Proceedings of the 2003 Winter Simulation Conference
S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, eds.

AN IMPROVED COMPUTATIONAL ALGORITHM FOR ROUND-ROBIN SERVICE

Jorge R. Ramos
Vernon Rego

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, U.S.A.

Janche Sang

Department of Computer and Info. Science
Cleveland State University
Cleveland, OH 44139, U.S.A.

ABSTRACT

We present an efficient algorithm for effecting round-robin service in discrete-event simulation systems. The approach generalizes and improves upon a previous approach in which a single arrival and a single departure event is considered and handled at a time; further, the previous approach is already an improvement over naive round-robin scheduling currently in use in simulation libraries. The prior proposal offered a run-time complexity of $O(n^2)$, because the processing of each event required an entire traversal of the job pool. We propose a generalized algorithm which includes the previous case and also accommodates burst arrivals and batch departures, further reducing run-time complexity to $O(n \log n)$. This is achieved through a detailed but efficient computation of multiple departure times, while simultaneously obviating the need for a job pool update with each departure. Empirical results are presented to compare performance with previously proposed algorithms.

1 INTRODUCTION

Good simulation models are a powerful tools used to answer performance questions related to waiting-line phenomena (Schwetman 1988), that are difficult to answer using other means. Because these models are usually computationally intensive, efficient techniques for implementing algorithms, particularly those that reduce run-time, are very useful as simulation execution enhancements.

The round-robin(RR) service discipline is a popular and widely used discipline in many real-world time-sharing systems because of its fairness. In this discipline, a job or customer is serviced for a single quantum q at a time. If the remaining service time required by a job exceeds the quantum size q, the job's processing is interrupted at the end of its quantum and it is returned to the rear of the queue, awaiting service quantum in the next round. A naive approach to implementing the RR discipline in simulation is to physically dole out service quanta to the jobs in a round-robin fashion. For jobs with very large service times this leads to very high event-handling overhead or context-switch overhead in event-based simulations (McHaney 1991, Fishman 2001), and in process-oriented simulations (Law and Kelton 2000), respectively.

The high cost of event-scheduling in the naive RR approach can be greatly reduced through a computational device that was introduced in Sang, Chung, and Rego (1994). The idea is to run an algorithm which first predicts and then schedules the next departure from the state of the system which is defined by the remaining service requirement of each job, the number of jobs and the next job in line for service. A simple analysis shows that if an RR system has n jobs in service and no more jobs enter the pool, the time complexity of the algorithm is $O(n^2)$. In this paper, we develop a novel batch departure computation in which multiple departures can be scheduled without having to update the state of the pool on each departure. This yields a new algorithm which further reduces simulation time complexity to $O(n \log n)$.

The remainder of the paper is organized as follows. In Section 2, we examine the components of the original single-departure computational algorithm. In Section 3, we develop a new batch departure formula which can significantly reduce simulation time. We also analyze the problem of cancellations in the batch departure formula and introduce the concept of "look ahead" as a desirable primitive in a simulation kernel. In Section 4 we present an algorithm to handle the update of state and insertion of new arrivals after a batch departure. In Section 5 we present the results of several experiments, comparing performance with previously proposed algorithms. Finally, we present a brief conclusion in Section 6.

2 THE SINGLE-DEPARTURE COMPUTATIONAL ALGORITHM

An appropriate representation of the job pool in the computational algorithm is a circular linked list, which the server traverses in a circular fashion. The original algorithm presented in Sang, Chung, and Rego (1994) keeps three fields for each job record: a process identifier (PID), remaining service-time, and a link to the next element in the circular list. An additional pointer HEAD, which is associated with pools of jobs, is used to indicate the job which is to receive the next quantum of service.

The computational algorithm uses a scheduling function, that traverses the pool and determines the next job to depart — the one with minimum remaining service time. It keeps track of this time ($minrem$), and also counts the number of steps ($steps$) between the current head (i.e., the current job to be serviced) and the job to depart. Upon determining these two parameters, the next departure is identified by using the size of the pool ($poolsize$) and service quanta (q) through a simple formula:

$$current\_time + ((minrem - 1) \times poolsize + steps) \times q. \tag{1}$$

Consider a pool of jobs (A, B, C, D, E) that have remaining service-time (5, 6, 3, 4, 8) respectively and HEAD points to A. it is easy to see that the next job to leave the system is C, if no new arrivals occur prior to the departure, after $(3 - 1) \times 5 + 3 = 13$ quanta. Observe that if an arrival event occurs before the scheduled departure event, the scheduled departure event has to be cancelled.

3 NEW BATCH DEPARTURE ALGORITHM

The original single-departure computational algorithm identifies one potential departure event and handles one arrival event at a time. We propose a novel algorithm in which we consider the possibility of processing burst arrivals and batch departures, to handle the simulation of models with bursty traffic. Figure 1 illustrates the difference between the original algorithm and the new batch algorithm in terms of the number of events. The new formula is a generalization of the formula used in the single-departure algorithm. A simple approach, such as repeatedly applying the Formula 1 presented in Section 2, is not trivial, since there is no easy way to keep track of the different remaining service quanta for jobs in the pool and the position of the head as elements get removed.

3.1 Derivation of the Generalized Formula

The new method exploits complete traversals of a somewhat artificially simplified pool, where the head and pool size are determined by the initial state, and used for all subsequent computations. With this approach we only need to subtract the extra quanta to get an exact solution. The subtraction of quanta can be expressed mathematically, and can thus be easily accounted for. Further, all departures from the pool can be scheduled solely from initial state information. The formula will be derived assuming that no new jobs arrive between the invocation of the algorithm and the scheduled n departures. We will, however, address the issue of new arrivals later.

We build a table T that contains the pool elements, the remaining service times, the relative positions and the relative order of distinct departures. The scheduled departure time of the next job from the pool is then given by:

$$current\_time + [v_i \times size\_of\_pool - (term1) - (term2)] \times q,$$

where v_i = remaining time for job i, and (term 1) and (term 2) are explained in detail below.

This formula reflects three constraints imposed on the pool, namely:

- Complete traversal of the fixed-size pool
- Extra steps due to counting elements that have already departed, which is term 1
- Extra steps due to traversal from a particular element to tail, which is term 2.

We now examine these terms in detail, assuming a pool of n elements.

Term 1: For a (potentially) departing job i ($i = 1, 2, ..., n$), term 1 is given by:

$$\sum_{j=1}^{i-1}(v_i - v_j) = (i-1) \times v_i - \sum_{j=1}^{i-1} v_i.$$

On the left side of the equation, we subtract the extra steps counted between the current job i and all previously departing jobs j. The expression on the right side of the equation is the one useful for implementation of the algorithm, since the term $\sum_{j=1}^{i-1} v_i$ can be stored in a single variable.

Term 2: Term 2 is a little more difficult, as we will shortly see. For departing job i ($i = 1, 2, ..., n$), term 2 is given by:

(Pool_size_at_round_i) – (Relative position of job i to head).

The pool size at a given round is obtained as (remembering that Pool_size is fixed at the instant the algorithm is invoked):

$$Pool\_size\_at\_round\_i = (Pool\_size - i + 1). \tag{2}$$

For relative positions, we take into account that the relative positions of jobs change as jobs are removed and scheduled for departure, and that the relative positions of jobs stationed between the head and a departing job don't change, though the positions of others must change.

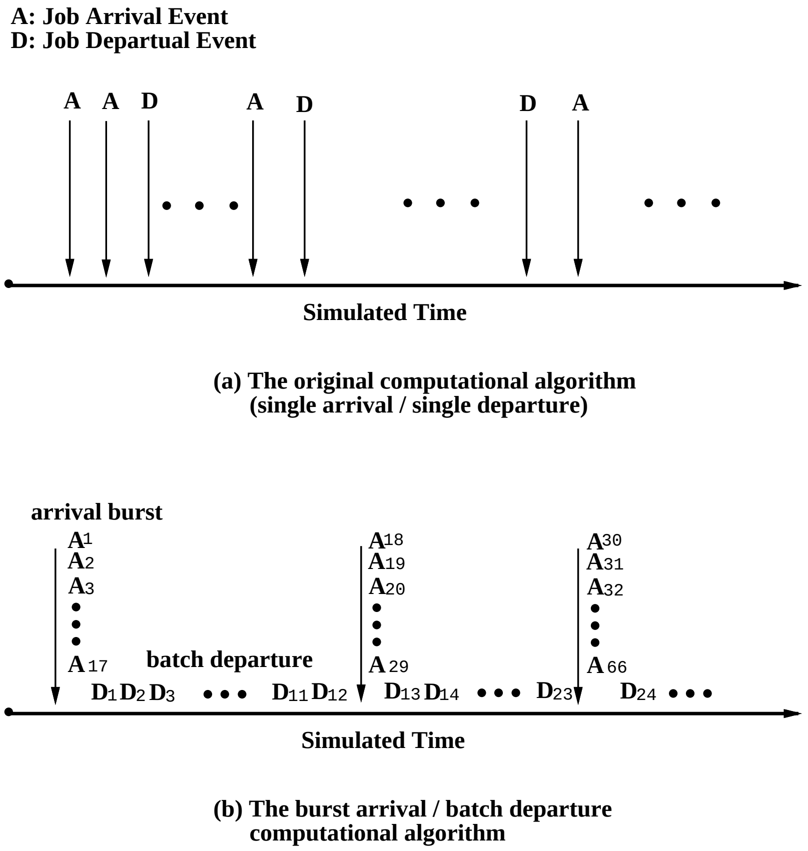

Figure 1: A Comparison of Single Arrival/Departure and Batch Arrival/Departure

Let p_i be the position of job i relative to the fixed head at the start of algorithm's invocation on the pool. Based on the above explanation, the new relative position is determined as:

$$relative\ position = p_i - \phi(p_i), \tag{3}$$

where the function $\phi(p_i)$ defines the number of jobs with relative positions smaller than p_i that have already departed. A straightforward approach to computing $\phi(p_i)$ by one-by-one comparison will drive the time complexity of the algorithm up to $O(n^2)$. Therefore, we have to resort to a more efficient but also more complicated data structure for the calculation of $\phi(p_i)$. We propose the use of a widely-used data structure, called augmented red-black tree or order-statistic tree (Cormen et al. 2001), which can support fast rank operations. The rank operation can be done in a time that is proportional to the height of the red-black tree, i.e. in $O(\log n)$ time.

Given the rank, the relative position can be easily calculated by following relationship:

$$\phi(p_i) = rank(p_i) - 1.$$

Hence, computing $\phi(p_i)$ can also be done in $O(\log n)$.

Combining Equation 2 and Equation 3, we obtain a final form for term 2:

$$(pool\_size - i + 1) - p_i + \phi(p_i).$$

With Term 1 and Term 2 thus defined, we are finally in a position to define a precise expression for the departure time of job $i (i = 1, 2, ..., n)$ for the batch-departure case (the batch-departure formula), which has the following form:

$$\boxed{\begin{aligned} & current\_time + [v_i \times pool\_size - ((i-1) \times v_i \\ & - \sum_{j=1}^{i-1} v_i) - ((pool\_size - i + 1) - p_i + \phi(p_i))] \times q. \end{aligned}}$$

This formula is computationally simple to implement. When invoked, it yields departure times starting from departure $i = 1$ to departure $i = n$, based solely on the initial state of the pool.

3.2 Computational Algorithm with Look-Ahead

In the original computational algorithm, if an arrival event occurs before a scheduled departure, the departure event is cancelled to preserve consistency of the pool. In replacing single departures with batch departures, if cancellations are used when arrivals occur, we may arrive at a situation where we compute and schedule the departure times of a large number n of jobs only to later find that nearly all such departures must be cancelled because of arrivals. This problem can be solved by resorting to a special look-ahead primitive which looks ahead in the simulation to determine the time of the next arrival. This makes for an efficient computation that determines only what is needed through constant monitoring via look-ahead, and because this method does not alter a simulation's trajectory, the resulting simulation produces consistent results.

The algorithm has three major steps. Firstly, we traverse the queue and build a table T containing relative positions and remaining service times. Secondly, we sort the table (using an efficient sorting algorithm such as quicksort) in increasing order of remaining service times v_i, determine the relative departure order and put it in the table T. After obtaining necessary information, we use the batch departure formula to schedule batch departure events. The algorithm terminates when it completes the departure time computation for each of the jobs in the table T or when the next arrival time (via look-ahead) is reached.

Both the computational algorithm and the naive algorithm yield the same results, serving to verify that the computational algorithm is indeed a correct and more efficient $O(n \log n)$ algorithm for the prescribed task.

3.3 An Example

Consider the following illustration of the use of the batch-departure formula. The traversal is done from left to right to obtain a table T containing the remaining service times v_i, the relative positions p_i and the departure order i. After sorting T by remaining service time v_i, we obtain the data shown in Table 1.

Table 1: Sample Pool

Job PID:	C	D	A	B	E
Remaining time v_i	3	4	5	6	8
Departure order i	1	2	3	4	5
Relative position p_i	3	4	1	2	5

Now applying the batch departure formula applied for $i = 1$ through $i = 5$, we get:

- **Departure 1**: Job C, $i = 1$:
  $v_i \times$ pool_size = $3 \times 5 = 15$
  term1 = $(i-1) \times v_i - \sum_{j=1}^{i-1} v_i = 0 - 0 = 0$
  term2 = (pool_size $-i+1) - p_i + \phi(p_i) = (5 - 1 + 1) - 3 + 0 = 2$
  departure = current_time + $[15 - 0 - 2]q$ = current_time + 13 q
- **Departure 2**: Job D, $i = 2$:
  departure = current_time + $[20 - 1 - 1]q$= current_time + 18 q
- **Departure 3**: Job A, $i = 3$:
  departure = current_time + $[25 - 3 - 2]q$= current_time + 20 q
- **Departure 4**: Job B, $i = 4$:
  departure = current_time + $[30 - 6 - 1]q$ = current_time + 23 q
- **Departure 5**: Job E, $i = 5$:
  departure = current_time + $[40 - 14 - 0]q$ = current_time + 26 q.

A graphical explanation of the computation is demonstrated in Figure 2. Consider the computation of the third departure, i.e. Job A departs. A total of 25 ticks are doled out to 5 jobs because Job A requires 5 ticks service time. These 25 ticks include 3 extra ticks (i.e. term1), as shown circled in Figure 2(a), given to C and D even after they have been marked as having departed. Furthermore, because A's relative position (i.e., p_i=1) is smaller than C's (i.e., 3) and D's (i.e., 4), the value of A's $\phi(p_i)$ is 0. This results in $term2 = 3 - 1 - 0 = 2$. There are two extra ticks, marked by rectangles in the last row of Figure 2(a), distributed to other in-pool jobs stationed after A (i.e., Jobs B and E). Thus, deducting these 5 extra ticks from the total of 25 ticks, we obtain the value 20. A similar calculation for Job B's departure is depicted in Figure 2(b). Note that the value of B's $\phi(p_i)$ is 1 because, among jobs that have already departed, only A has a smaller relative position (i.e., 1) than B (i.e., 2). Hence term2, which is $2 - 2 + 1 = 1$, shows that one extra tick is given to an in-pool job (i.e., Job E). Subtracting the extra quanta in term1 and term2, we obtain the value 23 for Job B.

4 HANDLING CHANGES OF STATE

In the original single-departure computational algorithm ($n = 1$), upon departing, the job updates the pool to the correct state at the simulation time of the event and leaves the pool. For the batch departure case ($n > 1$), we have to find a way of obtaining all necessary information in a single traversal of the pool and then update all jobs accordingly. Since each departure must have a corresponding departure-event, any one of these n departure events may be used to update the state of the pool. Different discounts have to be applied to different elements in the pool, depending on the position of the head with respect to these elements.

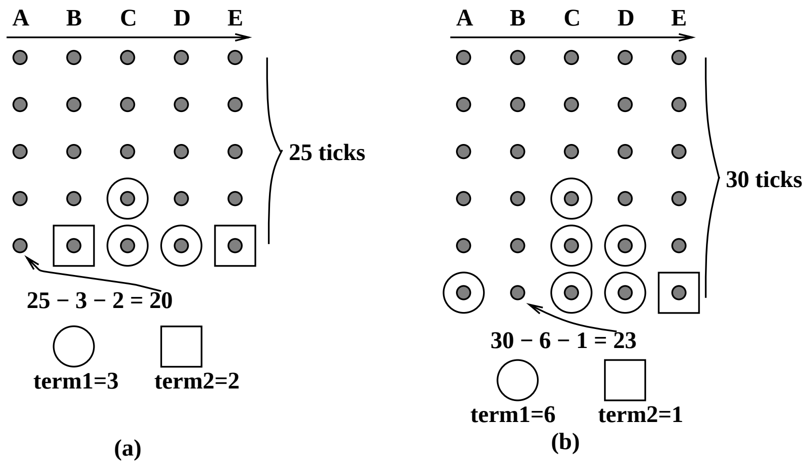

Figure 2: Sample Departures - (a) Job A Departs, (b) Job B Departs

The algorithm is as follows: Assume that there are n jobs to depart in a batch. The remaining service-time of the nth job will be used as the discount quantity. Next, simply traverse the pool from head to tail, subtracting $discount$ for each job lying between head and the nth job and subtracting $discount - 1$ for the rest. Those jobs that have remaining service-times less than or equal to zero are deleted from the pool, i.e., they have been scheduled for (potential) departure. Once the update is done, the pool is reindexed and the new head is defined. The pool is then ready for the next invocation of the update algorithm.

Consider the batch departure of jobs C and D in our previously defined example. Since job D is the last to leave in the batch departure, we use D's remaining service time (i.e., 4) as the quantity $discount$. Applying the proposed algorithm we get the results in Table 2.

Table 2: Pool after Departure of C and D

Job PID	A	B	C	D	E
Remaining time v_i	5	6	3	4	8
Relative position p_i	1	2	3	4	5
discount (–)	4	4	4	4	3
Updated remaining time v_i	1	2	–1	0	5

After deleting all jobs with zero or negative remaining service-times from the pool (i.e., C and D), we obtain an up-to-date pool with consistent state at $time = clock + 18q$, as shown in Table 3. The new head will now point to job E.

Table 3: Updated Pool

Job PID	A	B	E
Remaining time v_i	1	2	5

5 PERFORMANCE EVALUATION

We ran a number of experiments to evaluate the performance of the batch departure algorithms. A single, unrestricted queue served in round-robin fashion was used to implement and test the algorithms. Further, the algorithms were implemented within an application-layer residing above the kernel of a thread-based process-oriented simulator based on the Purdue Ariadne threads library (Mascarenhas and Rego 1996). The input parameters used were quantum q, exponentially distributed job interarrival times with mean $1/\lambda$, exponentially distributed job departure times with mean $1/\mu$, and discretized exponentially distributed batch sizes with mean $1 + 1/\beta$. The output parameter measured was the amount of CPU time required to do the simulations, given specific values for the input parameters described above. Several variations of the proposed algorithms were implemented within the application-layer on the simulator kernel, to evaluate the performance of the different ideas presented in the paper. To help identify the different runs, we use the following notation:

- **orCA** - the original single-departure computational algorithm
- **nuBD** - the batch departure formula with one-departure at a time
- **buBD** - the batch departure formula with batch departures
- **BD** - nuBD or buBD.

Each experiment was repeated 20 times with different random number seeds for each run, and the results then averaged. As explained in Sang, Chung, and Rego (1994), the use of averages does not represent the absolute performance of the algorithms but rather their relative performance given a particular configuration of parameters.

5.1 Benchmarks

The first experiment was designed to evaluate the behaviour of the different variations of the algorithms to arrivals that occur one at a time.

5.1.1 Experiment 1: Sensitivity to Traffic Intensity with Single Arrivals

The purpose of this experiment was to measure the performance of the algorithms as the ratio $\rho = \lambda/\mu$ is varied. The service time, $ST = 1/\mu$ and the number of jobs N were fixed at values $ST = 200$ and N=20,000 jobs, while λ was varied. The results are shown in Figure 3.

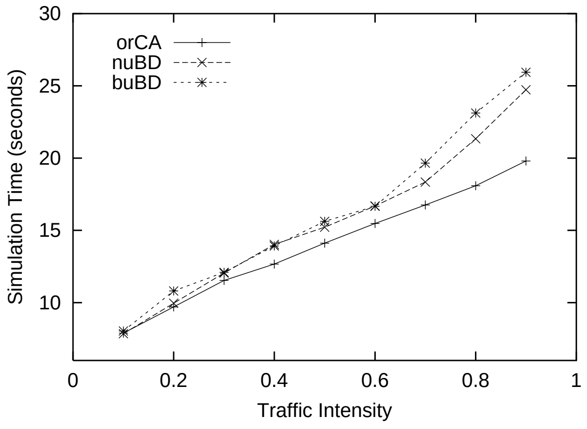

Figure 3: Simulation Time vs. Traffic Intensity for Single Arrivals

The following experiments were designed to evaluate the behavior of the different variations of the algorithms for arrivals that occur in distinct batches. For each arrival event, where interarrival mean is $1/\lambda$, a (discretized exponential) batch size BA with mean $1 + 1/\beta$ was defined, and BA arrival events were generated. The service time ST, with mean $1/\mu$, was divided by BA to obtain the service time for each job in an arriving batch to ensure system stability.

5.1.2 Experiment 2: Sensitivity to Batch Size

The purpose of this experiment was to measure the performance of the algorithms as batch size BA is varied. The systems evaluated include orCA, nuBD and buBD. The parameters used were fixed $1/\lambda$, $1/\mu = 160$, with $N = 10,000$ jobs, while $1 + 1/\beta$ was varied. The experiment was repeated for different values of ρ, by varying $1/\lambda$, with results for $1/\lambda = 200, 320$ and 533 ($\rho = 0.8, 0.5$ and 0.3) shown in Figures 4, 5 and 6, respectively.

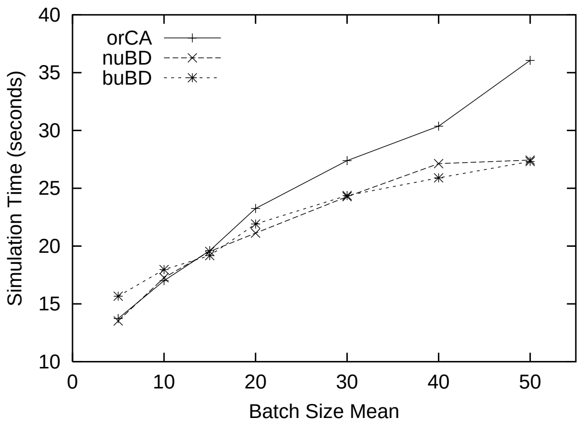

Figure 4: Simulation Time vs. Batch Size (ρ = 0.8)

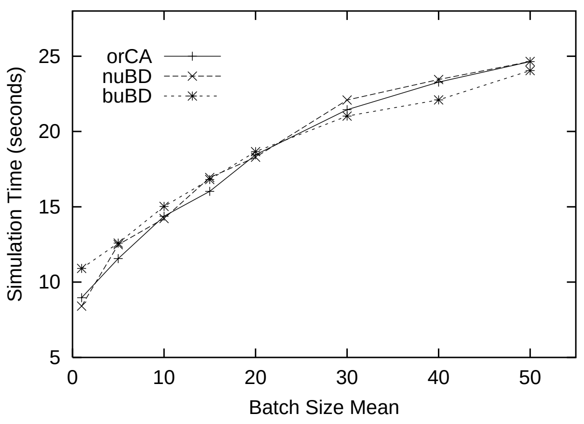

Figure 5: Simulation Time vs. Batch Size (ρ = 0.5)

5.1.3 Experiment 3: Sensitivity to Traffic

The purpose of this experiment was to measure the performance of the algorithms with the batch size BA fixed, while α is varied. The systems evaluated include orCA, nuBD and buBD. The parameters used were $(1 + 1/\beta) = 30$, $1/\mu = 160$ with $N = 10,000$ jobs, and $1/\lambda$ was varied. The results are shown in Figure 7.

5.1.4 Interpretation of Results

The main cost incurred by the orCA algorithm is due to repeated traversals of the pool and cancellations of many departures. The main cost incurred by the batch departure algorithm is due to sorting. The results indicate the relative cost of these two algorithms, and how costs change with the pool size. Experimentally, the pool size grows with both large batch arrival sizes and a high traffic intensity.

Our experiments enable us to identify three different performance regions:

- The first region involves a single arrival at a time, or very few arrivals. This behavior is witnessed in Experiment 1. In this region, the pool sizes

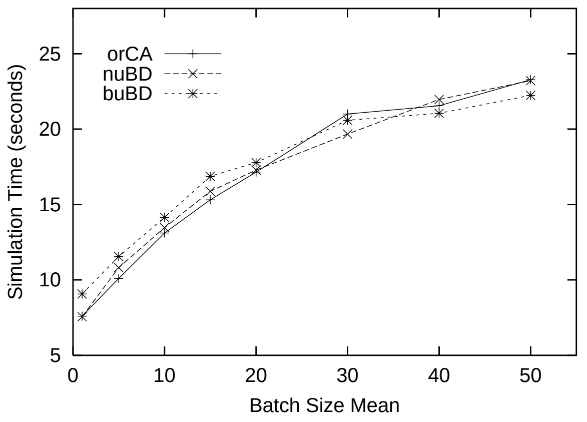

Figure 6: Simulation Time vs. Batch Size (ρ = 0.3)

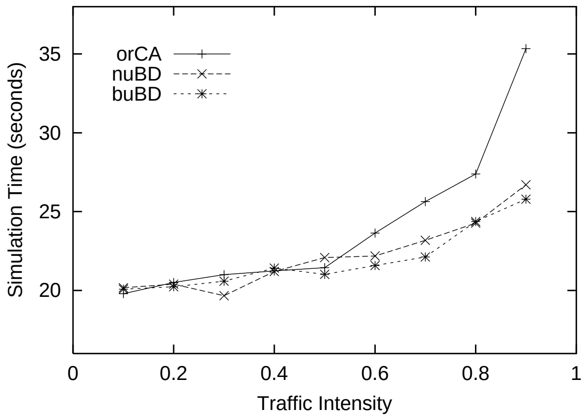

Figure 7: Simulation Time vs. Traffic Intensity for Batch Arrivals

are relatively small, and it is cheaper to traverse the pool many times instead of performing sorting operations. So orCA performs better than BD.

- The second region involves low to medium traffic intensity. According to the figures, this is caused by two variations: (1) $\alpha = 0$ to 0.5 and batch arrivals of any size; (2) batch arrivals, with batch-size below a critical size (in the case of the experiments, this is mean batch-size < 20). This is the behavior witnessed in Experiments 2 and 3. In this region, we have pools of moderate size, and the cost of sorting is roughly the same as the cost of traversing the pool repeatedly. Here, orCA and BD peform equally well.
- The third region involves medium to high traffic intensity (according to the figures, this corresponds to the experiments with $\alpha = 0.5$ to 1) and a batch size over a certain threshold (here, mean batch-size > 20 for the experiments). In this region it costs significantly more to traverse a large pool repeatedly than to perform a sort operation. Thus, BD offers better performance.

The reason why the performance-behavior of the different algorithms reverse when going from region I to region III is that the repeated traversal of a large pool exhibits a theoretical asymptotic growth rate of $O(n^2)$, whereas a sort operation with n red-black tree insert/rank operations can be done both in time $O(n \log n)$. Thus, for large pool sizes, the sorting and red-black tree algorithms tend to offer better performance. The regions are clearly demarcated in Table 4.

Through our experiments, we have determined that the batch departure formula-based algorithm works better than the original single-departure computational algorithm for traffic intensities $a > 0.5$ and batch sizes $BA > 20$, which includes situations of burstiness and high traffic. Examining region III, we see that here traffic intensity approaches 1 and more cancellations tend to occur for orCA. Also, the difference between BA and orCA increases with increasing pool size.

6 CONCLUSION

It is well-known that simulations of CPU scheduling or general waiting-line models can consume large amounts of processing time, especially when discrete-event simulations are supported by threads-based systems. We exploit the fact that a reduction in the number of scheduled events offers a correspondingly sharp reduction in simulation time. We built upon an algorithm we proposed previously, namely, a computational algorithm based on a formula which predicts the next (potential) job departure. This was shown to be a significant improvement over a simple schedule of re-entrant events for simulating round-robin service. Here, we generalized the idea to batches, to make efficient simulations that accommodate traffic that occurs in bursts. By obtaining a new batch-departure formula, we conclude that it is possible to reduce simulation run-time even further. The idea of infrequent pool-state updates reduces the time complexity from $O(n^2)$ to $O(n \log n)$, and our experiments show the idea to be effective. Further, the empirical results show that the new algorithms perform significantly better that the original single-departure computational algorithm, especially when traffic intensity is high. We believe that generalizations to the multiprocessor case are achievable.

ACKNOWLEDGMENTS

Research supported in part by DoD DAAG55-98-1-0246 and PRF-6903235.

REFERENCES

Cormen, T., C. Leiserson, R. Rives, and C. Stein. 2001. *Introduction to Algorithms, Chapter 14: Augmenting Data Structures*. 2d ed. Boston, MA: McGraw-Hill.

Table 4: Performance of Round-Robin Algorithms

		Traffic intensity	
Service discipline		0 – 0.5	0.5 – 1
Single arrivals		Region I: orCA has better performance	
Batch arrivals, with mean size	1– 20	Region II: equal performance for orCA and BD	
	> 20		Region III – BD performs better

Fishman, G.S. 2001. *Discrete-Event Simulation*. New York: Springer-Verlag.

Law, A. and W.D. Kelton. 2000. *Simulation Modeling and Analysis*. 3d ed. Boston, MA: McGraw-Hill.

Mascarenhas, E. and V. Rego. 1996. Ariadne: Architecture of a Portable Threads system supporting Thread Migration. *Software - Practice and Experience* 26(3):327–357.

McHaney, R. 1991. *Computer Simulation: A practical perspective*. San Diego, CA: Academic Press. 1991.

Sang, J., K. Chung, and V. Rego. 1994. Efficient algorithms for simulating service disciplines. *Simulation Practice & Theory* 1:223–244.

Schwetman, H.D. 1988. Using CSIM to model complex systems. In *Proceedings of the 1988 Winter Simulation Conference*, ed. M. Abrams, P. Haigh, and J. Comfort, 246 – 253. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers.

AUTHOR BIOGRAPHIES

JORGE R. RAMOS is a Ph.D. student in Computer Sciences at Purdue University. He received a Masters degree in Computer Sciences from Purdue University in 2003. His current research interests include simulation systems, bandwidth trading, real-time data mining and knowledge discovery. His e-mail address is `<jrramos@cs.purdue.edu>`.

VERNON REGO is a Professor of Computer Sciences at Purdue University. He received his M.Sc.(Hons) in Mathematics from B.I.T.S. (Pilani, India), and an M.S. and Ph.D. in Computer Science from Michigan State University (East Lansing) in 1985. He was awarded the 1992 IEEE/Gordon Bell Prize in parallel processing research, and is an Editor of *IEEE Transactions on Computers*. His research interests include parallel simulation, parallel processing and software engineering. His e-mail address is `<rego@cs.purdue.edu>`.

JANCHE SANG received the Ph.D. degree from Purdue University, W. Lafayette, IN, in 1994, and then joined the Department of Computer and Information Science, Cleveland State University, where he is currently an associate professor. His research interests include parallel and distributed computing, networks, software engineering, and simulation. He has received research grants from NASA, NSF, OBOR, and OSC. His e-mail address is `<sang@cis.csuohio.edu>`.