Chapter 6

VCE Design Validation

In order to demonstrate that the VCE design is implementable and demonstrate and quantify its value, two prototype systems and a simulation model have been constructed. The focus of these experimental efforts has been on studying the scheduling, dispatching and load balancing methodologies described in Section 5.1. An interpreter for the *application description framework* syntax described in Chapter 4 has also been implemented and used as a component in one of the prototypes and in the simulator.

All three of these systems serve different purposes in validating the VCE design. The first prototyping effort involved building a small version of the VCE Execution Suite on a workstation cluster. This prototype included implementations of all the major Execution Suite components described in Section 3.4.1 (scheduling/dispatching daemons, execution processes, group leaders, etc.). The goal of this first system was to gain experience with the construction of these components and refine their interfaces. It also was used for experimentation with the VCE load measurement algorithm (Section 5.1.2) and to experiment with redundant scheduling (Section 5.1.2) and low priority scheduling (Section 5.1.2).

The second prototype demonstrates the value of the VCE scheduling and application description techniques in managing the execution of multi-stage applications on multiple high performance parallel computers. This prototype includes a more robust execution process which can parse application and task description files.

A simulation model was built to allow for more controlled load balancing experiments than could be performed on the prototype systems using shared hardware. The simulation system was developed using an object oriented approach and utilizes a processor simulation algorithm which more closely matches actual processor behavior than many commonly used techniques (see Section 6.3.2).

This chapter will describe the prototypes and simulation system. Preliminary performance results gathered from experiments with all three systems will be discussed.

6.1 A Workstation Cluster Based Prototype

The simplest possible VCE configuration involves a single cluster of homogeneous workstations. Such a configuration was used to construct the first VCE prototype [90]. While this configuration does not include some features of the VCE, such as exploitation of multiple high-performance parallel machines, it was useful to establish designs for the Execution Suite components and implement and exercise their interfaces. Further, it may be beneficial to demonstrate that the Execution Suite is capable of "scaling down". That is, that it can be useful in very small systems as well as very large ones. Finally, since networked workstations are common and widely underutilized (see Section 1.1.3), demonstrating that the VCE can effectively use their excess capacity will make it attractive to a larger user community.

6.1.1 Scope of the Prototype

A schematic showing all the components of the VCE was given in Figure 2. Figure 20 showed just the components that constitute the VCE execution suite. The goal of the VCE workstation cluster prototype was to implement a single group and execution process. These components were used to validate the scheduling algorithms discussed in Section 5.1 and demonstrate their applicability to a workstation cluster environment. The application description framework discussed in Chapter 4 was not implemented for this prototype. Instead, the dispatching sequences of three test applications (Section 6.1.5) were hard coded into the prototype execution processes.

To allow for rapid prototyping, the system uses the Isis Distributed Toolkit [19] for communication among the scheduling/dispatching daemons, group leader and execution process rather than the World Wide Web technology mentioned in Section 5.2. Isis is particularly well suited to a workstation based VCE implementation. It supports the concept of a *process group*. When processes join an Isis process group they are supplied with information about other members of the group. It is natural to implement a VCE scheduling/dispatching daemon group as an Isis group. When a process joins the group it can use the information provided by Isis to determine which scheduling/dispatching daemon was initialized first. This daemon serves as the group leader. Isis also allows non-group members to broadcast messages to a group. This function is used by the execution process to send its machine requests to the group leader.

Because this prototyping effort was focused on studying scheduling and dispatching issues, the inter-task communication capabilities described in Section 5.2 were not implemented. Application components exchanged data using Isis functions or shared files.

6.1.2 Low Priority Dispatching

The VCE design seeks to insure that conventional (non-VCE) users of workstations do not experience poor performance because their machines are being used to host VCE tasks. VCE tasks should only consume excess workstation capacity. This is accomplished by dispatching VCE tasks on shared workstations at very low priority (Section 5.1.2). VCE tasks are only placed on hosts that are lightly loaded when the task is initiated. If other workstation users increases their use of the machine, the operating system will reduce the amount of resources made available to the low priority VCE task [8] so that its imact on the workstation users will be negligable. Of course, this causes starvation of the VCE task. Redundant scheduling is employed so that the VCE application user will experience performance equivalent to that provided by the least loaded host.

6.1.3 Workstation Load Measurement

Section 5.1.2 introduced the general VCE load measurement algorithms and their underlying philosophy. These algorithms involve taking periodic snap shots of the processor load state, using several consecutive snap shots to derive a "smoothed load average" and determining how heavily the machine is loaded relative to its *nuisance threshold* (the load level at which users notice poor performance). This section will discuss those algorithms were implemented for the VCE workstation cluster prototype.

A workstation's run queue length is generally a useful measure of system load [69]. Experience with the VCE has confirmed the utility of monitoring the workstation run queue length, however, by itself, it does not adequately characterize the state of a workstation. Specifically, a machine executing several I/O bound jobs may experience performance degradations. These degradations are caused by competition for I/O facilities rather than competition for the CPU and are not detected by monitoring the length of the run queue. A load measurement limited to examination of the run queue might incorrectly indicate that a workstation being used for local program compilation or document processing had excess capacity. To compensate for this deficiency, the VCE monitors both the amount of interrupt activity (which is indicative of the I/O load) and the run queue length.

Before a machine is added to the VCE workstation cluster prototype, a one time experiment is performed. A large number of I/O bound jobs are run concurrently on the machine. The number of system interrupts resulting from this load is noted and establishes the I/O capacity of the machine. This experiment is performed only once for each machine. The results are recorded and used subsequently by the scheduling algorithms. Because the experiment measures a hardware characteristic (I/O capacity) it does not need to be repeated.

After a machine is added to the VCE the *rstat()* function is called every six seconds to sample the average run queue length and number of system interrupts which occurred during the preceding interval. If the average length of the run queue during the interval was greater than two (indicating significant CPU contention) or I/O activity was greater than two-thirds of capacity (the capacity was estimated experimentally as described above) then the load on the machine is considered high. At this load level performance degradations are noticeable. Conversely, if the average run queue length is less than one (indicating the CPU was idle for some period) and I/O activity was less than one-third of capacity then the load is considered low. When load conditions are between these two extremes the load is considered to be adequate.

Figure 28 showed the general algorithm for processor load measurement in the VCE. The algorithm involves taking periodic snapshots of the load on the machine (*instantaneous* load measurements). A *smoothed* load measure is adjusted any time several consecutive instantaneous load readings are higher or lower than the current smoothed load measure. For workstations the algorithm is customized as follows:

- The sampling interval is six seconds.
- The smoothed load measure is raised if the instantaneous load measure is higher for two consecutive intervals.
- the smoothed load measure is lowered if the instantaneous load measure is lower for 10 consecutive intervals.

Six second sampling intervals are used for the following reasons:

- Some Unix systems only update load measurements every five seconds, so more frequent sampling may not be useful.
- Run queue lengths are reported in terms of a one minute average. Values for shorter periods are estimated by extrapolating between consecutive samples. Sampling every tenth of a minute simplifies this computation.
- Longer sampling intervals increase the chance of placement errors because it takes the system longer to notice and react to changing conditions.

As described in Section 5.1.2, The algorithm is designed to react to increases in local load faster than decreases. Increases in load cause performance degradations for local tasks. The VCE must act quickly to avoid starting remote tasks on a machine where the local load is increasing. However, after a period of high or medium load conditions a machine should not be considered eligible to host remote tasks until a sustained period of low load conditions is observed. The criterion for raising the smoothed measure is observation of two consecutive intervals of higher load. This was selected because it insures that, if a large job is started on a workstation, the workstation will be removed from the group leader's list of eligible machines within 12 seconds. The machine will accept tasks again after 10 consecutive intervals of low load.

6.1.4 Task Placement

In a workstation group the group leader maintains a list of machines with low load. When an execution process wishes to place tasks in a group it sends a message to the group leader with the number of machines needed. The group leader responds with a list of available machines.

6.1.5 Experimentation

Three test scenarios have been constructed to exercise various capabilities of the prototype. One is a numerical quadrature system. This application was used mainly to test the dispatching and communication facilities in the VCE. The execution times of the constituent tasks were only a few seconds, so this application did not provide much insight into the performance characteristics of the scheduling system.

In order to gauge the performance impact of redundant scheduling, a benchmark application was constructed. The application is made of two tasks. Both tasks execute an identical number of integer arithmetic operations and exit. The sizes of the tasks were chosen so they would be course grain and run on the order of hours on workstations. The performance data given in Table 4 summarizes data gathered

Table 4: Performance Benchmark

	Average Run Time (Minutes)	% Improvement
Random Placement	150	0
Selective Placement	108	28
2-Way Redundancy	102	32
3-Way Redundancy	85	43
4-Way Redundancy	82	45
5-Way Redundancy	82	45

from more than 30 runs of the benchmark with various levels of redundancy. The random placement entry in Table 4 corresponds to a policy of running one instance of each task on a randomly selected workstation. This policy occasionally results in the selection of a very heavily loaded machine and results in poor performance. Placing tasks on machines that are initially lightly loaded ("Selective Placement" in Table 4) improves performance considerably. Redundant task placements improve performance even more. The data shows that raising redundancy from two machines per task to three yields a substantial performance improvement. This is because, with three machines per task, the probability that at least one instance of each task will be running on a lightly loaded machine for a substantial proportion of its execution time is high. The performance benefits of using more than three machines per task are minimal.

The third application used to test the prototype is a text analysis system. Each task in the application examines a different text database and determines how many occurrences of a target string it contains. The tasks are of varying sizes. Two process databases containing about 240K bytes of text. The third searches 160K bytes of text.

This text processing application is representative of a class of applications which will be common in network based computing environments. The expansion of the

Table 5: Text Processing Application Performance

	Average Run Time (Minutes)	% Improvement
Random Placement	81	0
Selective Placement	57	30
3-Way Redundancy	48	41

internet will make vast text databases widely available. The VCE, with its redundant task scheduling capabilities, may provide an attractive platform for applications that process or analyze such data.

This application functions well in a redundant execution environment. All tasks accept their input from read-only files. Contention for the input files does not degrade performance because the distributed file system caches the data efficiently and the ratio of processing time to I/O time is high. When each task completes, it writes a single record of results. If one task writes its output and a second instance of the same task writes its output before the system can terminate it, this does not cause an error. The second write will overlay the first with identical correct data.

Table 5 shows average performance results when the application was run with random task placement, selective task placement and three-way redundancy.

6.2 A Multiple Parallel Computer Based Prototype

A somewhat more relevant VCE configuration is one that includes the participation of multiple parallel supercomputers. A VCE prototype has been developed to manage execution of multi-stage VCE applications in such an environment [92]. As in the workstation cluster based prototype, the focus is on studying the load balancing approaches layed out in Section 5.1. This prototype also implements the *application description framework* syntax of Chapter 4. The application description interpreter makes it easier to experiment with more complex applications and gives a basis to judge the expressiveness and implementability of the syntax.

6.2.1 Scope of the Prototype

Because Isis is not available on some of the platforms used in this prototype, a different message passing tool, p4 [22], was used for communication among the execution suite components. P4 is not as conducive to the VCE structure as Isis. Isis supports a group based communication model which lends itself naturally to the VCE design's process group orientation. The communication paradigm supported by p4 is less dynamic. P4 applications are structured as follows: a master program is started which then starts a group of slaves that can communicate with each other and with the master. The number of slaves is constant and must be established when the master program is initialized. In the VCE the scheduling/dispatching daemons run continuously and the execution process communicates with them when a user requests application execution and as the application executes. The daemons are not slaves to the execution process nor is the execution process a slave to the daemons.

For the purposes of this prototype study, the p4 communication functions were supplemented by Unix sockets [96] calls to overcome the limitations of p4's master/slave communication model. Specifically, the load monitoring routines are incorporated into separate daemons. These daemons perform the load monitoring algorithm and provide the current load state to client programs using a sockets interface. When a VCE application is to be run, the execution process starts a scheduling/dispatching routine on each machine. The scheduling/dispatching routine obtains its host's current load conditions from the load monitoring daemon and notifies the execution process if the load conditions will permit it to host a VCE task. This structure is inferior to the VCE design specified in Chapter 3 because excessive overhead is incurred when the execution process is initiated. Also, if several users were executing VCE applications simultaneously multiple scheduling/dispatching routines would be running on each host.

Of course, in production VCE systems VCE components will communicate using more robust distributed object technologies [1, 17, 59, 82]. Since the primary objective of both prototypes was to study scheduling and load balancing issues, fully implementing the communication design of Section 5.2 was not necessary.

While the communication daemons that will support intertask communication in the VCE are not implemented in this prototype, some communication facilities beyond those present in the workstation cluster based prototype are included. Specifically, this prototype manages data files on behalf of application tasks. When a task is redundantly dispatched, any files that are indicated by the application description as being input to that task, are automatically replicated in the local file systems of each host. Similarly, when a task instance completes, its output files are made available to subsequent application stages. Partially completed files of cancelled task instances are discarded. This makes redundant dispatching transparent as regards file I/O operations.

6.2.2 Hardware Platforms

Several supercomputers and high performance workstation clusters at the Northeast Parallel Architectures Center (NPAC) have been included in this prototype system. The NPAC environment consists of a wide set of state-of-the-art multi-computer systems (see Figure 42). The platforms used are briefly described below:

IBM SP-2: The SP-2 consists of a cluster of 16 RISC/6000 370 nodes interconnected by a crossbar switch (Allnode). Each node runs at a clock rate of 62.5 MHz.

ALPHA/FDDI: The ALPHA/FDDI configuration consists of 8 DEC ALPHA workstations interconnected by a high performance (100 Mbps) backbone composed of dedicated, switched FDDI segments. The ALPHA nodes have a clock rate of 150 MHz.

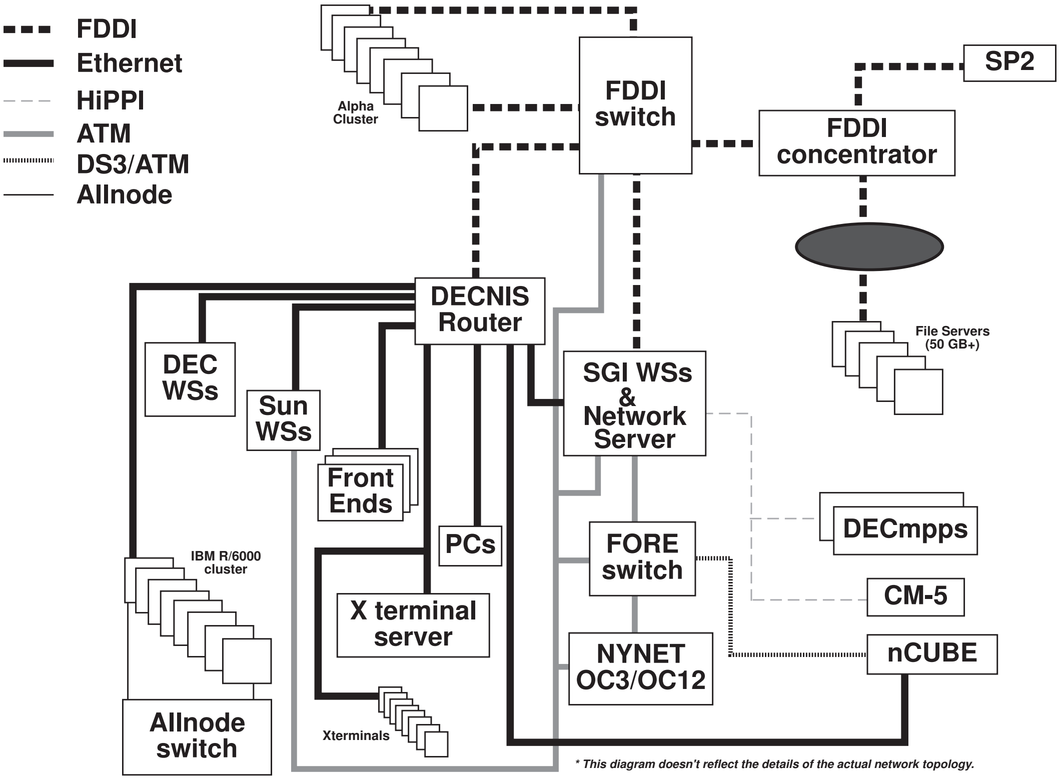

Figure 42: Computing Environment at NPAC

ATM LAN: This configuration consists of 2 SUN SPARCstation IPXs and 3 SGI workstation communicating over an ATM Local Area Network using ATM FORE ASX-100 switch. SUN IPX nodes operate on an approximately 40 MHz clock. Host computers are connected to the ATM switch through Fore's 200 series adaptors. Each adaptor has a dedicated Intel i960 processor (running at 25 MHz) to support segmentation and reassemble functions and to manage data transfer between the adaptor and the host computer. The 200 series adaptors also have special hardware for AAL CRC and special-purpose DMA hardware.

6.2.3 Applications

Three distributed applications have been implemented on the multiple parallel computer based prototype and have been used to test the runtime support.

Linear Equation Solver

A linear equation solver constructed from a set of general purpose linear algebra tasks (shown in Figure 26) was discussed in Chapter 4.

Multi-target Tracking System

The Multi-target tracking system demonstrates the multi target tracking capabilities that is required by a Battle Management Command Control and Communication System [56]. The multi target tracker, shown in Figure 43, is designed to provide an estimation of launch vehicle parameters for individual targets/missiles in multi-target scenarios. The tracker receives input in terms of launch sites for missiles or targets. The launch sites are specified in terms of latitudes and longitudes. This information is fed to two focal plane tracking (FPT) modules (2 dimensional tracking) at 5 second intervals. The focal plane tracking modules process this data using kinematic filtering algorithms and track pruning and prediction algorithms. The output of this module is an initial prediction of trajectories of launched missiles. This data is then fed to a three dimensional tracking system which uses the data from the two focal plane tracking modules to prune duplicate tracks (if any), extend existing tracks, prune bad tracks and initiate new tracks. The output of the system is a list of target trajectories.

JPEG Compression/Decompression

JPEG (Joint Photographic Experts Group) is an emerging standard for image compression. JPEG standard aims to be generic and can support a wide variety of applications for continuous-tone images. We have used data parallel programming model to implement a distributed JPEG algorithm on a cluster of workstations. In this implementation half of the computers participate in compression of an image file

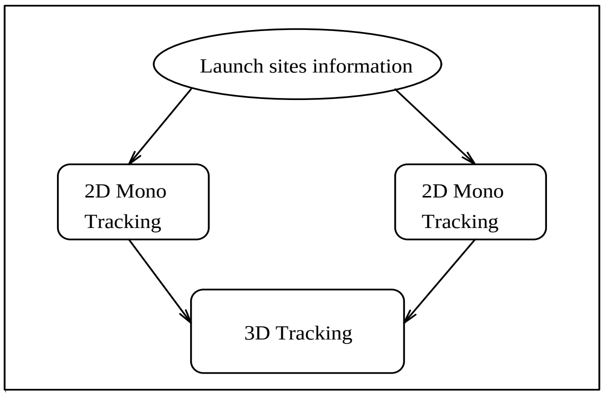

Figure 43: Multi-Target Tracker System

while the second half reconstruct the compressed image. The image to be compressed is divided into N/2 equal parts (where N denotes the number of processors) by the master process and then shipped to one half of the processors. Each processor performs the sequential JPEG compression algorithm on its portion of the image. After compression the processors send the compressed image to another set of N/2 processors which perform the decompression. Once decompression is done, the results are sent back to the master process which combines them into one image. Consequently, this algorithm involves five stages *viz.* distribution of uncompressed image , compression of the image, transmission of compressed image, decompression of the image, and finally combining the decompressed images.

6.2.4 Performance

Of the three VCE applications described in in Section 6.2.3 the linear equation solver is most representative of the kind of application that will benefit from development and execution in the VCE. It is constructed from tasks which could be reused in other applications and for large data sets it has a long running time. So the developer benefits from the application description framework and the user benefits from the automatic runtime scheduling and dispatching capabilities.

A series of performance tests using the linear equation solver application were performed to determine under what conditions running an application in the VCE with the scheduling protocols described in Section 5.1 would be better than running it in a single computer. The IBM SP2 and the DEC ALPHA workstation cluster have similar performance characteristics [55], so often a user who has an application that could run on either platform would simply choose one at random. Observations in [55] suggest that even a moderate difference in load conditions at runtime can effect performance enough to impose a considerable penalty for choosing the wrong machine. Performance experiments done with the VCE support these findings and show that redundant scheduling will reduce turnaround times in many situations.

In order to obtain controlled testing conditions, testing was performed on the DEC ALPHA cluster and the IBM SP2 when transient load conditions were low. A load generating utility was used to slightly elevate the load on one or the other as the tests were performed. The load generator allows load conditions to be varied between *low* and *moderate*. It places two dummy processes on each node in the configuration for a randomly determined period to simulate an interval of high activity. These *busy* periods are alternated with randomly determined *idle* periods where no artificial load is placed on the machine. When *low* load is specified, the busy periods vary from four to eight minutes while the idle periods are between 20 and 30 minutes. To simulate *moderate* load, idle periods are between four and eight minutes and busy periods are from 20 to 30 minutes.

The results of performance tests run using the linear equation solver application to solve a system of 1024 equations are shown in Table 6. The table shows application performance on each platform under low and moderate load conditions. It also shows performance where instances of the application tasks are placed on both machines ("Redundant Scheduling") or placed only on the machine that is less loaded at the time the tasks start ("Selective Scheduling"). The table shows the degradation which results when just two other processes per node are competing for computing resources. Neither redundant nor selective scheduling will have a significant impact on performance when the loads on all the candidate machines are roughly equivalent. That is, if all machines are heavily loaded, or all machines are lightly loaded, there

Table 6: Performance of the Linear Equation Solver Application

	Load	Execution time in seconds
SP2	Moderate	1153
	Low	849
ALPHA	Moderate	1440
	Low	732
Reduntant Scheduling	SP2-Moderate ALPHA-Low	741
	SP2-Low ALPHA-Moderate	868
Selective Scheduling	SP2-Moderate ALPHA-Low	744
	SP2-Low ALPHA-Moderate	886

is little a scheduling system can do to improve performance. However, when some candidate machines are less active than others, both the scheduling approaches studied tend to prevent the performance loss that could be encountered with a random placement.

Redundant scheduling yields slightly better performance than selective scheduling. If there is a significant load change after a task is placed with selective scheduling performance will degrade. Redundant scheduling insures that the best possible performance will be obtained because all candidate machines run the tasks and the result produced by the fastest machine is returned to the user or made available for subsequent computations. Of course, redundant scheduling incurs higher costs for the application because the computations performed by some machines are discarded. These costs are somewhat offset by eliminating the overhead required for gathering load measurements.

Both of the scheduling protocols insulate the user from the effects of poor random placements without imposing the considerable overheads associated with process migration or checkpointing techniques.

In production Virtual Computing Environments it is expected that the choice between redundant and selective task scheduling would be based on the nature and frequency of VCE jobs to be executed. If VCE applications are submitted only occasionally, or tend to be of short duration then the use of redundant scheduling would be preferred. If many large VCE jobs are to be executed then selective scheduling would provide lower costs and preserve improved performance.

6.3 A VCE Simulation Model

The VCE design incorporates many features which are not commonly found in distributed computing environments. These features include a simplified load measurement and information exchange policy, group oriented load balancing and redundant scheduling techniques. Because the approaches and technologies employed in the VCE are novel, a variety of experiments have been performed to understand the characteristics of the system and quantitatively measure its utility. Two different avenues have been used to gather data on the system's performance. A series of experimental prototype systems have been implemented and a simulation model has been developed. This section describes the simulation model which has been constructed to study VCE scheduling and load balancing policies. The simulation is constructed to model each processing element of parallel computers and each workstation in workstation clusters. Multiple supercomputers and workstation clusters can be modeled in each simulation run. The model employs an object oriented approach where parallel computers and workstation clusters are formed as aggregations of simple processor objects. Similarly, parallel applications are modeled as aggregations of sequential tasks and VCE applications are constructed as aggregations of parallel applications. Simulated processors use a round robin scheduling policy which is more realistic than the first-come first-served approach used in many other simulation studies. The model of a parallel application relaxes the common assumption that task arrivals at each node are independent.

6.3.1 Evaluating Distributed System Designs

There are three techniques which are commonly used to evaluate distributed system designs: analysis, simulation and prototyping [57]. Both simulation and prototyping have been used to explore various aspects of the VCE design. Mathematical (queuing) studies have not been performed. The limitations of mathematical modeling make it particularly difficult to use in studying VCE configurations. For instance, analytical models characterize processors as servicing tasks on a first-come first-served (FCFS) basis [57, 66, 77, 93]. Assuming FCFS scheduling protocols allows a distributed system to be characterized as a network of M/M/1 queues. The queuing theoretic properties of such systems are well understood [57]. Unfortunately, the FCFS assumption usually does not hold because most operating systems use round robin or prioritized scheduling [100]. Inappropriately assuming that distributed processing elements provide FCFS service can lead to invalid results [34].

Another shortcoming of mathematical modeling is that arrivals at each node of a system are assumed to be independent [57, 66, 77, 93]. However, in a high performance computing environment where parallel applications may have components distributed across many nodes, this assumption also may not hold.

These questionable assumptions commonly made in analytical models of distributed systems (FCFS node processing and independent task arrivals) are also made in many published simulation studies [3, 4, 39, 50, 79, 84, 95, 110]. However, the VCE simulation model described below incorporates more reasonable characterizations of these parameters. In this way, simulation provides a more flexible means for studying system performance than does rigorous mathematical analysis. Parameters like scheduling discipline and task arrival behavior can be modified to reflect conditions of the systems of interest in a simulation. With analytical models these assumptions are required by the queuing theory which underlies the methodology [57, 93].

It might seem that prototyping experiments alone would be sufficient to characterize and validate the design of the VCE. This is not the case. The performance of VCE applications, as well as their impact on the non-VCE applications running on the same hardware, must be precisely measured. Performing controlled experiments on hardware is difficult. Acquiring several parallel computers simultaneously and reserving them for the exclusive use of this research effort is not practical. Further, the instrumentation needed to measure the quantities of interest could themselves impact the outcomes of the experiments [57]. Finally, since network attached high performance computers are expected to be more common in the near future, it is desirable to demonstrate the scalability of the VCE computing approach. Obviously, it is not possible to show experimentally that the VCE (or any computing system) will perform well on hardware configurations larger than those that currently exist.

Prototyping has played an important role in evaluating several aspects of the VCE. It has been used to validate simulation results and identify issues that must be addressed when a production VCE system is developed. However, the simulation model provides opportunities to observe how system behavior is impacted by subtle changes in parameters. These kinds of observations are difficult to perform using prototype systems alone.

6.3.2 Basic Elements of the Simulation Model

The VCE simulation software is a discrete event simulator written in an object oriented style in C++ [97]. Simulation software is extremely amenable to object oriented design methodologies. The motivation for the programming language "C with Classes", which was a direct precursor to C++, was to facilitate development of efficient distributed system simulation programs [98]. A simulation program describes (models) objects in the real world and how they interact with one another. It is natural to write a single module to represent each object being simulated and to encapsulate the behavior of the object within the module. This encapsulation maximizes the flexibility and modifiability of the simulator. The `processor` class, which is described below, models a CPU time-shared among many tasks using round robin scheduling. The object oriented structure of the simulator makes it easy to change features like this, so changing the model to use FCFS scheduling would only require a small change to a single module. This modifiablity will ease future enhancement of the VCE by allowing new hardware elements and software approaches to be quickly reflected in the simulator.

The simulation model uses a C language [62] library called **smpl** [75]. The library provides functions to manage an event queue. An event queue is a linked list. Each item on the list represents an event which is going to occur later in the simulation. Each list item contains an indicator of the type of event and the time at which it is scheduled to occur. **smpl** functions are provided to add an event to the list (schedule it), remove an event from the list (cancel it), and fetch the event which is to occur next (cause it). The **smpl** library also provides functions to generate random variables from various distributions, collect statistics about simulation objects and format reports.

The simulator employs a class hierarchy. The simplest classes are *tasks* and *processors*. Processors represent a single CPU. Tasks are sequential jobs which run on a single processor. More complex classes are achieved by aggregating these classes. Supercomputers and workstation clusters contain arrays of processors while parallel applications are modeled using arrays of tasks and VCE applications are constructed as arrays of parallel applications. Details of all these constructs are discussed in this section.

Besides being useful as a performance evaluation tool, the simulator also has been helpful in formulating the overall VCE design. It has served as a means to document the assumptions of the design and to experiment with and debug algorithms before they are implemented in prototypes.

Processor Modeling Issues

The most fundamental constructs in the VCE simulation are processors and tasks. Every supercomputer and cluster in a simulation contains processors which can execute

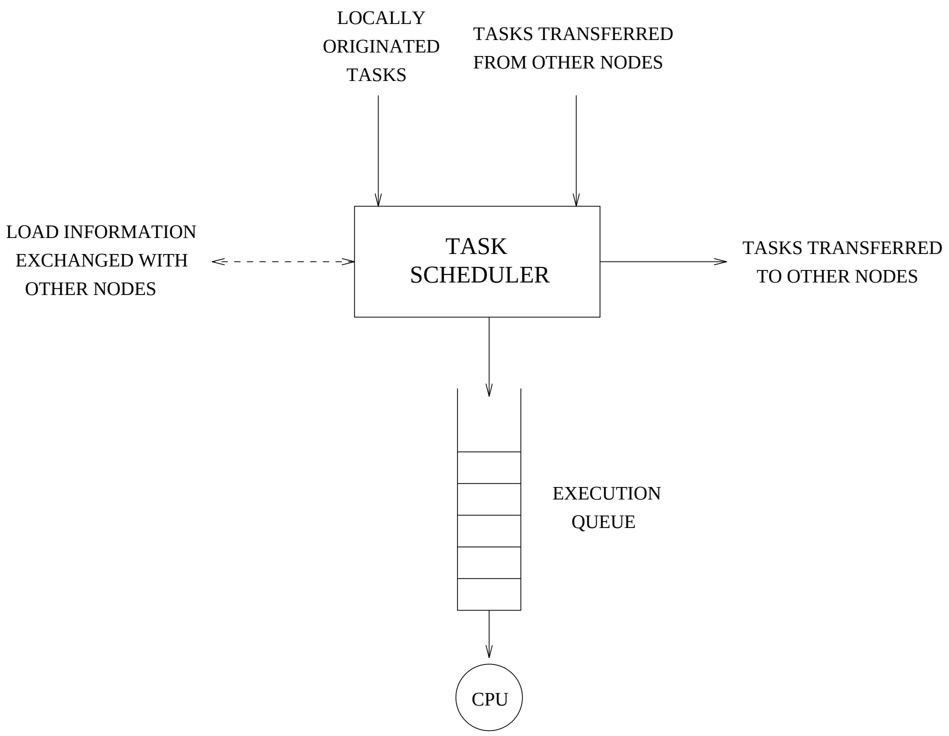

Figure 44: A Typical Node Model

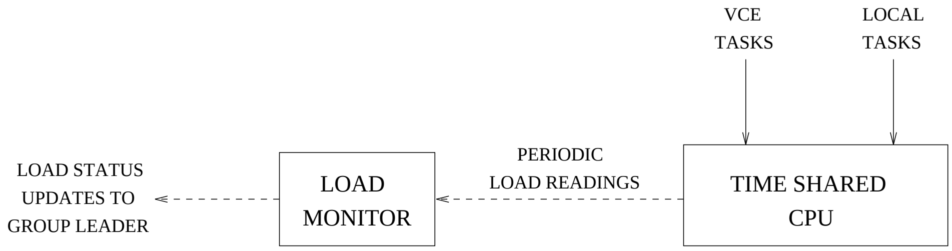

Figure 45: The VCE Node Model

tasks. As was noted earlier, most distributed system simulators used to study load balancing algorithms assume a first-come first-served processor scheduling model [3, 4, 39, 50, 79, 84, 95, 110]. Figure 44 shows how processors are typically modeled in such studies. The *task scheduler* in the figure is a software process which implements the processor load measurement technique, information exchange policy, transfer policy and location policy (see Section 5.1.2) of the load balancing algorithm being considered. Tasks arrive at the node due to actions of a local user or as a result of transfers from heavily loaded nodes. When a task arrives the task scheduler performs the load balancing algorithm and either adds the task to the queue of the local processor or transfers it to another node.

The node model of Figure 44 is not used in the VCE model. The FCFS protocol implied by the figure is not appropriate. If the processor being considered is a high performance workstation or an element of a MIMD parallel computer, it is very likely that processor resources are shared among all locally executing tasks on a round robin basis [100]. The choice of whether to model processors as time shared resources or FCFS queues has a significant impact on the simulation results [34]. In FCFS models, once a task is scheduled on a node its completion time is a function of its running time and the running times of the tasks queued ahead of it. It is unaffected by subsequent load changes. On a time shared processor each time a task is accepted for execution the performance of all the tasks already executing on that node is degraded.

While most load balancing simulations adopt a FCFS node model, several studies have considered a round robin local scheduling protocol [34, 108, 110]. The node model used in the VCE simulation is shown in Figure 45. The *Load Monitor*, in a real VCE implementation, is a process within the scheduling/dispatching daemon which implements the processor load measurement algorithm described in Section 5.1.2. The processor load measurement policy determines whether the node is eligible to accept VCE tasks. When the load monitor detects a significant and sustained change in the local load, a message indicating the new load level is sent to the group leader. Tasks are generated either locally or are submitted by remote VCE execution processes when the node is lightly loaded.

Processor and Task Simulation Algorithm

In the VCE simulation model each processor $\mathbf{P^n}$ and task $\mathbf{T^m}$ is assigned a unique integer identifier. The static characteristics of processors and tasks are: $\mathbf{P^n_{sp}}$, the speed of processor n, and $\mathbf{T^m_{sz}}$, the size of task m. The latter quantity is defined as the time that would be required to complete task T^m as the only task running on a processor of speed 1.0.

The following dynamic quantities can be measured at any time during the simulation:

CurrentTime The current simulation time.

$\mathbf{P^n_{ld}}$ The current load on processor P^n (that is, the number of tasks it is running).

$\mathbf{P^n_{cm}(T^m)}$ The proportion of task T^m which has been completed on processor P^n.

$\mathbf{P^n_{ex}(T^m)}$ The expected time of completion of task T^m on processor P^n.

The time required to execute a task on a specific processor is called the $lifespan$ of the task. The lifespan increases if the load on the processor is high and is influenced by the speed of the processor. More formally, the lifespan of task T^m on processor P^n, $P^n_{ls}(T^m)$, is computed as:

$$P^n_{ls}(T^m) = \frac{T^m_{sz}\,P^n_{ld}}{P^n_{sp}}. \tag{1}$$

Three types of events can occur on a processor: a task can begin execution, a task can complete, or a task can be cancelled. Each of these events results in an ordinal change in the load on the processor. When any of these events occurs $CurrentTime$ and P^n_{ld} are updated.

When a task begins on a processor its load is incremented and the task's initial expected completion time is computed as:

$$P^n_{ex}(T^m) = CurrentTime + P^n_{ls}(T^m). \tag{2}$$

An *interval* is the time between consecutive events on a processor. The duration of intervals is variable and may be zero. When an event occurs the length of the terminating interval is computed and stored in the variable **int**. Before the load of the processor is incremented or decremented the amount of progress made by each task is calculated using the function:

$$P^n_{cm}(T^m) = P^n_{cm}(T^m)^{old} + \frac{int}{P^n_{ls}(T^m)}. \tag{3}$$

That is, The duration of the interval is divided by the total time needed to complete the task to determine what portion of the task was completed during the interval. This is added to the portion of the task which was completed in prior intervals to obtain the total amount of progress made on the task so far.

Once the processor load is updated and Equation 2 is performed for newly arriving tasks, a new expected completion times must be computed for any task which was executing on the processor in the previous interval and has not yet completed. The updated load value, P^n_{ld} is used to calculate a new lifespan, $P^n_{ls}(T^m)$, for each task using Equation 1. Since $P^n_{ls}(T^m)$ represents the total time needed to execute the task and $P^n_{cm}(T^m)$ represents the fraction of that time which has already elapsed, the function which gives the new expected completion time for each task is:

$$P^n_{ex}(T^m) = CurrentTime + (1 - P^n_{cm}(T^m)) \cdot P^n_{ls}(T^m) \tag{4}$$

Figure 46 shows the complete processor simulation algorithm. The algorithm is executed each time an event occurs on a processor (a task arrival, completion or

for each running task
  compute proportion completed, $P^n_{cm}(T^m)$, using Equations 1 and 3

update the processor load P^n_{ld}

for each arriving task
  compute the initial expected completion time, $P^n_{ex}(T^m)$,
  using Equations 1 and 2

for each running task
  compute updated expected completion time, $P^n_{ex}(T^m)$,
  using equations 1 and 4

Figure 46: Processor Event Simulation Algorithm

Table 7: Example Simulated Tasks

Task	Size (T^m_{sz})	Arrival Time
1	20	0
2	5	10
3	10	14

Table 8: Processor Simulation Example

		Updated Simulation Variables						
Time	Event	P^1_{ld}	$P^1_{cm}(T^1)$	$P^1_{ex}(T^1)$	$P^1_{cm}(T^2)$	$P^1_{ex}(T^2)$	$P^1_{cm}(T^3)$	$P^1_{ex}(T^3)$
0	T^1 arrives	1	0	20				
10	T^2 arrives	2	.5	30	0	20		
14	T^3 arrives	3	.6	38	.4	23	0	34
23	T^2 completes	2	.75	33	1	23	.3	37
33	T^1 completes	1	1	33	1	23	.8	35
35	T^3 completes	0	1	33	1	23	1	35

cancellation). The algorithm produces a new expected completion time for each task resident on the processor after the event. Once the expected completion time is known for all running tasks smpl routines are called to schedule the next completion event.

Processor Simulation Example

To clarify these concepts the three tasks in Table 7 will be used in a small example of how the behavior of an individual processor is modeled in the VCE simulation. The table shows arrival times and sizes of the tasks. In an actual simulation these values are determined using pseudorandom variables (Section 6.3.5).

Table 8 shows the events and calculations which would occur when execution of the tasks is simulated on processor P^1 whose processing speed, which is a user specified simulation parameter, is 1.0 ($P^1_{sp} = 1$). When the first task arrives at time 0 it is beginning execution so no part of the task has been executed ($P^1_{cm}(T^1) = 0$). Since it is the only task on the processor and $CurrentTime = 0$ Equations 1 and 2 reduce to $P^1_{ex}(T^1) = T^1_{sz} = 20$. When the second task arrives ten time units later, Equation 3 shows that the first task is half done ($P^1_{cm}(T^1) = .5$). The arrival of T^2 causes the load on the processor to go to two ($P^1_{ld} = 2$). Now the processor is being shared by two tasks, so the lifespan of both is doubled (Equation 1). The initial expected completion time of T^2, $P^1_{ex}(T^2)$, is set to 20 using Equation 2 and the expected completion time of T^1, $P^1_{ex}(T^1)$, is updated to 30 by Equation 4. The simulation continues in this fashion with the portion of each task that has been completed ($P^n_{cm}(T^m)$) and its expected completion time ($P^n_{ex}(T^m)$) being recomputed each time a task arrives or completes. When a task completes its expected completion time ($P^n_{ex}(T^m)$) becomes an actual completion time.

Simulating Low Priority Tasks

In some cases the VCE design requires VCE tasks to be scheduled at low priority. Therefore, low priority tasks must be accommodated by the simulation model. This is done by maintaining a separate queue of low priority and normal priority tasks. When a simulated processor has both normal and low priority tasks running the low priority tasks are disregarded. That is, they are not figured in to the processor load and they do not make progress towards completion. When a processor has only low priority tasks, they are processed using the time sharing algorithms described above. While most operating systems attempt to give some resources to low priority tasks [100], very low priority CPU-bound tasks competing against higher priority CPU-bound tasks in Unix based systems will make minimal progress [8].

6.3.3 Parallel Applications

The processor and task models described above can be used in a straightforward way to model a MIMD parallel computer. For the purposes of the simulation study, a parallel machine is considered to be a collection of processors with identical speed values. A primary use of parallel computers is to execute parallel applications.

A parallel application is one that simultaneously assigns tasks to several nodes in the supercomputer. These tasks exchange data with each other periodically. The parallel applications in the simulation study are simplified somewhat to avoid the complexities of excessive detail which would not appreciably improve the accuracy of the results.

Each parallel application has a unique integer identifier and a set of nodes which will perform its computations. That is:

$$\mathbf{A^l_{nodes}} = \{P^h..P^i\} \tag{5}$$

where the sequence $h..i$ represents a sequence of unique though not necessarily consecutive integers. Each processor in the set must belong to the same parallel computer. No processor belongs to more than one parallel computer.

For the purposes of the model, we characterize a parallel application as dispatching some computation on each node. After these computations complete the nodes exchange data and begin a new computation. This period of computation followed by a period of communication constitutes one *phase* of the application. If a node completes its computation for a given phase before the other nodes, then the faster node becomes idle (with respect to this application) until the slower nodes complete the phase. Since parallel application designers strive to equalize the amount of work done by each node in each phase, the model treats each phase as a single task. For an application with k phases this gives the ordered sequence of tasks:

$$\mathbf{A^l_{tasks}} = (T^j, T^{j+1}, ..T^{j+k}). \tag{6}$$

Where the size of each task is equal to the combined computation and communication time of the corresponding phase of the application. That is, T^j_{sz} equals the total computation and communication time of the first phase of application A^l, T^{j+1}_{sz} equals the total time of the second phase, and so on.

Execution of a parallel application is simulated by starting the first task in A^l_{tasks} on each processor in A^l_{nodes}. When each instance of the first task completes, the second task is dispatched, and so on. Various metrics can be produced relating to the execution of a given parallel application A^l. Two that are of particular interest are the total time to completion of the application and the the average time that processors in A^l_{nodes} where idle with respect to A^l.

6.3.4 Workstation Clusters

Workstation clusters are modeled largely as groups of independent machines. In the VCE the only restriction on workstation clusters is that they be object code compatible. The assumption made for supercomputers that all nodes have the same processing speed is not reasonable for workstation clusters.

6.3.5 Modeling Processor Workload

The main objective of the VCE simulation model is to study the performance of redundantly scheduled VCE-tasks and their impact on conventionally generated computing tasks. Therefore, generating a realistic background workload for simulated machines is important to the results. For workstations, making the common assumption that each machine's workload is characterized by a exponentially distributed interarrival time and an exponentially distributed service time is reasonable. However, this does not imply that each machines conforms to the M/M/1 queue model because Unix workstations are timeshared rather than being single server FIFO queues [8, 70]. Therefore, the background workload generator for a workstation generates task arrival times and task sizes from exponential distributions and the tasks are serviced as described in Section 6.3.2.

Modeling the workload of a parallel supercomputer is even more complex. Supercomputing tasks have more parameters than the usual arrival time and service requirement. As discussed in Section 6.3.3, the characteristics of a parallel application include the number of nodes required, the number of phases and the length of each phase. Determining reasonable estimates for these parameters is difficult. Several thorough studies have been done regarding usage patterns of workstations [80, 81].

Similar studies of supercomputers have not been done. One reason for this may be the relative small number of machines deployed. While it is possible for a researcher to select a sample of a few dozen workstations and monitor their usage over several months, this approach will not work for parallel computing systems because there are rarely more than a few systems at any one site.

For the VCE simulation study the following approach was taken for generating the background workload: Application interarrival times, number of phases and amount of processing per phase were all selected from exponential distributions. Observation has shown that parallel application programmers often favor using a number of nodes which is a power of two. Therefore, the number of nodes for an application is selected by choosing a number from a uniformly distributed discrete random distribution and raising two to that power.

Sets of load generator parameters were developed for both parallel computers and workstation clusters that resulted in a high, medium or low load on the system. The load generator for workstation clusters can also be used to achieve a mixed workload which is typical of many workstation environments.

6.3.6 Modeling VCE Components

In addition to the structures discussed in the previous sections the VCE simulation effort also has to model some system elements that are unique to the Virtual Computing Environment. Specifically, VCE group leaders and execution processes must be modeled. To support these functions a unique feature was incorporated into the processor model. The Unix rstat() system call is simulated by the processor model. The VCE load balancing functions, like many load balancing functions, consider the length of a processors ready queue as an indicator of its load. On real Unix systems, programs running at the user level do not have direct access to this information. Instead, the VCE uses the rstat() system function. rstat() returns, among other things, an indicator of the average length of the processor ready queue over the preceding 60 seconds. The VCE load monitoring function issues rstat() every six seconds and extrapolates consecutive readings to estimate the ready queue length over shorter periods. So that the same load monitoring algorithm could be used in the simulator as is used in the VCE prototypes, the simulator,s processor class includes an rstat() method which returns the ready queue length over averaged over the past 60 simulated seconds.

Simulated group leaders execute code based very closely on that deployed in the experimental prototype system group leaders. Each group leader monitors one workstation cluster or supercomputer and maintains a list of the nodes in the group which have been lightly loaded for an extended of period of time.

The execution process used in the VCE simulator also mirrors that used in the actual system. It includes a parser for application description framework files described in Section 4.2. The syntax of the application description frameworked used by the simulator is enhanced so the size of each executable file can be encoded in the application description. This size information can be obtained experimentally using the VCE prototype system. This insures that the characteristics of the VCE applications studied with the simulator matches those of actual applications as closely as possible.

The simulated execution process allows for experimentation with many system parameters. VCE tasks can be dispatched at normal or low priority and the redundancy levels can be varied.

6.3.7 Instrumentation

The model provides ample opportunity for collecting a variety of performance and utilization data. Each task (Section 6.3.2) has an indicator of its size. The task's arrival time, completion time and total running time are recorded during the simulation. The difference between a task's size and its running time constitute its *delay*. As discussed in Section 6.3.3, a parallel application is divided into several phases where each phase contains multiple tasks which are executed in parallel. The delay of an application phase is given by the maximum delay of any task, not the sum of the task delays. The delay of an application is computed as the sum of the delays of each phase. When a VCE application is simulated, the delays it causes for other applications can be precisely measured. This statistic is an important measure of how non-VCE users are impacted by VCE scheduling policies.

In addition to gauging application performance, the model also provides hardware utilization data. Each processor is modeled using a smpl *facility* [75]. Software in smpl automatically collects a variety of information about each facility (processor) including the average number of tasks using it and the amount of time it was idle.

The model tracks the amount of processing performed by each cancelled task. This statistic is important in judging the effectiveness of VCE scheduling and is difficult to measure on prototype systems. On hardware systems when a task is cancelled before it produces output there is no indication of much computation it had performed. The only way to measure this on a prototype is through the use of software monitors which can skew the results of the experiments [57]. On the simulator the portion of each task which is already completed is recorded for each task running on a processor any time an event occurs which changes the number of tasks on the processor. Recording that data within cancelled tasks is quite natural. This data is useful in measuring the cost of redundant scheduling practices.

6.3.8 Simulator Results

A wide variety of experiments are possible using the simulator. Two will be discussed in this section.

Measuring the Effect of Various Redundancy Levels

One series of experiments was conducted using a small simulated system in order to further confirm that a redundancy level of more than three yields only nominal performance improvements. These experiments also demonstrated the importance of the VCE's load measurement and task placement techniques.

The simulated configuration for these experiments was kept simple. The hardware simulated was a single 32 node parallel computer and a cluster of 64 homogeneous

Table 9: Application Performance at Low Load Levels With Selective Placement

REDUNDANCY LEVEL	RUNNING TIME	CPU SECONDS USED
1	7839	20000
2	7341	36521
3	6605	50963
4	6568	64435

workstations. The workstations were simulated as being 25% slower than the nodes of the parallel supercomputer. A small non-VCE workload was placed on the parallel computer. The workstation cluster was established as containing an approximately equal mix of lightly, moderately and heavily loaded machines.

The application simulated was the linear equation solver whose structure was shown in Figure 26 in Chapter 4. The application has five tasks. For the purpose of the simulation it was assumed that each task required four nodes of the parallel computer or four workstations and required 1000 CPU seconds of processing on each node or workstation. In order to allow comparisons between the workstations and parallel computer nodes, but still account for the workstation's lower speed, each "CPU second" on a workstation is characterized as needing 1.25 wall clock seconds to elapse. This means the application, if dispatched non-redundantly, would require 20000 CPU seconds to run on either platform. One of the application tasks can run in parallel with another portion of the application, so if the application had exclusive use of eight nodes of the parallel computer it would complete in 4000 seconds of wall clock time. It would require 5000 seconds of wall clock time to complete on eight otherwise idle workstations.

Tables 9 and 10 show average results of five simulation runs for each of eight experiments. The results in Table 9 reflect the simulated application running on the simulated hardware described above using the scheduling methods described in

Table 10: Application Performance at Low Load Levels Without Selective Placement

REDUNDANCY LEVEL	RUNNING TIME	CPU SECONDS USED
1	13859	20000
2	10817	32391
3	9427	44590
4	9182	57568

Section 5.1. Table 10 represents the same scenario, except that the VCE load measurement routines were not used. Task instances were placed on randomly selected hosts.

These results show that both selective initial task placement and redundant scheduling improved performance significantly, and using both produced better results that using either one alone. The first entry in each table (REDUNDANCY LEVEL 1) measures application performance without redundant scheduling. The difference between these two numbers represents the performance degradations that results from placing application tasks without regard for the load on each computing node when the tasks are initiated (Table 10). Simply placing tasks on nodes that have excess capacity at the time the tasks are initiated improves performance by 43% (Table 9). The size of the performance improvement is reasonable because one third of the workstations in the simulation are heavily loaded. When selective placement methods are used it is unlikely that any of these machines will be chosen to host any portion of the VCE application. Without redundancy or selective placement a small number of heavily loaded machines may degrade application performance significantly.

The other entries in Table 10 show that redundant scheduling does somewhat (but not entirely) offset the lack of a selective task placement policy. Three way redundancy reduces average application run time by 32%, which is still not as fast as running with selective placement and without redundancy.

Even with selective scheduling policies and an environment with high levels of excess capacity, redundant scheduling can further improve performance. As seen in Table 9, two- and three-way redundancy produces performance gains of 6% and 16%.

These simulation results make it possible to measure the extent to which computing cycles are wasted doing redundant processing. The data shows that when selective scheduling is used, more CPU seconds are wasted than when it is not. This is because, without selective placement, some task instances are placed on overloaded hosts where they are canceled before they consume much CPU time. With selective placement, it is more likely that multiple instances of a task will be placed on lightly loaded machines and the slowest instance will near completion before the fastest instance completes.

Measuring the Effect of Low Priority Task Scheduling

The VCE simulator has also been used to study the feasibility of scheduling VCE tasks at low priority in order to minimize interference with non-VCE users. The advantages of low priority scheduling are considerable. It insures that only excess computing resources are consumed by VCE applications. When a low priority VCE task is executing on a machine, it will make only nominal progress if regular priority non-VCE jobs are present and competing for resources [8]. When other work is not active on a node, the low priority VCE task can benefit from computing cycles that would otherwise go unused. Of course, low priority tasks are always in danger of starvation. The purpose of the simulation experiments described below was to determine to what extent the VCE's redundant scheduling policy compensates for the performance impact of low priority dispatching.

These experiments used the same simulated hardware configuration as described earlier. That is, a 64 node workstation cluster and a parallel computer with 32 faster nodes were used. The linear equation solver was used again as a test application. The load conditions were changed somewhat from the experiments described above. For these experiments the load on the parallel computer was set to *moderate* rather than *low*. The additional load was used because the question under consideration relates to whether redundant scheduling can avoid delays due to task starvation. Starvation

Table 11: Application Performance at Moderate Load Levels With Selective Placement

REDUNDANCY LEVEL	RUNNING TIME	CPU SECONDS USED
1	9595	20000
2	8334	36112
3	6939	50889
4	6853	61121

Table 12: Application Performance at Moderate Load Levels With Low Priority Workstation Dispatching

REDUNDANCY LEVEL	RUNNING TIME	CPU SECONDS USED
1	20881	20000
2	11550	33711
3	10068	45114
4	8002	51704

is unlikely to occur with low load levels. The workstation cluster was again modeled as containing a roughly equal mix of lightly, moderately and heavily loaded machines. Table 11 gives observations of application performance at these higher load levels with all VCE tasks dispatched at normal priority. The results are comparable to those in Table 9 except for the generally poorer performance due to the higher non-VCE workload.

The first set of experiments in this area involved modifying the simulated VCE execution process to dispatch VCE tasks on workstations at low priority. This is expected to be a common method of scheduling VCE work because workstation owners expect predictable performance from their machines. Using low priority for VCE tasks is less likely to cause noticeable workstation performance degradations. Parallel computers are commonly a shared resource, so non-VCE users will be less sensitive

Table 13: Application Performance at Moderate Load Levels With Low Priority Dispatching on All Machines

REDUNDANCY LEVEL	RUNNING TIME	CPU SECONDS USED
1	25003	20000
2	21130	32509
3	12826	45813
4	11601	54089

to competition from VCE applications. Table 12 Shows the result of rerunning the scenario of Table 11 with low priority dispatching of VCE workstation tasks. It is apparent that reducing the dispatching priority of VCE tasks on the workstations in this configuration significantly reduces application performance, particularly when redundant scheduling is not used. It can also be seen that redundant scheduling does mitigate much of the performance loss due to lower scheduling priority. While the running time of 20,881 seconds for non-redundant scheduling with low priority scheduling (Table 12) more than twice as long as the 9,595 seconds needed with normal priority (Table 11), with three way redundancy the performance loss is only 45% (10,068 seconds vs. 6,939 seconds).

To further explore the impact of low priority scheduling, another set of simulation runs was performed with another change to the VCE execution process. Table 13 shows the impact of dispatching VCE tasks at low priority on the configuration's simulated parallel computer as well as the workstation cluster. The advantage of such a technique is that virtually all of the resources used to execute the application will come from excess CPU cycles and have negligible effect on conventionally executed (non-VCE) tasks. Of course, using very low priority for all VCE tasks does further reduce performance. But again, redundant execution lowers the impact markedly.

To emphasize the conclusions that can be drawn from the data in Tables 11, 12 and 13 it is instructive to focus on the **REDUNDANCY LEVEL 3** entry in each table. If the application were run on the simulated supercomputer in this configuration with no competing load, it would require 4,000 seconds to complete. Using three-way redundant scheduling and normal dispatching priority (Table 11) the application completes in 6,939 seconds despite a considerable transient load on the hardware. This performance is 28% better than the 9,595 seconds observed without redundant scheduling. Modifying the dispatching to use low priority scheduling of VCE tasks on workstations, but maintaining three-way redundancy, increases running time to 10,068 seconds, a tolerable level compared to the 20,881 seconds achieved without redundancy. Finally, using low priority scheduling for all VCE tasks and three-way redundancy increases application running time to 12,826 seconds, Low priority scheduling insures nominal interference with normal priority (non-VCE) work [8]. This shows that the VCE scheduling protocols can provide tractable performance with no interference to conventional users. Using low priority scheduling without redundant execution doubles application running time and is unlikely to provide acceptable performance. These experiments demonstrate the types of performance impacts that result from low priority scheduling. System adminstrators and planners can use this data to determine which scheduling policies are to be used in their installations.

6.3.9 Use of the Simulator to Design VCE Configurations

The experiments described in the previous section demonstrate the value of the VCE simulation model. Using it, it was possible to evaluate how changes in redundancy levels and dispatching policies would impact application performance under different load conditions. The experiments described above support the value of key VCE design features including redundant scheduling and simplified load measurement. Simulation experiments make it possible to precisely quantify how a VCE parameter like task dispatching priority can be used to improve VCE application performance or reduce interference with non-VCE workloads. The simulator can be used by system administrators to determine how VCE scheduling policies can best be customized to satisfy the objectives of their installations.

6.4 Summary

This chapter described the tools which have been implemented to study the characteristics of the VCE execution suite. The prototype systems have been used to build preliminary implementations of key elements of the VCE run time environment (scheduling/dispatching daemons, group leaders, execution processes and application description framework parsers). They have also been used to construct, execute and gauge the performance of several VCE applications in real world environments. Experimentation with the prototypes have demonstrated the implementability of the execution suite and the effectiveness of redundant scheduling. The prototypes can be enhanced in the future to include other aspects of the execution suite such as the communication functions described in Section 5.2. The prototypes will serve as a foundation for future execution suite systems.

The VCE simulation model provides a vehicle for determining how the VCE execution suite will perform in, and impact, possible host configurations. By precisely and efficiently modeling key elements of the run time environment, the simulator will enable planners and system administrators to anticipate what levels of performance they can expect from the VCE execution suite. It can also be used to determine how system parameters can be set, and system components customized, to deliver the best application performance without imposing unacceptable impacts to non-VCE users.

Bibliography

[1] Adler, R. M., *Emerging Standards for Component Software*, Computer, Vol. 28, No. 3, pp. 68-76, March 1995.

[2] Advanced Visual Systems Inc., **AVS Developer's Guide (Release 4.0)**, Part No. 320-0013-02, Revision B, Advanced Visual Systems Inc., Waltham Mass., May 1992.

[3] Ahmad, I., Ghagoor, A., and Mehrotra, K., *A Decentralized Task Scheduling Algorithm and its Performance Modeling for Computer Networks*, Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing, pp. 314-321, December 1991.

[4] Ahmad, I., Ghagoor, A., Mehrotra, K., Mohan, C. K., and Ranka, S., *Performance Modeling of Load Balancing Models Using Neural Networks*, Concurrency: Practice and Experience, Vol. 6, No. 5, pp. 393-409, August 1994.

[5] Allia, M., Arlat, J., Fabre, J.-C., Kanoun, K., and Powell, D., *Dependability Evaluation of the Saturation Technique*, Proceedings of the 5th Annual European Computer Conference (CompEuro '91), pp. 437-441, May 1991.

[6] Anderson, T. E., Culler, D. E., and Patterson, D. A., *A Case for NOW (Networks of Workstations)*, IEEE Micro, Vol 15, No. 1, pp. 54-64, February 1995.

[7] Armstrong, R., and Macfarlane, J., *The Use of Frameworks for Scientific Computation in a Parallel Distributed Environment*, Proceedings of the Third International Symposium on High Performance Distributed Computing, San Francisco, California, pp. 15-25, August 1994.

[8] Bach, M. J., **The Design of the Unix Operating System**, Prentice-Hall Inc., 1986.

[9] Barak, A., Guday, S.,and Wheeler, R. G., **The MOSIX Distributed Operating System, Load Balancing for UNIX**, Lecture Notes in Computer Science, Springer-Verlag, Vol. 672, 1993.

[10] Barak, A., Laden, O., and Yarom, Y., *The NOW MOSIX and its Preemptive Process Migration Scheme*, Bulletin of the Technical Committee on Operating Systems and Application Environments, IEEE Computer Society, Vol. 7, No. 2, pp. 5-11, June 1995.

[11] Barak, A., and Wheeler, R. G., *MOSIX: An Integrated Multiprocessor UNIX*, Proceedings of the Winter 1989 USENIX Conference, San Diego, CA, pp. 101-112, February 1989.

[12] Barbacci, M. R., Doubleday, D. L., Weinstock, C. B., and Wing, J. M., *The Durra Language and Runtime Environment: Tools for PMS Level Programming*, **Decentralized Systems**, Proceedings of the IFIP WG 10.3 Working Conference, pp. 25-27, North Holland, 1990.

[13] Barbacci, M. R., Weinstock, C. B., Doubleday, D. L., Gardner, M. J., and Lichota, R. W., *Durra: a structure description language for developing distributed applications,* Software Engineering Journal, pp. 83-94, March 1993.

[14] Barbacci, M. R., and Wing, J. M., *Specifying Functional and Timing Behavior for Real-Time Applications*, Proceedings of Parallel Architectures and Languages Europe (PARLE) Volume 2. Springer-Verlag, Lecture Notes in Computer Science, Vol 259, pp. 124-140, June 1987.

[15] Benmohammed-Mahieddine, K., Dew, P. M., and Kara, M., *A Periodic Symmetrically-Initiated Load Balancing Algorithm for Distributed Systems*, Proceedings of the 14th International Conference on Distributed Computing Systems, pp. 484-492, June 1994.

[16] Besana, V., De Luca, J., Giard, E., and Reger, J., **Experiences with SOMobjects: Distributed System Object Model (DSOM)**, IBM International Technical Support Centers, Document No. GG24-4165-00, November 1993.

[17] Betz, M. *Interoperable Objects*, Dr. Dobb's Journal, Vol. 19, No. 11, pp. 18-32, October 1994.

[18] Birman, K. P., Schiper, A., and Stephenson, *Lightweight Causal and Atomic Group Multicast*, ACM Transactions on Computer Systems, Vol 9, No 3. pp. 272-299, August 1991.

[19] Birman, K., Cooper, R., Joseph, T., Kane, K. and Schmuck, F., **The Isis System Manual.**

[20] Blumofe, R. D., and Park, D. S., *Scheduling Large-Scale Parallel Computations on Networks of Workstations*, Proceedings of the Third International Symposium on High Performance Distributed Computing, San Francisco CA., pp. 96-105, August 1994.

[21] Bricker, A., Litzkow, M. and Livny, M., *Condor Technical Summary*, Computer Science Department, University of Wisconsin - Madison, January 1992.

[22] Butler, R., and Lusk, E., *User's Guide to the p4 Programming System*, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439-4801, April 1994.

[23] Carriero, N., and Gelernter, D., *Linda in Context*, Communications of the ACM, Vol. 32, No. 4, pp. 444-459, April 1989.

[24] Carriero, N., Gelernter, D., Kaminsky, D., Westbrook, J., *Adaptive Parallelism with Piranha*, Department of Computer Science, Yale University, New Haven, Connecticut, YALE/DCS/RR-954, February 1993.

[25] Carriero, N., Gelernter, D., Mattson, T. G., and Sherman, A. H., *The Linda Alternative to Message-Passing Systems*, Parallel Computing, Vol. 20, No. 4, April 1994.

[26] Casavant, T. L., and Kuhl, J. G., *A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems*, IEEE Transactions on Software Engineering, Vol. 14, No. 11, pp. 141-154, November 1988.

[27] Chang, J., and Maxemchuck, N. F., *Reliable Broadcast Protocols*, ACM Transactions on Computer Systems, Vol. 2, No. 3, pp. 251-273, August 1984.

[28] Chen, J., Xie, L., and Sun, Z.-X., *An Intelligent Network Manager for Distributed Task Scheduling*, Proceedings of the 11th International Conference on Computer Communication, pp. 493-499, October 1992.

[29] Cheng, D.Y., *A Survey of Parallel Programming Languages and Tools*, RND-93-005, NASA Ames Research Center, March 1993.

[30] Cheriton, D.R., *The V Distributed System,* Communications of the ACM, pp. 314-333, Vol. 31, No. 3, 1988.

[31] Chiu, J-F, and Chiu, G-M, *Process-Replication for Fault-Tolerance and Performance Improvement in Distributed Computing Systems*, Proceedings of the Third International Symposium on High Performance Distributed Computing, San Francisco, California, pp. 236-243, August 1994.

[32] Chu, W. W., Holloway, L. J., Lan, M., and Efe, K., *Task Allocation in Distributed Data Processing*, Computer, Vol 13, No. 11, pp. 57-69, November 1980.

[33] Clark, H. and McMillin, B., *DAWGS a distributed compute server utilizing idle workstations,* Journal of Parallel and Distributed Computing, pp. 175-186, Vol.14, No.2, 1992.

[34] Dandamudi, S., *Performance Impact of Scheduling Discipline on Adaptive Load Sharing in Homogeneous Distributed Systems*, Proceedings of the 15th International Conference on Distributed Computing Systems, pp. 484-492, June 1995.

[35] Deremer, F., and Kron, H. H., *Programming-in-the-Large Versus Programming-in-the-Small*, IEEE Transactions on Software Engineering, Vol. 2, No. 2, June 1974.

[36] Eager, D. L., Lazowska, E. D., and Zahorjan, J., *A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing*, Performance Evaluation, Vol. 6, pp. 53-68, March 1986.

[37] Eager, D. L., Lazowska, E. D., and Zahorjan, J., *Adaptive Load Sharing in Homogeneous Distributed Systems*, IEEE Transactions on Software Engineering, Vol. 12, No. 5, pp. 662-675, May 1986.

[38] Elnozahy, E. N., and Zwaenepoel, W., *Replicated Distributed Processes in Manetho*, Proceedings of the Conference on Fault-Tolerant Computing Systems, pp 222-230, 1991.

[39] Efe, K., and Schaar, M. A., *Performance of Co-Scheduling on a Network of Workstations*, Proceedings of the 13th International Conference on Distributed Computing Systems, pp. 525-531, May 1993.

[40] Fabre, J. C., *Improving Fault Tolerance in Distributed Systems: The Saturation Approach*, Proceedings of the 5th Annual European Computer Conference (CompEuro '91), pp. 634-641, May 1991.

[41] Farrell, W. E., Fox, W. Given, J. W., MacRitchie, B., and Wang, J., *Data Processing on Networks of Workstations*, Bulletin of the Technical Committee

on Operating Systems and Application Environments, IEEE Computer Society, Vol. 7, No. 2, pp. 35-42, June 1995.

[42] Fox, G. C., et. al., **Solving Problems on Concurrent Processors, Volume 1,** Prentice Hall, 1988.

[43] Fox, G. C., **The Architecture of Problems and Portable Parallel Software Systems,** Northeast Parallel Architectures Center, Technical Report SCCS-134, Syracuse University, July 1991.

[44] Fox, G. C., and Furmanski, W., *The Use of National Information Infrastructure and High Performance Computers in Industry*, Proceedings of the 2nd International Workshop on Massively Parallel Processing Using Optical Interconnections, October 1995.

[45] Fox, G. C., Furmanski, W., Chen, M., Rebbi, C., and Cowie, J. H., *WebWorks: Integrated Programming Environment Tools for National and Grand Challenges*, Northeast Parallel Architectures Center, Technical Report sccs-0715, June 1995.

[46] Fox, G. C., Hiranandani, H., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C, and Wu, M., **Fortran D Language Specifications**, Northeast Parallel Architectures Center, Syracuse University, 1990.

[47] Garcia-Molina, H., and Spauster, A., *Ordered and Reliable Multicast Communication*, ACM Transactions on Computer Systems, Vol. 9, No. 3, pp. 242-271, August 1991.

[48] Geist, A., Beguelin, A., Dongarra, J., Weicheng, J., Manchek, R., and Sunderam, V., **PVM 3 User's Guide and Reference Manual**, Oak Ridge National Laboratory, Document No. ORNL/TM-12187, September 1994.

[49] Gelernter, D., and Carriero, C., *Coordination Languages and their Significance*, Communications of the ACM, Vol. 35, No. 2, pp. 97-107, February 1992.

[50] Ghafoor, A., and Ahmad, I., *An Efficient Model of Dynamic Task Scheduling for Distributed Systems*, Proceedings of the 14th Annual International Computer Software and Applications Conference, pp. 442-447, November 1990.

[51] Graham, I. S., **The HTML Sourcebook**, John Wiley & Sons Inc., 1995.

[52] Grimshaw, A. S., *The Mentat Runtime System: Support for Medium-Grain Parallel Computation*, Proceedings of the Fifth Distributed Memory Computing Conference, Los Alamitos, CA., Vol II, pp. 1064-1073, 1990.

[53] Grimshaw, A. S., *Easy-to-Use Object-Oriented Parallel Processing with Mentat*, Computer, Vol. 26, No. 5, May 1993.

[54] Hariri, S., *High Performance Distributed Systems: Concepts and Designs*, Artech House, in preparation.

[55] Hariri, S., Park, J., Yu, F., Parashar, M., and Fox, G., *A Message Passing Interface for Parallel and Distributed Computing*, Proceedings of the Second International Symposium on High Performance Distributed Computing, pp. 84-91, July 1993.

[56] Hariri, S., Yadav, R., Thiagarajan, B., Park, S. Y., Subramanyan, M., and Reddy, R., "A Concurrent Multi Target Tracker: Benchmarking and Portability", International Conference on Parallel Processing (ICPP), August 1994.

[57] Heidelberger, P., and Lavenberg, S. S., *Computer Performance Evaluation Methodology*, IEEE Transactions on Computers, Vol. 33, No. 12, pp. 1195-1220, December 1984.

[58] Hwang, K., and Briggs, F.A., **Computer Architecture and Parallel Processing**, McGraw-Hill, Second Edition.

[59] Jacob, B., **SOMobjects: A Pratical Introduction to SOM and DSOM**, IBM International Technical Support Centers, Document No. GG24-4357-00, July 1994.

[60] Ju, J., Xu, G., and Tao, J., *Parallel Computing Using Idle Workstations,* Operating Systems Review, Vol. 27, No. 3, 1993.

[61] Kaashock, M. F., and Tanenbaum, A. S., *Group Communication in the Amoeba Distributed Operating System*, Proceedings of the 11th International Conference on Distributed Computing Systems, pp. 222-230, May 1991.

[62] Kernighan, B. W., and Ritchie, M. R., **The C Programming Language - Second Edition**, Prentice Hall, 1988.

[63] Kramer, J., *Configuration Programming - A Framework for the Development of Distributable Systems,* The Proceedings of the IEEE International Conference on Computer Systems and Software Engineering (CompEuro 90), pp. 374-384, 1990.

[64] Kramer, J., Magee, J., and Finkelstein, A., *A Constructive Approach to the Design of Distributed Systems*, Proceedings of the 10th International Conference on Distributed Computing Systems, pp. 580-587, June 1990.

[65] Kramer, J., Magee, J., and Sloman, M., *Configuration Support for System Description, Construction and Evolution*, **Managing Complexity in Software Engineering**, pp. 247-260, edited by R. J. Mitchell, Peter Pereginus Ltd., 1990.

[66] Kremien, O., and Kramer, J., *Methodical Analysis of Adaptive Load Sharing Algorithms*, IEEE Transactions on Parallel and Distributed Systems, Vol. 3, No. 6, pp. 747-760, November 1992.

[67] Krueger, P., and Cawla, R., *The Stealth Distributed Scheduler,* Proceedings of the 11th IEEE International Conference on Distributed Computing Systems, May 1991.

[68] Krueger, P., and Livny, M., *A Comparison of Preemptive and Non-Preemptive Load Distributing*, Proceedings of the 8th International Conference on Distributed Computing Systems, pp. 123-130, June 1988.

[69] Kunz, T., *The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme*, IEEE Transactions on Software Engineering, Vol. 17, No. 7, pp. 725-730, July 1991.

[70] Leffler, S. J., McKusick, M. K., Karels, M. J., and Quaterman, J. S., **The Design and Implementation of the 4.3BSD Unix Operating System**, Addison-Wesley, 1989.

[71] Litzkow, M. and Solomon, M., *Supporting Checkpointing and Process Migration Outside the Unix Kernel,* Proceedings of the Winter 1992 USENIX Conference, pp. 283-290, 1991.

[72] Liu, R.-C., *Analysis on the Effects of System Parameters on Load Balancing*, Information Sciences, Vol. 81, No. 1-2, pp. 37-54, November 1994.

[73] Lord, H. D., *Improving the Application Development Process with Modular Visualization Environments*, Computer Graphics, Vol. 29, No. 2, May 1995.

[74] Mechoso, C. R., Farrara, J. D., Spahr, J. A., *Running a Climate Model in a Heterogeneous, Distributed Computer Environment,* Proceedings of the Third International Symposium on High Performance Distributed Computing, pp. 79-84, August 1994.

[75] MacDougall, M. H., **Simulating Computer Systems - Techniques and Tools**, MIT Press, 1987.

[76] Mattson, T. G., *Programming Environments for Parallel Computing: A Comparison of CPS, Linda, P4, PVM, POSYBL, and TCGMSG*, Proceedings of the 27th Annual Hawaii International Conference on Systems Science, Vol. II, pp. 586-594, January 1994.

[77] Mirchandaney, R., Towsley, D., and Stankovic, J. A., *Analysis of the Effects of Delays on Load Sharing*, IEEE Transactions on Computers, Vol. 38, No. 11, pp. 1513-1525, November 1989.

[78] Mullender, S.J., and van Rossum, G., *Amoeba: A Distributed Operating System for the 1990s,* Computer, pp. 44-53, Vol. 23, No. 5, 1990.

[79] Munetomo, M., Takai, Y., and Sato, Y., *A Genetic Approach to Dynamic Load Balancing in a Distributed Computing System*, Proceedings of the 1st IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence, Vol. 1, pp. 418-421, June 1994.

[80] Mutka, M. W., *Estimating Capacity for Sharing in a Privately Owned Workstation Environment,* IEEE Transactions on Software Engineering, Vol. 18, No. 4, 1992.

[81] Mutka, M. W., and Livny, M., *The Available Capacity of a Privately Owned Workstation Environment,* Performance Evaluation, pp. 269-284, Vol. 12, No. 4, July 1991.

[82] The Object Management Group, **The Common Object Request Broker: Architecutre and Specification**, OMG Document Number 93.12.43, December 1993.

[83] Parasoft Corporation, *Express Reference Manual,* 1988.

[84] Park, C., and Kuhl, J. G., *A Fuzzy-Based Distributed Load Balancing Algorithm for Large Distributed Systems*, Proceedings of the 2nd International Symposium on Autonomous Decentralized Systems, pp. 366-273, April 1995.

[85] Peterson, L., Hutchinson, N., O'Malley, S., and Rao, H., *The x-Kernel: A Platform for Accessing Internet Resources,* Computer, pp. 23-33, Vol. 23, No. 5, May 1990.

[86] Prieto-Diaz, R., and Neighbors, J.M., *Module Interconnection Languages,* The Journal of Systems and Software, No. 6, pp. 307-334, 1986.

[87] Rehmet, G. M., **Remora - Implementing Adaptive Parallelism on a Heterogeneous Cluster of Networked Workstations**, Master's Thesis, Rhodes University, South Africa, September 1994.

[88] Rinaldo, F., and Wolbers, S., *Loosely Coupled Parallel Processing at Fermilab*, Computers in Physics, Vol. 7. No. 2, pp. 184-190, March 1993.

[89] Rosenberry, W., Kenney, D., and Fisher, G., **Understanding DCE**, O'Reily & Associates, 1992.

[90] Rousselle, P., Hariri, S., and Tymann, P, *Redundant Task Scheduling in a Heterogeneous Computing Environment*, Northeast Parallel Architectures Center, Technical Report SCCS-658, October 1994.

[91] Rousselle, P., Tymann, P., Hariri, S., and Fox, G., *The Virtual Computing Environment* Proceedings of the Third International Symposium on High Performance Distributed Computing, San Francisco, California, pp. 7-14, August 1994.

[92] Rousselle, P., Yadav, R., Hariri, S., Kim, D., Boddapati, S., and Tymann, P., *Application Development and Execution in a Virtual Computing Environment*, Northeast Parallel Architectures Center, Technical Report SCCS-705, April 1995.

[93] Schaar, M., Efe, K., and Delcambre, L. M., *An Analytical Model for Predicting Performance in a Heterogeneous System*, Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Computing Systems, pp. 328-335, May 1991.

[94] Shoja, G.C., *A Distributed Facility for Load Sharing and Parallel Processing Among Workstations,* Journal of Systems Software, pp. 163-172, Vol. 14, No. 3, 1991.

[95] Simpson, R. A., and Harget, A. J., *A Simulation Study to Determine the Importance of Load Balancing Algorithms for Loosely Coupled Distributed Systems*, IFIP Transactions on Computer Science and Technology, IFIP 13th World Computer Congress, Vol. A-51, pp. 69-74, August 1994.

[96] Stevens, W. R., **UNIX Network Programming**, Prentice Hall, 1990.

[97] Stroustrup, B., **The C++ Programming Language - Second Edition**, Addison-Wesley, 1991.

[98] Stroustrup, B., **The Design and Evolution of C++**, Addison-Wesley, 1994.

[99] V. S. Sunderam, **PVM: A Framework for Parallel Distributed Computing**, Oak Ridge National Laboratory, 1991.

[100] Tannenbaum, A. S., **Modern Operating Systems**, Prentice-Hall, 1992.

[101] Theimer, M., and Lantz, K., *Finding Idle Machines in a Workstation-Based Distributed System,* IEEE Transactions on Software Engineering, pp. 1444-1458, Vol. 15, No. 11, 1989.

[102] Theimer, M., and Hayes, B., *Heterogeneous Process Migration by Recompilation*, Proceedings of the 11th IEEE International Conference on Distributed Computing Systems, May 1991.

[103] Thornborrow, C., Wilson, A. J. S., and Faigle, C., *Developing Modular Application Builders to Exploit MIMD Parallel Resources*, Proceedings of Visualization '93, pp. 134-141, San Jose CA, October 1993.

[104] Waldspurger, C.A., Hogg, T., Huberman, B.A., Kephart, J.O., and Stornetta, W.S., *Spawn: a distributed computational economy,* IEEE Transactions on Software Engineering, pp. 103-117, Vol.18, No.2, 1992.

[105] Walker, B., Popek, G., English, R., Kline, C., and Thiel, G., *The LOCUS Distributed Operating System,* Proceedings of the 1983 SIGOPS Conference, pp. 49-70.

[106] Wang, C. -J., Krueger, P. E., and Liu, M. T., *Intelligent Job Selection for Distributed Scheduling,* Proceedings of the Thirteenth International Conference on Distributed Computing Systems, pp. 517-524, May 1993.

[107] Wood, M. D., *Replicated RPC Using Amoeba Closed Group Communication*, Proceedings of the 13th International Conference on Distributed Computing Systems, pp. 222-230, 1993.

[108] Zhou, S., *A Trace-Driven Study of Dynamic Load Balancing*, IEEE Transactions on Software Engineering, Vol. 14, No. 9, pp. 1327-1341, September 1988.

[109] Zhou, S., Zheng, X., Wang, J. and Delisle, P., *Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems*, Software - Practice and Experience, pp. 1305-1336, Vol. 23, No. 12, December 1993.

[110] Zhu, W., and Goscinski, A., *Low Cost Load Balancing Algorithms*, Proceedings of the 27th Annual Simulation Symposium, pp. 226-235, April 1994.

Biographical Data

Name:	Philip Rousselle
Date and Place of Birth:	September 10, 1962 Buffalo, New York
Elementary & High School:	Buffalo Public Elementary School #84 Graduated 1976
	Kensington High School Buffalo, New York Graduated 1980
College:	SUNY College at Buffalo BS (Info Systems Management), 1983
	Union College Schnectady, New York MS (Computer Science), 1989
Employment	International Business Machines Inc Endicott NY