A PERFORMANCE-ORIENTED APPROACH TO THE DESIGN OF DISTRIBUTED SYSTEMS

Rajive Bagrodia

November 1988
CSD-880090

A Performance-Oriented Approach to the Design of Distributed Systems[1]

Rajive L. Bagrodia

3531 Boelter Hall
Computer Science Department
University of California at Los Angeles
Los Angeles, CA 90024.
Tel: (213) 825-4943

e-mail:rajive@cs.ucla.edu

[1]This research was supported by a grant from NSF

Abstract

Efficient distributed systems may be designed by incorporating performance evaluation as an integral part of the software development cycle. This paper proposes a system design strategy in which performance models are viewed as continuously evolving implementations of system specifications, where the specifications include desired performance characteristics for the system. In this strategy, systems at an intermediate level of design consist of operational modules and partially implemented (simulation) modules. The goal is to ensure that the partially implemented design provides the desired performance. The paper presents a methodology to develop partially implemented designs, and describes algorithms to execute the design in a distributed environment.

1 Introduction

A number of distributed systems have stringent constraints on their performance. Among other parameters, the constraints may deal with response time, reliability or throughput. It is advantageous and less expensive to ensure that a given design meets its performance constraints *before the design is implemented.* We present an approach that uses simulations to integrate performance evaluation with the top-down design of integrated distributed systems. An integrated system emphasises the interaction among the software and hardware components and may also involve human interaction.

As integrated systems become more complex, they become harder to evaluate using analytical techniques. Example systems include those where the cooperation between processors is described by means of complex algorithms (e.g. network access algorithms in distributed systems, resource contention/allocation algorithms, and memory access algorithms in shared-memory systems) as well as distributed operating systems and applications. Even when analytical models can be constructed, lack of sufficient data generally precludes doing a comprehensive study of the performance issues. The usual alternative to analyzing the performance of a system is to use simulations. For large systems, simulation techniques prove to be prohibitively expensive mainly due to the effort required to construct and maintain separate simulation models of proposed or exiting systems. Any modification in the system design necessitates a corresponding change in the simulation model. Model validation becomes increasingly difficult leading to doubtful consistency between the model and the actual system being developed.

The key issue addressed in this paper is to identify an appropriate abstraction for performance models of distributed systems that avoids the problems mentioned above. We suggest the use of Partially Implemented Performance Specification (or PIPS); this abstraction views performance models as continuously evolving implementation of system specifications. The next section presents our approach. Section 3 describes related research projects. Section 4 gives a brief overview of the language used to write PIPS programs. Section 5 presents the centralized and distributed algorithms to execute PIPS programs. Section 6 describes some limitations of our approach and discusses implementation issues.

2 Approach

A distributed system consists of a collection of *communicating* sequential processes that execute concurrently on a number of processors linked by an arbitrary interconnection network. A processor may interleave the execution of a number of processes. Processes communicate exclusively via messages and a process may not directly modify the data-space of another process. Each component of the distributed system is classified either as a PP or a PE. Although the classification of components as a PE or PP is arbitrary, in general a PE is realized primarily in hardware, whereas a PP represents a program module. For instance, consider a distributed inventory management system implemented on a network of workstations. Each workstation and the network is considered to be a PE; each modules in the inventory manager program may be considered a PP. A PP is reactive, in that its actions are initiated on receipt of a message. In the absence of any messages, the PP is quiescient. The behaviour of a generic PP may be expressed as follows:

```
while (not terminated)
{   (wait to) receive the next message;
    perform local computation to process the message;
}
```

The local computation performed by a PP is termed a computation step; it includes the creation of messages sent to other PPs in the system. A PP is associated with a PE. Execution of a computation step by a PP may be viewed as "using the PE on which the PP is resident" and takes a finite amount of time. We assume that each PE is associated with a *clock* which measures the execution-time of a computation step. The computation steps of two PPs associated with the same PE cannot be executed in parallel.

We now consider the message-based simulation model of a distributed system. In the model, PPs and PEs are represented by logical processes, henceforth referred to as LPs. An LP that represents a PE is sometimes refered to as a *server* LP. Interactions among PPs are represented by an exchange of messages among the corresponding LPs. The computation step executed by a PP on a PE is modeled by the execution of a simulation step by a server LP that models the PE. In executing a simulation step, an LP waits for a certain amount of simulation time (equal to the estimated time required by the physical system to process the message) to elapse. It does this by scheduling a *time-out* message to itself at an appropriate time in the future. In addition to LPs that model PPs and PEs, a model also contains *instrumentation* and *housekeeping* LPs to drive the model and collect and print statistics. The behaviour of a generic LP may be expressed as follows:

```
while (not terminated)
{   (wait to) receive the next message;
    perform local computation to process the message
    OR simulate the processing of the message;
}
```

An LP simulates the processing of a message by executing a simulation step.

An LP either executes a computation step or a simulation step. If each LP executed only computation steps, the model is an operational system and each LP is really a PP (or PE); an LP that executes simulation steps is an abstraction of the corresponding PP. The basic contribution of this research is in presenting an approach to iteratively transform a model (where the abstractions are expressed as simulation steps of LPs) into an operational system; at every intermediate step, the expected performance of the system is monitored to ensure that it lies within the desired range.

Figure 1 displays the system development strategy. Given the performance specification for a proposed system and an initial system design, an analyst develops a simulation model of the software. The simulation model is initially constructed at a very coarse, logical level and contains estimates of the execution times and resource requirements for various modules in a program. It also contains local invariants that specify correct execution of each module. A model of the hardware environment of the proposed system is built using parameterized library modules. Each software module is explicitly mapped to some hardware module. The simulation model is executed to collect performance statistics. If system performance is unacceptable, the analyst modifies the design by appropriate changes in either the software or the hardware model. The software and hardware components are modeled separately so that changes may be made independently. It is often desirable to study the impact of reconfiguring a model on its performance. Primitives are provided to allow the mapping of modules to be changed with minimal changes to the model.

The modified design is executed as a simulation and the process is repeated until system performance is acceptable. At this stage, the model is refined by introducing more detail in its implementation. This may be done in a variety of ways: subdividing a module into more modules, elaborating

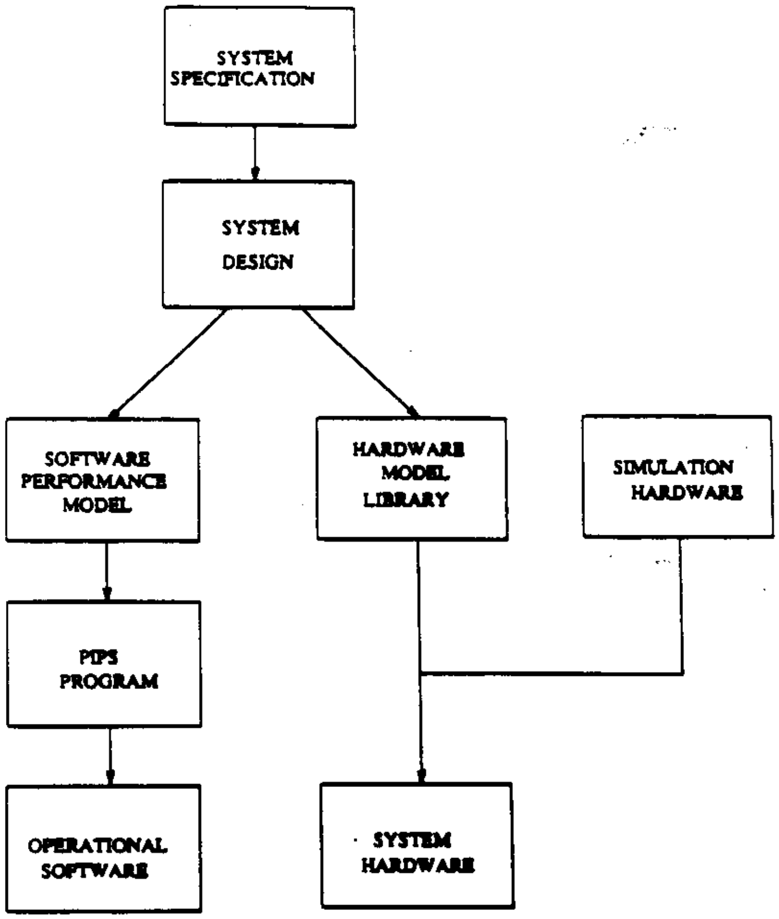

Figure 1: Software Development from a Performance Perspective

a simulation step into a computation step, elaborating the required processing for a message, or replacing a model of some hardware unit by actual hardware. The above process of refinement and elaboration implies that at some intermediate stage, the model may contain some modules that are (at least partly) operational, and the computation steps of these modules must be included in determining the overall performance characteristics of the evolving system. The intermediate form of the model is refered to as a PIPS program. In this manner the model is refined iteratively, while its performance is continuously monitored until the (software) model has been transformed into operational code. Note that the PIPS program may be executed on a distributed architecture. (For instance, the analyst may decide to replace an entity that models communication between remote processes by an operational network that links multiple computers.) It follows that the execution environment for PIPS programs must be capable of measuring computation steps and executing simulations in a distributed environment.

What advantages may be derived from the above approach? The primary advantage is that a performance model of the system does not have to be designed and maintained separately. *The evolving design is its own* *model.* The iterative refinement of the model is really the application of a top-down design methodology to the software design process with the final refinements producing the distributed software from the performance model. This can result in tremendous savings in terms of manpower and resources while at the same time ensuring the consistency between the software and its model. Model validation is an important concern when the model is developed separately from the system. Modifications in system specifications may require the model to be significantly reworked and revalidated. In the PIPS approach to performance evaluation, modifications in system design are directly incorporated in the model.

In addition to performance evaluation, the model is also useful in testing and debugging. In order to facilitate debugging, an assertion-based trace facility is provided. This facility may be used to ensure that a given assertion(s) is obeyed locally by a module at discrete points in its execution. Finally, our approach encourages reusability. A library facility is supported which provides hardware models, statistics collection and report generation entities as well as a collection of entities for services like termination detection and process allocation.

When is the integrated approach to software design useful? In order to include both operational and simulation modules in measuring performance, the simulation 'engine' (the architecture on which the model is executed) must be similar to the proposed hardware for the modeled system. If the actual hardware is available, our approach will yield maximum benefit. However, even if some characteristic hardware parameters like clock speed, memory and disk access times, instruction cycle time ... are scalable with respect to the simulation engine, our approach will be of significant help in predicting system performance.

3 Related Work

Performance modeling in the design stage has frequently been used in the design of hardware systems. The idea of integrating simulation models with system design was proposed by Zurcher and Randell [26] to develop a methodology for the design of computer systems and also explored by Parnas[18]. Sanguenetti [14] describes a technique for performance prediction by integrating simulation and software system design using PPML[20], a system modeling language.

Other researchers have suggested methodologies and tools to construct performance models *prior* to developing the system. Chandy et al [6] describes a top-down methodology for evaluating the performance of computer/communication systems in the early design stages. Smith & Browne [24] proposed a design methodology to integrate the software development process with performance evaluation using software execution graphs. Other design methodologies like the Design, Realization, Evaluation and Modeling System designed by Riddle et al [21] and the Software Tool for Evaluating System Design designed by Baker et al [4] have emphasised the use of simulation to study the behaviour of software systems. Estrin et al[9] suggest a system design methodology to study performance aspects in multiple domains: a control flow graph is constructed to analyze safety and liveness properties. Also, a data flow graph may be used in conjunction with the control flow graph to study performance characteristics by interactive simulation. Roman[22] describes a specification language called CSPS, to study correctness and performance characteristics of distributed systems. CSPS is an extension of CSP[12] and uses verification techniques that have been developed for CSP programs to prove properties of the CSPS programs. The above work emphasizes throughput and reliability as opposed to completion time as the important metric of system performance.

As an alternative to performance prediction, monitoring techniques have been suggested to *measure* system performance. A body of work exists on monitoring techniques to study the performance of operational systems. The main emphasis of performance monitoring is to extract information about the execution of a system without significantly affecting its behaviour or performance. The monitoring activity is usually done transparently and requires minimum input from the program being monitored. Miller [16] describes a methodology for non-intrusive measurement of communications in a distributed system. Joyce et al [15] describe a system to display and analyze the information collected during the monitoring of a distributed system. The two novel features of this work are its use of event abstraction and graphical animation to present monitored information; and its use of replay facilities to control non-determinism in the execution of a distributed program. Whereas the above projects have made a significant contribution to our understanding of the factors that impact on the performance of distributed computations, they are primarily used as *prescriptive* rather than *preventive* or *predictive* aids. By integrating performance evaluation as an integral aspect of the system development cycle, we hope to solve performance problems in the early stages of system development. In the next section, we describe language primitives for integrated software development.

4 Language Primitives

This section gives a brief description of language primitives that can be used to write distributed programs and their simulation models. A complete description of the primitives including justifications for their selection may be found in [3,2]. The primitives use the notion of **entity** to model objects, **message** to model their interactions and **clock** to schedule events. They may be implemented in any sequential programming language (e.g. FORTRAN, PASCAL, C etc) to develop a language for integrated design. The discussion in this section is developed in a language independent manner and uses Pascal-like syntax and semantics.

Entities are the basic building-blocks of a distributed program. An entity is an independent, sequential program module which is used to model processes. An entity may create other entities; terminate itself; and send (receive) messages to (from) other entities. An **entity** type is used to define objects of a given type. Various instances of an entity type may be created dynamically to represent the many objects of a given type. An entity instance is created by executing a **let** statement. Hereafter, we shall use the term entity to mean an instance of an entity type. The local variables of an entity cannot be accessed by other entities. Entities communicate via messages. On being created, an entity is assigned a unique identifier, which is bound to the entity for its lifetime. In order for an entity to send a message to another, it must have access to the latter entity's identifier. A message is viewed as a specific instance of a message type. A message type consists of a name and a list of message parameters. An entity sends a message to another by executing an **invoke** statement. Message sending is non-blocking: messages sent by an entity are deposited in the receiving entity's message buffer; the sending entity is not delayed. An entity accepts messages from its buffer by executing a **wait** statement. If a desired message is not present in the buffer, the entity waits for the message. The wait may be indefinite or specify a time-out interval. In the first case, the entity ceases to wait only when the desired message is received by it. In the latter case, if the desired message is not received by the entity within the specified time period, the entity will eventually time out and thus cease to wait. On ceasing to wait, an entity proceeds to the next statement in its code. Initially, every program written using our language fragment consists of a single entity called **main** executing on one processor. The purpose of entity **main** is to initiate the execution of the program. The particular programming language in which these constructs are to be embedded is called the host language.

We illustrate the concepts described above by means of an example which implements the sieve of Eratosthenes[12]. This algorithm identifies successive prime numbers from a sequence of consecutive natural numbers. We define an entity type called *sieve* to implement the algorithm. Multiple instances of the *sieve* entity are created - one for each prime number that has already been identified in the sequence. The various *sieve* entities form a pipeline. Each *sieve* entity in the pipeline inputs numbers from its predecessor, suppresses those that are multiples of the original prime and passes the rest to the successor entity.

The program to implement this algorithm is presented in pseudo-code in figure 2. Entity **main** (lines 0-10) is used to initiate the program. The **main** entity creates the first *sieve* entity whose unique identifier is stored in its local variable *first_sieve* (line 5). Main sends a stream of integers 2,3,4,5.... to entity *first_sieve* via messages of type *next_number* (lines 7-8). The types of all messages that may be received by an entity must be defined within the corresponding entity type definition. For instance, the *sieve* entities may receive messages of type *next_number*. This message type is defined in line 16. The first element of the stream of numbers received by a *sieve* entity is a prime. For instance, the first number (i.e. 2) received by the *sieve* entity *first_sieve* is a prime number, and is stored in the entity's local variable *my_prime* (line 19). At this point, entity *first_sieve* creates a new instance of the *sieve* entity-type. In general, the ith *sieve* (i>1), say s_i, is recursively created by the (i-1)th *sieve* (line 20). Subsequently, s_i removes all multiples of *my_prime* from the sequence of numbers received by it and passes the rest onto *sieve* s_{i+1} via messages of type *next_number* (lines 24-25). A *sieve* entity receives (or waits to receive) the next message of type *next_number* by executing the **wait** statement in line 23.

Figure 3 presents another program fragment that represents a simple FIFO server, using essentially the primitives introduced by the preceding example. Entity *server* is a FIFO server with two parameter: *mu*, which represents the average service time for a job, and *histogram* which is the identifier for a histogram entity (line 0). The actual service time required to process a job is generated by using an exponential distribution. (Alternatively a job may itself generate its service time). Jobs that require service send a *request* message to the *server* entity. This message has two parameters: the (simulation) time at which the request was generated and the identifier of the requesting job (line 3). When idle, the *server* entity accepts a *request* message (line 8), generates a service time and executes a wait

```
entity main;
{ Local Variable Declaration Section }
        first_sieve : entity_identifier;
        i:integer;
{ Entity Body }
        let first_sieve be sieve;
{ send a stream of numbers 2,3...1000 to first_sieve }
        for i := 2 to 1000 do
        invoke first_sieve with next_number(i);
end-entity;

entity sieve;
{ Local Variable Declaration Section }
        next-sieve : entity_identifier;
        my_prime: integer;
{ Message Receive Declaration Section }
        message next_number(number:integer);
{ Entity Body }
        wait for ( message-type = next_number);
        my_prime:= number;
        let next-sieve be sieve;
        while true do
        begin
            wait for (message-type = next_number);
            if (mod(number,my_prime) <> 0) then
                invoke next-sieve with next_number(number);
        end;
end-entity;
```

Figure 2: Sieve of Eratosthenes

statement (line 9) that simulates the processing of the request. In general, a **wait** statement has the following form:

wait [*t*] [for *b*]

where *t* is the **wait-time** and *b* represents the **wait-condition**. The wait-condition may reference any local variables of the entity and is normally used by the entity to specify the message(s) it is ready to accept. Execution of a **wait** statement causes the entity to wait if it specifies a non-null wait-time and no message satisfying the wait-condition is present in its message buffer. If the entity is waiting for a specific message(s), other messages received by the entity are stored (in the order they were received) in the message buffer. A waiting entity ceases to wait when it is delivered a message that satisfies the condition *b* or if it receives a **time-out** message from the monitor. A **time-out** message is sent to the entity if no message satisfying condition *b* is received by the entity within the specified wait-time.

In our example of a FIFO server, while the entity is servicing a request, it does not accept any requests until the current request has been serviced. As a result, the wait condition in line 9 evaluates to *true* only on receipt of a *time-out* message. On receiving this message, the entity sends a *reply* message to the job entity (line 10) to indicate that the requested service has been completed. The *server* also computes the total amount of time that elapsed from when the request was generated until the service was completed. It sends this time to a library entity called *histogram*(line 11). (Alternatively, each job entity may compute its own elapsed system time). Entity *histogram* is one among a variety of statistics collection routines available in the library. It is used to generate a histogram of values in specified intervals. The *server* is now ready to accept the next *request* message (line 8).

In the next section, we describe algorithms to execute programs which contain both operational and simulation modules.

5 Partially Implemented Specifications

In this section we present the discrete-event simulation algorithm and contrast it with the algorithm to execute partially implemented specifications on sequential and multicomputer architectures. The terms entity and LP are used interchangeably in this discussion.

```
entity server(mean-service-time:integer; histogram:entity-identifier);

{ Message Receive Declaration Section }
        message request(stim:integer;patient-id : entity-identifier);

{ Entity Body }
while true do
begin
        wait for (message-type = request);
        wait exp(mean-service-time) for (message-type = time-out);
        invoke patient-id with reply;
        invoke histogram with insert(clock-stim);
end;
end-entity;
```

Figure 3: FIFO Server Entity

5.1 Simulation Algorithm

Typical simulation algorithms use two data structures[17]: a *simulation clock* and an *event-list*. The simulation clock gives the time up to which the physical system has been simulated. The event-list is a partial order of tuples; each tuple consists of three fields:(m_i,p_i,t_i), where m_i represents a message, p_i the destination LP for m_i, and t_i a timestamp. The partial order is typically based on the timestamp. If two dependent events have the same timestamp, the dependencies must be expressed explicitly. At every step of the simulation, the algorithm selects the tuple with the smallest timestamp, say (m_i,p_i,t_i), from the event-list and delivers message m_i to LP p_i. The partial order guarantees that events are simulated in the order of their dependencies. The simulation clock advances in a monotonic non-decreasing manner through the timestamps associated with each tuple.

The preceding simulation algorithm is sometimes refered to as an *imperative* algorithm. Other simulation languages use an *interrogative* algorithm, also called a *wait until* algorithm[25]. In this algorithm, messages are not necessarily delivered in the partial-order specified by the event-list. Each LP may specify a *wait condition* which restricts the set of messages that it is willing to accept; a message is delivered to the destination LP only

```
$clock$:=0;
while ($simulation\ not\ terminated$) do
{       fetch next tuple ($m_i$,$p_i$,$t_i$) from event-list;
        if ($m_i$ is not accepted by $p_i$) then
                store $m_i$ in $tempq$;
        else    { if ($m_i$=$time\text{-}out$) then $clock$:=$t_i$;
                  deliver $m_i$ to $p_i$ for simulation ;
                  merge $tempq$ with $event\text{-}list$;
                }
}
```

Figure 4: Wait Until Simulation Algorithm

if it satisfies its wait-condition. The basic difference between the imperative and interrogative simulation algorithms lies in their treatment of conditional events: in the imperative algorithm, if a LP is not ready to process a message, it must do internal buffering and process the message at a later time; in the interrogative algorithm, the buffering is done directly by the simulation algorithm. The simulation clock in the interrogative algorithm cannot advance through the timestamps associated with the tuples because the messages need not be accepted by the LPs in the partial order implied by the timestamps. Instead, the simulation clock is advanced through the timestamps associated with the time-out messages in the event-list. An LP schedules its own *time-out* messages and must accept it when it is sent. The event-list always contains an entry for each entity that indicates the time at which a *time-out* message is to be sent; this time is set by default to an arbitrarily large value. The wait-until simulation algorithm is described in figure 4, in which *clock* refers to the simulation clock. Figure 4 is an inefficient implementation of the algorithm. However, it serves our purpose of outlining the basic nature of simulation. For a detailed discussion of implementation strategies for the above algorithm, the reader is refered to Franta[10, Chapter 7].

We consider the simulation for a producer consumer program. In the program, a producer process produces data and sends it to a buffer process; a consumer process requests data from the buffer process, and is blocked if the buffer is empty. The simulation code for this program is presented in figure 5, where buffer overflow has been ignored. We assume **main** creates one instance of each of the *producer*, *consumer* and *buffer* entity-types, respectively named *p1*, *c1* and *b1*. The service times for both *p1* and *c1* are generated from a negative exponential distribution with a mean of 5 units. Assume that entity *p1* takes 4, 6, 8, and 3 time units to generate successive units of data; and *c1* takes 8, 2, 6, and 9 time units respectively to consume successive data units. Figure 6 represents the progress of the simulation by changes in the event-list and the simulation clock. The underlined tuple in each entry contains the 'next' message to be processed in the simulation. The clock in figure 6 refers to the simulation time at which the message in the underlined tuple is delivered to the entity. The *time-out* message has been abbreviated to *tout* in the figure. When created, entity *c1* sends a *get* message to *b1*, and *p1* schedules a *time-out* message. The initial event-list also contains default entries with *time-out* messages for *c1* and *b1*. Figure 7 shows a time-line diagram for the simulation; each simulation step of an entity is shown by a dashed line segment.

In the physical system modeled by the simulation, the producer and consumer processes execute on separate processors. This mapping is modeled by executing the simulation steps of entities *p1* and *c1* in parallel. For instance, in figure 7 at time 4, simulation time elapses in parallel for both *p1* and *c1*. If processes in the physical system are interleaved on a single processor, the simulation steps of the corresponding LPs must be executed sequentially. This is typically achieved by defining a *server* entity of the type shown in figure 3. In the modified simulation, the producer and consumer entities simulate generation or consumption of data by explicitly requesting the appropriate amount of service from the *server* entity. In the new model, each simulation step of the *server* entity simulates the processing of a request from either the producer or the consumer entity, thus effectively mapping both entities to the same processor. In the following sections, we modify the wait-until simulation algorithm to execute PIPS programs.

5.2 PIPS Algorithm

In a partially implemented program, if an entity receives a *time-out* message it executes a simulation step; otherwise it executes a computation step. The PIPS algorithm differs from traditional simulation algorithms primarily in its treatment of the computation step executed by a LP. In simulations, the execution time of a computation step is completely ignored. For instance, in the time diagram in figure 7, the computation step executed by entity *b1*

```
entity producer(mu: integer; bufid:entity-identifier);
while (true) do
{    /* simulate the generation of data */
     wait exp(mu) for (message-type = time-out);
     invoke bufid with put(data);
}
end-entity;

entity consumer(mu: integer; bufid:entity-identifier);
message next(data:integer);
while (true) do
{    invoke bufid with get(myid);
     wait for (message-type = next);
     /* simulate the consumption of data */
     wait exp(mu) for (message-type = time-out);
}
end-entity;

entity buffer;
flag:boolean:=false;
message put(data:integer);
message get(hisid:entity-identifier);
while (true) do
{    wait for ((message-type=get) or (message-type=put));
     if (message-type = get) then
          if (buffer not empty) then
            invoke hisid with next(next-data-item)
          else flag:=true;
     else /* message must be a put */
          if (flag) then /* satisfy outstanding request */
          {  invoke hisid with next(data);
             flag:=false;}
          else store data in buffer;
}
end-entity;
```

Figure 5: Simulation of a Producer Consumer Program

No.	*Clock*	*Event-list*
1	0	$(\underline{\langle get,b1,0\rangle}, \langle tout,p1,4\rangle, \langle tout,c1,\infty\rangle, \langle tout,p1,\infty\rangle)$
2	4	$(\underline{\langle tout,p1,4\rangle}, \langle tout,c1,\infty\rangle, \langle tout,p1,\infty\rangle)$
3	4	$(\underline{\langle put,b1,4\rangle}, \langle tout,p1,10\rangle, \langle tout,c1,\infty\rangle, \langle tout,b1,\infty\rangle)$
4	4	$(\underline{\langle next,c1,4\rangle}, \langle tout,p1,10\rangle, \langle tout,b1,\infty\rangle, \langle tout,c1,\infty\rangle)$
5	10	$(\underline{\langle tout,p1,10\rangle}, \langle tout,c1,12\rangle, \langle tout,b1,\infty\rangle)$
6	10	$(\underline{\langle put,b1,10\rangle}, \langle tout,c1,12\rangle, \langle tout,p1,18\rangle, \langle tout,b1,\infty\rangle)$
7	12	$(\underline{\langle tout,c1,12\rangle}, \langle tout,p1,\infty\rangle, \langle tout,b1,\infty\rangle)$
8	12	$(\underline{\langle get,b1,12\rangle}, \langle tout,p1,18\rangle, \langle tout,c1,\infty\rangle, \langle tout,b1,\infty\rangle)$
9	12	$(\underline{\langle next,c1,12\rangle}, \langle tout,p1,18\rangle, \langle tout,c1,\infty\rangle, \langle tout,b1,\infty\rangle)$
10	14	$(\underline{\langle tout,c1,14\rangle}, \langle tout,p1,18\rangle, \langle tout,b1,\infty\rangle)$
11	14	$(\underline{\langle get,b1,14\rangle}, \langle tout,p1,18\rangle, \langle tout,c1,\infty\rangle, \langle tout,b1,\infty\rangle)$
12	18	$(\underline{\langle tout,p1,18\rangle}, \langle tout,c1,\infty\rangle, \langle tout,b1,\infty\rangle)$
13	18	$(\underline{\langle put,b1,18\rangle}, \langle tout,p1,21\rangle, \langle tout,b1,\infty\rangle, \langle tout,c1,\infty\rangle)$
14	18	$(\underline{\langle next,c1,18\rangle}, \langle tout,p1,21\rangle, \langle tout,b1,\infty\rangle, \langle tout,c1,\infty\rangle)$
15	21	$(\underline{\langle tout,p1,21\rangle}, \langle tout,c1,24\rangle, \langle tout,b1,\infty\rangle)$
	⋮	⋮

Figure 6: Events in a Simulation of the Producer Consumer Program

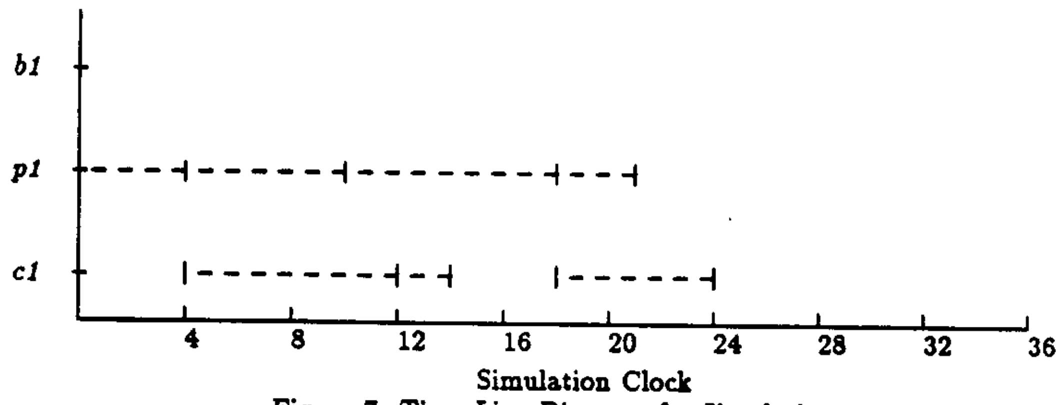

Figure 7: Time Line Diagram for Simulation

to process a *put* or *get* message does not have any effect on the simulation clock. By contrast, the execution time of the computation step is critical to the performance measurement of a PIPS program.

The simulation steps of two (or more) LPs may be executed sequentially by requiring both LPs to 'request' service from a common *server* LP. How do we specify sequential execution of the computation step of one LP with the simulation step of another? We introduce the concept of a Logical element (LE). When created, each entity is mapped to a specific LE; multiple entities may be mapped to a common LE. The simulation or computation steps of all entities mapped to a particular LE are executed sequentially. In order to allow entities to be mapped to an LE, the **let** primitive described in section 4 is extended as follows:

let e_1 **be** *entity-type-name* **on** le_i

where le_i is a variable of type **element-identifier**. Variables of this type are used to represent logical elements. Each distinct value for a variable of type **element-identifier** corresponds to a unique processor in the physical system. Each LE is associated with a unique *virtual* clock; we use $clock_i$ to refer to the *virtual* clock for LE_i. The *virtual* clock of an LE may be advanced by the execution of a computation step or a simulation step. In this section, we present a centralized algorithm that determines how each *virtual* clock in a PIPS program is advanced. The next section describes a distributed algorithm to execute PIPS programs on multicomputers.

The PIPS algorithm uses two data structures: a set of *virtual* clocks, one for each LE in the program, and an event-list. For a given LE, say LE_i, the value of $clock_i$ represents the time up to which the entities mapped to LE_i have been simulated or executed. The event-list is a partial order of tuples: each tuple in the event-list contains (m_i,p_i,l_i,t_i), where message m_i is to be delivered to LP p_i which is mapped on LE l_i. As in the simulation algorithm, the timestamp t_i is used to assign the partial order to messages in the event-list. However, there is one major difference in the semantic interpretation of t_i between the simulation and PIPS algorithms. In the simulation algorithm of figure 6, the timestamp on a *time-out* message indicated the time at which the message was delivered to the named LP. When multiple entities are mapped to an LE, their simulation or computation steps must be executed sequentially. Under these conditions, the *time-out* message to an entity cannot be scheduled at an absolute time in the future. The timestamp on the message is simply used to indicate the *earliest* time at which this message may be delivered to an entity. It is computed by adding the wait-time specified by the entity to the current value of the *virtual* clock; the wait-time is also carried separately in the *time-out* message. The main question that arises is the following: how is the *virtual* clock of an LE advanced during execution of a PIPS program? To motivate our answer, we first consider this question in two simple contexts.

Consider the PIPS code for the producer consumer program where the code for entity consumer is fully developed, and the producer entity is only partially implemented. Figure 8 gives the expanded code for entity consumer. The code for the producer and buffer entities is exactly as in figure 5. As in the simulation, we assume that **main** creates one instance of each entity-type, respectively named *p1*, *c1* and *b1*. The computation step of entity *c1* is assumed to take 5 time units, and that of *b1*, 3 time units. The wait-time for the producer entity is once again sampled from an exponential distribution with a mean value of 5 time units, and the first few values are again assumed to be 4, 6, 8 and 3 respectively. We consider two simple mappings of the PIPS program: one where each entity is mapped to a separate LE, and the other where all three entities are mapped to a common LE.

```
entity consumer(bufid:entity-identifier);
count: integer;
message next(data:integer);
count:=0;
while (true) do
{    invoke bufid with get;
     wait for (message-type = next);
     count:=count+1;
     print(data);
}
end-entity;
```

Figure 8: PIPS Code for Entity Consumer

Figure 9 represents the execution of the PIPS program where each entity is mapped to a separate LE. For brevity, we omit the LE-field from the tuples in the event-list. For each entry in figure 9, the tuple which contains the next message to be processed is underlined. The clock value refers to the time at which the message in the underlined tuple is delivered to the destination entity, and is equal to the time denoted by the *virtual* clock for the destination entity immediately after the message is accepted by it (we use '*virtual* clock of an entity or LP' to mean the *virtual* clock of the LE to which the entity has been mapped). The time taken for the execution (or simulation) of the computation (or simulation) step is illustrated in the time diagram in figure 10. The simulation steps are shown by a dashed line and the computation steps by a continuous line. At any point in its execution, the value of the PIPS clock is the minimum of the values of the virtual clocks for all LEs and indicates the time up to which the PIPS program has been executed or simulated.

Execution of the PIPS program proceeds by executing events in the partial order determined by the event-list and the wait-condition of the LPs. Consider the first three events from figure 9. The first event is the execution of a computation step by entity *b1* to process a *get* message; this event advances its *virtual* clock to 3. The next event is the transmission of a *time-out* message to entity *p1*; the processing of this message by *p1* causes its *virtual* clock to advanced in simulation time to 4 and also deposits a *put* message for entity *b1* with timestamp 4. Note that the simulation step for entity *p1* was initiated when its *virtual* clock was at 0, and procceeded in parallel with the computation step of *b1*. When the *put* message is delivered to *b1*, it executes another computation step which advances its *virtual* clock to 7 (the timestamp on the *put* message(4) + the execution time of the computation step (3)) and deposits a *next* message with timestamp 7 for delivery to entity *c1*. The first three entries should be contrasted with the corresponding entries for the simulation. As seen from figure 6, the latter two events happen at simulation time 4 because the execution time of a computation step is ignored in the simulation.

Consider a case where the computation step of an entity overlaps with the simulation step of another: the fourth event in the PIPS program of figure 9 is the processing of a *next* message by *c1* when its *virtual* clock reads 7; as seen in the time line diagram of figure 10, the computation step (which takes 5 time units) will overlap with the simulation step of entity *p1* which expires at 10 time units. At this point, the computation step of entity *b1* is initiated which then overlaps with the continuing computation step of entity *c1*. Overlapping processing steps need not cause the execution of an entity to be interrupted. In the above example, even though the simulation step of *p1* completes before the computation step of *c1*, executing the simulation step of *p1* (that is, sending a *time-out* message) *after* the computation step

No.	*Clock*	*Event-list*
1	0	(<*get,b1*,0>, <*tout,p1*,4(4)>, <*tout,c1*,0(∞)>, <*tout,b1*,0(∞)>)
2	4	(<*tout,p1*,4(4)>, <*tout,c1*,∞>, <*tout,b1*,∞>)
3	4	(<*put,b1*,4>, <*tout,p1*,10(6)>, <*tout,c1*,∞>, <*tout,b1*,∞>)
4	7	(<*next,c1*,7>, <*tout,p1*,10(6)>, <*tout,b1*,∞>, <*tout,c1*,∞>)
5	10	(<*tout,p1*,10(6)>, <*tout,c1*,∞>, <*tout,b1*,∞>)
6	10	(<*put,b1*,10>, <*tout,p1*,18(8)>, <*tout,c1*,∞>, <*tout,b1*,∞>)
7	12	(<*get,b1*,12>, <*tout,p1*,18(8)>, <*tout,c1*,∞>, <*tout,b1*,∞>)
8	16	(<*next,c1*,16>, <*tout,p1*,18(8)>, <*tout,c1*,∞>, <*tout,b1*,∞>)
9	18	(<*tout,p1*,18(8)>, <*tout,c1*,∞>, <*tout,b1*,∞>)
10	18	(<*put,b1*,18>, <*tout,p1*,21(3)>, <*tout,c1*,∞>, <*tout,b1*,∞>)
11	21	(<*get,b1*,21>, <*tout,p1*,21(3)>, <*tout,c1*,∞>, <*tout,b1*,∞>)
12	21	(<*tout,p1*,21(3)>, <*tout,c1*,∞>, <*tout,b1*,∞>)
13	21	(<*put,b1*,21>, <*tout,p1*,34(13)>, <*tout,c1*,∞>, <*tout,b1*,∞>)
15	24	(<*next,c1*,24>, <*tout,p1*,34(13)>, <*tout,c1*,∞>, <*tout,b1*,∞>)
17	29	(<*get,c1*,29>, <*tout,p1*,34(13)>, <*tout,c1*,∞>, <*tout,b1*,∞>)
19	32	(<*next,c1*,32>, <*tout,p1*,34(13)>, <*tout,c1*,∞>, <*tout,b1*,∞>)
	⋮	⋮

Figure 9: Events in the PIPS Model : Multiple LE Mapping

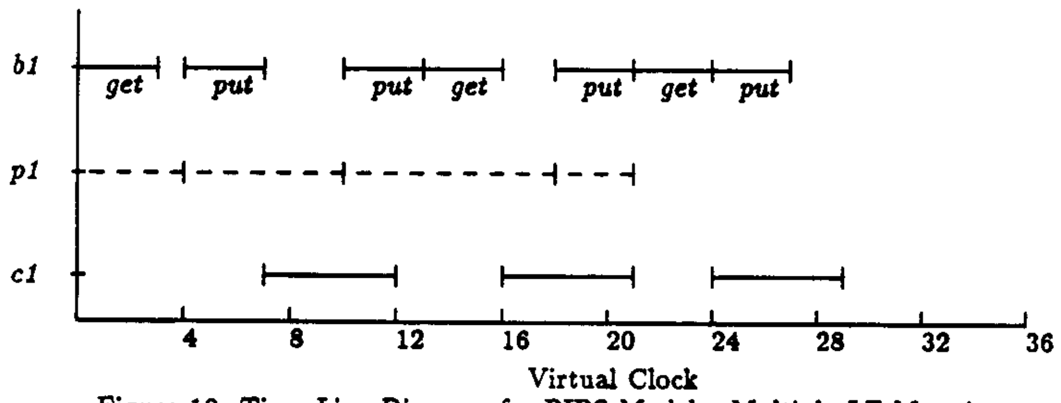

Figure 10: Time Line Diagram for PIPS Model – Multiple LE Mapping

of *c1* has been completed does not affect logical correctness. This is primarily due to the assumption that the computation step of an entity is atomic; thus no message can be received by the entity from the time it initiates the processing of a message until it completes its computation step. In particular, if on completing its simulation step, entity *p1* was to send a message to *c1*, entity *c1* may process the message only after its computation step has been executed. Of course, an implementation may choose to interrupt entity *c1*, process the *time-out* message for entity *p1*, and resume execution for *c1*. However, the context-switching implied by this approach may be expensive to implement.

We now consider a different mapping where the program is being designed for a centralized implementation. The new mapping implies that all computation and simulation steps are executed sequentially. This configuration may be tested with minimum modifications to the program; only entity main needs to be modified to map all entities to a common LE. For such mappings, the execution algorithm is particularly simple because a single *virtual* clock needs to be maintained. The events in the PIPS execution are illustrated in figure 11. The sequential nature of the program is clearly visible in the time line diagram for this mapping in figure 12. In the centralized mapping, the first *next* message is processed by *c1* when its *virtual* clock reads 10 units as compared to the previous mapping where the message was processed at time 7.

The previous discussion indicates how the virtual clocks in a PIPS program must be handled. Messages are (conditionally) delivered to entities in the order determined by their timestamps. If two messages have the same timestamp and must be processed in a specific order, in general, the program must explicitly indicate the dependency. The PIPS algorithm predefines the ordering in the following two cases: if the messages are generated by a single entity, delivery will be attempted in the order in which the messages were generated. Secondly, if exactly one of the messages is a *time-out* message, delivery of the other message will be attempted first. In all other cases, messages with the same timestamp may be delivered in an arbitrary order. The timestamp for messages other than *time-out* messages is the time shown by the *virtual* clock of the transmitting entity at the end of its computation (or simulation) step. The timestamp for a *time-out* message is the sum of the *virtual* clock of the entity and the wait-time specified in the wait statement executed by the entity.

When a *time-out* message is delivered to an entity, its *virtual* clock is

No.	*Clock*	*Event-list*
1	0	$(\underline{\langle \mathit{get}, \mathit{b1}, 0\rangle}, \langle \mathit{tout}, \mathit{p1}, 4(4)\rangle, \langle \mathit{tout}, \mathit{c1}, \infty\rangle, \langle \mathit{tout}, \mathit{b1}, \infty\rangle)$
2	7	$(\underline{\langle \mathit{tout}, \mathit{p1}, 4(4)\rangle}, \langle \mathit{tout}, \mathit{c1}, \infty\rangle, \langle \mathit{tout}, \mathit{b1}, \infty\rangle)$
3	7	$(\underline{\langle \mathit{put}, \mathit{b1}, 7\rangle}, \langle \mathit{tout}, \mathit{p1}, 13(6)\rangle, \langle \mathit{tout}, \mathit{c1}, \infty\rangle, \langle \mathit{tout}, \mathit{b1}, \infty\rangle)$
4	10	$(\underline{\langle \mathit{next}, \mathit{c1}, 10\rangle}, \langle \mathit{tout}, \mathit{p1}, 13(6)\rangle, \langle \mathit{tout}, \mathit{b1}, \infty\rangle, \langle \mathit{tout}, \mathit{c1}, \infty\rangle)$
5	15	$(\langle \mathit{get}, \mathit{b1}, 15\rangle, \underline{\langle \mathit{tout}, \mathit{p1}, 13(6)\rangle}, \langle \mathit{tout}, \mathit{c1}, \infty\rangle, \langle \mathit{tout}, \mathit{b1}, \infty\rangle)$
6	21	$(\underline{\langle \mathit{get}, \mathit{b1}, 15\rangle}, \langle \mathit{put}, \mathit{b1}, 21\rangle, \langle \mathit{tout}, \mathit{p1}, 29(8)\rangle, \langle \mathit{tout}, \mathit{c1}, \infty\rangle, \langle \mathit{tout}, \mathit{b1}, \infty\rangle)$
7	24	$(\underline{\langle \mathit{put}, \mathit{b1}, 21\rangle}, \langle \mathit{tout}, \mathit{p1}, 29(8)\rangle, \langle \mathit{tout}, \mathit{c1}, \infty\rangle, \langle \mathit{tout}, \mathit{b1}, \infty\rangle)$
8	27	$(\underline{\langle \mathit{next}, \mathit{c1}, 27\rangle}, \langle \mathit{tout}, \mathit{p1}, 29(8)\rangle, \langle \mathit{tout}, \mathit{c1}, \infty\rangle, \langle \mathit{tout}, \mathit{b1}, \infty\rangle)$
9	32	$(\langle \mathit{get}, \mathit{c1}, 32\rangle, \underline{\langle \mathit{tout}, \mathit{p1}, 29(8)\rangle}, \langle \mathit{tout}, \mathit{c1}, \infty\rangle, \langle \mathit{tout}, \mathit{b1}, \infty\rangle)$
10	40	$(\underline{\langle \mathit{get}, \mathit{c1}, 32\rangle}, \langle \mathit{put}, \mathit{b1}, 40\rangle, \langle \mathit{tout}, \mathit{p1}, 43(3)\rangle, \langle \mathit{tout}, \mathit{c1}, \infty\rangle, \langle \mathit{tout}, \mathit{b1}, \infty\rangle)$
	⋮	⋮

Figure 11: Events in the PIPS Model : Single LE Mapping

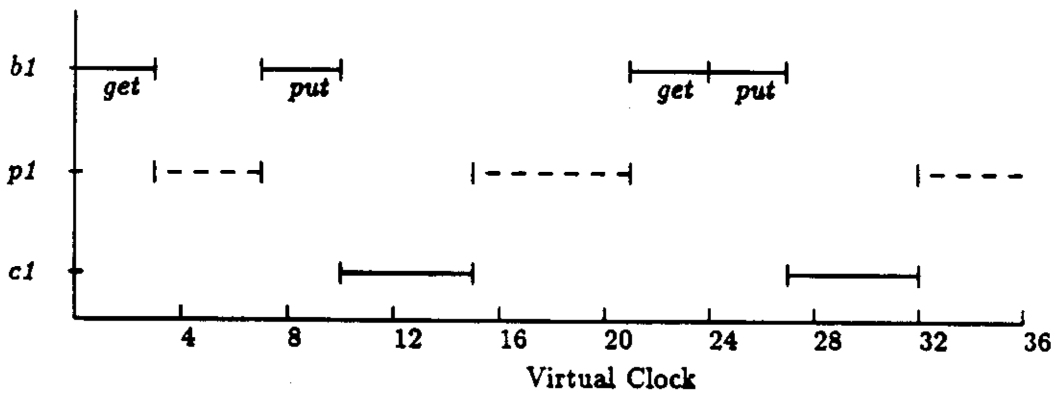

Figure 12: Time Line Diagram for PIPS Model – Single LE Mapping

```
$clock$:=0;
while ($execution$ $not$ $terminated$) do
{   fetch next tuple ($m_i$,$p_i$,$t_i$) from event-list;
    if ($m_i$ is not accepted by $p_i$) then store $m_i$ in $tempq$;
    else {
        if ($m_i$=$time$-$out$) then
        {   $clock_i$:=$clock_i$+$\Delta t_s$;
            $clock$:=$clock_i$;
            $p_i$ simulates processing of $m_i$;
        };
        else {$clock_i$:=max($t_i$,$clock_i$);
            $clock$:=$clock_i$;
            $p_i$ processes $m_i$; measure duration of the computation step ($\Delta t_c$);
            $clock_i$:=$clock_i$+$\Delta t_c$;
        };
        merge $tempq$ with $event$-$list$;
    };
};
```

Figure 13: Centralized Algorithm to execute PIPS Programs

simply incremented by Δt_s, the wait-time specified in the message. For other messages, the duration, say Δt_c, of the computation step executed by the entity is measured and the *virtual* clock of the entity set to the sum of Δt_c and the larger of its *virtual* clock or the message timestamp. The algorithm to execute PIPS programs is presented in figure 13. In the figure, *clock* refers to the *PIPS* clock whose value indicates the time up to which the PIPS program has been executed.

Why are *time-out* messages processed differently from other messages? The *time-out* message is an example of a conditional message. A conditional message is scheduled for the future and may be canceled by the sending entity. When an entity executes a simulation step, it schedules a *time-out* message as a conditional message: if the entity accepts another message in the interim, it may cancel the previously scheduled *time-out* message. The timestamp on a *time-out* message indicates the completion time of the simulation step executed by an entity. This ensures that the *time-out* message is processed only after ascertaining that no other message was indeed accepted by the entity. However, the scheduling of every *time-out* message need not be conditional. For instance, consider the simulation step executed by entity *put*. The wait-condition guarantees that the entity will not accept any message until its specified wait-time has expired. In this situation, the *time-out* message may be delivered to an entity, like other messages in the system, in the order determined by the time at which it is *created*.

How do we distinguish between conditional and unconditionally scheduled *time-out* messages? The wait-condition explicitly indicates whether or not the scheduling of the *time-out* message is conditional. Alternatively, a separate language construct may be used to specify a predetermined duration for a simulation step (similar to the hold primitve of SIMULA[8]). What is the benefit of scheduling unconditional *time-out* messages? If each entity executes only unconditional simulation steps, the difference between execution of a simulation step or a computation step is minimal. The computation step executed by an entity may be viewed exactly as a simulation step scheduled by an unconditional *time-out* message, except that its duration is actually measured rather than specified by the entity. However, conditional messages are essential in simulations to model objects like preemptible servers. The notion of conditional and unconditional events has been used by Chandy & Misra to suggest an elegant scheme for distributed simulation[5].

In a PIPS program, the execution time of every computation step may not be relevant to the performance of a system. Consider, for instance, a statistics collection entity, like the histogram entity in a previous example. Presumably this entity will not be a part of the eventual system, and the execution time of its computation step should not be included in the performance metrics being collected for the system. We define le_0 to be a null-valued **element-identifier**. For all entities mapped to le_0, the execution time of the computation steps are ignored and the simulation steps are executed in parallel. A PIPS program may be executed entirely as a simulation, simply by mapping all entities in the program to the logical element le_0.

In the example discussed in this section, we have tacitly assumed that the computation time used by the run-time system is not relevant to the performance measurements. As a result, the overhead of event-list management, message-transmission, ...have been ignored. This assumption was made only for simplicity in exposition; message-transmission time in the underlying system can be incorporated in performance measurements. Of course, the transmission medium may also be explicitly modeled by an entity.

5.3 Distributed Implementation

This section describes an algorithm to execute PIPS programs in a multicomputer environment. Once again, we modify existing algorithms for distributed simulation to execute PIPS programs on parallel architectures.

What are the benefits of executing a PIPS program on multiple processors? In the case of distributed simulation, the motivation was to decrease the elapsed time required to execute simulation programs. In our case, this simply represents another refinement step: replacing some server LPs in the model (for instance, the entity that models a communication network) by the actual hardware (use an operational network to transmit messages between entities) or execute different modules of the evolving design on a parallel computer. We also have an additional motive - validation. If the hardware architecture of the system being designed is (partially) available, it can be used to validate part of the PIPS model. For instance, assume that the communication network to be used in a system is available. In this case, the entity that modeled the network may be removed from the PIPS program, and the program can use the available network to transmit messages. Measurements of the transmission times can be used to validate the model. It is important to note that replacing an entity by actual hardware may increase the elapsed time for execution of the PIPS program. (This follows because simulating a message transmission may require the execution of only a few instructions on a processor. The time taken for actual transmission of the message over a network will be determined by the network itself.) Further, the refinement affects the performance characteristics of the PIPS system only to the extent that entities are an abstraction of actual components, and may not reproduce the exact behaviour of the physical device (for instance, the network in our example).

In distributed simulations, the event-list is not centralized. Instead, each LP locally selects the next message for processing in a manner such that all messages are *eventually* processed in their correct order of dependency. Various algorithms [7,17,13] have been designed to allow an LP to make this decision locally. We use the so-called conservative simulation technique suggested by Chandy & Misra and described in [17] to execute PIPS programs on parallel architectures. (A few performance studies[11,19] on the simulation of queuing networks using the above technique for distributed simulation found insignificant speed up in the completion time of the simulation program. We reiterate that in using distributed simulation, our goal is not to decrease the execution time of PIPS programs; rather it is to allow the

PIPS program to be executed in a multicomputer environment. If required, other techniques for distributed simulation that prove to be more efficient can be incorporated.)

We adapt the distributed simulation algorithm described in [17] for the execution of PIPS programs on parallel architectures. The simplest distribution of the PIPS program is to execute each LE on a separate processor. In general, multiple LEs may execute on a given processor; we assume one LE per processor, only for simplicity in exposition. In the distributed PIPS model, we assume that each LE is associated with a number of incoming channels, on which it receives messages from LPs on other LEs, and a number of outgoing channels, on which it sends messages to LPs on other LEs. Each LE, say l_i, waits until it has a message on each incoming channel, and picks the message, say m_i with the smallest timestamp, say t_i. As in the centralized algorithm, if m_i is a *time-out* message, the *virtual* clock of l_i is advanced by the simulation period; otherwise the clock is advanced by the amount of time required to process the message. The handling of the incoming message may possibly cause messages to be sent on some of the output channels associated with l_i. The main problem with this scheme is that if no messages are received on a particular channel, the destination LE for that channel will be blocked. A cycle of blocked processes may form, causing the system to deadlock. This may be prevented by requiring every LE to eventually send some message on every outgoing channel. If no messages are generated by the LPs on the LE, dummy messages refered to as null messages are sent to avoid deadlocks. The algorithm to execute a distributed PIPS program is presented in figure 14, where transmission of null messages has been omitted. The algorithm is similar to the centralized algorithm of figure 13. The major difference in the distributed implementation lies in that each LE *locally* decides its 'next' event.

The language discussed in section 4 needs to be extended in minor ways to handle distributed execution of PIPS programs. In particular primitives must be introduced to execute (simulate) the entities mapped to an LE on a specific processor. On the other hand, the run-time system may randomly distribute the load among available processors.

6 Summary and Remarks

We have described a performance-oriented approach to the design of distributed systems which are not amenable to analytical modeling. In this

```
$clock$:=0;
while ($execution\ not\ terminated$) do
{   wait until a message exists on every incoming channel;
    $m_i$:= message with minimum timestamp;
    if ($m_i$ is not accepted by $p_i$) then
        store $m_i$ in $tempq$;
    else
    {           if ($m_i$=$time$-$out$) then
        {   $clock_i := clock_i + \Delta t_s$;
            $p_i$ simulates processing of $m_i$;
        };
        else {$lock_i := \max(t_i, clock_i)$;
            $p_i$ processes $m_i$; measure duration of the computation step ($\Delta t_c$);
            $clock_i := clock_i + \Delta t_c$;
        };
        merge $tempq$ with message-list of appropriate channel(s);
    };
};
```

Figure 14: Distributed Algorithm to execute PIPS Programs

section, we examine major restrictions of our approach and briefly describe the implementation efforts in progress.

The approach suggested in this paper allows processes in distributed programs to execute in one of two modes: *simulation* mode or *computation* mode. A process (entity) executes in the simulation mode to *model* the processing of a message, and executes in the computation mode to actually process the messages. In the computation mode, the time taken by the entity to process the message is measured by the clock associated with the processor on which the process is executed. In the simulation mode, the actual time taken by the processor is ignored; instead the relevant time is the duration of the simulation step measured by the simulation clock. Both the simulation and computation time-periods are included to predict system performance.

The most serious restriction of the PIPS approach concerns the elaboration of a conditional simulation step – a simulation step scheduled by a conditional *time-out* message. Consider the representation of a preemptible program module. An entity may simulate such a module by scheduling a conditional *time-out* message which is canceled if the entity accepts another message before completion of the simulation step. In general, this effect cannot be directly elaborated into a computation step, becuase the computation step is atomic. Consider the following situation: PP_a and PP_b execute computation steps of durations t_a and t_b in parallel, where $t_a < t_b$. After executing its computation step PP_a sends a message m1 to PP_b, which must be processed by PP_b before the expiration of interval t_b. Assume that the PIPS model is being executed on a uniprocessor. If LP_b executes a simulation step, the preceding situation can be modeled correctly by requiring LP_b to schedule a conditional *time-out* message. However, if LP_b executes a computation step, the message cannot be processed before duration t_b expires. Note that this problem cannot be solved simply by allowing a specified message to interrupt the computation step of an entity (although such a facility would be necessary to remove this restriction). The problem arises because the execution of LP_a and LP_b must be *interleaved* in the PIPS environment. If the computation step of LP_b is executed before that of LP_a, message m1 will not exist when LP_b is executing. The problem may be resolved in one of two ways: execute the computation step of LP_b as a series of microcomputation steps, i.e. interleave the execution of LP_a and LP_b at a smaller level of granularity, thus allowing LP_a to generate message m1 before the computation step of LP_b is completed. A second alternative is to use rollback. Checkpoint the states of LP_a and LP_b before executing computation steps, and execute the computation steps in arbitrary order. If the computation steps were executed in an incorrect order, rollback the computation and reexecute to ensure that message m1 is, in fact, generated before the computation step of LP_b is initiated. These ideas are currently under further investigation.

A second restriction of our approach concerns the use of shared memory. The distributed system model described in section 2 restricts communication between PPs to be message-based. However, shared memory may be used by restricting how each entity updates the shared variables. The effect of these restrictions is to ensure that access to the shared variable does not affect the event dependencies that are represented by the event-list. The nature of these restrictions is similar to the restrictions placed on Ada programs that use shared variables[1][section 9.11]. Other restrictions have to do with the resolution and speed of the processor clock on which an entity is executed. The resolution of the clock limits the accuracy of the measurement of a computation step. The speed of the processor clock limits the 'speed' at which a PIPS program may be executed.

In order to use the integrated approach to performance prediction, it must be the case that the simulation hardware either be the same as, or be scalable to the hardware on which the proposed system will eventually be executed. On executing a computation step, the *virtual* clock of an LE may possibly be advanced by a duration that is proportional, rather than equal, to the execution time of a computation step. This extension will allow analysts to directly examine the consequence of upgrading some existing hardware by a component that is, for instance, 50% faster. If the simulation hardware and the proposed hardware are radically different, measurements of the computation steps on the simulation hardware are not meaningful. In such situations, all entities in the PIPS program must be mapped to the *null element*. The integrated approach to system design is still useful; however the nature and purpose of the iterative refinements must be modified. As refinements are progressively introduced in the design, performance metrics must also be refined such that they relate only to the portion of the design that is as yet abstract.

We have not addressed the problem of workload characterization, or the related problem that arises when enhancements have to be made to an existing system: how do we incorporate the performance of the existing system into the performance prediction for future systems? This problem is relatively simple, because the approach presented in this papper *relies* on the inclusion of operational modules in performance predictions. As such it is sufficient to build scaffolding around the separately designed, operational modules and treat it as a monolithic entity which interacts with the rest of the system via messages.

A PIPS environment is currently under development on a network consisting of SUN and HP workstations and a 32 node Intel iPSC hypercube. The environment consists of a simulation language, a run-time support system and a program development environment.

A portable, object-oriented simulator is being developed by implementing the primitives described in section 4 in the C programming language. The run-time system has two major responsibilities: implementing interprocess communication and implementing the distributed PIPS algorithm. Remote communication facilities (communication between processes resident on different nodes) are provided by the Cosmic C environment[23], which has been implemented on a number of parallel architectures including the Intel and Amatek Hypercubes as also on a network of SUN workstations. It should be noted that the run-time system could instead be based on some other operating environment. For instance, if complete transparency across different systems was desirable, the IPC could be based on a distributed operating system like LOCUS or CRONUS.

ACKNOWLEDGEMENTS

I am grateful to Dick Muntz for many stimulating discussions and useful suggestions. His comments on an earlier version of this manuscript significantly improved its readability.

References

[1] *Reference Manual for the Ada Programming Language.* United States Department Of Defense, 1982.

[2] R. Bagrodia. *An Environment For the Design and Performance Analysis of Distributed Systems.* PhD thesis, Dept. of Computer Sciences, University of Texas, Austin, Tx 78712., May 1987.

[3] R. Bagrodia, K.M. Chandy, and J. Misra. A message-based approach to discrete-event simulation. *IEEE Transactions on Software Engineering,* SE-13(6):654–665, June 1987.

[4] J.W. Baker, T. Chester, and R.T. Yeh. *Software Developement By Stepwise Refinement.* Report SDBEG-2, Dept. of Computer Science, University of Texas at Austin, January 1978.

[5] K.M. Chandy and J. Misra. *Conditional Knowledge as a Basis for Distributed Simulations.* Technical Report, Dept. of Computer Sciences, California Institute of Technology, Los Angeles., January 1988.

[6] K.M. Chandy, J. Misra, R. Berry, and D. Neuse. The use of performance models in systematic design. In *AFIPS, Proccedings of the National Computer Conference,* 1982.

[7] K.M. Chandy and R. Sherman. *Space-Time and Simulation.* Technical Report, Dept. of Computer Sciences, California Institute of Technology, Los Angeles., 1988. Submitted to Distributed Simulation Conference, 1989.

[8] Myhrhaug B. Dahl O.J. and Nygaard K. *Simula 67 Common Base Language.* Norwegian Computing Centre, Oslo, 1970.

[9] G. Estrin, R. Fenchel, R. Razouk, and M. Vernon. Sara (system architects apprentice): modeling, analysis and simulation support for design of concurrent systems. *IEEE Transactions on Software Engineering,* SE-12(2), February 1986.

[10] W.R. Franta. *The Process View Of Simulation.* Elsevier North-Hollabd Inc., New York, 1977.

[11] R. Fujimoto. Lookahead in parallel discrete event simulation. In *International Conference on Parallel Processing,* August 1988.

[12] C.A.R. Hoare. Communicating sequential processes. *CACM*, 21(8):666–677, August 1978.

[13] D. Jefferson and H. Sowizral. Fast concurrent simulation using the time-warp mechanism. In *Distributed Simulation 1985, Society for Computer Simulation Multiconference*, San Diego, 1985.

[14] Sanguinetti John. A technique for integrating simulation and system design. In *Conference on Simulation, Measurement and Modelling of Computer Systems*, Boulder, Colorado, August 1979.

[15] J. Joyce, G. Lomow, K. Slind, and B. Unger. *Monitoring Distributed Systems.* Project Jade Report No. J85/9/1, Dept. Of Computer Science, University of Calgary, Calgary, Canada, May 1985.

[16] B.P. Miller. *Performance Characterization of Distributed Programs.* Ph.D. thesis, University of California at Berkley, January 1985.

[17] J. Misra. Distributed discrete-event simulation. *Computing Surveys*, 18(1), March 1986.

[18] D. Parnas. Sodas and a methodology for system design. In *AFIPS Conference Proceedings*, 1967.

[19] D.A. Reed, A.D. Malony, and B.D. McCredie. Parallel discrete event simulation: a shared memory approach. In *Proceedings of the 1987 ACM SIGMETRICS Conference*, pages 36–39, May 1987.

[20] W.E. Riddle. *The Modeling And Analysis Of Supervisory Systems.* Ph.D. thesis, Stanford University, March 1972.

[21] W.E. Riddle, J.C. Wileden, J.H. Sayler, A.R. Segal, and A.M. Stavely. Behaviour modeling during software design. *IEEE Software Engg.*, SE-4(4), July 1978.

[22] G.C. Roman. Specifying software/hardware interactions in distributed systems. In *Proceedings of the International Conference on Software Engineering*, pages 126–139, May 1987.

[23] C.L. Seitz, J. Seizovic, and Wen-King Su. *The C Programmer's Abbreviated Guide to Multicomputer Programming.* Technical Report Caltech-CS-TR-88-1, Dept. of Computer Sciences, California Institute of Technology, Los Angeles., January 1988.

[24] C. Smith and J.C. Browne. Aspects of software design analysis : concurrency and blocking. In *Performance '80*, May 1980.

[25] J.G. Vaucher. A wait-until algorithm for general purpose simulation languages. In *Winter Simulation Conference*, pages 177–183, 1973.

[26] F. Zurcher and Randell. Iterative, multi-level modelling - a methodology for computer system design. In *Proceedings IFIP Congress 68*, 1968.