Experiences in Simulating a Declarative Multiprocessor

Gary S. H. Tan and Y. M. Teo
Dept. of Information Systems and Computer Science
National University of Singapore
Kent Ridge
Singapore 0511.
tel: (65)-772-6276
e-mail: {gtan, teoym}@iscs.nus.sg

Abstract

There has been extensive research into non-conventional, non-von Neumann parallel computer architectures and declarative programming languages. Dataflow *and* reduction *multiprocessors are examples of such machines which exhibit novel architectures. The Flagship Parallel Reduction Machine is one such multiprocessor using a packet-based graph reduction model of computation to exploit the parallelism inherent in functional languages. A* functional *simulator for the Flagship Machine has been written for studying the functional characteristics of the machine. However, the functional simulator only simulates the actions of the executional units, with no notion of time involved. For performance evaluation, the timing characteristics must be monitored. This paper describes a technique for introducing an event-driven timing scheme into the functional simulator. With the introduction of such a scheme, certain synchronisation issues arise due to the functionality of the simulator. This paper also describes ways of resolving these issues. The architecture is MIMD, based on a set of tightly-coupled processor-store pairs interconnected by a delta network. This is a commonly used architecture, so anyone intending to simulate a similar architecture can draw from the experiences as related in this paper.*

1 Introduction

Declarative languages are programming languages which are based on mathematics. Their use therefore leads to the development of clear, concise and easily maintainable programs. It is no wonder then that proponents of declarative languages argue that they will be the eventual solution to the software crisis faced by the computing world.

Many projects were started to develop architectures to exploit the inherent parallelism in declarative languages; one of these projects was the Flagship project, which was a collaborative university and industrial research project with the aim of designing and building a complete computing system based on a *declarative* style of programming. The collaborators included the University of Manchester, Imperial College of London and International Computers Limited (ICL). It was envisaged that such a machine would be the equivalent of the so-called Japanese *fifth generation* computer [4], with the flexibility and power capable of meeting the computing needs of the 1990s [3].

To aid in studying the functional characteristics of the machine, a *functional* simulator for the Flagship Machine was written. The functional simulator only simulates the actions of the executional units, with no notion of time involved. However, for performance evaluation, the timing characteristics must be monitored. This paper describes a technique for introducing an event-driven timing model into the functional simulator.

Section 2 first describes the Flagship Machine and its functional simulator, while section 3 discusses how the concept of time can be incorporated into a functional simulator. The timing model is described in section 4. With the introduction of such a model, certain synchronisation issues appear due to the functionality of the simulator. These issues, together with ways of resolving them, are discussed in section 5. This section also describes a method of improving the efficiency of the simulator. Section 6 lists out the assumptions made in the implementation of the simulator and finally, section 7 provides the conclusions to this paper.

2 The Flagship Machine

The Flagship Machine executes programs using the technique of *graph reduction*; a program is first compiled into a graph, and the machine transforms this graph through a series of reductions (or rewrites) until no further reductions are possible. The resulting graph is then the result of the program.

Graph reduction is implemented on the Flagship Machine using a Super-Combinator Model of computation [9], which determines which parts of the program graph can be reduced and also specifies the reduction rules.

1080-241X/95 $4.00 © 1995 IEEE

In the machine, the nodes of the graph are represented as packets. Packets which can be reduced immediately are said to be *active*, while those which are not ready for any reductions are said to be *dormant*. Packets waiting for arguments which are being evaluated to be returned are called *suspended*; a suspended packet will become active once the required arguments are evaluated and returned.

2.1 Flagship Machine Architecture

The architecture of the Flagship Machine comprises a set of tightly-coupled processor-store pairs interconnected by a multi-stage *delta* communication network as shown in figure 1. Each processor can access its own local store directly, but access to remote stores is made by message-passing via the communication network.

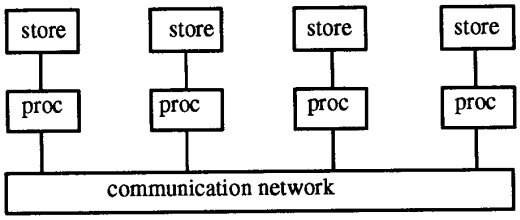

Figure 1: Flagship Machine Architecture

The architecture of each processor is illustrated in block form in figure 2; each processor comprises four processing units, a **Rewrite Unit (RU)**, an **Active Packet Scheduler (APS)**, a **Store Management Unit (SMU)** and a **Network Interface (NIF)**.

A program to be executed is first loaded into the Local Packet Store, the locations being allocated by the Store Management Unit. Addresses of active packets are given to the Active Packet Scheduler, which performs the task of scheduling these packets for rewriting. Rewriting of the active packets is performed by the Rewrite Unit. When the RU is free to perform a rewrite, it signals the APS which selects a packet for rewriting and passes its address to the RU.

During the course of rewriting, further active packets may be generated. These are given to the APS which determines whether to schedule them locally or export them to other less busy processors to ensure that the workload is well distributed. Work to be exported is given to the Network Interface which performs the sending and receiving of messages for the processor. The NIF also collects information on processor activity provided by the network to assist in the load balancing scheme of the machine.

2.2 The Network and Load Balancing

Communication between processors is provided by a high performance *delta* network; figure 3 shows an

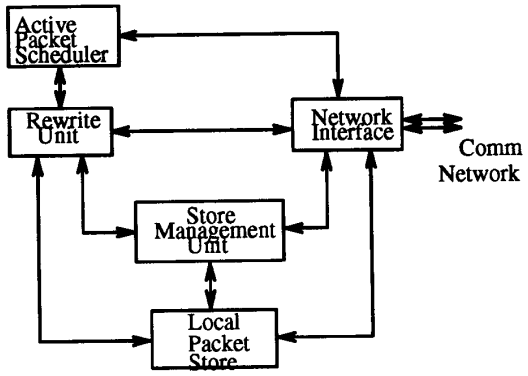

Figure 2: Flagship processor architecture (simplified)

example of an 8x8 delta network formed using 3 identical stages of perfect shuffle interconnections [7] using 2x2 crossbar switching elements.

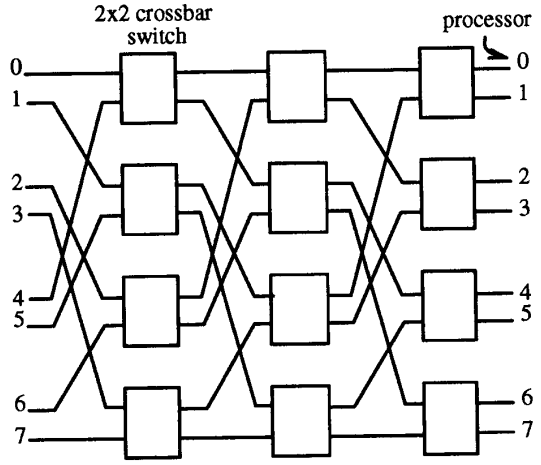

Figure 3: An 8x8 Delta Network for 8 processors

The network also provides support for the load balancing of the machine. For a multiprocessor to operate efficiently work must be distributed evenly, by static or dynamic means, among the processors so that maximum parallelism can be attained. Because of the dynamic nature of graph reduction, Flagship distributes work dynamically to 'map' sub-graphs produced at run-time over the available processing resources. This requires a means of ascertaining the global busyness of the system so that a busy processor can send work to a less busy processor. This is achieved by propagating

activity level information backwards through the communication network [6]. In the Flagship Machine, the activity level of a processor is a measure of the number of active packets awaiting reduction; this is held in a register termed the *local activity level* (LAL) register.

Figure 4 illustrates the backward propagation of this activity level in a 4x4 network connecting 4 processors. In the figure, forward flow is from left to right, and backward flow from right to left. The numbers immediately to the right of the rightmost network switching elements represent the activity levels (local activity levels) of the processors.

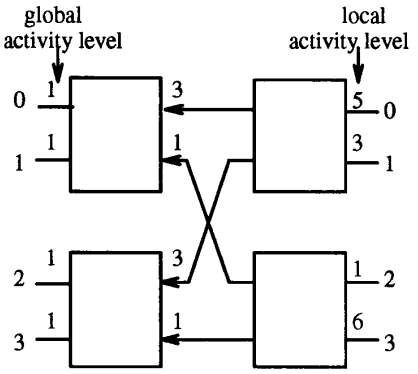

Figure 4: Activity level Propagation

Each output port (right side) of a switching element has a register which holds the activity level information. At each switching element the two activity levels are compared and the minimum is transmitted to the preceding stage. In this way the minimum processor activity level is progressively propagated backwards through the switches until it reaches the processors; here it is stored in the *global activity level* (GAL) register for each processor.

Each processor therefore maintains two registers to monitor activity levels, the LAL register giving the local activity, and the GAL register giving the minimum activity of the machine. By checking the LAL against the GAL each processor can determine whether other processors are less busy than itself and whether to export work.

2.3 The Flagship Functional Simulator

As part of the Flagship project, a functional simulator for the Flagship Machine was developed; this is a program, written in the high-level programming language C, which models the operation of the executional units of the Flagship Machine.

The simulation of program execution on the Flagship Machine is achieved by first loading program packets into the packet store and passing addresses of active packets to the Active Packet Scheduler (APS). The simulated execution of the program then proceeds by executing a sequence constituting a *lock-step* reduction cycle of the processors where the actions of each processor is simulated in turn, followed by the simulation of the communication network; this sequence is repeated until no active packets remain in the machine, which signifies the end of the program execution.

In the reduction cycle for each processor, a check is first made for any network messages which may have arrived; these would then be read by the NIF unit and processed, active packet messages for example are stored and their addresses passed to the APS. After the message reading is complete, the processor signals to the APS for the address of an active packet to be rewritten and performs its rewrite.

3 Introducing Time into a Functional Simulator

The Flagship functional simulator simulates only the *actions* of the executional units regardless of the time taken to perform their actions. A detailed investigation into the executional performance of the machine inevitably requires the introduction of some concept of timing.

Furthermore, the functional simulator executed in a lock-step mode by executing a reduction cycle for each processing unit followed by the simulation of the network, repeating until all rewrites had been performed. Performing rewrites in a lock-step simulation constrains rewrites to be of the same length and therefore take identical execution time. In practice, rewrites are not generally of the same length and take different times to execute. An inaccuracy arises because of this lock-step simulation, where the processor workload is re-calculated at the end of each rewrite and, if changed, propagated backwards through the network. In a real machine however, processors run asynchronously and workload information will also propagate asynchronously, and not synchronously as assumed in the lock-step simulation. The introduction of time will remove the synchronous nature of the simulator.

One technique which can be used to introduce time is the *event-driven* timing scheme, used previously in gate-level simulators [1] to model the behaviour of logic circuit networks. In this scheme, the passage of time in the network is modelled using an array indexed by time-quanta, termed a *time-wheel*. State changes in a network are represented as events, scheduled on this time-wheel at the time at which they are to occur. When simulated time advances to match that corresponding to a state change in the time-wheel, the effects of this state change may then be evaluated and again scheduled to take effect at the appropriate future time. For example, when an inverter's input signal

changes at time t_0, the new output state should occur at time $t_0 + D$, where **D** is the propagation delay; an event will therefore be scheduled at time $t_0 + D$ in the time-wheel.

The logic networks are usually represented by sets of tables giving descriptions of individual logic elements and their interconnections. For each element, a function is written whose input parameters represent the inputs of the element and which returns a value representing the output of the element. The effects of an element's state change can be evaluated by accessing its network table yielding the connected (fan-out) elements. Functions corresponding to these connected elements are used to evaluate subsequent state changes, which are then scheduled at the appropriate future time.

The timed simulation proceeds by stepping through the time-wheel evaluating the events in each time-slot. Events scheduled in the same time-slot are chained by a linked-list as shown in figure 5. Figures 6(b) and (c) show how the time-wheel is traversed for the simulation of the logic circuit shown in figure 6(a). At time **t**, the output of element **A** is changed to logic level 1, event 1 is therefore scheduled as shown in figure 6(b). Since **A** fans out to **B** and **C** both outputs of **B** and **C** will change after a propagation delay of **dt** (assuming identical propagation delays). Hence servicing event 1 will cause events 2 and 3 to be scheduled at time **t** + **dt** on the time-wheel as two chained events as shown in figure 6(c).

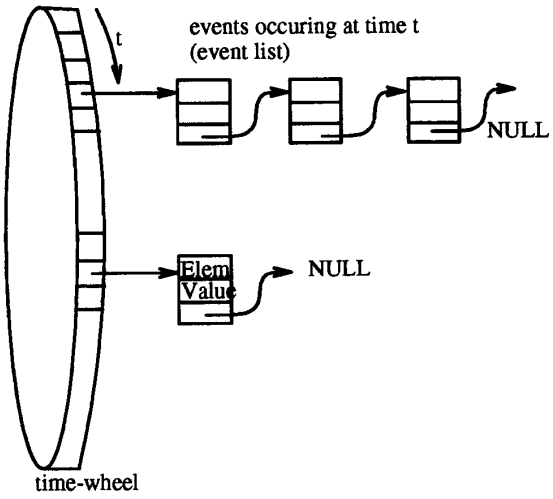

Figure 5: The time-wheel and event lists

For the Flagship simulator, events can be used to represent the *actions* of the executional units, e.g. a store access, reading of a network message etc. A timing scheme for such simulations must model *concurrency* and *contention* accurately. Take as a simple example two independent executional units **A** and **B** sharing a resource **R** where each unit can operate independently of the other, an example of concurrency. Only one unit can access the shared resource at any one time however; when one unit, say **A**, tries to access **R** while **B** is in the process of accessing it, contention is said to have arisen. In this circumstance, **A** will have to wait until **R** is free again before the access can be made.

The event-driven scheme allows concurrency and contention to be modelled accurately; concurrency is modelled implicitly by the use of the time-wheel to store the events while contention is modelled simply by re-scheduling the events when resources are busy. Such a scheme cannot be superimposed on an existing functional simulator however, but necessitates a complete re-development. The various executional actions will have to be partitioned into atomic operations and represented as events. In the limit, individual ma-

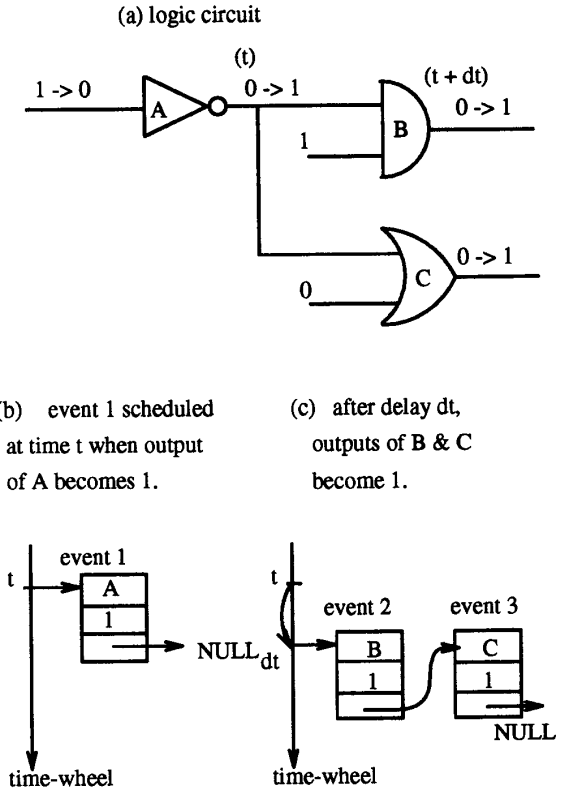

Figure 6: Simulation of a logic circuit

chine instructions will have to be represented as events scheduled one after the other, and the functionality of the simulation will be lost. Furthermore, the simulator will run slow due to the overheads of scheduling and processing of the atomic events.

The scheme proposed in this paper is therefore event-driven only at the functional level of the reduction cycle of the functional simulator. This removes the need for the lock-step nature of the reduction cycle for the simulator, since it is now event-driven, while maintaining the functionality of the simulator. It also models concurrency and enables the processors to run asynchronously. Furthermore, there is no need for a complete re-development of the simulator; its structure remains very much the same, only that part of the code which simulates the lock-step execution is replaced by the time-wheel concept. The next section describes how a timing model based on this 'modified' event-driven timing scheme is applied to the functional simulator.

4 A Timing Model for the Flagship Simulator

The implementation of an event-driven timing scheme in conjunction with a functional simulator requires the partitioning of the actions of the executional units of the machine into events. Since the simulation is to be driven at the level of the reduction cycle, it would *prima facie* be logical to represent the reduction cycle of a processor as an event. Care must be taken to ensure the atomicity of events as far as possible however, so that events interact correctly. As outlined in section 2.3, a reduction cycle comprises the reading of network messages followed by the rewriting phase. It would therefore be better to reduce the granularity of the reduction cycle event by dividing it further into smaller events, where each reading of a single network message will constitute a separate event, a Read Message event, and the rewriting of a packet will constitute another event, a Rewrite event. If this was not done, the reduction cycle event will be processed as a single block of functional code, and messages arriving at a processor while it was performing the reading of messages will not be read as their events will not be processed until the reduction cycle event is processed.

To simulate the actions of the Network Interface, two events are used, one for transmitting (Transmit Message event) and another for receiving (Receive Message event) a network message. An event is also needed to represent the backward propagation of activity level through the network to update the global activity levels of the processors (Update Activity Level event).

Events are represented by a record structure as shown in figure 7. Events occurring in the same time interval are chained in a linked list, through the use of the 'Chain' field of the record.

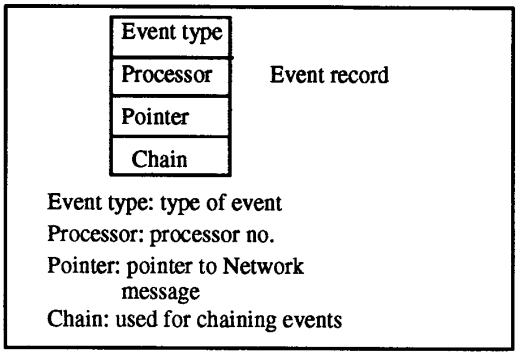

Figure 7: Structure of an Event record

How the above five events, i.e. Rewrite, Read Message, Transmit Message, Receive Message and Update Activity Level event, simulate the execution of the machine is illustrated by example in figure 8, which shows the events scheduled for two processors on the time-wheel during a passage of timed-simulation. The time-wheel is organised as a 1-dimensional array, indexed by the **time-interval** or **time-step** of the simulation. A single time-wheel is used for scheduling events generated by all the processors, thus facilitating the modelling of concurrency (in figure 8, the events are shown separated by processor for clarity). Each processor maintains a variable which represents its *timer*, to keep track of its own execution time as the functional code of each event is executed.

The actions of the executional units of the machine are simulated by functional blocks of C code; the actions taken during the evaluation of an event are therefore calls to the appropriate functional block. As the function code for a processor is executed, its *timer* keeps track of the machine time corresponding to the code executed in terms of the machine cycle time, i.e. processor clock time (see assumption 4 in section 6). This *timer* specifies the future time at which a consequent event should be scheduled.

For processor 0, when a Read Message event at the start of a reduction cycle is evaluated at time $\mathbf{t_1}$, the function code which reads a network message from the Network Interface buffer is executed, followed by the scheduling of another Read Message event at the appropriate future time ($\mathbf{t_2}$). If no messages are present, then a Rewrite event can be scheduled ($\mathbf{t_3}$). When the Rewrite event is serviced, the code which obtains an active packet address and the subsequent rewriting of the packet is executed. During the course of the rewrite, packets may be created and scheduled by the Active Packet Scheduler for running; this increases the activity level of the processor, which must be propa-

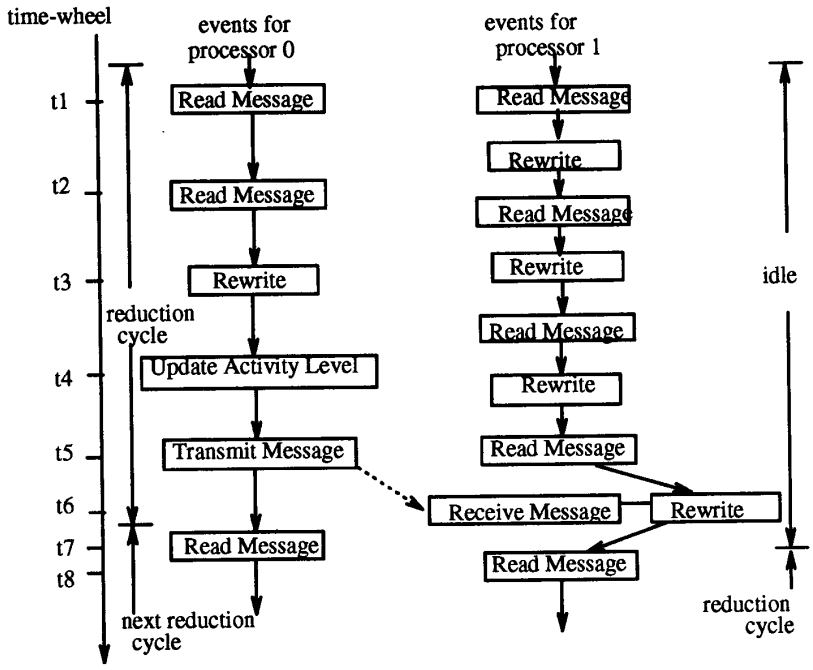

Figure 8: Events scheduled during passage of time through time-wheel

gated to the other processors by scheduling an Update Activity Level event (t_4). A created packet may also be exported in which case a Transmit Message event is scheduled to simulate the transmission of the packet message (t_5). When this event is evaluated, the code for encapsulation and sending of a message is executed. A Receive Message event is then scheduled for the destination processor 1 at the time t_6 when the message should arrive there. At the end of the code execution, a Read Message event is again scheduled (at time t_7) to start off the next reduction cycle.

For processor 1, at time t_1, there are no messages or active packets for rewriting and the processor is effectively idle. When a Read Message is evaluated, on finding no messages, it schedules a Rewrite event. When this Rewrite event is evaluated, a check is made to see if there are any packets for rewriting; since there are no packets available, it schedules a Read Message. This process of repeated scheduling of Read Message and Rewrite events is continued until a Receive Message event is evaluated (t_6), which places the message into the Network Interface buffer. The next Read Message event will then start a reduction cycle by reading this message.

4.1 Simulation Algorithm

Initially, program packets belonging to the user programs are loaded into the packet store and addresses of active packets are passed to the Active Packet Scheduler. A Rewrite event is then scheduled for each processor at the beginning of the time-wheel, i.e. at the first time-step, to initiate the simulation.

Simulation commences at the first time-step and steps through the time-wheel. At each time-step, a check is made to see if the time-slot contains a list of events for *servicing*. If no list is present, simulation progresses to the next time-step; if a list is present, its events are evaluated or *serviced* in list sequence until exhausted. Processing an event may lead to the creation of other events which are scheduled to take effect at appropriate later time-steps. The *driver* module of the simulator performs the function of stepping through the time-wheel, and records the number of time-wheel cycles; its algorithm is shown in figure 9.

The *driver* is a *while* loop which increments the time-step of the time-wheel, servicing events present in each time-step. Once the event list corresponding to the last time-step of the time-wheel has been serviced, the driver 'wraps' round to the start of the wheel again and increments a variable, COUNTER, which records the number of full cycles of the wheel traversed. Con-

```
COUNTER = 0; slot = 0;
/* driver loop */
while (Events still present in time-wheel)
{  /* search for next event in time-wheel */
  while (t_wheel[slot] == NULL)
  {
     slot++;   /* go to next time-step */
     if (slot == TIMEWHEELSIZE)
     { /* increment timewheel counter */
        COUNTER++;
       /* jump to start of time-wheel */
        slot = 0;
     }
  }
  while (t_wheel[slot] != NULL)
  { /* get event */
    EventPtr = t_wheel[slot];
    /* next event in current time-step */
    t_wheel[slot] = t_wheel[slot]->Chain;
    /* Service the event */
    ServiceEvent(EventPtr);
  }
}   /* end of time-wheel driver loop */
```

Figure 9: Time-wheel driver module

trol is passed out of the loop only when there are no scheduled events present in the time-wheel, indicating that all programs have terminated.

4.2 Servicing Events

The simulator driver services events by calling a function **ServiceEvent**, which takes the appropriate actions. In summary, the actions taken and functional code executed upon encountering the respective events are:

Read Message event:

```
if (Network messages present)
{
  Read Network Message from NIF
  Schedule next Read Message event
}
else  Schedule Rewrite event
```

Rewrite event:

```
if (active packet address present)
{
  Get Active Packet Address
  Process Active Packet (perform rewrite)
}
if (processors > 1) Schedule Read Message
else  Schedule next Rewrite event
```

Transmit Message event:

```
Propagate message through Network
(however, message is not put in NIF buffer
 of destination (dest) processor)
Schedule Receive Message event for dest
at time (current time + Network latency)
```

Receive Message event:

```
Put Network message into NIF buffer of dest
```

Update Activity Level event:

```
Call Propagate Activity Level function
(This event is called whenever the activity
 of the processor is changed)
```

4.3 Scheduling Events

The scheduling of events on the time-wheel is performed by invoking the **ScheduleEvent** function. The necessary information, viz. type of event, processor, pointer and offset time t from the current time-step, is passed to the function.

The function takes the time t (in terms of machine cycles) and calculates the offset from the current time-step in terms of time-step of the time-wheel using the formulae:

$$offset = integer(\frac{t}{t_s} + 0.5)$$

where t_s is the time-step of the time-wheel.

The offset is thus rounded to the nearest time-step and the event is then scheduled on the time-wheel. In the simulator described, the size of the time-wheel is chosen such that the time-span is longer than the maximum possible offset, i.e. no event can take longer than the time-span of the time-wheel. Thus, an event generated during the servicing of an event can be scheduled on the time-wheel without the possibility of overflow. The scheduling is thus straightforward; if **n** is the size of the time-wheel, and **c** the current time-step, the algorithm would be:

```
if (offset < (n - c))
then
    schedule at (c + offset)
   /* scheduling forward on the wheel */
else
    schedule at (c + offset - n)
   /* wrap around the wheel */
```

At any time during the simulation, the absolute execution time will be given by the formulae:

$$(COUNTER * n + c) * timestep + timer$$

where *timer* is the number of machine cycles executed in the current time-step.

5 Synchronisation & Efficiency Issues

5.1 Synchronisation of Events

The level of granularity of an event must be chosen so that it is as atomic as possible, so that its execution can be performed correctly and independently without being affected by any other events. For example, consider an event A of large granularity which at time 5 units accesses information stored in a register, and an event B, scheduled 2 time units after event A, which updates this register. Due to the functionality of event A, event B cannot be serviced until event A has finished; this causes event A to access an un-updated register and hence leads to inaccuracy in the simulation.

Reducing the granularity of events for the simulator was for example achieved by splitting the reduction cycle into a series of Read Message events and a Rewrite event. This was possible because the reading of a message and the rewriting of an active packet were written as two separate functions in the original functional simulator. There are cases however where the reduction of the granularity of an event is not possible; an example is the Rewrite event, where the execution of the Super-Combinator code is written as a single C function in the simulator. During the execution of the Super-Combinator code, packets may be created and given to the APS, which must decide whether to export or keep them based on the local and global activity levels. However, because of the functionality of the Super-Combinator code execution, checking of the global activity level is made in the current time-step of the Rewrite event and does not take into account the time elapsed while performing the rewrite, implying that the global activity level may have not been updated yet at the time of the check, as illustrated in figure 10.

In the figure, the horizontal line to the right of each Rewrite event represents the duration of the functional Rewrite event. At time $\mathbf{t_1}$, Rewrite event (a) is started for processor 0 and the functional code for the rewrite is executed; five time-steps into the rewrite, a packet is created and put into the queue (shown as APScall in the figure). Accordingly, an Update Activity Level event is scheduled on the time-wheel to signify that at time $\mathbf{t_3}$, five time-steps later, a packet address is added onto the queue and the change in activity level must be broadcast to the other processors. At time $\mathbf{t_2}$, however, processor 1 starts Rewrite event (b) and 3 time-steps into the rewrite, a packet is generated and given to the APS which at this stage makes a check on the global activity level; this check is made at time $\mathbf{t_2}$, since the simulation is at this time-step servicing Rewrite event (b), and not at the correct time of $\mathbf{t_2 + 3}$. Event (c) is therefore not serviced yet and thus the un-updated global activity will be read.

The approach to be adopted here is therefore one of synchronisation of the actions. If an action is dependent on another, then clearly this can only be correct if they have been synchronised in simulation time. The calling of the APS must therefore be represented as a new event (APSput event) to be scheduled at the correct time so that when the global activity level is checked, the correct information will be given.

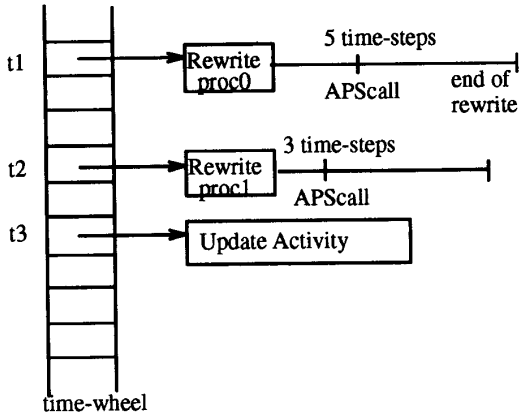

Figure 10: Checking un-updated activity level at $\mathbf{t_2}$

Now that the APS call is separated from the main rewrite function, future events generated during the rest of the Rewrite event, e.g. the sending of a message, will not have taken into account the time taken to execute the APS function code. An event which represents the sending of a network message (NIFput event) is therefore also necessary to allow the sending of a message to occur at the appropriate time.

The scheduling of the first delayed event of the rewrite is straightforward; the scheduling of subsequent delayed events on the time-wheel is not, since the times at which these events are to be scheduled depend on the execution times of previous delayed APSput or NIFput events. What is required is an additional list, a **PendingEventsList**, to hold these events together with their function invocation times as the servicing of the Rewrite event progresses. After the Rewrite event is serviced, the events of this list can then be scheduled and processed one at a time. Thus, during a rewrite, each time an APS call or Network Interface function is encountered, instead of immediately executing the code, their events are appended to the PendingEventsList, e.g. figure 11(a) shows 3 delayed events generated during the servicing of a Rewrite event, the times shown are those at which the function calls are encountered during the servicing of the event. At the end of the functional rewrite, a Read Message event (or Rewrite event if simulating only 1 processor) is appended as the last event to the PendingEventsList so that the next reduction cycle

can be scheduled.

The scheduling and execution of these delayed events on the list can now proceed; the first step is to time the occurrence of each event as an offset from the previous event to simplify the scheduling (figure 11(b)). Next, the first event on the list (APSput in the example) is extracted and scheduled at a time 10 units later than the current time-step. When simulation has progressed to that time-step, the APSput event is processed. Once this event is completed, the next event on the list can be scheduled at time offset + time taken to process previous event from this time-step. In this example, it is assumed that each APSput event takes 5 units and each NIFput event 10 units. Therefore the second APSput event will be scheduled at time (15 + 5 = 20) units from time-step 10. This whole process is repeated until all events on the list have been processed. Figure 12 shows how each event is scheduled one after another. In this way, active packets are scheduled, and messages are also sent off, at the right times.

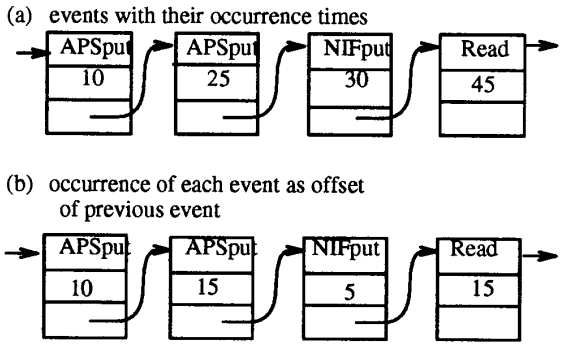

Figure 11: Events on the PendingEventsList

5.2 Busy-poll vs. Interrupt Simulation

This section describes a technique to increase the efficiency of running the simulator in terms of saving memory usage.

In the implementation of the timing model described above, the driving of events on the time-wheel is based on a polling concept, i.e. when a processor is idle, Read Message and Rewrite events are alternately scheduled for that processor until a message arrives for processing (see processor 1 in figure 8). This is wasteful since for each idle processor, events have to be scheduled in each time-step until a message arrives to be read.

A more efficient method would be to use an interrupt-driven concept; here, when a processor becomes idle, no more events are scheduled for it and simulation re-commences only when it becomes busy again due to the arrival of a network message. To implement this, an extra Idle event is needed. When a processor becomes idle, it schedules an Idle event which sets a **Busy** flag to FALSE. When a network message arrives at this processor, a Receive Message event will be scheduled. Servicing this event will reactivate the idle processor by scheduling a Read Message event and setting the **Busy** flag to TRUE. Using this scheme reduces the number of events generated in the simulation and also allows the monitoring of processor idle times, given by the times between Idle and Read Message events.

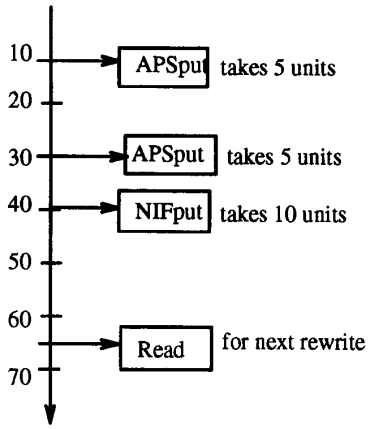

Figure 12: Scheduling events of the PendingEventsList

6 Assumptions of the Simulation

Several assumptions were made in the implementation of the Flagship Machine timed-simulator described in this paper. These are:

1. A uniprocessor architecture is assumed for each processing element, i.e. the actions of the Rewrite Unit, Active Packet Scheduler, Network Interface and Store Management Unit are all performed by one microprocessor.

2. Processors are assumed to have a machine instruction cycle time of 100 ns.

3. The network transmits information at a rate of 1 byte/100 ns. (i.e. 10 Mbytes per sec).

4. Timing the machine execution was achieved by translating the C code of the simulator into assembly code for a RISC architecture [5] instruction set, e.g. the AMD29300 chip family [2], and then counting the instructions. This yields greater accuracy than just counting the C language statements, which was reported in [10].

This assembly code is assumed to be stored as microcode held in a separate ROM in the processor.

7 Conclusion and Further Work

In this paper, we have described a simulation tool developed for investigating the workings of a declarative multiprocessor. This tool was evolved from building a timing model on top of an existing functional simulator. However, with the introduction of this timing scheme, certain synchronisation issues arose due to the functionality of the simulator. The paper also described ways of resolving these issues and also described a way to improve the efficiency of the simulator tool.

Although the timing scheme introduced above enables concurrency to be modelled accurately, the overall accuracy of the simulation depends on the choice of the time-step of the time-wheel. It also depends on how closely the processing actions carried out in the simulator resemble those of the real machine. The actions may also differ in an implementation of the machine; they may be re-arranged for performance, or may contain additional error trapping code for diagnostic purposes.

Related to this is the assignment of timing to the various actions to be performed in the firmware of the real machine. There is, however, good confidence in the assignment of timing, since it is translated from assembly code. All the various sequences of operations, e.g. sending of network messages, rewriting of packets etc., have been properly timed, tested out and verified to be correct. Furthermore, the test programs executed on the simulator produce the results expected, showing that the simulator is functionally correct.

This simulation tool has already been successfully developed and implemented, and has provided the author with a useful research tool for providing insights into the workings of a declarative multiprocessor. Investigations using this simulation tool have already been carried out [8], and are still ongoing, on the multiprogramming, load balancing and scheduling aspects of the Flagship Machine.

References

[1] M.d'Abreu, "Gate-Level Simulation", IEEE Design and Test, pg. 63–71, Dec 1985.

[2] Am29300 Family Handbook, High Performance 32-bit Building Block, Advanced Micro Devices Incorporated, Apr 1985.

[3] R.Dettmer, "Flagship – A fifth-generation machine", Electronics and Power, Mar 1986.

[4] Murakami,K et al. 1985, "Research on Parallel Machine Architecture for Fifth-Generation Computer Systems", IEEE Computer, Vol. 18, No. 6, pg. 76–92, Jun 1985.

[5] D.Patterson, "Reduced Instruction Set Computers", Communications of the ACM, Vol. 28, No. 1, pg. 8–21, Jan 1985.

[6] J.Sargeant, "Load balancing, Locality and Parallelism Control in Fine-Grain Parallel Machines", Internal Report, Flagship Project, Department of Computer Science, University of Manchester, Apr 1987.

[7] H.Stone, "Parallel Processing with the Perfect Shuffle", IEEE Transactions on Computers, Vol. C-20, No. 2, pg. 153–161, Feb 1971.

[8] G.S.H.Tan, "An Investigation into Multiprogramming Aspects of a Declarative Multiprocessor", Ph.D Thesis, Dept. of Computer Science, Univ. of Manchester, U.K., Nov 1992.

[9] P.Watson, "Evaluating Functional Programs on the Flagship Machine", Proceedings of Conference on Functional Programming Languages and Comp. Architecture. Portland, Oregon, edited by G.Kahn, Springer-Verlag, LNCS, Vol 274, pg.80-97, Sep. 1987.

[10] P.S.Wong, "Evaluation of a Proposed Architecture of a Rewrite-rule Machine", M.Sc Thesis, Department of Computer Science, University of Manchester, Jan 1987.