_offset_pos)|ip_u|ip_a0_absolute
 MOVE R0, IP
jog5071:
 MOVE R2, [SSLOT0,A1]  ; SSLOT0 = 16
MOVE [R3,A0], R0
```

```
SUB R3, 1, R3
MOVE R0, IP

;line 154
FACT_THREAD11:
 MOVE 0, R0
 MOVE R0, I
 NOP
 MOVE 1, R0
 MOVE R0, I
MOVE [R3,A0], R0
SUB R3, 1, R3
MOVE R0, IP

;line 156
FACT_THREAD12:
 MOVE 0, R0
 MOVE R0, I
 NOP
 MOVE 1, R0
 MOVE R0, I
;line 157
MOVE [R3,A0], R0
SUB R3, 1, R3
MOVE R0, IP

FACT_inlet_list:
FACT_inlet_0_pointer:
 DC msg:(FACT_inlet0+FACT_place<<ip_offset_pos)|msg_u|5
FACT_inlet_1_pointer:
 DC nil
FACT_inlet_2_pointer:
 DC nil
FACT_inlet_3_pointer:
 DC nil
FACT_inlet_4_pointer:
 DC nil
FACT_inlet_5_pointer:
 DC nil
FACT_inlet_6_pointer:
 DC nil
FACT_inlet_7_pointer:
 DC nil
FACT_inlet_8_pointer:
 DC nil
FACT_inlet_9_pointer:
 DC nil
FACT_inlet_10_pointer:
 DC nil
FACT_inlet_11_pointer:
 DC msg:(FACT_inlet11+FACT_place<<ip_offset_pos)|msg_u|4
FACT_inlet_12_pointer:
 DC msg:(FACT_inlet12+FACT_place<<ip_offset_pos)|msg_u|4
FACT_inlet_13_pointer:
 DC nil
FACT_inlet_14_pointer:
 DC msg:(FACT_inlet14+FACT_place<<ip_offset_pos)|msg_u|4
FACT_inlet_15_pointer:
 DC msg:(FACT_inlet15+FACT_place<<ip_offset_pos)|msg_u|2
FACT_inlet_16_pointer:
 DC msg:(FACT_inlet16+FACT_place<<ip_offset_pos)|msg_u|4
entry FACT_inlet0
entry FACT_inlet_0_pointer
entry FACT_inlet1
entry FACT_inlet_1_pointer
entry FACT_inlet2
entry FACT_inlet_2_pointer
entry FACT_inlet3
entry FACT_inlet_3_pointer
entry FACT_inlet4
entry FACT_inlet_4_pointer
entry FACT_inlet5
entry FACT_inlet_5_pointer
entry FACT_inlet6
entry FACT_inlet_6_pointer
entry FACT_inlet7
entry FACT_inlet_7_pointer
entry FACT_inlet8
entry FACT_inlet_8_pointer
entry FACT_inlet9
entry FACT_inlet_9_pointer
entry FACT_inlet10
entry FACT_inlet_10_pointer
entry FACT_inlet11
entry FACT_inlet_11_pointer
entry FACT_inlet12
entry FACT_inlet_12_pointer
entry FACT_inlet13
entry FACT_inlet_13_pointer
entry FACT_inlet14
entry FACT_inlet_14_pointer
entry FACT_inlet15
entry FACT_inlet_15_pointer
entry FACT_inlet16
entry FACT_inlet_16_pointer

end
```

```
ref FACT_0 = l_nnr
ref FACT_1 = addr:(FACT_place+FACT_codeblock_start<<ip_offset_pos)
ref GC_FACT_0 = l_nnr
ref GC_FACT_1 = addr:((FACT_place+GC_FACT)<<ip_offset_pos)|2
ref GC_0_FACT_0 = l_nnr
ref GC_0_FACT_1 = addr:((FACT_place+GC_0_FACT)<<ip_offset_pos)|8
ref FACT_inlet_0_pointer_ref = FACT_inlet_0_pointer+FACT_place
ref FACT_inlet_0_ref = msg:(FACT_inlet0+FACT_place<<ip_offset_pos)|msg_u|5
ref FACT_inlet_1_pointer_ref = FACT_inlet_1_pointer+FACT_place
ref FACT_inlet_2_pointer_ref = FACT_inlet_2_pointer+FACT_place
ref FACT_inlet_3_pointer_ref = FACT_inlet_3_pointer+FACT_place
ref FACT_inlet_4_pointer_ref = FACT_inlet_4_pointer+FACT_place
ref FACT_inlet_5_pointer_ref = FACT_inlet_5_pointer+FACT_place
ref FACT_inlet_6_pointer_ref = FACT_inlet_6_pointer+FACT_place
ref FACT_inlet_7_pointer_ref = FACT_inlet_7_pointer+FACT_place
ref FACT_inlet_8_pointer_ref = FACT_inlet_8_pointer+FACT_place
ref FACT_inlet_9_pointer_ref = FACT_inlet_9_pointer+FACT_place
ref FACT_inlet_10_pointer_ref = FACT_inlet_10_pointer+FACT_place
ref FACT_inlet_11_pointer_ref = FACT_inlet_11_pointer+FACT_place
ref FACT_inlet_11_ref = msg:(FACT_inlet11+FACT_place<<ip_offset_pos)|msg_u|4
ref FACT_inlet_12_pointer_ref = FACT_inlet_12_pointer+FACT_place
ref FACT_inlet_12_ref = msg:(FACT_inlet12+FACT_place<<ip_offset_pos)|msg_u|4
ref FACT_inlet_13_pointer_ref = FACT_inlet_13_pointer+FACT_place
ref FACT_inlet_14_pointer_ref = FACT_inlet_14_pointer+FACT_place
ref FACT_inlet_14_ref = msg:(FACT_inlet14+FACT_place<<ip_offset_pos)|msg_u|4
ref FACT_inlet_15_pointer_ref = FACT_inlet_15_pointer+FACT_place
ref FACT_inlet_15_ref = msg:(FACT_inlet15+FACT_place<<ip_offset_pos)|msg_u|2
ref FACT_inlet_16_pointer_ref = FACT_inlet_16_pointer+FACT_place
ref FACT_inlet_16_ref = msg:(FACT_inlet16+FACT_place<<ip_offset_pos)|msg_u|4
```

A.4 Flattened Implementation

A.4.1 Complex MDP

```
(global-struct (:label gc_fact) (:ref gc_0_fact_0) (:ref gc_0_fact_1))
(global-struct (:label gc_0_fact) (:ref fact_0) (:ref fact_1) 1
               (:tagged-literal sym 0) (:tagged-literal cfut -1)
               (:tagged-literal cfut -1) (:tagged-literal sym 0)
               (:tagged-literal sym 0))
(codeblock "fact")
(comment "file FACT.tl0")
(comment "line 22")
(comment "FRAME SLOTS")
(comment "Frame header")
(comment "Frame body")
(assembler-label islot0 1)
(assembler-label islot1 2)
(assembler-label islot2 3)
(assembler-label islot3 4)
(assembler-label islot4 5)
(assembler-label pfslot0_0 7)
(assembler-label pfslot0_1 8)
(assembler-label pfslot1_0 9)
(assembler-label pfslot1_1 10)
(assembler-label psslot0_0 11)
(assembler-label psslot0_1 12)
(assembler-label jslot0 6)
(assembler-label sslot0 13)
(assembler-label sslot1 14)
(assembler-label sslot2 15)
(assembler-label gslot0_0 17)
(assembler-label gslot0_1 18)
(comment "Frame size = 19")
(dc 19)
(dc 5)
(dc (:tagged-literal sym 0))
(dc (:tagged-literal local-ip (:label fact_inlet0)))
(comment "line 55")
(start inlet 0)
(label (:label fact_inlet0))
(move (:message 1) (:j-register A1))
(comment "line 57")
(comment "line 58")
(comment "line 59")
(comment "line 60")
(move 2 (:frame sslot2))
(comment "line 61")
(move 2 (:frame sslot1))
(comment "line 62")
(move 3 (:frame sslot0))
(comment "line 63")
(comment "line 64")
(comment "line 65")
(move 1 (:j-register i))
(send0 (:message 2))
(reserve (:register add-inlet))
(add (:message 4) -1 (:register add-inlet))
(send0 (:temporary (:register add-inlet)))
(free (:register add-inlet))
(send0 (:message 3))
(send0 (:j-register NNR))
```

```
(sende0 (:j-register A1))
(move 0 (:j-register i))
(comment "line 66")
(move (:message 2) (:frame pfslot0_0))
(move (:message 3) (:frame pfslot0_1))
(move (:message 4) (:frame jslot0))
(send0 (:j-register NNR))
(send0 (:ref fact_inlet_1_ref))
(sende0 (:j-register A1))
(suspend)
(end-inlet)
(end inlet 0)
(comment "special inlet")
(start inlet 1)
(label (:label fact_inlet1))
(move (:message 1) (:j-register A1))
(comment "line 66")
(reserve (:register genjump))
(sub (:frame sslot0) 1 (:register genjump))
(bz (:register genjump) (:tagged-literal rel (:label L0)))
(move (:register genjump) (:frame sslot0))
(free (:register genjump))
(suspend)
(label (:label L0))
(end-inlet)
(end inlet 1)
(comment "line 100")
(start thread 0)
(fall-through)
(comment "thread 0")
(comment "line 104")
(move 1 (:j-register i))
(send0 (:frame pfslot0_0))
(reserve (:register add-inlet))
(add (:frame jslot0) 0 (:register add-inlet))
(send0 (:temporary (:register add-inlet)))
(free (:register add-inlet))
(sende0 (:frame pfslot0_1))
(move 0 (:j-register i))
(comment "line 105")
(ffree)
(end thread 0)
(comment "line 68")
(start inlet 11)
(label (:label fact_inlet11))
(move (:message 1) (:j-register A1))
(comment "line 70")
(comment "line 71")
(comment "receive of psslot0 eliminated")
(comment "   fall through")
(end-inlet)
(end inlet 11)
(comment "line 108")
(start thread 1)
(fall-through)
(comment "thread 1")
(suspend)
(end thread 1)
(comment "line 74")
(start inlet 12)
(label (:label fact_inlet12))
(move (:message 1) (:j-register A1))
(comment "line 76")
(comment "line 77")
(comment "   fall through")
(end-inlet)
(end inlet 12)
(comment "line 111")
(start thread 2)
(fall-through)
(comment "thread 2")
(comment "line 113")
(move (:message 2) (:frame islot1))
(le (:frame islot1) 1 (:register temp0b))
(comment "line 114")
(tl0 start nswitch)
(bf (:register temp0b) (:tagged-literal rel (:label L1)))
(add (:j-register R3) 1 (:j-register R3))
(move (:tagged-literal local-ip (:label fact_thread5)) (:lcv (:j-register R3)))
(br (:tagged-literal rel (:label L2)))
(label (:label L1))
(add (:j-register R3) 1 (:j-register R3))
(move (:tagged-literal local-ip (:label fact_thread7)) (:lcv (:j-register R3)))
(label (:label L2))
(tl0 end nswitch)
(comment "line 115")
(tl0 start sswitch)
(bt (:register temp0b) (:tagged-literal rel (:label fact_thread3)))
(comment "   fall through")
(tl0 end sswitch)
(end thread 2)
(comment "line 120")
(start thread 4)
(label (:label fact_thread4))
(comment "line 121")
(move 1 (:j-register i))
(mp-falloc-setup)
(send1 (:ref sys_alloc_msg_ref))
```

```
(send1 (:ref fact_1))
(send1 (:j-register nnr))
(send1 (:j-register A1))
(sende1 (:ref fact_inlet_14_pointer_ref))
(move 0 (:j-register i))
(suspend-string)
(end thread 4)
(comment "line 80")
(start inlet 14)
(label (:label fact_inlet14))
(move (:message 1) (:j-register A1))
(comment "line 82")
(comment "line 83")
(comment only-use (:ref fact_inlet_14_pointer_ref) pfslot1 0.T)
(comment "line 84")
(reserve (:register genjump))
(sub (:frame sslot2) 1 (:register genjump))
(bz (:register genjump) (:tagged-literal rel (:label L5)))
(move (:register genjump) (:frame sslot2))
(free (:register genjump))
(move (:message 2) (:frame pfslot1_0))
(move (:message 3) (:frame pfslot1_1))
(suspend)
(label (:label L5))
(end-inlet)
(end inlet 14)
(comment "line 136")
(start thread 8)
(fall-through)
(comment "thread 8")
(comment "line 140")
(move 1 (:j-register i))
(send0 (:message 2))
(send0 (:ref fact_inlet_12_ref))
(send0 (:message 3))
(send0 (:frame islot2))
(sende0 (:tagged-literal sym 0))
(move 0 (:j-register i))
(suspend)
(end thread 8)
(comment "line 86")
(start inlet 15)
(label (:label fact_inlet15))
(move (:message 1) (:j-register A1))
(comment "line 88")
(comment "   fall through")
(end-inlet)
(end inlet 15)
(comment "line 150")
```

```
(start thread 10)
(fall-through)
(comment "thread 10")
(comment "line 152")
(tl0 start sfall)
(reserve (:register genjump))
(sub (:frame sslot0) 1 (:register genjump))
(bz (:register genjump) (:tagged-literal rel (:label fact_thread0)))
(move (:register genjump) (:frame sslot0))
(free (:register genjump))
(suspend)
(tl0 end sfall)
(end thread 10)
(comment "line 100")
(start thread 0)
(label (:label fact_thread0))
(comment "line 104")
(move 1 (:j-register i))
(send0 (:frame pfslot0_0))
(reserve (:register add-inlet))
(add (:frame jslot0) 0 (:register add-inlet))
(send0 (:temporary (:register add-inlet)))
(free (:register add-inlet))
(sende0 (:frame pfslot0_1))
(move 0 (:j-register i))
(comment "line 105")
(ffree)
(end thread 0)
(comment "line 91")
(start inlet 16)
(label (:label fact_inlet16))
(move (:message 1) (:j-register A1))
(comment "line 93")
(comment "line 94")
(comment only-use (:ref fact_inlet_16_pointer_ref) islot4 0.T)
(comment "line 95")
(reserve (:register genjump))
(sub (:frame sslot1) 1 (:register genjump))
(bz (:register genjump) (:tagged-literal rel (:label L6)))
(move (:register genjump) (:frame sslot1))
(free (:register genjump))
(move (:message 2) (:frame islot4))
(suspend)
(label (:label L6))
(end-inlet)
(end inlet 16)
(comment "line 142")
(start thread 9)
(fall-through)
```

```
(comment "thread 9")
(comment "line 146")
(mul (:frame islot1) (:message 2) (:frame islot3))
(comment "line 147")
(move (:frame islot3) (:frame islot0))
(comment "line 148")
(tl0 start nfall)
(comment "   fall through")
(tl0 end nfall)
(end thread 9)
(comment "line 127")
(start thread 6)
(label (:label fact_thread6))
(comment "line 128")
(move 1 (:j-register i))
(send0 (:frame pfslot0_0))
(reserve (:register add-inlet))
(add (:frame jslot0) 1 (:register add-inlet))
(send0 (:temporary (:register add-inlet)))
(free (:register add-inlet))
(send0 (:frame pfslot0_1))
(send0 (:frame islot0))
(sende0 (:tagged-literal sym 0))
(move 0 (:j-register i))
(comment "line 129")
(tl0 start sfall)
(reserve (:register genjump))
(sub (:frame sslot0) 1 (:register genjump))
(bz (:register genjump) (:tagged-literal rel (:label fact_thread0)))
(move (:register genjump) (:frame sslot0))
(free (:register genjump))
(suspend-string)
(tl0 end sfall)
(end thread 6)
(comment "line 117")
(start thread 3)
(label (:label fact_thread3))
(comment "line 118")
(tl0 start sfall)
(reserve (:register genjump))
(sub (:frame sslot0) 1 (:register genjump))
(bz (:register genjump) (:tagged-literal rel (:label fact_thread0)))
(move (:register genjump) (:frame sslot0))
(free (:register genjump))
(suspend-string)
(tl0 end sfall)
(end thread 3)
(comment "line 123")
(start thread 5)
(label (:label fact_thread5))
(comment "line 124")
(move (:frame islot1) (:frame islot0))
(comment "line 125")
(tl0 start nfall)
(br (:tagged-literal rel (:label fact_thread6)))
(tl0 end nfall)
(end thread 5)
(comment "line 131")
(start thread 7)
(label (:label fact_thread7))
(comment "line 132")
(sub (:frame islot1) 1 (:frame islot2))
(comment "line 133")
(tl0 start sfork)
(reserve (:register genjump))
(sub (:frame sslot2) 1 (:register genjump))
(bnz (:register genjump) (:tagged-literal rel (:label label7)))
(add (:j-register R3) 1 (:j-register R3))
(move (:tagged-literal local-ip (:label fact_thread8)) (:lcv (:j-register R3)))
(br (:tagged-literal rel (:label label8)))
(label (:label label7))
(move (:register genjump) (:frame sslot2))
(label (:label label8))
(free (:register genjump))
(tl0 end sfork)
(comment "line 134")
(tl0 start sfall)
(reserve (:register genjump))
(sub (:frame sslot1) 1 (:register genjump))
(bz (:register genjump) (:tagged-literal rel (:label fact_thread9)))
(move (:register genjump) (:frame sslot1))
(free (:register genjump))
(suspend-string)
(tl0 end sfall)
(end thread 7)
(comment "line 142")
(start thread 9)
(label (:label fact_thread9))
(comment "line 146")
(mul (:frame islot1) (:frame islot4) (:frame islot3))
(comment "line 147")
(move (:frame islot3) (:frame islot0))
(comment "line 148")
(tl0 start nfall)
(br (:tagged-literal rel (:label fact_thread6)))
(tl0 end nfall)
(end thread 9)
(comment "line 136")
```

```
(start thread 8)
(label (:label fact_thread8))
(comment "line 140")
(move 1 (:j-register i))
(send0 (:frame pfslot1_0))
(send0 (:ref fact_inlet_12_ref))
(send0 (:frame pfslot1_1))
(send0 (:frame islot2))
(sende0 (:tagged-literal sym 0))
(move 0 (:j-register i))
(suspend-string)
(end thread 8)
(comment "line 154")
(start thread 11)
(label (:label fact_thread11))
(suspend-string)
(end thread 11)
(comment "line 156")
(start thread 12)
(label (:label fact_thread12))
(comment "line 157")
(suspend-string)
(end thread 12)
(inlet-handler fact  (0 5)(1 2) (11 4) (12 4) (14 4) (15 2) (16 4))
(end-codeblock fact  (0 5)(1 2) (11 4) (12 4) (14 4) (15 2) (16 4))
```

A.4.2 Simple MDP

```
((GSLOT0_1 . 18)
 (GSLOT0_0 . 17)
 (SSLOT2 . 15)
 (SSLOT1 . 14)
 (SSLOT0 . 13)
 (JSLOT0 . 6)
 (PSSLOT0_1 . 12)
 (PSSLOT0_0 . 11)
 (PFSLOT1_1 . 10)
 (PFSLOT1_0 . 9)
 (PFSLOT0_1 . 8)
 (PFSLOT0_0 . 7)
 (ISLOT4 . 5)
 (ISLOT3 . 4)
 (ISLOT2 . 3)
 (ISLOT1 . 2)
 (ISLOT0 . 1))
((GLOBAL-STRUCT (:LABEL GC_FACT) (:TAGGED-LITERAL SPECIAL_TAG (:REF GC_0_FACT_0))
                (:TAGGED-LITERAL SPECIAL_TAG (:REF GC_0_FACT_1)))
 (GLOBAL-STRUCT (:LABEL GC_0_FACT) (:TAGGED-LITERAL SPECIAL_TAG (:REF FACT_0))
                (:TAGGED-LITERAL SPECIAL_TAG (:REF FACT_1)) (:TAGGED-LITERAL INT 1)
                (:TAGGED-LITERAL SYM 0) (:TAGGED-LITERAL CFUT -1)
                (:TAGGED-LITERAL CFUT -1) (:TAGGED-LITERAL SYM 0)
                (:TAGGED-LITERAL SYM 0))
 (CODEBLOCK "fact")
 (COMMENT "file FACT.tl0")
 (COMMENT "line 22")
 (COMMENT "FRAME SLOTS")
 (COMMENT "Frame header")
 (COMMENT "Frame body")
 (COMMENT "Frame size = 19")
 (DC (:TAGGED-LITERAL INT 19))
 (DC (:TAGGED-LITERAL INT 5))
 (DC (:TAGGED-LITERAL SYM 0))
 (DC (:TAGGED-LITERAL LOCAL-IP (:LABEL FACT_INLET0)))
 (COMMENT "line 55")
 (START INLET 0)
 (LABEL (:TAGGED-LITERAL LABEL FACT_INLET0))
 (MOVE (:MESSAGE (:TAGGED-LITERAL INT 1)) (:J-REGISTER R2))
 (MOVE (:J-REGISTER R2) (:J-REGISTER A1))
 (COMMENT "line 57")
 (COMMENT "line 58")
 (COMMENT "line 59")
 (COMMENT "line 60")
 (MOVE (:TAGGED-LITERAL INT 2) (:J-REGISTER R1))
 (MOVE (:J-REGISTER R1) (:FRAME (:TAGGED-LITERAL INT (SSLOT2 . 15))))
 (COMMENT "line 61")
 (MOVE (:J-REGISTER R1) (:FRAME (:TAGGED-LITERAL INT (SSLOT1 . 14))))
 (COMMENT "line 62")
 (MOVE (:TAGGED-LITERAL INT 3) (:J-REGISTER R2))
 (MOVE (:J-REGISTER R2) (:FRAME (:TAGGED-LITERAL INT (SSLOT0 . 13))))
 (COMMENT "line 63")
 (COMMENT "line 64")
 (COMMENT "line 65")
 (MOVE (:TAGGED-LITERAL INT 1) (:J-REGISTER R1))
 (MOVE (:J-REGISTER R1) (:J-REGISTER I))
 (SEND0 (:MESSAGE (:TAGGED-LITERAL INT 2)))
 (MOVE (:MESSAGE (:TAGGED-LITERAL INT 4)) (:J-REGISTER R2))
 (ADD (:J-REGISTER R2) (:TAGGED-LITERAL INT -1) (:J-REGISTER R1))
 (SEND0 (:TEMPORARY (:J-REGISTER R1)))
 (SEND0 (:MESSAGE (:TAGGED-LITERAL INT 3)))
 (MOVE (:J-REGISTER NNR) (:J-REGISTER R2))
 (SEND0 (:J-REGISTER R2))
 (SENDE0 (:J-REGISTER A1))
 (MOVE (:TAGGED-LITERAL INT 0) (:J-REGISTER R1))
```

```
(MOVE (:J-REGISTER R1) (:J-REGISTER I))
(COMMENT "line 66")
(MOVE (:MESSAGE (:TAGGED-LITERAL INT 2)) (:J-REGISTER R2))
(MOVE (:J-REGISTER R2) (:FRAME (:TAGGED-LITERAL INT (PFSLOT0_0 . 7))))
(MOVE (:MESSAGE (:TAGGED-LITERAL INT 3)) (:J-REGISTER R1))
(MOVE (:J-REGISTER R1) (:FRAME (:TAGGED-LITERAL INT (PFSLOT0_1 . 8))))
(MOVE (:MESSAGE (:TAGGED-LITERAL INT 4)) (:J-REGISTER R2))
(MOVE (:J-REGISTER R2) (:FRAME (:TAGGED-LITERAL INT (JSLOT0 . 6))))
(MOVE (:J-REGISTER NNR) (:J-REGISTER R1))
(SEND0 (:J-REGISTER R1))
(DC (:TAGGED-LITERAL SPECIAL_TAG (:REF FACT_INLET_1_REF)))
(SEND0 (:J-REGISTER R0))
(SENDE0 (:J-REGISTER A1))
(SUSPEND)
(END 12 INLET 0)
(COMMENT "special inlet")
(START INLET 1)
(LABEL (:TAGGED-LITERAL LABEL FACT_INLET1))
(MOVE (:MESSAGE (:TAGGED-LITERAL INT 1)) (:J-REGISTER R2))
(MOVE (:J-REGISTER R2) (:J-REGISTER A1))
(COMMENT "line 66")
(MOVE (:FRAME (:TAGGED-LITERAL INT (SSLOT0 . 13))) (:J-REGISTER R1))
(SUB (:J-REGISTER R1) (:TAGGED-LITERAL INT 1) (:J-REGISTER R2))
(BZ (:J-REGISTER R2) (:TAGGED-LITERAL REL (:LABEL L0)))
(MOVE (:J-REGISTER R2) (:FRAME (:TAGGED-LITERAL INT (SSLOT0 . 13))))
(SUSPEND)
(LABEL (:TAGGED-LITERAL LABEL L0))
(END 2.0 INLET 1)
(COMMENT "line 100")
(START THREAD 0)
(FALL-THROUGH)
(COMMENT "thread 0")
(COMMENT "line 104")
(MOVE (:TAGGED-LITERAL INT 1) (:J-REGISTER R1))
(MOVE (:J-REGISTER R1) (:J-REGISTER I))
(SEND0 (:FRAME (:TAGGED-LITERAL INT (PFSLOT0_0 . 7))))
(MOVE (:FRAME (:TAGGED-LITERAL INT (JSLOT0 . 6))) (:J-REGISTER R2))
(ADD (:J-REGISTER R2) (:TAGGED-LITERAL INT 0) (:J-REGISTER R1))
(SEND0 (:TEMPORARY (:J-REGISTER R1)))
(SENDE0 (:FRAME (:TAGGED-LITERAL INT (PFSLOT0_1 . 8))))
(MOVE (:TAGGED-LITERAL INT 0) (:J-REGISTER R2))
(MOVE (:J-REGISTER R2) (:J-REGISTER I))
(COMMENT "line 105")
(FFREE)
(END 4 THREAD 0)
(COMMENT "line 68")
(START INLET 11)
(LABEL (:TAGGED-LITERAL LABEL FACT_INLET11))
(MOVE (:MESSAGE (:TAGGED-LITERAL INT 1)) (:J-REGISTER R1))
```

```
(MOVE (:J-REGISTER R1) (:J-REGISTER A1))
(COMMENT "line 70")
(COMMENT "line 71")
(COMMENT "receive of psslot0 eliminated")
(COMMENT "   fall through")
(END 1 INLET 11)
(COMMENT "line 108")
(START THREAD 1)
(FALL-THROUGH)
(COMMENT "thread 1")
(SUSPEND)
(END 1 THREAD 1)
(COMMENT "line 74")
(START INLET 12)
(LABEL (:TAGGED-LITERAL LABEL FACT_INLET12))
(MOVE (:MESSAGE (:TAGGED-LITERAL INT 1)) (:J-REGISTER R2))
(MOVE (:J-REGISTER R2) (:J-REGISTER A1))
(COMMENT "line 76")
(COMMENT "line 77")
(COMMENT "   fall through")
(END 1 INLET 12)
(COMMENT "line 111")
(START THREAD 2)
(FALL-THROUGH)
(COMMENT "thread 2")
(COMMENT "line 113")
(MOVE (:MESSAGE (:TAGGED-LITERAL INT 2)) (:J-REGISTER R1))
(MOVE (:J-REGISTER R1) (:FRAME (:TAGGED-LITERAL INT (ISLOT1 . 2))))
(LE (:J-REGISTER R1) (:TAGGED-LITERAL INT 1) (:J-REGISTER R2))
(COMMENT "line 114")
(TLO START NSWITCH)
(BF (:J-REGISTER R2) (:TAGGED-LITERAL REL (:LABEL L1)))
(ADD (:J-REGISTER R3) (:TAGGED-LITERAL INT 1) (:J-REGISTER R3))
(DC (:TAGGED-LITERAL LOCAL-IP (:LABEL FACT_THREAD5)))
(MOVE (:J-REGISTER R0) (:LCV (:J-REGISTER R3)))
(BR (:TAGGED-LITERAL REL (:LABEL L2)))
(LABEL (:TAGGED-LITERAL LABEL L1))
(ADD (:J-REGISTER R3) (:TAGGED-LITERAL INT 1) (:J-REGISTER R3))
(DC (:TAGGED-LITERAL LOCAL-IP (:LABEL FACT_THREAD7)))
(MOVE (:J-REGISTER R0) (:LCV (:J-REGISTER R3)))
(LABEL (:TAGGED-LITERAL LABEL L2))
(TLO END NSWITCH)
(COMMENT "line 115")
(TLO START SSWITCH)
(BT (:J-REGISTER R2) (:TAGGED-LITERAL REL (:LABEL FACT_THREAD3)))
(COMMENT "   fall through")
(TLO END SSWITCH)
(END 2.0 THREAD 2)
(COMMENT "line 120")
```

```
(START THREAD 4)
(LABEL (:TAGGED-LITERAL LABEL FACT_THREAD4))
(COMMENT "line 121")
(MOVE (:TAGGED-LITERAL INT 1) (:J-REGISTER R1))
(MOVE (:J-REGISTER R1) (:J-REGISTER I))
(MP-FALLOC (:J-REGISTER R1))
(DC (:TAGGED-LITERAL SPECIAL_TAG (:REF SYS_ALLOC_MSG_REF)))
(SEND1 (:J-REGISTER R0))
(DC (:TAGGED-LITERAL SPECIAL_TAG (:REF FACT_1)))
(SEND1 (:J-REGISTER R0))
(MOVE (:J-REGISTER NNR) (:J-REGISTER R1))
(SEND1 (:J-REGISTER R1))
(SEND1 (:J-REGISTER A1))
(DC (:TAGGED-LITERAL SPECIAL_TAG (:REF FACT_INLET_14_POINTER_REF)))
(SENDE1 (:J-REGISTER R0))
(MOVE (:TAGGED-LITERAL INT 0) (:J-REGISTER R1))
(MOVE (:J-REGISTER R1) (:J-REGISTER I))
(SUSPEND-STRING)
(END 6 THREAD 4)
(COMMENT "line 80")
(START INLET 14)
(LABEL (:TAGGED-LITERAL LABEL FACT_INLET14))
(MOVE (:MESSAGE (:TAGGED-LITERAL INT 1)) (:J-REGISTER R1))
(MOVE (:J-REGISTER R1) (:J-REGISTER A1))
(COMMENT "line 82")
(COMMENT "line 83")
(COMMENT ONLY-USE)
(COMMENT "line 84")
(MOVE (:FRAME (:TAGGED-LITERAL INT (SSLOT2 . 15))) (:J-REGISTER R2))
(SUB (:J-REGISTER R2) (:TAGGED-LITERAL INT 1) (:J-REGISTER R1))
(BZ (:J-REGISTER R1) (:TAGGED-LITERAL REL (:LABEL L5)))
(MOVE (:J-REGISTER R1) (:FRAME (:TAGGED-LITERAL INT (SSLOT2 . 15))))
(MOVE (:MESSAGE (:TAGGED-LITERAL INT 2)) (:J-REGISTER R2))
(MOVE (:J-REGISTER R2) (:FRAME (:TAGGED-LITERAL INT (PFSLOT1_0 . 9))))
(MOVE (:MESSAGE (:TAGGED-LITERAL INT 3)) (:J-REGISTER R1))
(MOVE (:J-REGISTER R1) (:FRAME (:TAGGED-LITERAL INT (PFSLOT1_1 . 10))))
(SUSPEND)
(LABEL (:TAGGED-LITERAL LABEL L5))
(END 3.0 INLET 14)
(COMMENT "line 136")
(START THREAD 8)
(FALL-THROUGH)
(COMMENT "thread 8")
(COMMENT "line 140")
(MOVE (:TAGGED-LITERAL INT 1) (:J-REGISTER R2))
(MOVE (:J-REGISTER R2) (:J-REGISTER I))
(SEND0 (:MESSAGE (:TAGGED-LITERAL INT 2)))
(DC (:TAGGED-LITERAL SPECIAL_TAG (:REF FACT_INLET_12_REF)))
(SEND0 (:J-REGISTER R0))
(SEND0 (:MESSAGE (:TAGGED-LITERAL INT 3)))
(SEND0 (:FRAME (:TAGGED-LITERAL INT (ISLOT2 . 3))))
(SENDE0 (:TAGGED-LITERAL SYM 0))
(MOVE (:TAGGED-LITERAL INT 0) (:J-REGISTER R1))
(MOVE (:J-REGISTER R1) (:J-REGISTER I))
(SUSPEND)
(END 4 THREAD 8)
(COMMENT "line 86")
(START INLET 15)
(LABEL (:TAGGED-LITERAL LABEL FACT_INLET15))
(MOVE (:MESSAGE (:TAGGED-LITERAL INT 1)) (:J-REGISTER R2))
(MOVE (:J-REGISTER R2) (:J-REGISTER A1))
(COMMENT "line 88")
(COMMENT "   fall through")
(END 1 INLET 15)
(COMMENT "line 150")
(START THREAD 10)
(FALL-THROUGH)
(COMMENT "thread 10")
(COMMENT "line 152")
(TLO START SFALL)
(MOVE (:FRAME (:TAGGED-LITERAL INT (SSLOT0 . 13))) (:J-REGISTER R1))
(SUB (:J-REGISTER R1) (:TAGGED-LITERAL INT 1) (:J-REGISTER R2))
(BZ (:J-REGISTER R2) (:TAGGED-LITERAL REL (:LABEL FACT_THREAD0)))
(MOVE (:J-REGISTER R2) (:FRAME (:TAGGED-LITERAL INT (SSLOT0 . 13))))
(SUSPEND)
(TLO END SFALL)
(END 1 THREAD 10)
(COMMENT "line 100")
(START THREAD 0)
(LABEL (:TAGGED-LITERAL LABEL FACT_THREAD0))
(COMMENT "line 104")
(MOVE (:TAGGED-LITERAL INT 1) (:J-REGISTER R1))
(MOVE (:J-REGISTER R1) (:J-REGISTER I))
(SEND0 (:FRAME (:TAGGED-LITERAL INT (PFSLOT0_0 . 7))))
(MOVE (:FRAME (:TAGGED-LITERAL INT (JSLOT0 . 6))) (:J-REGISTER R2))
(ADD (:J-REGISTER R2) (:TAGGED-LITERAL INT 0) (:J-REGISTER R1))
(SEND0 (:TEMPORARY (:J-REGISTER R1)))
(SENDE0 (:FRAME (:TAGGED-LITERAL INT (PFSLOT0_1 . 8))))
(MOVE (:TAGGED-LITERAL INT 0) (:J-REGISTER R2))
(MOVE (:J-REGISTER R2) (:J-REGISTER I))
(COMMENT "line 105")
(FFREE)
(END 3 THREAD 0)
(COMMENT "line 91")
(START INLET 16)
(LABEL (:TAGGED-LITERAL LABEL FACT_INLET16))
(MOVE (:MESSAGE (:TAGGED-LITERAL INT 1)) (:J-REGISTER R1))
(MOVE (:J-REGISTER R1) (:J-REGISTER A1))
```

```
(COMMENT "line 93")
(COMMENT "line 94")
(COMMENT ONLY-USE)
(COMMENT "line 95")
(MOVE (:FRAME (:TAGGED-LITERAL INT (SSLOT1 . 14))) (:J-REGISTER R2))
(SUB (:J-REGISTER R2) (:TAGGED-LITERAL INT 1) (:J-REGISTER R1))
(BZ (:J-REGISTER R1) (:TAGGED-LITERAL REL (:LABEL L6)))
(MOVE (:J-REGISTER R1) (:FRAME (:TAGGED-LITERAL INT (SSLOT1 . 14))))
(MOVE (:MESSAGE (:TAGGED-LITERAL INT 2)) (:J-REGISTER R2))
(MOVE (:J-REGISTER R2) (:FRAME (:TAGGED-LITERAL INT (ISLOT4 . 5))))
(SUSPEND)
(LABEL (:TAGGED-LITERAL LABEL L6))
(END 2.5 INLET 16)
(COMMENT "line 142")
(START THREAD 9)
(FALL-THROUGH)
(COMMENT "thread 9")
(COMMENT "line 146")
(MOVE (:FRAME (:TAGGED-LITERAL INT (ISLOT1 . 2))) (:J-REGISTER R1))
(MUL (:J-REGISTER R1) (:MESSAGE (:TAGGED-LITERAL INT 2)) (:J-REGISTER R2))
(MOVE (:J-REGISTER R2) (:FRAME (:TAGGED-LITERAL INT (ISLOT3 . 4))))
(COMMENT "line 147")
(MOVE (:J-REGISTER R2) (:FRAME (:TAGGED-LITERAL INT (ISLOT0 . 1))))
(COMMENT "line 148")
(TLO START NFALL)
(COMMENT "   fall through")
(TLO END NFALL)
(END 3 THREAD 9)
(COMMENT "line 127")
(START THREAD 6)
(LABEL (:TAGGED-LITERAL LABEL FACT_THREAD6))
(COMMENT "line 128")
(MOVE (:TAGGED-LITERAL INT 1) (:J-REGISTER R1))
(MOVE (:J-REGISTER R1) (:J-REGISTER I))
(SENDO (:FRAME (:TAGGED-LITERAL INT (PFSLOT0_0 . 7))))
(MOVE (:FRAME (:TAGGED-LITERAL INT (JSLOT0 . 6))) (:J-REGISTER R2))
(ADD (:J-REGISTER R2) (:TAGGED-LITERAL INT 1) (:J-REGISTER R1))
(SENDO (:TEMPORARY (:J-REGISTER R1)))
(SENDO (:FRAME (:TAGGED-LITERAL INT (PFSLOT0_1 . 8))))
(SENDO (:FRAME (:TAGGED-LITERAL INT (ISLOT0 . 1))))
(SENDEO (:TAGGED-LITERAL SYM 0))
(MOVE (:TAGGED-LITERAL INT 0) (:J-REGISTER R2))
(MOVE (:J-REGISTER R2) (:J-REGISTER I))
(COMMENT "line 129")
(TLO START SFALL)
(MOVE (:FRAME (:TAGGED-LITERAL INT (SSLOT0 . 13))) (:J-REGISTER R1))
(SUB (:J-REGISTER R1) (:TAGGED-LITERAL INT 1) (:J-REGISTER R2))
(BZ (:J-REGISTER R2) (:TAGGED-LITERAL REL (:LABEL FACT_THREAD0)))
(MOVE (:J-REGISTER R2) (:FRAME (:TAGGED-LITERAL INT (SSLOT0 . 13))))
(SUSPEND-STRING)
(TLO END SFALL)
(END 3 THREAD 6)
(COMMENT "line 117")
(START THREAD 3)
(LABEL (:TAGGED-LITERAL LABEL FACT_THREAD3))
(COMMENT "line 118")
(TLO START SFALL)
(MOVE (:FRAME (:TAGGED-LITERAL INT (SSLOT0 . 13))) (:J-REGISTER R1))
(SUB (:J-REGISTER R1) (:TAGGED-LITERAL INT 1) (:J-REGISTER R2))
(BZ (:J-REGISTER R2) (:TAGGED-LITERAL REL (:LABEL FACT_THREAD0)))
(MOVE (:J-REGISTER R2) (:FRAME (:TAGGED-LITERAL INT (SSLOT0 . 13))))
(SUSPEND-STRING)
(TLO END SFALL)
(END 0 THREAD 3)
(COMMENT "line 123")
(START THREAD 5)
(LABEL (:TAGGED-LITERAL LABEL FACT_THREAD5))
(COMMENT "line 124")
(MOVE (:FRAME (:TAGGED-LITERAL INT (ISLOT1 . 2))) (:J-REGISTER R1))
(MOVE (:J-REGISTER R1) (:FRAME (:TAGGED-LITERAL INT (ISLOT0 . 1))))
(COMMENT "line 125")
(TLO START NFALL)
(BR (:TAGGED-LITERAL REL (:LABEL FACT_THREAD6)))
(TLO END NFALL)
(END 1 THREAD 5)
(COMMENT "line 131")
(START THREAD 7)
(LABEL (:TAGGED-LITERAL LABEL FACT_THREAD7))
(COMMENT "line 132")
(MOVE (:FRAME (:TAGGED-LITERAL INT (ISLOT1 . 2))) (:J-REGISTER R2))
(SUB (:J-REGISTER R2) (:TAGGED-LITERAL INT 1) (:J-REGISTER R1))
(MOVE (:J-REGISTER R1) (:FRAME (:TAGGED-LITERAL INT (ISLOT2 . 3))))
(COMMENT "line 133")
(TLO START SFORK)
(MOVE (:FRAME (:TAGGED-LITERAL INT (SSLOT2 . 15))) (:J-REGISTER R2))
(SUB (:J-REGISTER R2) (:TAGGED-LITERAL INT 1) (:J-REGISTER R1))
(BNZ (:J-REGISTER R1) (:TAGGED-LITERAL REL (:LABEL LABEL7)))
(ADD (:J-REGISTER R3) (:TAGGED-LITERAL INT 1) (:J-REGISTER R3))
(DC (:TAGGED-LITERAL LOCAL-IP (:LABEL FACT_THREAD8)))
(MOVE (:J-REGISTER R0) (:LCV (:J-REGISTER R3)))
(BR (:TAGGED-LITERAL REL (:LABEL LABEL8)))
(LABEL (:TAGGED-LITERAL LABEL LABEL7))
(MOVE (:J-REGISTER R1) (:FRAME (:TAGGED-LITERAL INT (SSLOT2 . 15))))
(LABEL (:TAGGED-LITERAL LABEL LABEL8))
(TLO END SFORK)
(COMMENT "line 134")
(TLO START SFALL)
(MOVE (:FRAME (:TAGGED-LITERAL INT (SSLOT1 . 14))) (:J-REGISTER R2))
```

```
(SUB (:J-REGISTER R2) (:TAGGED-LITERAL INT 1) (:J-REGISTER R1))
(BZ (:J-REGISTER R1) (:TAGGED-LITERAL REL (:LABEL FACT_THREAD9)))
(MOVE (:J-REGISTER R1) (:FRAME (:TAGGED-LITERAL INT (SSLOT1 . 14))))
(SUSPEND-STRING)
(TLO END SFALL)
(END 2.0 THREAD 7)
(COMMENT "line 142")
(START THREAD 9)
(LABEL (:TAGGED-LITERAL LABEL FACT_THREAD9))
(COMMENT "line 146")
(MOVE (:FRAME (:TAGGED-LITERAL INT (ISLOT1 . 2))) (:J-REGISTER R2))
(MUL (:J-REGISTER R2) (:FRAME (:TAGGED-LITERAL INT (ISLOT4 . 5))) (:J-REGISTER R1))
(MOVE (:J-REGISTER R1) (:FRAME (:TAGGED-LITERAL INT (ISLOT3 . 4))))
(COMMENT "line 147")
(MOVE (:J-REGISTER R1) (:FRAME (:TAGGED-LITERAL INT (ISLOT0 . 1))))
(COMMENT "line 148")
(TLO START NFALL)
(BR (:TAGGED-LITERAL REL (:LABEL FACT_THREAD6)))
(TLO END NFALL)
(END 2 THREAD 9)
(COMMENT "line 136")
(START THREAD 8)
(LABEL (:TAGGED-LITERAL LABEL FACT_THREAD8))
(COMMENT "line 140")
(MOVE (:TAGGED-LITERAL INT 1) (:J-REGISTER R2))
(MOVE (:J-REGISTER R2) (:J-REGISTER I))
(SENDO (:FRAME (:TAGGED-LITERAL INT (PFSLOT1_0 . 9))))
(DC (:TAGGED-LITERAL SPECIAL_TAG (:REF FACT_INLET_12_REF)))
(SENDO (:J-REGISTER R0))
(SENDO (:FRAME (:TAGGED-LITERAL INT (PFSLOT1_1 . 10))))
(SENDO (:FRAME (:TAGGED-LITERAL INT (ISLOT2 . 3))))
(SENDEO (:TAGGED-LITERAL SYM 0))
(MOVE (:TAGGED-LITERAL INT 0) (:J-REGISTER R1))
(MOVE (:J-REGISTER R1) (:J-REGISTER I))
(SUSPEND-STRING)
(END 3 THREAD 8)
(COMMENT "line 154")
(START THREAD 11)
(LABEL (:TAGGED-LITERAL LABEL FACT_THREAD11))
(SUSPEND-STRING)
(END 0 THREAD 11)
(COMMENT "line 156")
(START THREAD 12)
(LABEL (:TAGGED-LITERAL LABEL FACT_THREAD12))
(COMMENT "line 157")
(SUSPEND-STRING)
(END 0 THREAD 12)
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_list") :NO-BRANCH)
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_0_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL MSG "(FACT_inlet0+FACT_place<<ip_offset_pos)|msg_u|5"))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_1_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL MSG "(FACT_inlet1+FACT_place<<ip_offset_pos)|msg_u|2"))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_2_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL SYM 0))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_3_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL SYM 0))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_4_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL SYM 0))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_5_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL SYM 0))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_6_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL SYM 0))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_7_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL SYM 0))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_8_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL SYM 0))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_9_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL SYM 0))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_10_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL SYM 0))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_11_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL MSG "(FACT_inlet11+FACT_place<<ip_offset_pos)|msg_u|4"))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_12_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL MSG "(FACT_inlet12+FACT_place<<ip_offset_pos)|msg_u|4"))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_13_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL SYM 0))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_14_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL MSG "(FACT_inlet14+FACT_place<<ip_offset_pos)|msg_u|4"))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_15_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL MSG "(FACT_inlet15+FACT_place<<ip_offset_pos)|msg_u|2"))
(LABEL (:TAGGED-LITERAL LABEL "FACT_inlet_16_pointer") :NO-BRANCH)
(DC (:TAGGED-LITERAL MSG "(FACT_inlet16+FACT_place<<ip_offset_pos)|msg_u|4"))
(END-CODEBLOCK FACT (0 5) (1 2) (11 4) (12 4) (14 4) (15 2) (16 4)))
```

A.4.3 MDP Assembly

```
set GSLOT0_1 = 18
set GSLOT0_0 = 17
set SSLOT2 = 15
set SSLOT1 = 14
set SSLOT0 = 13
set JSLOT0 = 6
set PSSLOT0_1 = 12
set PSSLOT0_0 = 11
```

```
set PFSLOT1_1 = 10
set PFSLOT1_0 = 9
set PFSLOT0_1 = 8
set PFSLOT0_0 = 7
set ISLOT4 = 5
set ISLOT3 = 4
set ISLOT2 = 3
set ISLOT1 = 2
set ISLOT0 = 1

module fact ; size = 161
entry fact_codeblock_start
entry fact_inlet_list
entry GC_FACT

GC_FACT:
dc {GC_0_FACT_0}
dc {GC_0_FACT_1}
entry GC_0_FACT

GC_0_FACT:
dc {FACT_0}
dc {FACT_1}
dc 1
dc nil
dc cfut:-1
dc cfut:-1
dc nil
dc nil

fact_codeblock_start:

;file FACT.tl0
;line 22
;FRAME SLOTS
;Frame header
;Frame body
;Frame size = 19
 DC 19
 DC 5
 DC nil
 DC ip:((FACT_INLET0+fact_place)<<ip_offset_pos)|ip_u|ip_a0_absolute
;line 55
FACT_INLET0:
 MOVE [1,A3], R2
 MOVE R2, A1
;line 57
;line 58
;line 59
```

```
;line 60
 MOVE 2, R1
 MOVE R1, [SSLOT2,A1]  ; SSLOT2 = 15
;line 61
 MOVE R1, [SSLOT1,A1]  ; SSLOT1 = 14
;line 62
 MOVE 3, R2
 MOVE R2, [SSLOT0,A1]  ; SSLOT0 = 13
;line 63
;line 64
;line 65
 MOVE 1, R1
 MOVE R1, I
 MOVE [4,A3], R2
 ADD R2, -1, R1
 SEND0 [2,A3]
 SEND0 [R1,A0]
 MOVE NNR, R2
 SEND20 [3,A3], R2
 SENDE0 A1
 MOVE 0, R1
 MOVE R1, I
;line 66
 MOVE [2,A3], R2
 MOVE R2, [PFSLOT0_0,A1]  ; PFSLOT0_0 = 7
 MOVE [3,A3], R1
 MOVE R1, [PFSLOT0_1,A1]  ; PFSLOT0_1 = 8
 MOVE [4,A3], R2
 MOVE R2, [JSLOT0,A1]  ; JSLOT0 = 6
 MOVE NNR, R1
 DC {FACT_INLET_1_REF}
 SEND20 R1, R0
 SENDE0 A1
 SUSPEND
;special inlet
FACT_INLET1:
 MOVE [1,A3], R2
 MOVE R2, A1
;line 66
 MOVE [SSLOT0,A1], R1  ; SSLOT0 = 13
 SUB R1, 1, R2
 BZ R2, ^L0
 MOVE R2, [SSLOT0,A1]  ; SSLOT0 = 13
 SUSPEND
L0:
;line 100
;thread 0
;line 104
 MOVE 1, R1
```

```
 MOVE R1, I
 MOVE [JSLOT0,A1], R2  ; JSLOT0 = 6
 ADD R2, 0, R1
 SEND0 [PFSLOT0_0,A1]  ; PFSLOT0_0 = 7
 SEND0 [R1,A0]
 SENDE0 [PFSLOT0_1,A1]  ; PFSLOT0_1 = 8
 MOVE 0, R2
 MOVE R2, I
;line 105
dc {fact_1}
move R0, R1
dc ip:(ffree+library_place<<ip_offset_pos)|ip_a0_absolute
move R0, ip
;line 68
FACT_INLET11:
 MOVE [1,A3], R1
 MOVE R1, A1
;line 70
;line 71
;receive of psslot0 eliminated
;   fall through
;line 108
;thread 1
 SUSPEND
;line 74
FACT_INLET12:
 MOVE [1,A3], R2
 MOVE R2, A1
;line 76
;line 77
;   fall through
;line 111
;thread 2
;line 113
 MOVE [2,A3], R1
 MOVE R1, [ISLOT1,A1]  ; ISLOT1 = 2
 LE R1, 1, R2
;line 114
 BF R2, ^L1
 ADD R3, 1, R3
 DC ip:((FACT_THREAD5+fact_place)<<ip_offset_pos)|ip_u|ip_a0_absolute
 MOVE R0, [R3,A0]
 BR ^L2
L1:
 ADD R3, 1, R3
 DC ip:((FACT_THREAD7+fact_place)<<ip_offset_pos)|ip_u|ip_a0_absolute
 MOVE R0, [R3,A0]
L2:
;line 115
```

```
 BT R2, ^FACT_THREAD3
;   fall through
;line 120
FACT_THREAD4:
;line 121
 MOVE 1, R1
 MOVE R1, I
 MOVE [_ALLOC_LOC,A0], R1
 ADD R1, 1, R1
 AND R1, [_NPROC_MINUS_1_LOC,A0], R1
 MOVE R1, [_ALLOC_LOC,A0]
 MOVE [R1,A2], R1
 DC {SYS_ALLOC_MSG_REF}
 SEND21 R1, R0
 DC {FACT_1}
 MOVE NNR, R1
 SEND21 R0, R1
 DC {FACT_INLET_14_POINTER_REF}
 SEND2E1 A1, R0
 MOVE 0, R1
 MOVE R1, I
MOVE [R3,A0], R0
SUB R3, 1, R3
MOVE R0, IP

;line 80
FACT_INLET14:
 MOVE [1,A3], R1
 MOVE R1, A1
;line 82
;line 83
;ONLY-USE
;line 84
 MOVE [SSLOT2,A1], R2  ; SSLOT2 = 15
 SUB R2, 1, R1
 BZ R1, ^L5
 MOVE R1, [SSLOT2,A1]  ; SSLOT2 = 15
 MOVE [2,A3], R2
 MOVE R2, [PFSLOT1_0,A1]  ; PFSLOT1_0 = 9
 MOVE [3,A3], R1
 MOVE R1, [PFSLOT1_1,A1]  ; PFSLOT1_1 = 10
 SUSPEND
L5:
;line 136
;thread 8
;line 140
 MOVE 1, R2
 MOVE R2, I
 DC {FACT_INLET_12_REF}
```

```
 SEND2O [2,A3], R0
 SENDO [3,A3]
 SENDO [ISLOT2,A1]
 SENDEO nil  ; ISLOT2 = 3
 MOVE 0, R1
 MOVE R1, I
 SUSPEND
;line 86
FACT_INLET15:
 MOVE [1,A3], R2
 MOVE R2, A1
;line 88
;   fall through
;line 150
;thread 10
;line 152
 MOVE [SSLOT0,A1], R1  ; SSLOT0 = 13
 SUB R1, 1, R2
 BZ R2, ^FACT_THREAD0
 MOVE R2, [SSLOT0,A1]  ; SSLOT0 = 13
 SUSPEND
;line 100
FACT_THREAD0:
;line 104
 MOVE 1, R1
 MOVE R1, I
 MOVE [JSLOT0,A1], R2  ; JSLOT0 = 6
 ADD R2, 0, R1
 SENDO [PFSLOT0_0,A1]  ; PFSLOT0_0 = 7
 SENDO [R1,A0]
 SENDEO [PFSLOT0_1,A1]  ; PFSLOT0_1 = 8
 MOVE 0, R2
 MOVE R2, I
;line 105
dc {fact_1}
move R0, R1
dc ip:(ffree+library_place<<ip_offset_pos)|ip_a0_absolute
move R0, ip
;line 91
FACT_INLET16:
 MOVE [1,A3], R1
 MOVE R1, A1
;line 93
;line 94
;ONLY-USE
;line 95
 MOVE [SSLOT1,A1], R2  ; SSLOT1 = 14
 SUB R2, 1, R1
 BZ R1, ^L6
 MOVE R1, [SSLOT1,A1]  ; SSLOT1 = 14
 MOVE [2,A3], R2
 MOVE R2, [ISLOT4,A1]  ; ISLOT4 = 5
 SUSPEND
L6:
;line 142
;thread 9
;line 146
 MOVE [ISLOT1,A1], R1  ; ISLOT1 = 2
 MUL R1, [2,A3], R2
 MOVE R2, [ISLOT3,A1]  ; ISLOT3 = 4
;line 147
 MOVE R2, [ISLOT0,A1]  ; ISLOT0 = 1
;line 148
;   fall through
;line 127
FACT_THREAD6:
;line 128
 MOVE 1, R1
 MOVE R1, I
 MOVE [JSLOT0,A1], R2  ; JSLOT0 = 6
 ADD R2, 1, R1
 SENDO [PFSLOT0_0,A1]  ; PFSLOT0_0 = 7
 SENDO [R1,A0]
 SENDO [PFSLOT0_1,A1]  ; PFSLOT0_1 = 8
 SENDO [ISLOT0,A1]
 SENDEO nil  ; ISLOT0 = 1
 MOVE 0, R2
 MOVE R2, I
;line 129
 MOVE [SSLOT0,A1], R1  ; SSLOT0 = 13
 SUB R1, 1, R2
 BZ R2, ^FACT_THREAD0
 MOVE R2, [SSLOT0,A1]  ; SSLOT0 = 13
MOVE [R3,A0], R0
SUB R3, 1, R3
MOVE R0, IP

;line 117
FACT_THREAD3:
;line 118
 MOVE [SSLOT0,A1], R1  ; SSLOT0 = 13
 SUB R1, 1, R2
 BZ R2, ^FACT_THREAD0
 MOVE R2, [SSLOT0,A1]  ; SSLOT0 = 13
MOVE [R3,A0], R0
SUB R3, 1, R3
MOVE R0, IP
```

```
;line 123
FACT_THREAD5:
;line 124
 MOVE [ISLOT1,A1], R1  ; ISLOT1 = 2
 MOVE R1, [ISLOT0,A1]  ; ISLOT0 = 1
;line 125
 BR ^FACT_THREAD6
;line 131
FACT_THREAD7:
;line 132
 MOVE [ISLOT1,A1], R2  ; ISLOT1 = 2
 SUB R2, 1, R1
 MOVE R1, [ISLOT2,A1]  ; ISLOT2 = 3
;line 133
 MOVE [SSLOT2,A1], R2  ; SSLOT2 = 15
 SUB R2, 1, R1
 BNZ R1, ^LABEL7
 ADD R3, 1, R3
 DC ip:((FACT_THREAD8+fact_place)<<ip_offset_pos)|ip_u|ip_a0_absolute
 MOVE R0, [R3,A0]
 BR ^LABEL8
LABEL7:
 MOVE R1, [SSLOT2,A1]  ; SSLOT2 = 15
LABEL8:
;line 134
 MOVE [SSLOT1,A1], R2  ; SSLOT1 = 14
 SUB R2, 1, R1
 BZ R1, ^FACT_THREAD9
 MOVE R1, [SSLOT1,A1]  ; SSLOT1 = 14
MOVE [R3,A0], R0
SUB R3, 1, R3
MOVE R0, IP

;line 142
FACT_THREAD9:
;line 146
 MOVE [ISLOT1,A1], R2  ; ISLOT1 = 2
 MUL R2, [ISLOT4,A1], R1  ; ISLOT4 = 5
 MOVE R1, [ISLOT3,A1]  ; ISLOT3 = 4
;line 147
 MOVE R1, [ISLOT0,A1]  ; ISLOT0 = 1
;line 148
 BR ^FACT_THREAD6
;line 136
FACT_THREAD8:
;line 140
 MOVE 1, R2
 MOVE R2, I
 DC {FACT_INLET_12_REF}
 SEND20 [PFSLOT1_0,A1], R0  ; PFSLOT1_0 = 9
 SEND0 [PFSLOT1_1,A1]  ; PFSLOT1_1 = 10
 SEND0 [ISLOT2,A1]
 SENDE0 nil  ; ISLOT2 = 3
 MOVE 0, R1
 MOVE R1, I
MOVE [R3,A0], R0
SUB R3, 1, R3
MOVE R0, IP

;line 154
FACT_THREAD11:
MOVE [R3,A0], R0
SUB R3, 1, R3
MOVE R0, IP

;line 156
FACT_THREAD12:
;line 157
MOVE [R3,A0], R0
SUB R3, 1, R3
MOVE R0, IP

FACT_inlet_list:
FACT_inlet_0_pointer:
 DC msg:(FACT_inlet0+FACT_place<<ip_offset_pos)|msg_u|5
FACT_inlet_1_pointer:
 DC msg:(FACT_inlet1+FACT_place<<ip_offset_pos)|msg_u|2
FACT_inlet_2_pointer:
 DC nil
FACT_inlet_3_pointer:
 DC nil
FACT_inlet_4_pointer:
 DC nil
FACT_inlet_5_pointer:
 DC nil
FACT_inlet_6_pointer:
 DC nil
FACT_inlet_7_pointer:
 DC nil
FACT_inlet_8_pointer:
 DC nil
FACT_inlet_9_pointer:
 DC nil
FACT_inlet_10_pointer:
 DC nil
FACT_inlet_11_pointer:
 DC msg:(FACT_inlet11+FACT_place<<ip_offset_pos)|msg_u|4
FACT_inlet_12_pointer:
```

```
 DC msg:(FACT_inlet12+FACT_place<<ip_offset_pos)|msg_u|4
FACT_inlet_13_pointer:
 DC nil
FACT_inlet_14_pointer:
 DC msg:(FACT_inlet14+FACT_place<<ip_offset_pos)|msg_u|4
FACT_inlet_15_pointer:
 DC msg:(FACT_inlet15+FACT_place<<ip_offset_pos)|msg_u|2
FACT_inlet_16_pointer:
 DC msg:(FACT_inlet16+FACT_place<<ip_offset_pos)|msg_u|4
entry FACT_inlet0
entry FACT_inlet_0_pointer
entry FACT_inlet1
entry FACT_inlet_1_pointer
entry FACT_inlet2
entry FACT_inlet_2_pointer
entry FACT_inlet3
entry FACT_inlet_3_pointer
entry FACT_inlet4
entry FACT_inlet_4_pointer
entry FACT_inlet5
entry FACT_inlet_5_pointer
entry FACT_inlet6
entry FACT_inlet_6_pointer
entry FACT_inlet7
entry FACT_inlet_7_pointer
entry FACT_inlet8
entry FACT_inlet_8_pointer
entry FACT_inlet9
entry FACT_inlet_9_pointer
entry FACT_inlet10
entry FACT_inlet_10_pointer
entry FACT_inlet11
entry FACT_inlet_11_pointer
entry FACT_inlet12
entry FACT_inlet_12_pointer
entry FACT_inlet13
entry FACT_inlet_13_pointer
entry FACT_inlet14
entry FACT_inlet_14_pointer
entry FACT_inlet15
entry FACT_inlet_15_pointer
entry FACT_inlet16
entry FACT_inlet_16_pointer

end

ref FACT_0 = l_nnr
ref FACT_1 = addr:(FACT_place+FACT_codeblock_start<<ip_offset_pos)
ref GC_FACT_0 = l_nnr
ref GC_FACT_1 = addr:((FACT_place+GC_FACT)<<ip_offset_pos)|2
ref GC_0_FACT_0 = l_nnr
ref GC_0_FACT_1 = addr:((FACT_place+GC_0_FACT)<<ip_offset_pos)|8
ref FACT_inlet_0_pointer_ref = FACT_inlet_0_pointer+FACT_place
ref FACT_inlet_0_ref = msg:(FACT_inlet0+FACT_place<<ip_offset_pos)|msg_u|5
ref FACT_inlet_1_pointer_ref = FACT_inlet_1_pointer+FACT_place
ref FACT_inlet_1_ref = msg:(FACT_inlet1+FACT_place<<ip_offset_pos)|msg_u|2
ref FACT_inlet_2_pointer_ref = FACT_inlet_2_pointer+FACT_place
ref FACT_inlet_3_pointer_ref = FACT_inlet_3_pointer+FACT_place
ref FACT_inlet_4_pointer_ref = FACT_inlet_4_pointer+FACT_place
ref FACT_inlet_5_pointer_ref = FACT_inlet_5_pointer+FACT_place
ref FACT_inlet_6_pointer_ref = FACT_inlet_6_pointer+FACT_place
ref FACT_inlet_7_pointer_ref = FACT_inlet_7_pointer+FACT_place
ref FACT_inlet_8_pointer_ref = FACT_inlet_8_pointer+FACT_place
ref FACT_inlet_9_pointer_ref = FACT_inlet_9_pointer+FACT_place
ref FACT_inlet_10_pointer_ref = FACT_inlet_10_pointer+FACT_place
ref FACT_inlet_11_pointer_ref = FACT_inlet_11_pointer+FACT_place
ref FACT_inlet_11_ref = msg:(FACT_inlet11+FACT_place<<ip_offset_pos)|msg_u|4
ref FACT_inlet_12_pointer_ref = FACT_inlet_12_pointer+FACT_place
ref FACT_inlet_12_ref = msg:(FACT_inlet12+FACT_place<<ip_offset_pos)|msg_u|4
ref FACT_inlet_13_pointer_ref = FACT_inlet_13_pointer+FACT_place
ref FACT_inlet_14_pointer_ref = FACT_inlet_14_pointer+FACT_place
ref FACT_inlet_14_ref = msg:(FACT_inlet14+FACT_place<<ip_offset_pos)|msg_u|4
ref FACT_inlet_15_pointer_ref = FACT_inlet_15_pointer+FACT_place
ref FACT_inlet_15_ref = msg:(FACT_inlet15+FACT_place<<ip_offset_pos)|msg_u|2
ref FACT_inlet_16_pointer_ref = FACT_inlet_16_pointer+FACT_place
ref FACT_inlet_16_ref = msg:(FACT_inlet16+FACT_place<<ip_offset_pos)|msg_u|4
```

Appendix B

Id Code for Paraffins

```
% -*- Package: ID-COMPILER -*-
%%% Id Example Program
%%% Title: Paraffins
%%% Original Author: Steve Heller
@include "basic-library";

%%% derived from /df/monsoon/009/paraffins-published.id, bounded outer loops.

% Generation of radicals.

type radical = H | C radical radical radical;

def length l =
  { n = 0
  in {for e <- l do
       next n = n + 1
     finally n }};

def 3_partitions m =
  {: (i,j,k) || i <- 0 to div m 3
              ; j <- i to div (m-i) 2
              ; k = m - (i+j) };


Def remainders l1 =
  { @!opcode hd,"HD";
    @!opcode tl,"TL";
    @!opcode store_hd,"STORE-HD";
    @!opcode store_tl,"STORE-TL";
    typeof hd = (list *0) -> *0;
    typeof tl = (list *0) -> (list *0);
    typeof store_hd = ((list *0),*0) -> void;
    typeof store_tl = ((list *0),(list *0)) -> void;

    handle = make_cons ();
    last = {while cons? l1 sequential do
              elt, next l1 = hd l1, tl l1;
              next handle = make_cons ();
              _ = store_hd (next handle, l1);
              _ = store_tl (handle,next handle);
            Finally handle };
```

```
    _ = store_tl (last, nil);
    result = tl handle
   in result };

%%%
%%% def remainders nil = nil
%%%  |  remainders (r:rs) = (r:rs) : (remainders rs);
%%%

def radical_generator n b =
  { radicals = {array (0,n) of
                | [0] = H:nil
                | [j] = rads_of_size_n radicals j || j <- 1 to n bound b}
  in
    radicals};


def rads_of_size_n radicals n =
  {: C ri rj rk || (i,j,k) <- 3_partitions (n-1)
                 ; ri:ris <- remainders (radicals[i])
                 ; rj:rjs <- remainders (if (i == j) then ri:ris
                  else radicals[j])
                 ; rk     <- if (j == k) then rj:rjs
                      else radicals[k] };



% Generation of paraffins.

type paraffin = BCP radical radical | CCP radical radical radical radical;


def BCP_generator radicals n =
  if (odd? n) then
    nil
  else
    {: BCP r1 r2 || r1:r1s <- remainders (radicals[div n 2])
                  ; r2 <- r1:r1s };

def 4_partitions m =
  {partitions = nil;
   in
     {for i <- 0 to div m 4 do
        next partitions =
          {for j <- i to div (m-i) 3 do
             next partitions =
               {for k <- max j (ceiling ((float m)/(float 2)) - i - j)
                    to (div (m-i-j) 2) do
                  next partitions = (i,j,k,(m-(i+j+k))): partitions
               finally partitions}
          finally partitions}
     finally partitions}};

def CCP_generator radicals n =
  {: CCP ri rj rk rl || (i,j,k,l) <- 4_partitions (n-1)
                      ; ri:ris <- remainders (radicals[i])
                      ; rj:rjs <- remainders (if (i==j) then ri:ris
                       else radicals[j])
                      ; rk:rks <- remainders (if (j==k) then rj:rjs
                       else radicals[k])
                      ; rl <- if (k==l) then rk:rks
                       else radicals[l] };
```

```
def BCP_until n rb genb =
  { radicals = radical_generator (div n 2) rb;
  in
    {array (1,n) of
     | [j] = (BCP_generator radicals j) || j <- 1 to n bound genb }};

def CCP_until n rb genb =
  { radicals = radical_generator (div n 2) rb;
  in
    {array (1,n) of
     | [j] = (CCP_generator radicals j) || j <- 1 to n bound genb }};

def paraffins_until n rb genb =
  { radicals = radical_generator (div n 2) rb;
  in
    {array (1,n) of
     | [j] = (BCP_generator radicals j),
             (CCP_generator radicals j) || j <- 1 to n bound genb }};


%%% $[0 1 0 1 0 3 0 10 0 36]
def test_BCP_until n rb genb =
  { result = BCP_until n rb genb;
  in
    {array (1,n) of
     | [i] = { bv = result[i];
             in
               (length bv)}
             || i <- 1 to n bound genb }};

%%% $[1 0 1 1 3 2 9 8 35 39]
def test_CCP_until n rb genb =
  { result = CCP_until n rb genb;
  in
    {array (1,n) of
     | [i] = { cv = result[i];
             in
               (length cv) }
             || i <- 1 to n bound genb }};

%%% $[1 1 1 2 3 5 9 18 35 75]
def test_paraffins_until n rb genb =
  { result = paraffins_until n rb genb;
  in
    {array (1,n) of
     | [i] = { bv,cv = result[i];
             in
               (length bv) + (length cv) }
             || i <- 1 to n bound genb }};

def foo x y z = (x,y,z);

def main nf =
  {n = floor nf;
   rb = 2;
   genb = 2;
   bcp_result = test_BCP_until n rb genb;
   ccp_result = test_CCP_until n rb genb;
   paraffins_result = test_paraffins_until n rb genb;
  in
    foo bcp_result ccp_result paraffins_result};

def top = main 10.0;
```

Bibliography

[1] Ang, Boon S., Alejandro Caro, Stephen Glim, and Andrew Shaw. An Introduction to the Id Compiler. Computation Structures Group Memo 328. MIT Laboratory for Computer Science, Cambridge, MA, 1991.

[2] 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.

[3] Barth, Paul S., Nikhil, Rishiyur S., and Arvind. "M-Structures: Extending a Parallel, Non-strict, Functional Language with State." *Conference on Functional Programming Languages and Computer Architecture*, Cambridge, MA, 1991.

[4] Buehrer, Richard and Kattamuri Ekanadham. Incorporating Data Flow Ideas into von Neumann Processors for Parallel Execution. *IEEE Transactions on Computers* C-36(12):1515–1522 (December 1987).

[5] Chiou, Derek. "Frame Memory Management for the Monsoon Processor". *Proceedings of the 1992 MIT Student Workshop on VLSI and Parallel Systems.*

[6] Culler, D. E. Managing Parallelism and Resources in Scientific Dataflow Programs. Technical Report 446, MIT Laboratory for Computer Science, March 1990. (PhD Thesis, Dept. of EECS, MIT.)

[7] Culler, David E., Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. "Fine-grain Prallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine," *Proceedings of the Fourth International Con-*

ference on Architectural Support for Programming Languages and Operating Systems, 1991, pages 164 – 175.

[8] Dally, William J. Dataflow on the J-Machine. An unnummbered MIT Concurrent VLSI Architecture memo, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1988.

[9] Dally, William, et al. Message-Driven Processor Architecture. MIT Artificial Intelligence Lab Memo 1069, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1988.

[10] 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.

[11] Dally, William J., et al. The J-Machine: A Fine-Grain Concurrent Computer. *Information Processing 89, Proceedings of the IFIP Congress*, 1989.

[12] Gajski, D. D., D. A. Padua, D. J. Kuck, and R. H. Kuhn. A Second Opinion on Data Flow Machines and Languages. *IEEE Computer*, February 1982, pages 58–69.

[13] 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.)

[14] Nikhil, Rishiyur S., ID Language Reference Manual Version 90.1. Computation Structures Group Memo 284-2, MIT Laboratory for Computer Science, Cambridge, MA, 1991.

[15] Nikhil, Rishiyur S. and Arvind. Id: a language with implicit parallelism. Computation Structures Group Memo 305, MIT Laboratory for Computer Science, Cambridge, MA, 1991.

[16] Noakes, Michael, MDP Programer's Manual. MIT Concurrent VLSI Architecture Memo 40, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1991.

[17] 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.)

[18] Papadopoulos, Gregory M., and David E. Culler, Monsoon: An Explicit Token Store Architecture. In *Proceedings of the* 17^{th} *International Symposium on Computer Architecture*, Seattle, Washington, May 1990.

[19] Schauser, Klaus Erik. Compiling Dataflow into Threads: Efficient Compiler-Controlled Multithreading for Lenient Parallel Languages. Master's Project, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 1991.

[20] Spertus, Ellen. Preliminary Dataflow on the MDP. MIT Concurrent VLSI Archtiecture Memo 21, MIT Artificial Intelligence Laborary, Cambridge, MA, 1989.

[21] Spertus, Ellen. Dataflow Computation for the J-Machine. MIT Artificial Intelligence Laboratory Technical Report 1233, Cambridge, MA, 1990. (Bachelor's Thesis, Department of EECS, MIT.)

[22] Spertus, Ellen and William J. Dally. Experiments with Dataflow on a General-Purpose Parallel Computer. MIT Artificial Intelligence Laboratory Technical Memo 1272, Cambridge, MA, 1991.

[23] Spertus, Ellen and William J. Dally. Experiences Implementing Dataflow on a General-Purpose Parallel Computer. *Proceedings of the 1991 International Conference on Parallel Processing*, pages II-231–II-235.

[24] Spertus, Ellen. An Analysis of the J-Machine's Ability to Support Fine-Grained Computation. MIT Artificial Intelligence Laboratory Technical Memo 1380, Cambridge, MA, expected in 1993.

[25] 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.

[26] Traub, Kenneth R., David E. Culler, and Klaus E. Schauser. Global Analysis for Partitioning Non-Strict Programs into Sequential Threads. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, San Francisco, June 1992.

[27] 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.

[28] 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.)

[29] 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.)

[30] von Eicken, Thorsten; David E. Culler, Klaus Erik Schauser, and Seth Copen Goldstein. TL0 version 2.1: An implementation of Threaded Abstract Machine (draft). University of California, Berkeley, 1991.