1992

Efficient Techniques for Simulating Service Disciplines

Janche Sang

Ke-hsiung Chung

Vernon J. Rego

Purdue University, rego@cs.purdue.edu

Report Number:
92-068
EFFICIENT TECHNIQUES FOR
SIMULATING SERVICE DISCIPLINES

Janche Sang
Ke-hsiung Chung
Vernon Rego

CSD-TR-92-068
September 23, 1992
Efficient Techniques for Simulating Service Disciplines

Janche Sang
Ke-hsiung Chung
Vernon Rego*

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

September 23, 1992

Abstract

Efficient techniques for simulating the Round-Robin and Processor-Sharing service disciplines are presented. These schemes are general in that they can be applied in any discrete-event simulation view to improve execution performance. For the Round-Robin discipline, the proposed approach is based on computation instead of process switching. For the Processor-Sharing discipline, we introduce a lazy approach in combination with any efficient priority queue structure to reduce scheduling-related computation. Using the Si simulation testbed for Simulating (process) interactions, the proposed algorithms are shown to be more efficient than the standard algorithms for simulating these service disciplines.

* This research was supported in part by NSF award CCR-9102331, NATO CRG-900108 and the Mathematical Sciences Section of Oak Ridge National Laboratory under contract Contract DE-AC05-84OR21400 with Martin Marietta Energy Systems, Inc.
1 Introduction

Service disciplines such as First-Come-First-Served (FCFS), Round-Robin (RR), Processor-Sharing (PS), etc. and issues related to their performance have attracted considerable attention in computer systems modeling research since the 1960s [19]. Despite myriad analyses (for example, see [4] or [11]) of waiting-line behaviour, questions of waiting-line performance still remain, especially in the context of multiqueue and multiserver systems. In the design of more complex computing systems (i.e., systems that are large in terms of number of resources and activities, and/or systems that exhibit complicated patterns of behaviour [17]), we usually have little choice but to rely on good simulation models to answer questions related to system performance. Since simulation models of complex systems can require tremendous amounts of computing time, potential reductions in time resulting from the implementation of more efficient algorithms for routine activities are well worth investigation.

In this paper, we propose efficient approaches to the simulation of the RR and PS service disciplines. These are shown to be improvements over the standard algorithms currently in use and can be applied to system simulations that are process-oriented, event-oriented, or activity-oriented [5]. Without loss of generality, we focus our attention on the process-oriented view in using the proposed algorithms. A process-oriented simulation is a simulation technique which models any kind of system in terms of interacting processes [5]. These processes employ a variety of statements to define the flows of entities (transactions, customers, jobs, etc.) through the system. Two or more processes may compete for resources and exchange messages with one another for the purpose of synchronization. Several well-known simulation systems based on the process-oriented view include GPSS [7], SLAM II [12] and CSIM [16], among others. To the best of our knowledge, none of these popular systems use efficient algorithms for the RR and PS disciplines. Further, with the exception of some methods discussed in [9], techniques discussing the implementation of queueing algorithms, in particular the RR and PS algorithms, for different simulation views are virtually nonexistent.

We model the single server queue as a function which takes as input customer (i.e., process) arrival instants and generates a sequence of customer departure instants from the queue. A process becomes blocked upon entry to the (single server) queueing center, and becomes active
again when its departure event occurs. The proposed approaches compute the departure times of consecutive departing customers before scheduling the corresponding departure events, instead of merely scheduling service quanta for queued jobs. The computed departure instant of a particular customer may be invalidated by one or more newly arriving customers, i.e., one or more arriving after the corresponding departure event is scheduled, but before the departure event can occur. This is because the sever must now attend to one or more previously unaccounted for customer arrivals, and decrease the amount of attention it planned on giving to customers already in the system prior to the arrival of the new customer(s). Consequently, a scheduled departure event that has been invalidated in this manner must be cancelled, and an updated departure instant for the same or another customer scheduled in its place. In addition, it is necessary that the remaining service time requirements of each customer in the system be adjusted whenever an arrival event or a departure event occurs. This idea has appeared in earlier studies of interrupt processing [8] and hybrid modeling [9, 15], though the study of computational schemes for different scheduling disciplines is apparently new.

In order to test the proposed algorithms and evaluate their performance we implemented them using the Si simulation system [14]. Si (a system for Simulating process interactions) is a process-oriented discrete-event simulation package in C which is built on top of the Sun Lightweight Process Library [18]. It is designed to be a research vehicle for conducting experiments related to simulation algorithms and for demonstrating applications of modeling techniques to real-world problems.

The remainder of the paper is organized as follows. Section 2 contains a description of general concepts embedded in the simulation of scheduling disciplines. In Section 3 we describe two approaches to simulating the RR discipline. One approach is based on process-switching, and the other is based on computation. We report the results of three experiments in which the time quantum offered by the server, the ratio of customer arrival rate to service rate, and the number of servers are varied. In Section 4 we describe two approaches to simulating the PS discipline. One approach is based on process switching and an exhaustive information update, while the other approach is a lazy approach which potentially reduces the amount of computation required. We analyze these approaches in terms of the number of operations performed, and conduct two experiments to support the results of the analysis. Finally, we
present a brief conclusion in Section 5.

2 Simulating Disciplines: General Concepts

A single server queueing system consists of a waiting-line area for customers (i.e., a queue), a single server, and a service discipline which determines how the server attends to customers in the system. As shown in Figure 1, the system can be treated as a black box, represented by a function that takes as input a stream of tuples \((A_1, a_1, s_1), (A_2, a_2, s_2), \ldots\) and yields as output another stream of tuples \((D_1, d_1), (D_2, d_2), \ldots\). For an input tuple \((A_i, a_i, s_i)\), \(A_i\) is an identifier representing the the \(i\)th job entering the black box at time \(a_i\), and \(s_i\) is the amount of service time requested by this job. For an output tuple \((D_i, d_i)\), \(D_i\) is an identifier for the \(i\)th job departing from the black box at time \(d_i\). Clearly, each stream comprises a non-decreasing sequence of times, i.e., \(a_i \leq a_{i+1}\) and \(d_i \leq d_{i+1}\), for all \(i \geq 1\). Such a black box view can be used to represent any service discipline for a single server system. For example, for the FCFS discipline, the black box can be represented by a function [6] which computes:

\[
D_i = A_i \\
D_i = \text{if } (a_i > d_{i-1})? (a_i + s_i) : (d_{i-1} + s_i)
\]

for all \(i \geq 1\), with where \(d_0 = 0\). Equation (1) depicts the first-in, first-out property of the queue, i.e., the sequence of job arrivals is identical to the sequence of job departures. The condition in Equation (2) tests the status of the server (i.e., busy or idle). If \(a_i > d_{i-1}\), the server is idle when the \(i\)-th job \(A_i\) arrives, and job \(A_i\) begins service at time \(a_i\). Otherwise, the server is busy, and job \(A_i\) must wait until the previous job \(A_{i-1}\) completes service and departs from the system at time \(d_{i-1}\). Unfortunately, determination of the output sequence \((D_i, d_i)\), \(i \geq 1\), is decidedly nontrivial for an arbitrary scheduling discipline, since in general it is not true that \(D_i = A_i\), for all \(i \geq 1\). The scope of our work is restricted to the determination of efficient algorithms for the generation of output sequences \((D_i, d_i)\), \(i \geq 1\), for the RR and PS service disciplines.

In contrast to the event-scheduling approach, where the occurrence of each event is explicitly associated with a corresponding event-handling routine, process-oriented simulations view event occurrences as being implicitly generated by processes. As a consequence, in process-
oriented simulations, clock advancement is associated with process-switching. There is potential for simulation time to pass when a process suspends itself, either definitely or indefinitely, by transferring control to another process, possibly itself. In general, a suspended process is reactivated when the condition responsible for its suspension is satisfied. This transfer of control between processes is managed by a simulation process scheduler. Once a process is given control by the scheduler, it resumes execution at the statement following its point of suspension.

In the simple single server queueing system, there are two kinds of implicit events: job arrivals and job departures. When either of these two events occurs, the service requirements of some or all of the queued jobs, depending on the service discipline in effect, undergo a potential reduction. If \( t_l \) denotes the time at which the last (arrival or departure) event occurred, and \( t_n \) denotes the time of the new (arrival or departure) event, then some or all queued jobs undergo a potential reduction in service time proportional to \( (t_n - t_l) \) at time instant \( t_n \). In other words, the time interval \( (t_n - t_l) \) is shared by a subset of queued customers in some manner. Determining how this time is to be allocated between queued jobs, and determining which jobs qualify for the server's attention depends on the service discipline.

The generation of event times is another issue of importance. Since arrival times are assumed to be inputs to the function representing the black box, they are not a concern to the algorithm implementing a service discipline. On the other hand, the algorithm is required to predict completion times of departing jobs and schedule corresponding departure events. The process-oriented simulation mechanism proceeds by inserting corresponding departure events into the simulation event calendar. If a new job arrives after such an event has been scheduled, and before the event can occur, its own service time requirement and interaction with the server causes the previously scheduled departure event to no longer be associated with
a correct departure time; this makes the departure event invalid. Consequently, it is necessary for the algorithm to cancel this invalid departure event and schedule a new departure event, for the same customer or a customer who requires to depart earlier.

A sketch of pseudo-code outlining a typical algorithm for implementing a service discipline is shown in Figure 2. This pseudo-code can be implemented with little modification in any process-oriented simulation environment, and can also be applied in other simulation views. In the process-oriented view, a job is represented by a process which describes all activities undertaken by the job, in particular the manner in which it flows through the system. In the text, we use the terms job and process interchangeably. The word “general” in Figure 2 may be replaced with the name of an appropriate scheduling discipline, for example “rr” for the round-robin and “ps” for the processor-sharing disciplines.

A job which is fed into a service center will invoke the function general\((f, t)\), where \(f\) is the name of the service center and \(t\) is the service time requirement of the job. For example, if the service center adopts the round robin service discipline, an arriving job will invoke function rr\((f, t)\). Function general() is essentially made up of three distinct steps. First, it invokes the function insert_to_pool() which inserts the current process’s (i.e., the caller’s) information, such as process identification, service time requirement, etc., into the pool of information relating to processes already in the system. Second, the current process blocks (suspends) itself by giving control to a scheduler, called scheduler. This is accomplished by the si_yield() action performed by the caller. The scheduler then determines which process is next in line to execute based on simulation time and event ordering; upon making such a determination, the scheduler relinquishes control to the selected process. One way to implement such a scheduler is as a special process which executes a simple scheduling function schedule() in a tight loop for as long as the simulation event calendar is not empty.

For a queue operating in stable mode, a suspended process eventually regains control from the scheduler at some instant in simulated time. This instant coincides with the instant at which the process’s service time requirements are satisfied by the server. The suspended process resumes execution at the statement following the si_yield() action, taking us to the third step in algorithm general\((f, t)\). In the third step, a process departs from the service center by invoking function delete_from_pool(). As mentioned earlier, both functions
insert_into_pool() and delete_from_pool() must call function adjust_pool() to determine precisely which jobs receive service during the interval between events, and perform an appropriate reduction in remaining service time requirement for these jobs. In addition, they call function schedule_next_to_leave() to determine the completion time of the next process to depart from the queue, under the condition that no new jobs arrive before this time. Under this assumption, a corresponding departure event is scheduled; the departure event will cause an appropriate suspended process associated with this event to resume execution at the departure time. The two procedures adjust_pool() and schedule_next_to_leave() are inherently discipline-dependent and their algorithms are left to be discussed in Sections 2 and 3.

3 The Round-Robin Discipline

In the FCFS service discipline, a job is allowed to resume execution and leave the system after a single suspension. It is suspended for the duration of its requested service time, after which it regains control and resumes execution. In contrast, a job operating in a pool of jobs under a RR service discipline suffers a number of suspensions. This number is a function of the size of the time quantum $q$ that the server can afford each job, the service time requirement of the job, and the number of jobs in the pool. If the remaining service time requirement of a job at the head of the queue exceeds $q$, the job’s processing is interrupted at the end of its quantum; following this, the job is returned to the rear of the queue, where it is forced to await its turn for the server and its next quantum.

3.1 The Naive Algorithm

In any discrete-event simulation view, a naive but reasonably efficient (especially in terms of simplicity) approach to implementing the RR discipline is to treat the pool of processes awaiting service in the queue as real-world processes, and physically dole out service quanta to these processes in a round-robin fashion. An arriving process is placed at the tail of the queue (pool) since it has lowest priority among jobs waiting for quanta. After each job receives its service quantum from the server, it moves to the tail of queue. In this way, a process requesting RR service gets its quantum from the server when its turn arrives, utilizes the server for this quantum of length $q$, and then relinquishes control to the scheduler so that the next job in the
insert_into_pool(f,t)
{
    if (size_of_pool(f) > 0)
        cancel previously scheduled departure event;
    adjust_pool(f);
    insert current running process's pid into pool f;
    schedule_next_to_leave(f); /* schedule a departure event */
}
delete_from_pool(f)
{
    adjust_pool(f);
    delete current running process's pid from pool f;
    schedule_next_to_leave(f); /* schedule a departure event */
}
general(f,t)
{
    insert_into_pool(f,t); /* implicit arrival event occurs */
    si_yield(scheduler); /* relinquish control to scheduler */
    /* regain control from scheduler */
    delete_from_pool(f); /* implicit departure event occurs */
}
schedule()
{ /* only the system process scheduler executes this routine */
    ....initialization...
    while(future_event_set is not empty) {
        E = extract.min(future_event_set);
        clock = E.clock;
        si_yield(E.pid); /* resume execution of process pid */
    }
}

Figure 2: The General Algorithm
pool can get its quantum from the server. A job whose service time requirements have been met through the scheduling of several such quantum service-allocations will leave the pool. A succinct description of this scheme can be seen in the pseudo-code given in Figure 3.

Using this approach in $Si$, a process invokes function $rr(f, t)$ to imitate the behaviour of a job with service requirement $t$ arriving at round-robin queue $f$. The function $rr()$ is built on top of the three $Si$ functions: request(), release(), and delay(). The functions request() and release() are similar to the operating system primitives lock and unlock, respectively, while function delay() has the same semantics as the hold operation in Simula [2] or CSIM [16]. Details of these functions can be found in [14].

While the naive approach has a certain elegance and is simple in description, there is considerable potential for high execution overhead due to context switching between processes. When a process issues a request to a server which has already been reserved by another process, the requesting process is blocked and a process switch (context switch) occurs. Though $Si$ implements simulation processes as lightweight processes and consequently enjoys relatively small context switch overheads, context switching in large amounts tends to be a major contributor to simulation execution time.

3.2 A Computational Algorithm

Instead of taking the naive approach and implementing the round-robin discipline by physically switching control from process to process, we view the job pool as an entity which changes state upon either a job arrival or a job departure. As discussed in the previous section, job
arrivals are controlled by an external source and are not a responsibility of the algorithm. However, the algorithm must compute job departure times as a function of the state of the pool and size of service quantum q. The state of the pool is defined by the remaining service requirements of jobs, number of jobs, and next job in line for service. While the general idea behind this approach has already been described in the previous section, it remains to discuss the implementation of the two key functions adjust_pool() and schedule_next_to_leave().

Before going into a description of function details, we remark on a characteristic of the round-robin discipline and the use of a data structure employed by both functions. By definition, the term round-robin suggests that waiting jobs are attended to by the server in a cyclic order. This order must be maintained if the algorithm is to repeatedly re-adjust the remaining service time requirements of processes in the pool, to reflect service these jobs may have already received through ongoing service-quanta allocations. The cyclic order can be maintained by utilizing pool state in conjunction with job arrival times.

For example, assume that the pool of jobs in a queue consists of the jobs A, B, C, and D; assume also that these jobs are to be served in this order, cyclically. If a new job X arrives while job C is in service (i.e., during C’s quantum), then the pool of jobs will now exhibit the cyclic order A, B, X, C, and D. Accordingly, a ring data structure may be used to preserve this cyclic order, as shown in Figure 4. Each element in the ring has three fields: process id, remaining ticks (i.e., remaining service time), and a pointer to the next element in the ring. An additional pointer head, which is associated with pool of jobs, is used to maintain the cyclic order. The pointer head points to the process that is next in line to receive a time-quantum of service from the server. Since the remaining service time of each job in the pool is readjusted only when the system state changes (i.e., an arrival or a departure from the pool), the job that will be pointed to by head following an event depends on the duration of time between state changes, the number of jobs in the pool, and the cyclic ordering of jobs.

The operations in functions adjust_pool() and schedule_next_to_leave() are detailed in Figure 5. In function adjust_pool(), we first calculate the time interval delta between consecutive adjustments. This is precisely the time between arrival or departure events at the queue. Next, we determine the number of quanta (ticks) in the interval by computing delta / q. Finally, these quanta are allocated to the processes in the ring using the cyclic
Assume 7 ticks available when Process X arrives

Next process to depart: C

(a) Before process X joins the pool

(b) After process X joins the pool

Figure 4: The ring structure
ordering, starting with the process pointed to by the pointer \textit{head}. When all quanta have been exhausted, the pointer \textit{head} is updated to point to the next process in line for a quantum. For ease of presentation, our discussion assumes that the service time requirement of each job is an integral number of ticks. The implementation of the algorithm in \textit{Si} does not have this limitation.

To accurately predict the earliest departure time of a job leaving the pool of jobs in the queue, the function \texttt{schedule.next.to.leave()} first searches the ring to find the process with the minimum number of remaining ticks. This process will be the first to leave the pool provided no new job, with a shorter (remaining) service time requirement, arrives before its departure. The search procedure originates at the process pointed to by pointer \textit{head}, to be consistent with the order of service. If two or more jobs have the same number \textit{m} of remaining ticks, the process selected by the search procedure is the first job found in the cycle starting at \textit{head} with \textit{m} ticks of required service remaining. The minimum tick requirement \textit{m} is stored in the variable \texttt{minrem}, and the cyclic distance between the job pointed to by pointer \textit{head} and the job found by the search procedure is stored in variable \texttt{steps}. The departure time of the next job to leave the pool is obtained by the formula:

\[
\textit{current.time} + ((\texttt{minrem} - 1) \times \texttt{size.of.pool(f)} + \texttt{steps}) \times q
\]

under the condition no new job arrives before this time to invalidate the departure time. Thus, the next process to depart from the pool can depart only if it gets \texttt{minrem} ticks of service. The first \texttt{(minrem} - 1) ticks are obtained while the server cycles the ring, giving each job a service tick. The last tick is obtained while the server cycles the portion of the ring starting at \textit{head} and terminating at the departing job, giving each job in this segment of the ring a single tick. Finally, a departure event based on the computed departure time is scheduled by inserting appropriate information into the simulation event calendar.

3.3 Performance Evaluation

A theoretical analysis of algorithms for implementing such disciplines is complicated by the fact that several factors must be taken into account, and some of these are difficult to measure. These include lightweight process context switching, cost of floating point computation, and the
adjust_pool(f)
{
    delta = clock - prev_clock;
    ticks = delta / q;
    q = ticks / size_of_pool(f);
    r = ticks % size_of_pool(f);
    for(ptr = head, i = 0; i<size_of_pool(f); i++) {
        ptr->remain -= q;
        ptr = ptr->next;
    }
    for(ptr = head, i = 0; i<r; i++) {
        ptr->remain -= 1;
        ptr = ptr->next;
    }
    head = ptr;
}
schedule_next_to_leave(f)
{
    pid = head->pid;
    minrem = head->remain;
    steps = 1;
    for(ptr=head->next, i=2; i<=size_of_pool(f); i++) {
        /* search for the process with min remaining ticks */
        if(ptr->remain < minrem) {
            minrem = ptr->remain;
            pid = ptr->pid;
            steps = i;
        }
        ptr = ptr->next;
    }
    schedule a departure event E for process pid at time
    (clock + (((minrem - 1) * size_of_pool(f) + steps) * q));
}

Figure 5: The computational RR algorithm
cost of operating a simulation event calendar in a particular environment. For a very simple comparison, assume that we only have to consider the three factors mentioned above. The execution time of the naive algorithm, say $T_{naive}$, and the execution time of the computational algorithm, say $T_{comp}$, can be expressed in terms of the quantities $t_{cs}$: the time for a context switch, $t_{fc}$: the time for a floating point operation, and $t_{eq}$: the time for an operation on the simulation event calendar. This relation is expressed by the equations:

$$T_{naive} = s_1 t_{cs} + s_2 t_{fc} + s_3 t_{eq}$$

$$T_{comp} = c_1 t_{cs} + c_2 t_{fc} + c_3 t_{eq}$$

where $s_i$, and $c_i$, $i = 1, 2, 3$, denote the number of operations of each type, for the two different algorithms, respectively. Unfortunately, these numbers are influenced by the number of customers simulated, the size of the time quantum $q$, parameters of the arrival and service-time distributions, etc. This complicates matters in an analytic derivation of $T_{naive}$ and $T_{comp}$ under general conditions.

An alternative scheme for evaluating the relative performance of these algorithms is to run benchmark models in a simulation environment. We conduct three experiments using the $Si$ system, which executes on SUN SPARC IPC workstations (each configured with 15.7 MIPS, 8 MB memory, and 64KB cache).

### 3.3.1 Benchmark I: A Single Server Model

The application program we use is for the execution of a single-server operating a single, unrestricted queue in round-robin fashion. This simple model has previously been used as a benchmark in comparing simulation packages [13]. The following input parameters are used in our experiments:

- quantum $q$.
- exponentially distributed job interarrival times with mean $1/\lambda$.
- Hyper-exponentially (two-component) distributed job service times with mean $1/\mu$. The distribution function $F(x)$ is given by

$$F(x) = \alpha F_1(x) + (1 - \alpha) F_2(x),$$

14
where $F_1$ and $F_2$ are exponential distributions with mean $1/\mu_1$ and $1/\mu_2$, respectively. The composite mean $1/\mu = \alpha/\mu_1 + (1 - \alpha)/\mu_2$.

**Experiment 1: Sensitivity to the quantum $q$**

The computational algorithm is more complicated than the naive algorithm in that it uses relatively complex data structures and operations to achieve the same results. This experiment attempts to measure the difference in performance between both implementations as the size of the quantum $q$ is varied. We chose $1/\lambda = 5$ and $1/\mu = 4$ by fixing $1/\mu_1 = 10$, $1/\mu_2 = 2.5$ and $\alpha = 0.2$.

**Experiment 2: Sensitivity to $\rho$**

This experiment is designed to see how the ratio $\rho = \lambda/\mu$ affects the performance of both implementations. The size of quantum $q$ is kept fixed in this experiment. In order to vary $\rho$, we change the parameter $\lambda$ while fixing $\mu$ at the value given in Experiment 1.
Figure 6: Sensitivity to the quantum $q$

Figure 7: Sensitivity to $\rho$
Interpretation of results

In both experiments, we measure the amount of CPU time required to simulate 10000 job completions at the single server queue. In each case, the experiment was repeated 25 times, with different random number seeds in each case, and the results were averaged; the results of both experiments can be seen in Figures 6 and 7, respectively. It should be stressed that the measured averages are not intended to represent the absolute performance of the algorithms but rather their relative performance for a particular parameter configuration. Thus the comparison of average times is of more interest than a comparison of raw numerical data.

The results of both experiments indicate that the computational algorithm outperforms the naive algorithm for all values of $q$, and $\rho \leq 0.98$. This is simply due to the fact that the cost of context-switching in the naive algorithm is high relative to operating with the more complex data structures and floating-point computations of the improved algorithm.

It is interesting to observe that when $q$ is small, the performance difference between the two algorithms is large, because a smaller value of $q$ implies that a larger number of context switches is required of the naive algorithm. On the other hand, the difference becomes small for values of $\rho$ close to 1; this is because the number of jobs in the pool is now significantly large, causing the computational algorithm to perform more work in maintaining the cyclic order and predicting departure times.

3.3.2 Benchmark II: A Multiple Queue Model

Here, we use a small generalization of the the previous model in that the system consists of several independent queues, and each queue is allowed a dedicated server which operates in round-robin fashion.

Experiment 3: Scalability

The purpose of this experiment is to determine how each algorithm responds to an upward scaling of queues, customers and servers. That is, we measure the effect of an increasing number of queues/servers on both algorithms. The multiple queue model, depicted in Figure 8, is used as our benchmark. This model is categorized as distributed/no splitting in [10], where distributed means that each server has its own queue, and no splitting means that a job cannot be decomposed into smaller tasks and should be run on a server until completion. In such a
A Multiple Server Model

system, jobs are assigned to the different servers with equal probabilities. We use the same workload as in Experiment 2 with $1/\mu = 4$, and $q = 1$; here $1/\lambda = 5n$, where $n$ is the number of queues/servers in the system. Thus, the mean job interval arrival time at each queue is $5n/n = 5$.

**Interpretation of results**

For the experiment just described, we measure the amount of CPU time required to process $2000 \times n$ job completions. The variable $n$ is incremented in steps of 10; for each value of $n$, the experiment is repeated 25 times, with different random number seeds each time, and the results averaged. Figure 9 shows how average execution time varies with the number of queues/servers. The computational algorithm consistently performs better than the naive algorithm, even attaining a 100% improvement in execution time for 100 servers. Both algorithms exhibit a significant performance difference when the number of queues/servers increases. Most of the cost exhibited by the naive algorithm can be attributed to the high cost of frequent context switches and event calendar operations.
4 The Processor Sharing Discipline

In the Processor Sharing (PS) discipline, jobs are scheduled for service as if the server were attending to all jobs simultaneously. Given a pool of $n$ jobs, each job receives service at a rate which is $1/n$ times the rate that is received by a job in a pool of size one, i.e., a rate which is inversely proportional to the number of competing jobs. In fact, the limiting behaviour of jobs in the RR discipline coincides with the PS discipline as the time quantum $q$ is made to approach 0. However the similarity is not of much practical use because it is infeasible to use either the naive RR or the computational RR algorithms to simulate a pool of $n$ PS jobs. Adopting the naive RR algorithm would almost certainly guarantee high execution costs because of a correspondingly large number of context switches, particularly for small $q$. This should be clear from the results seen in Experiment 1. On the other hand, the computational RR algorithm is also impractical because the potential for numerical error arising is high in the computation of $\delta t/a/q$, particularly for small $q$. Recall that this expression is used to determine the number of ticks that must be allocated to the pool of jobs between events. Therefore, it is apparent that an efficient PS algorithm is not directly obtainable from the RR algorithms discussed earlier. We now look at two different mechanisms for effecting the PS
discipline.

4.1 The Naive Algorithm

A naive algorithm for simulating the PS discipline is one that is based on its definition. That is, each job in the pool receives the same amount of service during any time interval. The term time interval refers to the interval of time between any two consecutive events which cause the state of the pool to change (i.e., either arrivals to or departures from the pool). Given an interval of length delta, the algorithm allocates portions of delta equally to all jobs in the pool, so that the portion of time, say share, received by one job is equal to the portion of time received by any other job in the pool. Hence at the end of such a time interval of length delta, each job in the pool has a remaining service time requirement that is less than its requirement at the beginning of the interval by the amount share. An algorithm that effects this procedure is shown in Figure 10.

It should be apparent that the algorithm described above suffers in that it is computationally demanding, though less so than the computational RR algorithm. Though the required computation is similar to the latter algorithm, the naive PS algorithm does not need to maintain a cyclic order of jobs. It does not need a ring data structure because all jobs in the pool are treated equally, regardless of when they enter the pool. A simple list structure is sufficient to preserve the necessary pool information. Each time the function adjust_pool() is invoked, the remaining service time of each job in the list is reduced by the quantity share. At the end of this adjustment, the job with minimum remaining service time is located through a simple linear search and its departure event from the pool is scheduled. This event is scheduled to occur at a departure time computed as

\[ \text{clock} + \text{minrem} \times \text{size_of_pool}(f). \]

That is, an amount of time equal to \( \text{minrem} \times \text{size_of_pool}(f) \) must elapse before this job can leave because the server must devote an equal amount of attention to all jobs in the pool.

4.2 A Lazy-Update Algorithm

We propose a simple lazy-update algorithm which is more efficient than the naive PS algorithm. Upon examining the code in Figure 10, we find that it is unnecessary to adjust
adjust_pool(f)
{
    delta = clock - prev_clock;
    share = delta / size_of_pool(f);
    for(i = 0; i<size_of_pool(f); i++) {
        ptr->remain -= share;
        ptr = ptr->next;
    }
}

schedule_next_to_leave(f)
{
    pid = head->pid;
    minrem = head->remain;
    for(ptr=head->next, i=2; i<=size_of_pool(f); i++) {
        if(ptr->remain < minrem) {
            minrem = ptr->remain;
            pid = ptr->pid;
        }
        ptr = ptr->next;
    }
    schedule a departure event E for process pid at time
    (clock + minrem * size_of_pool(f));
}

Figure 10: The naive PS algorithm
each job's remaining service time at the end of each time interval simply because the quantity being subtracted (i.e., share) is the same for each job. Instead of performing this subtraction exhaustively, we utilize a special variable to keep track of repeated share adjustments to the pool. This variable, say f.qty, is initialized to be zero when the PS facility is created. At the end of every time interval, the required amount of readjustment share is added to variable f.qty. Since we do not actually perform the subtraction, a snapshot of the contents of the pool at any given time will reveal list of nodes with remaining service time subfields remain that are not representative (i.e., contain values greater than or equal to) their true remaining service times. This property may be summarized as:

**Property 4.1** For each process in the pool, the true remaining service time is given by

\[
\text{remain} - f\text{.qty}.
\]

This simple idea is implemented by the code shown in Figure 11. Two operations are required to preserve the above property. Whenever a new process arrives and is added to the pool (list), the field containing its remaining service time must be incremented by the amount f.qty (and this is done when function insert_into_pool() is invoked). Whenever the departure time of the earliest process to leave the pool is to be computed (i.e., when function schedule_next_to_leave() is invoked), the addition performed in the previous step is undone by subtracting the quantity f.qty from the contents of each process's remaining service time subfield. Because of this modification, the only required operations for handling the pool are INSERT and EXTRACT MIN primitives. It can readily be concluded that the list structure we have advocated for maintaining the pool can be replaced by any other efficient priority queue structure, such as a heap structure. A final caution relating to this PS implementation is that the quantity f.qty should not be allowed to grow so as to cause numerical overflow; this is achieved by resetting f.qty to zero whenever the pool is empty, easily achievable when routine delete_from_pool() is invoked.

### 4.3 A Simple Static Analysis

To simplify the analysis, we assume a PS system with \( n \) initial jobs and no incoming jobs. In using the naive PS algorithm, \( n \) computations (i.e., subtractions) are required before the
adjust_pool(f)
{
    delta = clock – prev_clock;
    f.qty += delta / size_of_pool(f);
}
schedule_next_to_leave(f)
{
    ptr = find_min(f);
    pid = ptr->pid;
    minrem = ptr->remain - f.qty;
    schedule a departure event E for process pid at time
        (clock + minrem * size_of_pool(f));
}
insert_into_pool(f,t)
{
    ...
    pid->remain += f.qty;
    insert current running process's pid into pool f;
    ...
}
delete_from_pool(f)
{
    ...
    delete current running process's pid from pool f;
    if( size_of_pool(f) == 0)
        f.qty = 0;
    ...
}

Figure 11: The lazy-update PS algorithm
first job can depart from the system, \( n - 1 \) computations are required before the second job can depart from the system, etc. The total amount of computation required is thus \( \sum_{i}^{n} = O(n^2) \). On the other hand, the lazy-update algorithm requires a single computation (i.e., addition) to update the state (i.e., remaining service times of jobs) of the pool, and a single extract minimum operation to determine the departure time of the next job to leave the pool. Using a heap structure to maintain the facility pool of size \( n \), operations \textit{EXTRACTMIN} and \textit{INSERT} have a time complexity of \( \log(n) \) [1]. Reasoning thus, the total computational effort required by the lazy-update PS algorithm is \( O(n \log n) \). It is possible that a further reduction in complexity can be had by resorting to more efficient algorithms for pool maintenance. An example of such a structure is the calendar queue [3], in which the \textit{EXTRACTMIN} and \textit{INSERT} operations have empirically been shown to be of complexity \( O(1) \). It is a simple data structure, similar to a multiple list except for the fact that it does not require an overflow list. In using a calendar queue, the time complexity of the lazy-update PS algorithm could be further reduced to \( O(n) \).

4.4 Performance Evaluation

While the previous static analysis gives an indication of how the two algorithms perform relatively, it ignores the effects of incoming jobs. In this sense it does not reflect realistic job departures from the pool. To remedy this deficiency, we conduct two experiments using \( S_i \) to further compare the performance of both algorithms.

**Experiment 4: Sensitivity to \( \rho \)**

This experiment is similar to Experiment 2, where we examines how the performance of each algorithm varies with \( \rho = \lambda/\mu \). As before, job interarrival times and service times are assumed to be exponential random variables. The arrival rate parameter \( \lambda \) is allowed to vary while service rate \( \mu \) is kept fixed. We measure the average amount of CPU time required for the algorithms to process 10000 jobs. The average was computed using 25 runs, each time using different random number seeds.

**Experiment 5: Scalability**

In this experiment, we attempt to measure the effect of varying the number of queues/servers on each algorithm. This experiment is similar to Experiment 3. We use parameters \( 1/\mu = 4 \),
and $1/\lambda = 4.44n$, where $n$ is the number of queues/servers; this yields $\rho = 4/4.44 \approx 0.90$. We measure the average amount of CPU time required by the algorithms to process $2000 \times n$ jobs in the multiqueue/server model. The variable $n$ is increased in units of 10, and each average is estimated using 25 runs with different random number seeds.

**Interpretation of results**

The results obtained in Experiment 4 (shown in Figure 12) are consistent with our simple analysis. In essence, the lazy-update PS algorithm outperforms the naive PS algorithm for all values of $\rho$. Further, the difference in performance between both approaches increases with $\rho$, and can be significant for $\rho$ near 1. This can be explained by the fact that large values of $\rho$ correspond to large queues, and the naive PS algorithm suffers from an increase in computational work.

The results obtained in Experiment 5 demonstrate that simulation studies of large parallel systems based on the naive PS algorithm may suffer higher execution costs because the differences between both algorithms increases with an increasing number of queues/servers.

5 Conclusion

It is clearly the case that efficient techniques for implementing simulation scenarios can significantly enhance simulation executions. The approaches that we propose are versatile and can be applied in any discrete-event simulation view. We have implemented these algorithms successfully on the Si research testbed for simulation ideas and demonstrated the feasibility (especially in terms of performance) of efficient algorithms, and their robustness under varying conditions.

Of the two approaches shown for effecting RR service in simulation, the naive algorithm has the advantage of simplicity in description and implementation. It is this simplicity which causes high context-switch overheads under high loads. In contrast, the proposed computational algorithm requires more programming effort, but suffers less context-switching overhead. This conclusion is supported by Experiments 1, 2 and 3, which show improved performance for the latter algorithm. In particular, Experiment 1 demonstrates that the performance difference can be significant as quantum size $q$ decreases. A similar effect is seen in Experiment 2 as $\rho$ is made to approach unity.
Figure 12: Sensitivity to $\rho$

Figure 13: The effect of scaling
Of the two approaches shown for effecting PS service in simulation, the frequent pool state
updates performed by the naive algorithm can be replaced by infrequent updates. This is the
essence of the proposed lazy-update PS algorithm. By combining such infrequent updates with
the use of an efficient priority queue data structure, the lazy-update algorithm can be shown
to reduce the $O(n^2)$ complexity of the naive algorithm to $O(n \log n)$, or possibly even $O(n)$
through use of a calendar queue. This is supported by the results of Experiment 4.

It would be of some interest to see if the lazy-update approach can be applied in effecting
the RR discipline. Clearly, the unequal adjustment of time for each process in the RR pool
makes it difficult to use a single variable to represent the amount of adjustment for all jobs in
the pool.

References


via recurrence relations. In *Proceedings of Conference on Measurement & Modeling of


