Contents

A Portable Run-Time System for PCN

Ian Foster, Steve Tuecke
Argonne National Laboratory

Stephen Taylor
Caltech

December 7, 1992

Abstract

This report describes a run-time system to support Program Composition Notation (PCN), a high-level concurrent programming notation. The run-time system is described in terms of an abstract machine. We specify an abstract architecture that represents the state of a PCN computation and executes abstract machine instructions that encode tests on or modifications to the computation state. Programs to be executed on the abstract machine are encoded as sequences of abstract machine instructions. The abstract machine may be implemented by an emulator written in a low-level language. Alternatively, sequences of abstract machine instructions may be further compiled to machine code. The run-time system is designed to run on uniprocessors, multiprocessors, and multicomputers.

1 Introduction

This report describes a run-time system to support Program Composition Notation (PCN), a high-level concurrent programming notation [1]. The run-time system is described in terms of an abstract machine. An abstract machine for a programming notation implements its computational model. It provides an *abstract architecture* that represents the state of a computation in the chosen model and executes *abstract machine instructions* that encode tests on or modifications to the computation state. Programs to be executed on the abstract machine are encoded as sequences of abstract machine instructions. The abstract machine may be implemented by an emulator written in a low-level language. Alternatively, sequences of abstract machine instructions may be further compiled to machine code.

In designing an abstract machine for PCN, we have emphasized architectural simplicity, sometimes apparently at the expense of efficiency. In particular, we provide no special support for sequential composition or nested choice blocks. Hence, the machine can only execute directly programs expressed in a PCN subset referred to as *Core PCN*. We expect this "RISC"-like approach to permit novel optimizations in an implementation, and hence provide good overall performance. However, experience may motivate extensions to support particular language features.

1.1 Core PCN

Definitions. A *kernel* is a process that invokes a primitive operation. A *call* is a process that invokes another program. A kernel is *non-suspending* if it is a definition or if it is an assignment or other kernel for which the input arguments are known to be available at the time it is called. A kernel is *suspending* if it is not known to be non-suspending.

Core PCN is the language subset in which programs have the following restricted form.

1. A program has the form P {? C_1, ..., C_n}, n > 0.
2. Each choice C_i has the form G $\rightarrow$ B, where G is either a sequence of guard tests or the empty test `true`.
3. Each body B has the form {; K_1, ..., K_k, { || P_1, ..., P_l}}, k, $l \geq 0$, where the K_i are *non-suspending kernels* and the P_j are *calls*. If $k = 0$, the sequential block may be omitted; if $l < 2$, the parallel block may be omitted.

1.2 The Program Composition Machine

The Program Composition Machine (*PCM*) consists of three components: a *reduction* component, a *communication* component, and a *garbage collection component.* These execute at every processor.

Reduction Component. This provides the facilities required to execute Core PCN programs. That is, it maintains a *pool* of processes and repeatedly selects and attempts to execute processes in this pool. Execution of a process involves trying each of the choices in the associated program. If the guard associated with any choice evaluates to true, any kernels in the body of that choice are executed, and processes are created to execute any calls. Otherwise, the process is replaced in the pool.

Two important optimizations improve the efficiency of the basic model. These are support for *tail recursion* and a *scheduling structure*. Tail recursion permits execution to continue with a body process when a choice with one or more calls in its body is used to reduce a process. This avoids the overhead of adding the process to the process pool and subsequently selecting it. To ensure that reduction is *just*, tail recursion is only applied a finite number of times before the current process is added to the process pool and a new process is selected for reduction. The number of tail recursive calls permitted before such a process switch occurs is termed the *timeslice*.

The scheduling structure avoids the overhead of repeatedly attempting to reduce processes whose data is not available. It consists of a single *active queue* containing all reducible processes plus a *suspension structure* that links together processes requiring particular data.

Recall that guard execution in PCN reads terms, while definition statements may provide values for definitions. Both read and definition operations may generate communication if they encounter references to remote terms (i.e., remote references). This effectively provides a global address space.

Communication Component. This operates at the end of a time-slice and receives messages that arrive at a processor. It can modify local data structures and/or send outgoing messages. Five types of message can be received: *Read*, *Define*, *Value*, *Cancel*, and *Collect*.

The *Read* message signifies that a remote processor requests a copy of local data, to be provided when it becomes available. The *Value* message carries a data structure to be used locally and is received in response to a *Read* message. The *Define* message signifies that a remote processor has executed a definition operation that refers to a local definition. The *Cancel* message indicates that a remote processor no longer requires certain interprocessor references. The *Collect* message signifies that a remote processor requires this processor to perform a local garbage collection.

Garbage Collection Component. Programming systems that support automatic storage allocation and dynamic data structures generally require a garbage collector to reclaim inaccessible storage. PCN is no exception. Global analysis techniques and program annotations can support optimizations that allow certain programs to execute in constant space. However, a garbage collector is required in the general case.

The garbage collector employed in the PCM has a global and a local component. The global component supports asynchronous garbage collection: that is, it permits individual processors to reclaim inaccessible storage independently [3]. The local component employs a stop and copy algorithm [2].

1.3 Instruction Set

The PCM instruction set is summarized in Appendix A. To briefly illustrate the use of these instructions, consider the following program:

```
movej(lb,j,s,a,R)
int lb,j,s,a[]
{ ?
    j ≥ lb, a[j] > s → { ; j := j − 1, movej(lb,j,s,a,R)},
    default → R = []
}
```

This compiles into the following instruction sequence:

```
movej:  try(L1)                      % Start of 1st choice
          le(A0,A1)                  % j ≥ lb
          build_static(A5,int,1)     % Create space for a[j]
          get_element(A1,A3,A5)      % Access a[j]
          lt(A2,A5)                  % a[j] > s
          put_data(A6,1)             % Build integer 1
          sub(A1,A6,A7)              % j := j − 1
          copy_mut(A7,A1)
          recurse(movej,5)           % Recurse as movej
L1:     default(5)                   % Succeed if 1st choice failed
          build_static(A5,tuple,0)   % Create []
          define(A4,A5)              % R = []
          halt                       % Terminate process
```

The try instructions encode the beginning of choices; their arguments are labels which indicate where execution should continue if a choice's guard does not succeed. Matching and test operations in guards are encoded using instructions such as le; the build_static and get_element instructions are used to access arguments. The body of a choice is encoded using instructions such as put_data, which creates a new integer, sub, which performs subtraction, and define, which encodes a definition. The recurse instruction encodes a tail-recursive call to a new program; halt encodes process termination. Another instruction, fork, is used to encode process creation.

1.4 Foreign Interface

A call to a program written in a language such as Fortran or C is compiled to a sequence of put_foreign instructions, which set up a vector of pointers to arguments, followed by a call_foreign instruction, which invokes the foreign program. Further, machine-dependent primitives are required to load foreign code.

1.5 Performance Tools

The PCM incorporates low-level support for the Gauge profiling system. Each halt, recurse, and default instruction takes an offset to a counter as an argument and increments this counter each time it is executed. The call_foreign instruction takes an offset to a timer as an argument. This is used to accumulate the total time spent in the foreign program. The counters and timers are stored in code modules, and can be accessed using special primitives.

Support is also provided for animation tools. A segment of memory may optionally be reserved for storing information about program events; a special primitive allows programs to record events in this area. System facilities support the dumping of logged events to external storage.

2 Data Structures

In the rest of this report, we define first the various PCM data structures and then the abstract instructions that operate on these data structures. This constitutes a specification for the reduction component. The communication and garbage collection components are described in separate sections.

The PCM uses a set of registers to hold important components of abstract machine state. Otherwise, it uses only a single data area, the heap. This is a contiguous sequence of 32-bit cells. All application program data structures and system control structures are allocated on the heap; all structures are *cell aligned.*

2.1 Control Structures

The PCM maintains various control structures representing processes, the suspension structure, interprocessor references, etc.

Process Records. A process record is a contiguous block of two or more cells. The first cell is the *next* field; it contains a pointer and is used for attaching the process into either the active queue or a suspension structure. The second cell is the *program* field which contains a pointer to the code which the process is to execute when it is scheduled. Additional cells contain references to process arguments.

Active Queue. The active queue is a list of process records, linked together using the *next field* in each record. Pointers to the first and last entries in the queue are held in abstract machine registers (c.f. AF and AB respectively).

Incoming Reference Table (IRT). A processor's IRT is used to record references from other processors to terms on the local heap. This permits garbage collection to be performed on a single processor independently of other processors. The IRT consists of a contiguous array of IRT entries, initially located at the bottom of the heap. A fixed-sized IRT is allocated initially; free entries are linked in a free list. If this free list becomes empty, the IRT is extended, as described in Section 5.1.

An IRT entry comprises two cells. The first is a 32 bit quantity representing the weight associated with the referred-to object. The second contains a pointer to either the referred-to object or, if the weight is zero, the next entry in the IRT free list.

Outgoing Reference Table (ORT). A processor's ORT is used to record local references to terms located on other processors. This permits the processor to cancel these references when local garbage collection indicates that they can be discarded. The ORT is maintained as a linked list of entries located on the heap. A pointer to the first element of this list is kept in a register (c.f. OH). The list is compacted at local garbage collection by removing free entries: i.e., those with weight zero. At other times, free entries are linked in both a free list and the ORT list.

Each ORT entry represents an outgoing reference to another processor and, if it has not become inaccessible, is referenced by a remote reference or value note (described below) located within local memory. An entry consists of four cells. The first contains a 31-bit integer value representing the weight associated with the interprocessor reference, plus a flag (the top bit) that indicates whether or not a *Read* request has been generated on that remote reference. The second contains an integer value representing an IRT index. The third cell contains either an integer value representing a processor identifier, or a reference to the next entry in the ORT free list. The fourth contains a reference to the next entry in the ORT. In addition, the high bit of the second cell is used as a mark bit during local garbage collection. The third cell is used to link an ORT entry into the ORT free list when the weight of that ORT entry is zero, indicating that the remote reference has been deleted.

Suspension Structure. During a reduction attempt, matching or guard evaluation may require values that are not yet available. These values may be either local definitions or remote terms referenced via remote references. A *suspension register* (c.f. SU) is used to record the suspension status of the current process. This register is set to 0 before a reduction attempt. If matching or guard evaluation requires a value, SU is set to the address of the required definition or remote reference. If subsequent evaluation requires a further value, SU is set to the value −1 to indicate that the current process requires more than one value. If a process requires more than one value, a *Read* message is generated for remote references, unless other processes have already read them. A flag associated with a remote reference indicates whether a *Read* message has already been generated.

If guard evaluation succeeds for any choice, the suspension register is *reset* causing entries to be discarded. If execution reaches the default choice, indicating that all previous choices have failed or suspended, the register's value is examined to determine whether the current process must be added to the suspension structure, as follows. If the value is zero, all previous choices have failed, and so execution proceeds with the body of the default choice. If the value is neither 0 nor −1, the process requires the value of a single definition or remote reference: the process is then added to a circular queue associated with the definition or remote reference. This case occurs most often and is the prime suspension technique.

A process that requires the value of more than one variable or remote reference (SU = −1) is added to a *Global Suspension Queue.* Pointers to the first and last entries in this queue are held in machine registers (c.f. AF and AB respectively). If the active queue ever becomes empty, all processes recorded on the global suspension queue are moved to the active queue. Execution then proceeds normally. However, if the active queue becomes empty again without any intervening reductions, the abstract machine enters a suspended state, which it exits only upon receipt of input from the keyboard or other processors. In order to ensure that reduction is just, the global wakeup is also performed at regular (but infrequent) intervals, even if the active queue never becomes empty.

Value Note. The communication component attaches a *value note* to an unbound definition when it receives a *Read* message requesting its value. This permits a *Value* message to be generated if the variable becomes bound. A value note consists of four contiguous

Table 1: Tag values

Type	Tag	2^2	2^1	2^0
Reference	REF	–	0	0
Undefined	UNDEF	0	1	0
Remote Ref	RREF	1	1	0
Tuple	TUP	0	0	1
String	STR	1	0	1
Integer	INT	0	1	1
Real	REAL	1	1	1

cells. The first is the *next field*; it points at the next process or suspension record in the suspension structure. The second contains a null value. The third is the ORT field; it contains a pointer to the ORT entry for the remote term for which a *Read* request is pending. The fourth field contains a pointer to the definition, so that its value can be accessed when the value note is processed.

2.2 PCN Data Types

Both application program data structures and executable code are represented on the heap by sequences of cells. Tuples, integers, reals, strings, arrays, and code are represented by *header cell* containing the tag and size information followed by one or more untagged *data cells* representing the data structure. This organization simplifies communication with programs written in other languages. Four data types are supported: *Tuple*, *String*, *Integer*, and *Real*. In addition, *Definitions* are represented by tagged cells, and reference chains are represented by tagged *Remote Reference* and *Local Reference* cells.

Tagged data types have a *three bit tag* located in the low bits of a cell. The *Local Reference* data type is distinguished by zeros in bits 0 and 1; all other data types have at least one of these bits set to 1. The tag values used to represent the different types are given in Table 1.

Data Headers. A data header cell has the general form:

$$< \text{SIZE(31-4), INLINE(3), TAG(2-0)} >$$

The SIZE field contains a positive size, expressed in terms of the number of elements (integers, reals, characters, tuple arguments) contained in the data structure. The INLINE field is 1 if the data structure is embedded in a module, and 0 otherwise.

Integers are stored as 32-bit quantities and reals as double-precision, 64-bit quantities. Strings contain four bytes per cell and are padded with null characters to a cell boundary.

References. References are represented by a single cell with the two low bits zero. Hence, a reference cell's contents can be interpreted as a pointer to a cell-aligned data structure.

Definitions. Definitions are represented by a single cell with a tag value of 2 (hexadecimal). The remainder of the cell contains a 29 bit cell offset from the beginning of the heap, which is used to construct a pointer to a circular suspension queue.

Remote References. A remote reference is represented by two cells. The first has a tag value of 3 (hexadecimal); the remainder of the cell contains a 29 bit cell offset from the beginning of the heap, which is used to construct a pointer to a circular suspension queue. The second cell contains a pointer to an ORT entry.

Code. Compiled code modules are represented as strings with a particular internal structure. A code module cannot be distinguished from other strings except by context. Figure 1 illustrates the format of a code module.

A module's header cell has a string tag, an inline flag of 0, and a size corresponding to the actual size of the module in bytes. The code string itself contains the following fields, in the following order:

1. *Counter Offset.* The offset, in cells, to the Counters field.

2. *Export Table.* A sequence of cell offsets to *programs* which are exported by the module (i.e., that appear in its export declaration). The last entry in the export table is a cell containing zero.

3. *Programs.* A sequence of compiled programs.

4. *Counters.* A sequence of counters and timers.

Each compiled program has the following fields in the following order:

1. *Idle Offset.* The offset to the timer used to accumulate idle time attributed to the program.

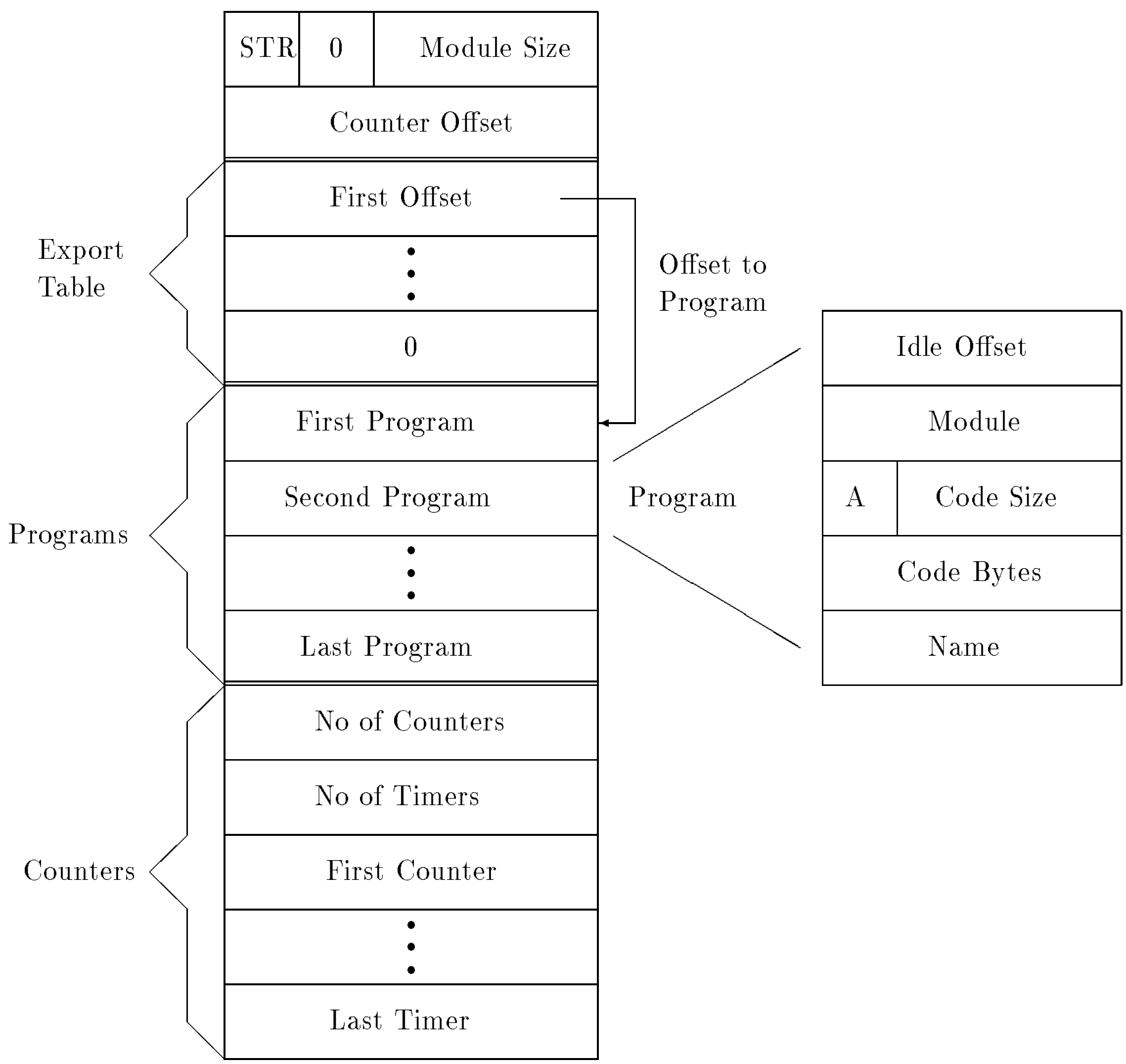
2. *Module.* A cell containing the cell offset from itself to the beginning of the module. This field is only used during garbage collection.

3. *Arity and Code Size.* A cell whose top byte is the arity of the program and whose low three bytes contain the *size* of the program in cells; this corresponds to the offset to the string for the program name. The Arity field must be a number less than 256, and is used when scheduling a process for execution.

4. *Code Bytes.* The assembled abstract machine instructions for the program. All abstract machine instructions are cell aligned.

5. *Name.* The program's name, as a null-terminated string padded with nulls to a cell boundary.

Figure 1: The Module Data Type



Note that offsets within a module are always to the code cells for a program, not the start of the program. The *Module*, *Code Size*, and *Arity* fields are accessed using negative offsets from the code cells.

Finally, the Counters area contains a one-cell counter for each **halt**, **recurse** and **suspend** instruction in the module, and a two-cell timer for each program and for each **call_foreign** instruction. The first two cells of this area specify the number of counters and timers. Subsequent cells contain first a sequence of counters, then a possible alignment cell to ensure double-cell alignment, and finally a sequence of timers. The order of the counters and the timers corresponds to the order of the corresponding instructions in the program definitions.

Code modules may be stored on disk in files. In this case, the module is preceded by a cell containing a magic number and version number, and the second cell contains the size (in cells) of the remainder. To ensure portability across machines with different byte ordering conventions, offsets, reals, and integers contained inside code are stored in a portable format.

Code modules on disk also contain three sections which are not placed on the heap when the module is loaded. The first two section contain lists of the foreign object files and foreign libraries, respectively, that are needed to resolve foreign function references in this module. These sections are (optionally) used to dynamicly link in foreign code at run time. The third section contains a list of all foreign functions that are referenced by this module as well as where in the code each function is called. This section is used during both dynamic and static linking of resolve foreign function references in the module.

2.3 Registers

The state of the abstract machine is held in registers. The following registers hold pointers to cells located on the heap:

- *HP* Heap Pointer, points to the top of the heap.
- *AF* Active queue Front, points to the first process in the active queue.
- *AB* Active queue Back, points to the last process in the active queue.
- *GF* Global queue Front, points to the first process in the global queue.
- *GB* Global queue Back, points to the last process in the global queue.
- *CP* Current Process, points to the process currently being reduced.
- *SP* Structure Pointer, a pointer used for building structures.
- *IFL* Irt Free List, points to the first entry in the IRT free list.
- *OFL* Ort Free List, points to the first entry in the ORT free list.
- *OH* Ort Head, points to the first entry in the ORT.

- *SU* SUspension, either points to a definition or remote reference for which the value is required by the current process; or contains the value 0 or -1.
- *ES* Event Stream, points to the tail of the global event stream.
- *KS* Keyboard Stream, points to the tail of the keyboard input stream.
- *PC* Program Counter, points to instruction cells on the heap.
- *FL* Failure Label, points to an instruction to branch to in case of choice failure.

Finally, the following auxiliary registers are used:

- *TS* Time Slice, an integer designating the remaining time slice for the current process.
- *BU* Buffer Flag, a boolean value which is set to true during tail recursion; otherwise false.
- *CA* Current Arity, an integer designating the arity of the current process.
- *A registers* A set of 256 registers which may hold references to *heap cells*. The A registers are used to hold process arguments and temporary values.
- *F registers* A set of 64 registers which may hold pointers to untagged data structures on the heap. The F registers are used to hold arguments to foreign procedures.
- *FP* Foreign Pointer, a pointer used for building calls to foreign procedures.
- *ISZ* Irt Size, an integer designating the number of entries in the IRT.
- *RF* Resize Flag, a boolean value which is set to true when the IRT has been resized; otherwise false.

3 Abstract Instruction Set

The instruction set includes six types of instruction:

1. *Control Instructions:* used to encode process scheduling and manipulation of various machine registers.
2. *Build Instructions:* used to construct data structures on the heap.
3. *Put Instructions:* used to place references to data structures in memory cells.
4. *Test Instructions:* used to encode guard execution.
5. *Term Manipulation Instructions:* used to encode various operations on terms.
6. *Foreign Instructions:* used to encode calls to foreign procedures.

The instruction set is summarized in Appendix A; example encodings are presented in Appendix B. Each instruction is assembled into one or more cell-aligned values. The top byte of the value is an op-code in the range zero to $N-1$ (where N is the number of abstract machine instructions). The main emulation loop simply *inspects* the op-code at the current program counter, *increments* the program counter and *dispatches* to execute the code for that instruction.

In the sections that follow a Pascal-like notation is used to explain the operation of each abstract machine instruction. Block structure is represented by *indentation.* The following notations are also used:

- $\underline{X := Y}$ Assignment of one variable to another.
- $\underline{=,\neq}$ Equality and inequality.
- $\underline{TAG\langle value\rangle}$ Denotes a cell with TAG and $value$ fields.
- $\underline{HEADER\langle tag,size\rangle}$ Denotes a data header cell with *tag* and *size* fields (and *inline* $= 0$).
- $\underline{is\_xxx(P)}$ Testing for specific data types (e.g., *is_integer(P)* tests if the cell at location P is a reference to an integer).
- $\underline{cell\_at(P),\ real\_at(P)}$ Denotes the byte, cell, real, etc. at address P.
- $\underline{byte\_1(P),\ byte\_2(P),\ byte\_3(P)}$ Denotes byte 1, 2 or 3 of the cell at address P.
- $\underline{tag\_at(P)}$ Denotes the *tag* value of the cell at address P.
- $\underline{offset\_to\_pointer(P)}$ Adds the 32-bit integer offset at location P to the pointer P and yields a new pointer to a cell.
- $\underline{case}$ A case statement in which execution enters and exits a single case and may not fall through to alternatives.

- *size_in_cells(tag,count)* Returns the number of cells required to hold *count* elements of type *tag*.

The following auxiliary functions will be used to define the instruction set:

- *is_unknown(C)* returns true if the heap cell C is a variable or a remote reference; false otherwise.

- *suspend_on(P)* manipulates the suspension register (SU) to record the fact that the current process requires the value at location P. Execution continues at the current failure label (FL).

- *fail()* causes execution to continue at the current failure label (FL).

- *dereference(P)* causes the reference P to be followed until P does not point to a reference.

- *enqueue_process(P)* places process P at the rear of the active queue using the active queue back register (AB).

- *schedule_process()* schedules a process from the front of the active queue using the active queue front register (AF). The process is made the current process by loading a pointer to it into the current process register (CP). Its arguments are then loaded into consecutive A registers beginning at register 0. This can be achieved since the program associated with a process includes the number of arguments (arity) in the process (c.f. Arity in Figure 1). The program counter (PC) is initialized to point at the encoded program associated with a process. In addition, the BU flag is set to false to indicate that the process arguments are currently unbuffered, and the CA register is loaded with the process arity.

- *process_susp_list(P)* processes the list of suspended processes and value notes at P. Processes are added to the active queue; *Value* messages are generated for value notes. Note that processes and value notes can be distinguished by the value of their second cell: processes contain a non-null pointer in this field and value notes a null value.

- *suspend_process()* suspends a process according to the value of the suspension register (c.f. Section 2.1).

- *save_arguments(N)* saves the contents of the first N A registers in the current process, if it is large enough (register CA $\geq$ N), or in a new process record otherwise.

- *signal(M)* appends a message M to the global event stream (c.f. register ES).

- *increment_counter(P)* increments the counter located at *offset_to_pointer*(P).

- *try_events()* checks whether garbage collection needs to be performed and processes any pending keyboard input and messages from other processors.

Detailed specifications are provided for most instructions in following sections. Unless stated otherwise, instructions assume that their arguments are dereferenced and available at the time of call. Hence, calls to most instructions cannot suspend. Type and range checking is optional; if performed, errors are signaled on the global event stream. In the specifications that follow, no type or range checking is performed.

3.1 Control Instructions

The process pool computational model is implemented using five control instructions: fork, halt, recurse, default, and try. The instructions are responsible for scheduling processes from the active queue, generating suspension structures, testing if garbage collection should be performed, etc. In addition, the run instruction is used to initiate execution of a process using a module and the send instruction to send a message on the global event stream.

fork(Label,Arity) allocates a new process record with a specified Arity and adds it to the rear of the active queue. The program field in the process is set to be Label and SP is set to point at the first argument of the process.

```
P := make_process(byte_1(PC))
cell_at(program_field(P)) := REFERENCE<offset_to_pointer(PC+1)>
SP := arguments_of(P)
PC := PC + 2
enqueue_process(P)
```

recurse(Label,Arity,CountOff) encodes tail recursion. It uses the values of the A registers and the current process record for the next reduction and thus saves process scheduling. If the time-slice (TS) is zero then Arity arguments (buffered in A registers) are saved in a process record (c.f. save_arguments). The process is then placed in the active queue with its program field set to Label and a new process is scheduled. If the time-slice is not over, execution proceeds from Label and the time-slice is decremented. The Buffer flag (BU) is set to true indicating that the process arguments are now buffered, and the Current Process Arity (CA) register is set to Arity. The suspension register is reset to indicate that there are no suspensions for the next reduction attempt. A check is made to determine if garbage collection is required. The counter associated with the instruction is incremented.

```
increment_counter(PC+1)
if (TS = 0) then
    save_arguments(byte_at(PC))
    cell_at(program_field(CP)) := REFERENCE<offset_to_pointer(PC+2)>
    enqueue_process(CP)
    schedule_process()
else
    TS := TS – 1
    BU := True
    cell_at(program_field(CP)) := REFERENCE<offset_to_pointer(PC+2)>
    PC := program_field(CP) + offset_to_code_bytes
SU := 0
try_event()
```

halt(CountOff) is used when a process reduces using a choice which has an empty body and thus *terminates*; this necessarily signifies the end of the current time-slice. The suspension register is reset. Another process is then scheduled and a test is made to determine if garbage collection is required. The counter associated with the instruction is incremented.

```
increment_counter(PC+1)
SU := 0
schedule_process()
try_event()
```

default(Arity,CountOff) causes the current process to proceed to the next instruction, to suspend on a single value, or to suspend on the global suspension queue, according to the value of the suspension register (c.f. Section 2.1). If suspension follows a recursive call then the arguments, buffered in A registers, must be saved in a process record. Suspension also requires that another process be scheduled and that the counter associated with the instruction be incremented.

```
if (SU = 0) then
    PC := PC+2
else
    increment_counter(PC+1)
    if (BU) then save_arguments(byte_1(PC))
    suspend_process()
    schedule_process()
    try_event()
```

try(Label) is used to encode conditional execution. It sets the failure label (FL) to Label; execution continues at the next instruction.

```
FL := offset_to_pointer(PC+1)
PC := PC + 2
```

run(M,P) is used to initiate execution of a process represented by a string or tuple P using the module M. An error is signaled if the program to be executed by P is not exported by M.

```
P1 := A[byte_1(PC)]
P2 := A[byte_2(PC)]
C := mlookup(P1,P2)
if (C = Null) then send(undefined(P1))
NP := make_process()              % Create a new process record
program_field(NP) := C            % New process is to execute P
"copy args from P2 to NP"
enqueue_process(NP)               % Add new process to active queue
PC := PC + 1
```

send(Reg) is used to append the term referenced by Reg to the global event stream. Any processes suspended on the definition referenced by ES are woken up; this definition is then overwritten with a reference to a new list structure. The contents of Reg are copied to the head of the new list structure, and a reference to a new definition is placed in both the list tail and ES.

```
P1 := A[byte_1(PC)]
if ( suspensions_at(ES) ) then process_susp_list(ES)
cell_at(ES) := REFERENCE<HP>
cell_at(HP) := HEADER<tuple_tag,2>
cell_at(HP+1) := REFERENCE<P1>
cell_at(HP+2) := REFERENCE<HP+3>
cell_at(HP+3) := DEFINITION<0>
ES := HP+3
HP := HP+4
PC := PC + 1
```

3.2 Build Instructions

The instructions **build_static**, **build_dynamic**, and **build_def** construct a data structure on the heap and place a reference to the new structure in a register. They differ only in the structures that they build.

build_static(Reg,Byte,Tag,Size) constructs a data area of specified Tag and Size on the heap, and places a reference to this area in Reg. If the data area represents a tuple, the structure pointer (SP) is set to point to its first element. This instruction is used to build arrays, mutable values, and tuples of size less than 31.

```
A[byte_1(PC)] := REFERENCE<HP>
cell_at(HP) := HEADER<byte_2(PC),integer_at(PC+1)>
if (byte_2(PC) = tuple_tag) then
    SP := HP + 1
HP := HP + 1 + size_in_cells(byte_2(PC), integer_at(PC+1))
PC := PC + 2
```

build_dynamic(Tag,Register1,Register2) constructs a integer array, real array, character array, or tuple filled with definitions (as specified by Tag), of size specified by the integer referenced by Register1, and places a reference to the structure in Register2.

```
tag := byte_1(PC)
size := integer_refed_by(A[byte_2(PC)])
A[byte_3(PC)] := REFERENCE<HP>
cell_at(HP) := HEADER<tag,size>
HP := HP + 1
if (tag = tuple_tag) then
    i := size
    while (i > 0) do
        cell_at(HP+size) := DEFINITION<0>
        HP := HP + 1
        i := i - 1
HP := HP + 1 + size_in_cells(tag,size)
PC := PC + 1
```

build_def(Register) constructs a definition on the heap and places a reference to the definition in a Register.

```
A[byte_1(PC)] := REFERENCE<HP>
cell_at(HP) := DEFINITION<0>
HP := HP + 1
PC := PC + 1
```

3.3 Put Instructions

The instructions put_data, put_value, and copy place a reference to a data structure in a memory cell. They differ only in the type of reference that they construct and where they put it.

put_data(Reg,Tag,Size,Value) places a reference to a value with tag Tag in Reg and increments PC by Size. The instruction is used to encode strings, single integers, and reals.

The instruction is followed by a negative offset to the beginning of the module, a data header cell with the INLINE field set to 1, and one or more data cells containing the data value.

A[byte_1(PC)] := REFERENCE<PC+3>
PC := PC + byte_3(PC)

put_value(Register) places the value in Register at the structure pointer, SP.

cell_at(SP) := A[byte_1(PC)]
SP := SP + 1
PC := PC + 1

copy(Register1,Register2) copies the contents of Register1 to Register2.

A[byte_2(PC)] := A[byte_1(PC)]
PC := PC + 1

3.4 Test Instructions

Test instructions encode test operations on process arguments. They are type, data, equal, neq, get_tuple, lt, and le. Of these, only type, data, and get_tuple can suspend. The other instructions assume that their arguments are available and dereferenced at the time of call.

In general, these instructions first obtain the number of an A register using the current program counter (PC). The register contents is then compared against some value. If the comparison succeeds execution proceeds at the next abstract machine instruction. Otherwise, execution proceeds at the current failure label. If the argument to a type, data or get_tuple instruction dereferences to a variable or remote reference, this fact is recorded using the suspension register, and execution proceeds at the current failure label (FL).

type(Register,Tag) tests that Register dereferences to a cell with the specified Tag. If the test succeeds, Register is overwritten with a reference to the dereferenced value.

```
P := A[byte_1(PC)]
dereference(P)
if (is_unknown(P)) then suspend_on(P)
if ( (tag_at(P) ≠ byte_2(PC) ) then fail()
A[byte_1(PC)] := REFERENCE<P>
PC := PC + 1
```

data(Register) succeeds when the value of the term referenced by Register becomes available. Register is overwritten as in the type instruction.

```
P := A[byte_1(PC)]
dereference(P)
if (is_unknown(P)) then suspend_on(P)
A[byte_1(PC)] := REFERENCE<P>
PC := PC + 1
```

get_tuple(Register1,Arity,Register2) is used to match structures. It tests that Register1 dereferences to a tuple of size Arity. If the test succeeds then the arguments of the tuple are loaded into consecutive A registers beginning with Register2.

```
P := A[byte_1(PC)]
dereference(P)
if (is_unknown(P)) then suspend_on(P)
arity := byte_2(PC)
if ( not is_tuple(P) OR arity ≠ cell_size_at(P)) then fail()
B := byte_3(PC)
while (arity > 0) do
    A[B] := cell_at(P)
    P := P + 1
    arity := arity − 1
    B := B + 1
PC := PC + 1
```

le(Register1,Register2) tests that the value of the integer or real referenced by Register1 is less than the value of that referenced by Register2.

```
P1 := A[byte_1(PC)]
P2 := A[byte_2(PC)]
PC := PC + 1
if (is_integer(P1) and is_integer(P2)) then
        if (integer_at(P1+1) > integer_at(P2+1)) then fail()
else if (is_real(P1+1) and is_real(P2+1)) then
        if (real_at(P1+1) > real_at(P2+1)) then fail()
else if (is_integer(P1+1) and is_real(P2+1)) then
        if (integer_at(P1+1) > real_at(P2+1)) then fail()
else if (is_real(P1+1) and is_integer(P2+1)) then
        if (real_at(P1+1) > integer_at(P2+1)) then fail()
else fail()
```

equal(Register1,Register2) tests that Register1 and Register2 reference single integers, single reals, or strings with the same value; or tuples of the same arity with equal subterms. Strings are compared using the C functions strcmp(): i.e. they are compared on a character by character basis until the first null character. Hence, this function cannot be used to test equality of character arrays.

3.5 Term Manipulation Instructions

The instructions get_element, put_element, and get_arg provide access to arrays and tuples. The instruction sizeof determines the size of a data structure. Arithmetic expressions on the right-hand side of calls to := are compiled to calls to five arithmetic kernels: add, sub, div, mul, and mod, which implement addition, subtraction, division, multiplication, and modulus respectively. Each takes two values as input, each either an integer or a real, and constructs either a real (if either input is a real) or an integer (if both inputs are integers) as output. The copy_mut instruction copies one mutable value to another of the same type. The coerce_mut instruction copies a mutable value to another that may be of a different type; this may require type coercion. The arguments to all these instructions are assumed to be available, dereferenced, and of the correct type. Finally, the define instruction is used in conjunction with build instructions to encode definition statements.

define(Register1,Register2) checks that Register1 dereferences to a variable or remote reference and generates an error if it does not. Otherwise, it processes any suspension queue attached to this location and assigns the location the contents of Register2. If the location is a remote reference then a *Define* message is generated.

```
P1 := A[byte_1(PC)]
P2 := A[byte_2(PC)]
PC := PC + 1
dereference(P1)
if (is_unknown(cell_at(P1))) then
    if (is_rem_ref(cell_at(P1))) then send_define(P1,P2)
    if (suspensions_at(P1)) then process_susp_list(P1)
    cell_at(P1) := cell_at(P2)
else
    signal_bad_define()
```

add(Register1,Register2,Register3) adds the numerical values referenced by Register1 and Register2, constructs a real or integer result on the heap, and places a reference to the result in Register3. The sub, mul, div, and mod instructions are similar; the mod instruction expects integer arguments.

```
P1 := A[byte_1(PC)]
P2 := A[byte_2(PC)]
A[byte_3(PC)] := REFERENCE<HP>
PC := PC + 1
if (is_integer(P1) and is_integer(P2)) then
    cell_at(HP) := HEADER<integer_tag,1>
    integer_at(HP+1) := integer_at(P1+1) + integer_at(P2+1)
    HP := HP + 2
else
    cell_at(HP) := HEADER<real_tag,1>
    if (is_integer(P1)) then d1 := integer_at(P1+1)
        else d1 := real_at(P1+1)
    if (is_integer(P2)) then d2 := integer_at(P2+1)
        else d2 := real_at(P2+1)
    real_at(HP+1) := d1 + d2
    HP := HP + 3
```

get_element(Register1,Register2,Register3) extracts an element of an integer, real, or character array. Register1 is assumed to reference an integer index, and Register2 an array. A data structure of the correct type to hold the retrieved element (an integer in the case of a character or integer array; otherwise a real) is constructed on the heap and a reference to this structure is placed in Register3.

```
P1 := A[byte_1(PC)]
P2 := A[byte_2(PC)]
P3 := REFERENCE<HP>
if (is_integer(P2)) then
    cell_at(HP) := HEADER<integer_tag,1>
    integer_at(HP+1) := integer_at(P2 + integer_at(P1+1))
    HP := HP + 2
else if (is_real(P2)) then
    cell_at(HP) := HEADER<real_tag,1>
    real_at(HP+1) := real_at((real *) P2 + integer_at(P1+1))
    HP := HP + 3
else if (is_string(P2)) then
    cell_at(HP) := HEADER<integer_tag,1>
    integer_at(HP+1) := (integer) character_at((char *) P2 + integer_at(P1+1))
    HP := HP + 2
PC := PC + 1
```

put_element(Register1,Register2,Register3) copies an integer or real value in a data item to a specified index in an integer, real, or character array. Register1 provides the index, Register2 the array, and Register3 the element. Type coercion is performed if necessary (integer to character, real to integer, integer to real).

```
P1 := A[byte_1(PC)]
P2 := A[byte_2(PC)]
P3 := A[byte_3(PC)]
if (is_integer(P2) and is_integer(P3)) then
    P := P2 + integer_at(P1+1)
    integer_at(P) := integer_at(P3+1)
else if (is_real(P2) and is_real(P3)) then
    P := (real *) P2 + integer_at(P1+1)
    real_at(P) := real_at(P3+1)
else if (is_character(P2) and is_integer(P3)) then
    P := (char *) P2 + integer_at(P1+1)
    character_at(P) := (character) integer_at(P3+1)
else if (is_real(P2) and is_integer(P3)) then
    P := (real *) P2 + integer_at(P1+1)
    real_at(P) := (real) integer_at(P3+1)
else if (is_integer(P2) and is_real(P3)) then
    P := (integer *) P2 + integer_at(P1+1)
    integer_at(P) := (integer) real_at(P3+1)
PC := PC + 1
```

get_arg(N,Tuple,Register) places a reference to the Nth argument of Tuple in Register.

```
P1 := A[byte_1(PC)]
P2 := A[byte_2(PC)]
A[byte_3(PC)] := REFERENCE<P2 + 1 + integer_at(P1+1)>
PC := PC + 1
```

sizeof(Register1,Register2) determines the size (i.e. number of elements) of the term referenced by Register1 and places an integer representing this size in the mutable integer referenced by Register2.

```
P1 := A[byte_1(PC)]
P2 := A[byte_2(PC)]
integer_at(P2+1) := cell_size_at(P1)
PC := PC + 1
```

copy_mut(Register1,Register2) copies the value of the mutable data structure referenced by Register1 into the mutable data structure referenced by Register2. The two structures are assumed to be of the same type and size. The coerce_mut instruction is similar but performs coercion.

```
P1 := A[byte_1(PC)]
P2 := A[byte_2(PC)]
i := size_in_cells(cell_tag_at(P1), cell_size_at(P1))
while (i > 0) do
    cell_at(P2) := cell_at(P1)
    i := i − 1
    P1 := P1 + 1
    P2 := P2 + 1
PC := PC + 1
```

3.6 Foreign Instructions

Two instructions, put_foreign and call_foreign, are used to encode calls to procedures written in languages such as C and Fortran.

put_foreign(Register) places a pointer to the data structure referenced by Register in the F register referenced by the register FP, and increments FP. Register is assumed to contain a previously dereferenced reference to a structure of the correct type.

cell_at(FP) := A[byte_at(PC)] + 1
FP := FP + 1
PC := PC + 1

call_foreign(Arity,Address,TimeOff) places a null pointer in the F register referenced by FP and invokes the procedure at **Address**, passing the address of F[0] as the base address for the argument vector. It resets FP and, upon return, increments the timer associated with the instructions.

then := time()
cell_at(FP) := NULL
FP := address_of(F[0])
CALL(byte_1(PC),integer_at(PC+1),FP)
FP := address_of(F[0])
increment_timer(PC+2, time() – then)
PC := PC + 3

4 The Communication Component

There are five types of message: *Read*, *Value*, *Define*, *Cancel*, and *Collect*. At the end of each time-slice, the communication component is invoked to process any pending messages. Each message received is deposited directly onto the heap. When sending a term it is *copied* into contiguous locations of a message buffer and absolute pointers are translated into relative offsets. On arrival, the term is *scanned* to perform the reverse translation.

Copying Terms. The copy_term procedure copies a term from the heap to the message buffer to be sent as part of a *Value* or *Define* message. The size of the term copied is determined by the size of the message buffer. A term is copied depth first with remote references to unbound subterms and to subterms that do not fit. A depth first traversal is used because it is particularly efficient for commonly used stream structures.

An IRT entry must be allocated when a new remote reference is created in a message. A reference to the local term is stored in this entry and a remote reference to the IRT entry (consisting of the node identifier and the IRT entry's index) is placed in the message. Both the newly created IRT entry and the remote reference are given the same large initial weight.

A remote reference may be encountered during copying. In this case, the ORT entry that it references is located. If this entry contains a weight greater than 1, a copy of the initial remote reference is included in the message. The weight associated with the ORT entry is split: a proportion (e.g., one third) is allocated to the remote reference, and the remainder retained locally.

If an interprocessor reference is repeatedly duplicated, its weight will eventually become one. Such a reference is *chained* next time it is duplicated: that is, a new reference to the reference with weight one is created, with a large initial weight. This involves creating a new IRT entry, as if a definition had been encountered.

A remote reference in a message has a different format to a remote reference on the heap. It consists of three cells. The first cell contains an integer value representing a weight, with a remote reference tag. The second and third cells contain integer values representing a node identifier and a location within a node respectively.

The copy_term procedure should be coded as an iterative algorithm which traverses a term and continues copying while there is room in the message buffer. In order to copy the term depth-first, the algorithm should maintain a stack of uncopied subterms. If remaining space in the message buffer is only adequate for remote references to stacked subterms, copying terminates and these remote references are created.

Scanning Terms. Scanning a term serves two purposes: it converts relative references into absolute references and allocates an ORT entry for each remote reference in the message. An ORT entry records the weight, processor identifier and IRT index associated with the remote reference. The remote reference in the message is replaced in the heap by a remote reference data type containing a reference to the ORT entry.

```
scan_term(P)
  while (not end_of_message(P)) do
    if (relative_reference(P)) then
      cell_at(P) := REFERENCE<P + offset_at(P)>
    if (remote_reference(P) then
      P1 := allocate_ort_entry(Weight(P),Node(P),Locn(P))
      cell_at(P) := REMREF<P1>
    P := P + size_of(term_at(P))
```

4.1 Message Processing

Define. The *Define* message contains a term to be copied locally and two integer values, which specify the IRT entry corresponding to the location to be assigned to and the weight associated with the remote reference to which the define operation was initially applied.

The destination IRT entry specified in the message is accessed to determine the location that is to be defined. The weight contained in the message is then subtracted from the weight associated with this IRT entry. If the location to be defined is a variable or a remote reference, the message is scanned. The location is then overwritten with the scanned term and suspended processes are woken up. If the location is a remote reference the message is forwarded; if it already has a value, a definition error is signaled.

```
define()
      extract_from_message(index,weight)
      locn := irt_address(index)
      cancel_irt_entry(index,weight)
      dereference(locn)
      scan(term)
      case tag_at(locn)
        Variable   : process_susp_list(locn)
                     cell_at(locn) := REFERENCE<term>
        RemRef     : process_susp_list(locn)
                     send_assign(locn,fnode,findex,term)
                     cell_at(locn) := REFERENCE<term>
        Otherwise  : send(deferror(locn,term))
```

A *Define* message is generated when a definition operation is applied to a remote reference. The remote reference specifies an ORT index; the ORT entry with this index is accessed to determine the remote processor identifier, remote address and weight components of the message. The ORT entry is then canceled.

Value. A *Value* message contains a term to be copied locally and an IRT index representing a location at which the term is to be placed. If this location is a remote reference or definition, the message is scanned. The location is then overwritten with the scanned term and suspended processes are woken up. Otherwise the message is discarded. In both cases the weight of the IRT entry is decremented by one.

```
value()
        extract_from_message(index,term)
        locn := irt_address(index)
        cancel_irt_entry(index,1)
        dereference(locn)
        if (is_unknown(cell_at(locn)) then
             scan(term)
             process_susp_list(locn)
             heap_at(locn) := REFERENCE<term>
```

Read. A *Read* message contains three integer values, which specify: the IRT entry representing the location to be read; the source processor; and the IRT entry representing the location to which the value is to be returned. The latter two fields specify the location of the remote reference at which the *Read* message originated.

If a *Read* message is received and it refers to a variable, a *value note* is attached to the variable. If it refers to a value, a *Value* message is generated to the requesting processor. If it refers to a remote reference, the message is forwarded. In neither case is the IRT entry modified.

```
read()
             extract_from_message(index,fnode,findex)
             locn := irt_address(index)
             dereference(locn)
             case tag_at(locn)
                Variable   : V := make_value_note(locn,fnode,findex)
                             enqueue_on_variable(V,locn)
                RemRef     : send_read(locn,fnode,findex)
                Otherwise  : send_value(fnode,findex,locn)
```

Cancel. A *Cancel* message contains an integer value specifying how many cancellations are encoded in the rest of the message, followed by a sequence of integer pairs that encode these cancellations. Each pair consists of an IRT index and a weight. A processor receiving a *Cancel* request decrements the weight of each referenced IRT entry by the associated weight. If the weight becomes zero, the IRT entry is added to the free list.

Collect. A *Collect* message requests a processor to perform immediate garbage collection.

5 Garbage Collection

Recall that the PCM locates all data structures created in the course of a computation – such as terms, code, and process records – in a memory area termed the *heap*. New structures are continually being created. However, most structures are never explicitly destroyed. *Garbage collection* must therefore be performed periodically to free heap space occupied by structures that are no longer required.

The garbage collector employed in the PCM has a global and a local component. The *global* component permits individual processors to reclaim inaccessible storage independently. In many situations, this can reduce both garbage collection overhead and real-time delays due to garbage collection. The global algorithm is based on that described in [3].

Many PCN processes iterate over lists. The iterative aspect of program behavior may result in a high proportion of useless verses useful data. It is thus beneficial to base the *local* algorithm on a stop and copy method [2]. Hence, the heap at each processor is organized as two semi-spaces: *oldspace* and *newspace*. Garbage collection causes accessible structures in the *oldspace* to be copied to the *newspace*; the two semi-spaces then exchange roles.

5.1 Global Collection

The global component of the garbage collector uses an extended reference counting algorithm (first described in [4]) to determine when memory cells accessed by remote references can be reclaimed. This algorithm can maintain reference counts in a multicomputer with substantially less communication than conventional reference counting. It associates weights with both interprocessor references and referenced objects. When an interprocessor reference is created, the reference is given an initial large weight and the weight of the referenced object is incremented by the same value. If the reference is subsequently duplicated, the weight in the reference is shared between the original reference and the copy (unless the weight is one, as discussed later). If the reference is reclaimed, a cancellation request is sent containing its weight to the object that it references. The weight of the reference is subtracted from the weight of the object. The algorithm hence maintains the invariant that the sum of weights of all references to an object (whether located on other processors or in messages in transit between processors) is equal to the weight associated with that object. When this weight becomes zero, there are no interprocessor references to the object, and it may be reclaimed by local garbage collection. Optimizations reduce the number of interprocessor communications generated by garbage collection.

Indirection tables are introduced to support the integration of reference-counting interprocessor garbage collection and copying intraprocessor collection. These permit relocation of data during local collection, identification of terms referenced from other processors, and concise storage of weights. The indirection tables are termed the Incoming Reference Table (*IRT*) and Outgoing Reference Table (*ORT*). A processor's IRT records references from other processors to local objects. A processor's ORT records references to terms located on other processors. Hence, every interprocessor reference has an entry in two indirection tables: one in the originating processor's ORT, and one in the IRT of the

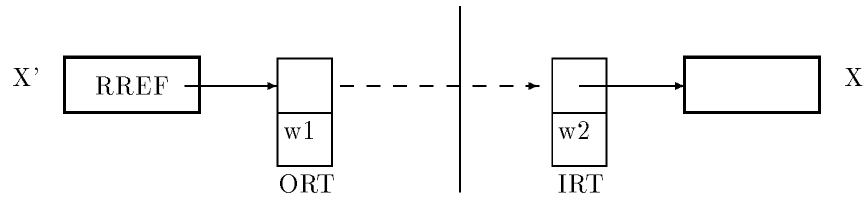


Figure 2: Incoming and outgoing reference tables

processor on which the referred-to object is located. Figure 2 illustrates this: a remote reference X' to a variable X passes via an ORT entry on one processor and an IRT entry on the other. The terms $w1$ and $w2$ represent the weights associated with the reference and the referred-to object, respectively. Note that $w1$ must be less than or equal to $w2$.

The form of the IRT and ORT has been described in Section 2.1. Here, we define the procedures that are used to manipulate these data structures.

Operations on the IRT

The IRT and its three associated registers (IFL, ISZ, RF) are manipulated using the following procedures.

irt_address(N) converts an IRT index into a heap address.

```
if (RF = True) then return(current_space + N)
else return(other_space + N)
```

irt_index(P) converts a heap address into an IRT index.

```
if (RF = True) then return(P − current_space)
else return(P − other_space)
```

allocate_irt_entry(Address, Weight) allocates an IRT entry from the IRT free list, extending the IRT if the free list is empty.

```
if (IFL = NULL) then extend_irt()
T := IFL
Index := irt_index(IFL)
IFL := cell_at(IFL+1)
cell_at(T) := Weight
cell_at(T+1) := REFERENCE<Address>
return(Index)
```

cancel_irt_entry(Index, Weight) decrements the weight associated with an IRT entry. If the entry's weight becomes zero, the entry is added to the IRT free list.

```
Address := irt_address(Index)
cell_at(Address) := cell_at(Address) – Weight
if (cell_at(Address) = 0) then
    cell_at(Address+1) := REFERENCE<IFL>
    IFL := Address
```

extend_irt() is used to extend the IRT when the IRT free list becomes empty. This involves copying the contents of the IRT to the bottom of the inactive semi-space, if this has not already be done, and setting the Resize Flag (RF); allocating space in the inactive semi-space, contiguous with the existing IRT, and linking this space into the IRT free list; and modifying the contents of the IRT Size register (ISZ) to reflect the new size of the IRT; and decrementing the amount of space available for the heap proper (free_space).

```
if (RF = False) then
    I := 0
    while (I < ISZ*2) do
        cell_at(other_space + I) := cell_at(current_space + I)
        I := I + 1
    RF := True
IFL := other_space + ISZ
P := other_space + ISZ
while (P < other_space + IRT_INCREMENT – 2) do
    cell_at(P) := 0
    cell_at(P+1) := REFERENCE<P+2>
    P := P + 2
cell_at(P) := 0
cell_at(P+1) := REFERENCE<0>
ISZ := ISZ + IRT_INCREMENT
free_space := free_space – IRT_INCREMENT
```

Operations on the ORT

Operations on the ORT and the associated registers OFL and OH are defined as follows.

allocate_ort_entry(Weight,Node,Location) allocates a new ORT entry, from the free list if possible and otherwise from the heap.

```
if (OFL == NULL) then
     OAddress := HP
     HP := HP + 4
     cell_at(OAddress + 3) := REFERENCE<OH>
     OH := OAddress
else
     OAddress := OFL
     OFL := cell_at(OFL+2)
cell_at(OAddress) := Weight
cell_at(OAddress+1) := Node
cell_at(OAddress+2) := Location
return(OAddress)
```

cancel_ort_entry(Address) returns an ORT entry to the ORT free list.

```
cell_at(Address+1) := 0
cell_at(Address+2) := REFERENCE<OFL>
OFL := Address
```

5.2 Local Collection

Local garbage collection causes accessible structures in the *oldspace* to be copied to the *newspace*. Accessible structures are those accessible from pointers into the heap. It is assumed that local garbage collection is performed only between reductions and that there is no current process. Hence, the pointers that must be followed comprise the queue registers (AF, AB, GF, and GB), the IRT, and pointers supporting asynchronous input and event handling (KS and ES).

The local garbage collector marks ORT entries that it encounters when copying; these correspond to accessible remote references. Once copying is completed, the ORT is scanned; unmarked entries can be discarded from the ORT. A cancellation request is generated for each unmarked entry. These requests are bundled in *Cancel* messages.

In outline, the algorithm operates as follows:

```
collect()
  hp := lp := newspace
  copy_irt_to_new_space()
  OH := copy_ort_to_new_space(OH)
  AF, AB, GF, GB, KS, ES := copy_to_new_space(AF, AB, GF, GB, KS, ES)
  while(lp < hp) do
    if ( newspace_at(lp) refers to oldspace ) then
      P := newspace_at(lp)
      dereference(P)
      if (in_new_space(P)) then
        newspace_at(lp) := P
      else
        newspace_at(lp) := copy_to_new_space(P)
    lp := lp + sizeof(object_at(lp))
  process_ort()
  "swap oldspace and newspace"


copy_to_new_space(pointer)
  "copy single level of structure at pointer to newspace at hp, represent
    substructures by references to their location in oldspace"
  "replace each copied item by a reference to its location in newspace"
  "increment hp by size of data copied"
  "return value of hp prior to increment"
```

Minor elaborations to this basic algorithm are required to ensure that untyped data structures are copied in their entirety. An untyped data structure is one whose structure can only be determined by its context and not by its representation on the heap. This is the case with process records and IRT and ORT entries. Hence, the IRT and ORT are copied before other structures, and are traced in a separate stage. Similarly, when tracing the active process queue (which originates in the register AF) and suspension structures, copying proceeds until all process records encountered have been copied to the new space. Finally, PCN data structures must be dealt with specially, to ensure that mutable data structures are not duplicated.

A process's code pointer references a program in a module. An entire module must be copied if any of its programs are referenced. The *module* field associated with each program is used to locate the start of a module and perform this operation when copying a process. The module's header cell is then overwritten with the new address of the module, to indicate that copying has occurred.

A reference to a tuple in the old space may be encountered when scanning the new space. The following algorithm is used to copy a tuple of arity A at location P. It returns a pointer to the location of the tuple in the new space.

```
copy_tuple(P,A)
  copy_tuple_size
  foreach argument
    copy arguments to new heap
  first argument := reference to new_heap
  return new address of first argument
```

When copying encounters a reference to a data structure other than a tuple, the header cell is examined. If the INLINE bit is set, the item is in a module. The negative offset contained in the preceding cell is applied to locate the beginning of the module. If the module's preceding cell is a reference to the new heap, then the module has already been copied; otherwise, the entire module is copied. If the INLINE bit is not set, the data structure is copied and header cell is set to reference the new location on the new heap. Finally, the position of the data structure in the new space (whether copied or not) is returned.

The algorithm used to copy data structures other than tuples (i.e., integer, real, or string) is summarized in the following description:

```
copy_data_item(P)
  if (inline(P)) then                  % Data in module
    P1 := P + integer_at(P–1)          % access module beginning
    P1 := copy_data_item(P1)           % copy the module
    P2 := P1 – integer_at(P-1)         % new location of data
    cell_at(P) := REFERENCE<P2>        % set old to point to new
    return(P1 – integer_at(P–2))       % return new location
  else                                 % Ordinary data structure
    copy_cells(cell_size_at(P))        % copy it
    cell_at(P) := REFERENCE<newspace location>
    return(newspace locations)
```

5.3 Garbage Collection Failure

Garbage collection is said to *fail* when computation cannot continue because of lack of free space. In *local* failure, a single processor runs out of space; in *global* failure, all processors run out of space. A garbage collector can in principle recover from local failure but not from global failure. Two mechanisms address the problem of *local* failure.

Broadcast. A processor in which local collection fails requests other processors to perform local collection, by broadcasting a *Collect* message. It then waits until either it receives a *Cancel* message, in which case it performs local garbage collection, or a time-out period elapses. If local free memory remains insufficient, the broadcast is repeated. An error is reported if a broadcast is repeated some fixed number of times without storage being reclaimed. A broadcast may be directed at all processors or, in a large parallel machine, to some subset of all processors.

Idle Collection. Idle processors perform local collection after a specified idle period. This optimization tends to reduce the frequency with which the broadcast mechanism must be invoked.

5.4 Deficiencies

We claim (but have not proved) that the garbage collector described here will eventually reclaim all garbage created by successful execution of a legal PCN program. However, illegal or non-terminating programs can create circular structures that cannot be reclaimed.

Illegal Programs. Illegal PCN programs that create circular data structures (i.e., definitions of the form X = Y, where Y contains X) can generate garbage that cannot be reclaimed.

Non-terminating Programs. If a non-terminating (and hence erroneous) program creates processes that suspend on more than one variable, these processes will remain on the global suspension queue indefinitely. Memory occupied by these processes and their data structures will never be reclaimed.

6 System Bootstrap

The bootstrapping of a *PCN* system involves loading a bootstrap module to each processor, building a collection of streams to link different processors, and creating an initial process that invokes the bootstrap process on each processor. By convention, the initial process is called boot. It is provided with a tuple as an argument; this contains the initial interprocessor streams. The boot argument has the form:

{Keybd,Events,ProcId,ProcCnt,UseHost,[In1,..,InN],[Out1,...,OutN]}

where Keybd is a stream from the keyboard, Events is the processor's event stream, ProcId is the processor identifier of this processor, ProcCnt is the number of processors being used, UseHost indicates whether or not the host node is to be used for process mapping (value = "y" or "n"), and the In's and Out's are streams to neighboring processors, used for input and output respectively. Processors are assumed to be numbered 0...N-1, for N nodes, where node N-1 is the host.

The input streams (In's) from other processors will be remote references to variables at known locations in the heap. The output streams are variables written into known locations for the benefit of other processors.

```
M = load_initial_module
C = mlookup(M,boot)
if ( C ≠ Null ) then
    P := make_process()
    procedure_field(P) := C
    S := build_boot_tuple()
    argument(P,0) := REFERENCE<S>
    enqueue_process(P)
    schedule_process()
else
    exit
```

In addition, the bootstrap procedure must initialize the various machine registers.

7 Asynchronous Keyboard Input

A convenient interface to asynchronous keyboard input is provided by a *keyboard event stream* which is incrementally bound to a list of characters typed at the keyboard. Architectural support for the event stream consists of a single machine registers, KS. This points to the tail of the stream. A check is made at the end of each time slice to determine whether input is pending; if so, the event stream is bound and KB is modified to point to the new tail. The nature of the check performed at each time slice depends on the capabilities of the underlying operating system. Two alternative techniques can be used. The first, preferred, technique is to provide an interrupt service routine that sets a flag when input is available. A new machine register, IP (the *Input Pending* flag), is used for this purpose. The implementation then only needs to check this flag at the end of each time slice. The second technique should only be used if the underlying operating system does not provide access to interrupts. In this case, the implementation must physically poll for events. If this is an expensive operation, polling may need to be performed less frequently (e.g., once every 100 time slices).

The keyboard event stream is made available to the initial process created at bootstrap time (c.f. Section 6).

References

[1] Chandy, M. and Taylor, S. The composition of parallel programs, *Proc. Supercomputing 89*, Reno, 1989.

[2] Cohen, J. Garbage collection of linked data structures, *Computing Surveys*, 13(3), 341-367, 1981.

[3] Foster, I. A multicomputer garbage collector for a single-assignment language, *International Journal of Parallel Programming*, 19(6), 1989.

[4] Weng, K., An abstract implementation for a generalized data flow language. MIT Laboratory for Computer Science TR-228, 1980.

A Abstract Instruction Set and Encoding

Abstract Instruction	OP	B1	B2	B3	Cell 1	Cell 2
fork(Procedure,A)	0	A	0	0	Offset	
recurse(Procedure,A)	1	A	0	0	CountOff	Offset
halt	2	0	0	0	CountOff	
default(A)	3	A	0	0	CountOff	
try(Label)	4	0	0	0	Offset	
run(R1,R2)	5	R1	R2	0		
send(R)	6	R	0	0		
build_static(R,T,Size)	7	R	T	0	Size	
build_dynamic(T,R1,R2)	8	T	R1	R2		
build_def(R)	9	R	0	0		
put_data(R,T,S,Value)	10	R	T	S	Value1	Value2 ...
put_value(R)	11	R	0	0		
copy(R1,R2)	12	R1	R2	0		
get_tuple(R1,A,R2)	13	R1	A	R2		
equal(R1,R2)	14	R1	R2	0		
neq(R1,R2)	15	R1	R2	0		
type(R1,Tag)	16	R1	Tag	0		
le(R1,R2)	17	R1	R2	0		
lt(R1,R2)	18	R1	R2	0		
data(R)	19	R1	R2	0		
sizeof(R1,R2)	20	R1	R2	0		
define(R1,R2)	21	R1	R2	0		
get_arg(R1,R2,R3)	22	R1	R2	R3		
get_element(R1,R2,R3)	23	R1	R2	R3		
put_element(R1,R2,R3)	24	R1	R2	R3		
add(R1,R2,R3)	25	R1	R2	R3		
sub(R1,R2,R3)	26	R1	R2	R3		
mul(R1,R2,R3)	27	R1	R2	R3		
div(R1,R2,R3)	28	R1	R2	R3		
copy_mut(R1,R2)	29	R1	R2	0		
coerce_mut(R1,R2)	30	R1	R2	0		
put_foreign(R)	31	R	0	0		
call_foreign(A,Address)	32	A	0	0	TimeOff	Address
exit()	33	0	0	0		

B Coding Examples

B.1 Partition: Array Version

PCN

```
partition(lb,ub,i,j,s,a)                           % partition array
int i,j,a[];
{ ? i<j →                                          % not done yet
    { ; { || movej(lb,j,s,a), movei(ub,i,s,a) },   % move in parallel
        i<j → { ; swap(i,j,a), i := i+1, j := j–1 },
        partition(lb,ub,i,j,s,a)                   % continue
    }
}
```

Core PCN

```
partition(lb,ub,i,j,s,a,L,R)
int i,j,a[];
{ ? data(L), i<j →
        { ||
            movej(lb,j,s,a,L,M1),
            movei(ub,i,s,a,L,M2),
            barrier2(M1,M2,R1),
            new$1(i,j,a,R1,R2),
            partition(lb,ub,i,j,s,a,R2,R)
        },
    default → R = [ ]
}
```

Abstract Instructions

```
partition/8: try(L1)
             data(R6)
             data(R2)
             data(R3)
             lt(R2,R3)
             build_def(R8)              % M1
             build_def(R9)              % M2
             build_def(R10)             % R1
             build_def(R11)             % R2
             fork(movei)
               put_value(R1)
               put_value(R2)
               put_value(R4)
```

```
          put_value(R5)
          put_value(R6)
          put_value(R8)
        fork(barrier2)
          put_value(R8)
          put_value(R9)
          put_value(R10)
        fork(new$1)
          put_value(R2)
          put_value(R3)
          put_value(R4)
          put_value(R10)
          put_value(R11)
        fork(partition)
          put_value(R0)
          put_value(R1)
          put_value(R2)
          put_value(R3)
          put_value(R4)
          put_value(R5)
          put_value(R11)
          put_value(R7)
        copy(R3,R1)
        copy(R4,R2)
        copy(R5,R3)
        copy(R6,R4)
        copy(R8,R5)
        recurse(movej,6)
L1:   default(8)
        build_static(R9,tuple,0)
        define(R8,R9)
        halt
```

Auxiliary Program barrier2

Core PCN

```
barrier2(M1,M2,R)
{? data(M1), data(M2) → R = [] }
```

Abstract Instructions

```
barrier2/3: try(L1)
              data(R0)
              data(R1)
              build_static(R3,tuple,0)
              define(R2,R3)
              halt
L1:         default(3)
```

Auxiliary Program new$1

Core PCN

```
new$1(i,j,a,L,R)
int i,j,a[];
{ ?  data(L), i<j →
        { ||
          swap(i,j,a,L,R1),
          new$2(i,j,R1,R)
        },
     default → R = []
}
```

Abstract Instructions

```
new$1/5: try(L1)
           data(R3)
           data(R0)
           data(R1)
           lt(R0,R1)
           build_def(R5)
           fork(new$2)
             put_value(R0)
             put_value(R1)
             put_value(R5)
             put_value(R4)
           copy(R5,R4)
           recurse(swap,5)
L1:      default(5)
           build_static(R5,tuple,0)
           define(R4,R5)
           halt
```

Auxiliary Program new$2

Core PCN

```
new$2(i,j,L,R)
int i,j;
{ ? data(L) → { ; i := i+1, j := j–1, R = [ ] }}
```

Abstract Instructions

```
new$2/4:  try(L1)
            data(R2)
            put_data(R4,1)
            add(R0,R4,R5)
            copy_mut(R5,R0)
            sub(R1,R4,R6)
            copy_mut(R6,R1)
            build_static(R7,tuple,0)
            define(R3,R7)
            halt
L1:       default(4)
```

Auxiliary Program swap

Core PCN

```
swap(i,j,a,L,R)
int i,j,a[ ];
{? data(L) → { ; tmp = a[i], a[i] := a[j], a[j] := tmp, R = [ ]} }
```

Abstract Instructions

```
swap/5:  try(L1)
           data(R3)
           get_element(R0,R2,R5)       % tmp1 := a[i]
           build_def(R6)               % tmp
           define(R6,R5)               % tmp = tmp1
           get_element(R1,R2,R6)       % tmp2 := a[j]
           put_element(R0,R2,R6)       % a[i] := tmp2
           put_element(R1,R2,R5)       % a[j] := tmp
           build_static(R7,tuple,0)
           define(R4,R7)
           halt
L1:      default(5)
```

Auxiliary Program movej

Core PCN

```
movej(lb,j,s,a,L,R)
int j,a[];
{ ?
     data(L), j >= lb, a[j] > s → { ; j := j − 1, movej(lb,j,s,a,L,R) },
     default → R = [ ]
}
```

Abstract Instructions

```
movej/6: try(L1)
            data(R4)
            data(R0)                        % lb
            data(R2)                        % s
            le(R0,R1)
            get_element(R1,R3,R6)
            lt(R2,R6)
            put_data(R7,1)
            sub(R1,R7,R8)
            copy_mut(R8,R1)
            recurse(movej,6)
L1:       default(6)
            build_static(R6,tuple,0)
            define(R5,R6)
            halt
```

B.2 Partition: Definitional Version I

PCN

```
part(X,Y,L,R)
{ ? Y ?= [N|Ns], X > N → { || L = [N|Ns1], part(X,Ns,Ns1,R) },
    Y ?= [N|Ns], X ≤ N → { || R = [N|Ns1], part(X,Ns,L,Ns1) },
    Y ?= [] → { || L = [], R = [] }
}
```

Core PCN

```
part(X,Y,L,R,Lc,Rc)
{ ? data(Lc), Y ?= [N|Ns], X > N → { || L = [N|Ns1], part(X,Ns,Ns1,R,Lc,Rc) },
    data(Lc), Y ?= [N|Ns], X ≤ N → { || R = [N|Ns1], part(X,Ns,L,Ns1,Lc,Rc) },
    data(Lc), Y ?= [] → { || L = [], R = [], Rc = [] },
    default → Rc = []
}
```

Abstract Instructions

```
part/6:   try(L1)
            data(R4)
            get_tuple(R1,2,R6)              % R6: N, R7: Ns
            data(R0)                        % X
            data(R6)                        % N
            lt(R6,R0)
            build_def(R8)                   % Ns1
            build_static(R9,tuple,2)
            put_value(R6)
            put_value(R8)
            define(R2,R9)                   % L = [N|Ns1]
            copy(R7,R1)
            copy(R8,R2)
            recurse(part,6)
          ... etc ...
```

B.3 Partition: Definitional Version II

An alternative version that makes N mutable and hence performs snapshots.

Core PCN

```
part(X,Y,L,R,Lc,Rc)
int N;
{ ? data(Lc), Y ?= [N|Ns], X > N → { || L = [N|Ns1], part(X,Ns,Ns1,R,Lc,Rc) },
    data(Lc), Y ?= [N|Ns], X ≤ N → { || R = [N|Ns1], part(X,Ns,L,Ns1,Lc,Rc) },
    data(Lc), Y ?= [] → { || L = [], R = [], Rc = [] },
    default → Rc = []
}
```

Abstract Instructions

```
part/6: build_static(R12,int,1)
        try(L1)
            data(R4)                        % data(Lc)
            get_tuple(R1,2,R6)              % R6: N, R7: Ns
            data(R0)                        % X
            data(R6)                        % N
            copy_mut(R6,R12)                % local copy of N
            lt(R12,R0)                      % X > N
            build_static(R13,int,1)
            copy_mut(R12,R13)
            build_def(R8)                   % Ns1
            put_tuple(R9,2)
            put_value(R13)
            put_value(R8)
            define(R2,R9)                   % L = [N|Ns1]
            copy(R7,R1)
            copy(R8,R2)
            execute(part,6)
        ... etc ...
```