2002

An Efficient Burst-Arrival and Batch- Departure Algoriths for Round-Robin Service

Jorge R. Ramos

Janche Sang

Report Number:
02-030

AN EFFICIENT BURST-ARRIVAL AND BATCH-DEPARTURE ALGORITHM FOR ROUND-ROBIN SERVICE

Jorge R. Ramos
Vernon Rego

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR #02-030
December 2002

An Efficient Burst-Arrival and Batch-Departure Algorithm for Round-Robin Service*

Jorge R. Ramos
Vernon Rego
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Janche Sang
Department of Computer and Info. Science
Cleveland State University
Cleveland , OH 44139

Abstract

Simulations of CPU scheduling or waiting-line models that involve a server dispersing shared service quanta across jobs can require significant processing time, especially when simulations are supported by thread-based systems. To effect a reduction in simulation time we reduce the number of scheduled events, without affecting simulation results. We present an algorithm for such enhanced round-robin service in discrete-event simulation and implement and test it on a threads-based simulator. The algorithm predicts potential job departures and schedules them in advance, using cancellation and rescheduling when necessary. We generalize and improve upon a previous approach in which a single arrival and a single departure event is handled at a time. While the prior proposal offered a run-time complexity of $O(n^2)$, the new algorithm accomplishes this in time $O(n \log n)$. Further, the generalization also accommodates burst arrivals and batch departures with the reduced time complexity. Empirical results are presented to compare performance with previously proposed algorithms.

Keywords: scheduling, event, threads, kernel, red-black, lookahead, batch arrivals, batch departures

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

1 Introduction

Service disciplines such as first-in first-out, highest-priority first, round-robin etc. are basic parameters in the design of computing and communication systems. They specify how a service mechanism (server) attends to work (task execution, packet processing) in a queueing system. As such systems become increasingly complicated in functionality, it is hard for analytical models to incorporate and account for all the underlying and often interrelated factors. In most instances of practical interest the best approach is to rely on good simulation models which answer questions related to waiting-line phenomena[1]. Because such simulations are usually time consuming, techniques for implementing faster algorithms are particularly useful.

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 in order to share the service resource with other jobs requiring the same service. 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 the service quantum it will receive in the next round. A naïve approach to implementing the RR discipline in simulation is to physically dole out service quanta to the jobs in a round-robin fashion. For a job with a large service request, this approach necessarily schedules many arrival and departure events in the simulation calendar; this leads to very high event-handling overhead in event-based [2] and process-oriented simulations [3]. Because of associated context-switching, costs are particularly high in thread-based simulation systems.

The high cost of event-scheduling in the naïve RR approach can be greatly reduced through a computational device that was introduced in [4]. The idea is to run an algorithm which first predicts and then schedules the next departure from the system. A view illustrating the differences between the naïve and computational RR approaches is shown in Figure 1. This single-departure computational algorithm improves upon the naïve version by utilizing the notion of state. The state of the pool is defined by the remaining service requirement of each resident job, the number of jobs and the next job in line for service. Each arrival and each departure changes the state of the pool, and necessitates a state update. The computational algorithm exploits a formula that enables the determination of the identity of the next job to (potentially) leave the pool based on pool state; this job's departure is then scheduled. Upon the next arrival or departure, the state of the pool is updated. If an arrival occurs before the scheduled departure, the departure event is cancelled to preserve consistency of the pool. A similar idea has appeared in earlier studies of interrupt processing [5] and hybrid modeling [6, 7]. Large simulation run-times — especially in thread-based systems with context-switching overheads — warrant efficient algorithms for the simulation of service disciplines.

In the single-departure computational RR algorithm [4], a traversal of the pool is required for each departure. A simple analysis shows that if a 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 take a different approach and develop a novel batch departure computation which determines the identity of multiple departures using special index fields augmenting the state of the pool. With the aid of such a computation, multiple departures can be scheduled without having to update the state of the pool on each departure, causing a reduction in the number of scheduled events and hence, the simulation time. The key, however, is to dynamically determine which events can be eliminated from simulation-calendar processing. This yields a new algorithm which further reduces simulation

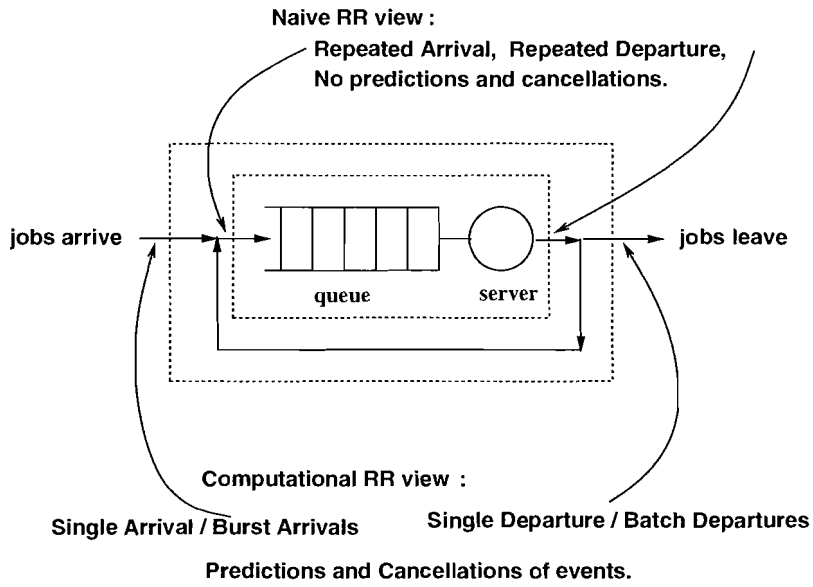


Figure 1: A Comparison of naïve and computational RR approaches

time complexity to $O(n \log n)$.

To test variations of the algorithm, we implemented it on a thread-based process-oriented simulator that is much like CSIM [1]. The system is layered, so that a simulation-kernel is supported by a thread-system in a modular way. Between the application layer — which is the level at which the user develops applications — and the kernel layer, is the domain layer. While various domains may be supported, our algorithm pertains to a queueing domain which offers the user primitives that deal with customers (jobs) and servers (CPUs). Given a domain layer, both naïve as well as computational versions of RR can be implemented as functions in a highly portable way. Indeed, the function is easy to develop given the algorithm, as long as there is a way to access pool state and the simulation calendar. As a practical matter, our tests were done on a simulator which is also highly portable: if the thread-system is portable then the entire system is portable because the layers are developed in the C language.

The remainder of the paper is organized as follows. In Section 2, we examine the components of the single-departure computational algorithm and illustrate improvements to be had by exploiting an index field for each job in the pool. In Section 3, we develop a new batch departure formula which can significantly reduce simulation time. The complete pseudo-code for departures, arrivals and the expected run-time of the algorithm is also presented. 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.

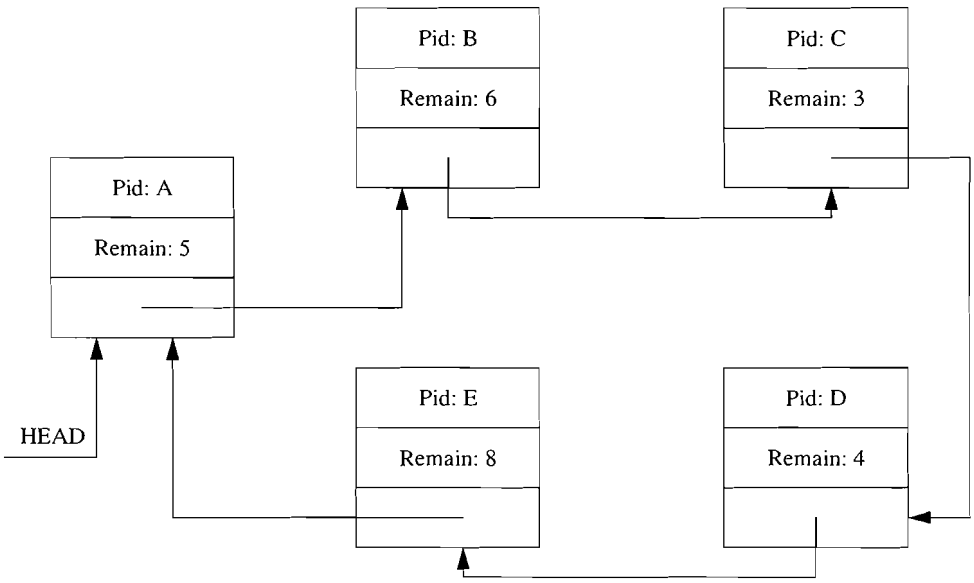


Figure 2: A circular queue for representing the Round-Robin service discipline

2 The Single-Departure Computational Algorithm and Improvements

In order to do a simulation in which the random number of quanta a job receives is not geometrically distributed, the service time of the job must be known. At the very latest, the scheduler must know this by the time the job starts its first quantum, so that its departure can be scheduled. Of course, we cannot know this time in a real system, but that is why we use various distributions in simulation. This view is an advantage of modeling. The random time (or random number of quanta) is selected before the job starts service.

When a number of jobs have already begun service, they reside in the pool. The scheduler knows that the job that will be first to leave the pool/system will be the job with the smallest number of remaining quanta that it needs served. This is not the same as a simple SJF (Shortest-Job-First) algorithm, because the server moves through the pool in cyclic fashion, so there may be several jobs with the minimum quanta requirement. The one to leave first will hold a unique position relative to the cyclic moving server in the pool.

In the following paragraphs, the single-departure computational round-robin algorithm presented in [4] is explained. The presentation is deliberately concise to avoid repetition. The interested reader is encouraged to consult the original paper for further explanations.

To determine the identity of the next potential departure from the pool, a computational algorithm needs to maintain the state of each job (or process) in the service pool. An appropriate representation of the job pool is a circular linked list, which the server traverses in a circular fashion. As illustrated in Figure 2, the algorithm presented in [4] 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. For clarity, we will use the example in Figure 2 to explain the main idea throughout the paper.

The computational algorithm consists of two basic functions:

- **ADJUST_POOL**: For an arrival or departure event, this function traverses the pool and adjusts remaining service-times, updating the state of the pool according to the time that has elapsed between two state updates (delta). Two adjustments are made. An amount of time is uniformly subtracted for each job — the number of complete traversals of the queue that occurred during the time delta. Next, for only those jobs stationed between the old head and the new head, one additional tick of time is subtracted.

- **SCHEDULE_NEXT_TO_LEAVE**: This function 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 (*pool_size*) and service quanta (q) through a simple formula:

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

Using the circular list in Figure 2 as an example, 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 [1]. When an arrival or departure event occurs, the algorithm first invokes the function ADJUST_POOL to update state. Next, it either inserts the newly arriving job into the pool or removes the departing job from the pool. The function SCHEDULE_NEXT_TO_LEAVE is invoked in the last step to predict the next departure from the pool. Observe that if an arrival event occurs before the scheduled departure event, the scheduled departure event has to be cancelled.

2.1 Enhancements with Indexing

In the original computational algorithm, the functions ADJUST_POOL and SCHEDULE_NEXT_TO_LEAVE are each required to traverse the pool. A simple but effective improvement is to merge both ADJUST_POOL and SCHEDULE_NEXT_TO_LEAVE so that only one traversal of the pool will suffice. This can be achieved by introducing an index field for each job in the pool, to indicate the service order. It is possible to achieve the same effect without the use of an index but a counter instead. The counter would keep track of the distance between the head and a particular job. The performance of both methods is the same, but the latter has the advantage that there is no need for an index field in each job record. We utilized the indexing technique because we will also require it in the batch departure scheme presented in Section 3.

This technique proceeds by doing a queue traversal and *minrem* finding simultaneously; the pseudo-code is shown in Figure 3. The number of steps from the last round (i.e., *rem*) is used along with an index (i.e., i) to adjust the states of jobs stationed between the old header and the new one. Clearly, jobs lying between index = 1 and index = *rem* require an extra tick subtracted. The

[1] These numbers may or may not be whole numbers. In a real system, a job may not actually require a whole quantum, but nevertheless consumes a whole quantum because of the granularity of operation. The algorithm handles this in the same way. If a job needs only the fractional part of a quantum, the system gives it the entire quantum. Because of granularity at the OS scheduling level, systems find it easier to schedule in small, but uniform quantum sizes. For convenience and clarity, we will use only whole numbers in this paper.

other use of the index is to determine the new value of *steps*, which must be equal to the index of the next job to (potentially) depart. Index updates are done during queue traversal with the help of a simple observation: the job immediately before the departing job must get an index = $pool\_size - 1$, and the job immediately after the departing job must get an index = 1. All other jobs are numbered accordingly, depending on their position. The same idea is used for arrivals. When an arrival occurs, if the pool is not empty, one job is being serviced, and the new job ends up with an index = $pool\_size + 1$, which is the new pool size. The job stationed immediately after this point of insertion becomes the new head, with index = 1. When two jobs have the same minimum remaining time, the job with a lower index departs first. The variable *minindex* is used to guarantee this condition.

Note that this algorithm only predicts one departure event every time it is invoked, which is when a new job arrives or when a job leaves the system. It is an improvement over the original computational algorithm in the sense that it reduces the number of pool traversals by one-half. This is practically faster, but not asymptotically faster[8], since theoretical run-time complexity is still $O(n^2)$. In the case where two or more jobs depart from the system in quick succession, repeatedly, this algorithm tends to becomes inefficient.

3 New Batch Departure Algorithm

As explained earlier, 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 4 illustrates the difference between the original algorithm and the new batch algorithm in terms of the number of events. We now proceed to derive a formula that enables the determination of a batch of n potential departures from a pool of $m \geq n$ jobs, based solely on the state of the pool. For simplicity, the formula will be derived assuming that no new jobs arrive between the invocation of the algorithm and the scheduled n departures. That is, departures can be identified because no new jobs arrive to change the state of the system and possibly change the identities of the departing jobs. We will, however, address the issue of new arrivals later.

The new formula is a generalization of the formula used in the single-departure algorithm, though not an easy one. One idea is to store the position and values of remaining service-time ticks in the pool while traversing the queue to determine the minimum. Then, Formula 1 — presented in Section 2 — can be applied multiple times. This approach, however, causes some complications which hinder its effective use:

- the position of the head changes,
- the size of the queue changes,
- some remaining-service time values are subjected to a decrement of one tick, while others undergo no subtraction, depending on whether they correspond to jobs that are stationed between the current head and the potential departure or not.
- the relative positions in the pool change, with respect to the head.

```
ADJUST_POOL_AND_SCHEDULE_NEXT_DEPARTURE(event_type)
    delta ← sim_clock − prev_clock
    ticks ← delta/q
    sub ← ticks/pool_size
    rem ← ticks mod pool_size
    first ← pool_size − rem        ◁ for reindexing
    minrem ← head.remain           ◁ for finding min remain
    minindex ← pool_size + 1
    steps ← 1
    job ← head
    for i ← 1 to pool_size
        do job.remain ← job.remain − sub      ◁ adjust remain
            if i ≤ rem      ◁ update indexes
                then job.remain ← job.remain − 1
                     job.index ← first + i
                else job.index ← i − rem
            if job.remain = 0
                then remove job from pool
                else if i = (rem + 1)      ◁ find new head
                       then newhead ← job
                     if job.remain < minrem or
                       (job.remain = minrem and job.index < minindex)
                       then minrem ← job.remain      ◁ find min remain
                            steps ← job.index
                            minindex ← steps
    if event_type = arrival
        then newjob's index ← pool_size + 1
            update minrem and steps if new job has a smaller remain
            insert new job to the pool
        else pool_size ← pool_size − 1      ◁ departure event
    next_departure ← current_time + ((minrem − 1) * pool_size + steps) * q
```

Figure 3: The Original Algorithm with Indexing

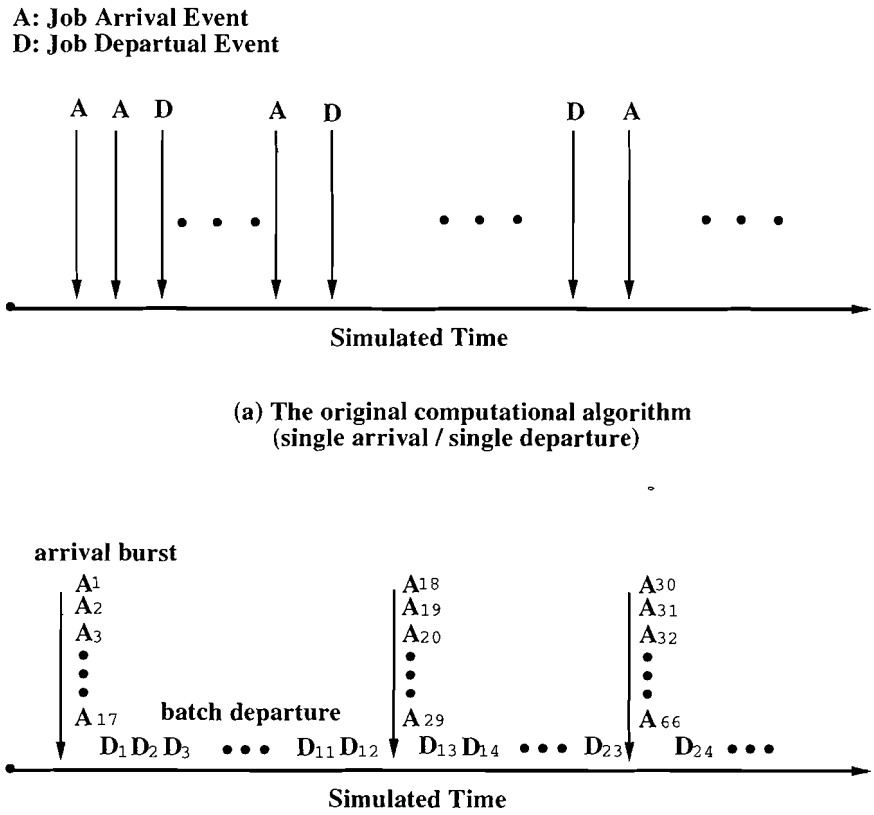


Figure 4: A Comparison of Single Arrival/Departure and Batch Arrival/Departure

There is no simple 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, preventing an easy generalization and application of the original formula. Further, because the pool size changes between updates, computational terms have to be added that require special handling during updates.

To illustrate the idea clearly, consider the state of the pool after C and D are removed from the original pool: this is shown in Figure 5. With each deletion, the head undergoes a change, ultimately moving from A to D to E. The relative positions of entries with respect to the head now all undergo change. That is, E is initially 5, but becomes 4 after C is removed, and 1 after D is removed, while both A and B maintain their positions 2 and 3, respectively. During the first round, 3 ticks are subtracted from each of A and B, while 2 ticks are subtracted from each of D and E. But, during the second round, 1 tick is subtracted from each. Thus, as the example shows, it can be quite a challenge to keep track of these differences efficiently as elements are deleted from a large pool, because of the different number of subtractions for different jobs during each round as their relative positions change.

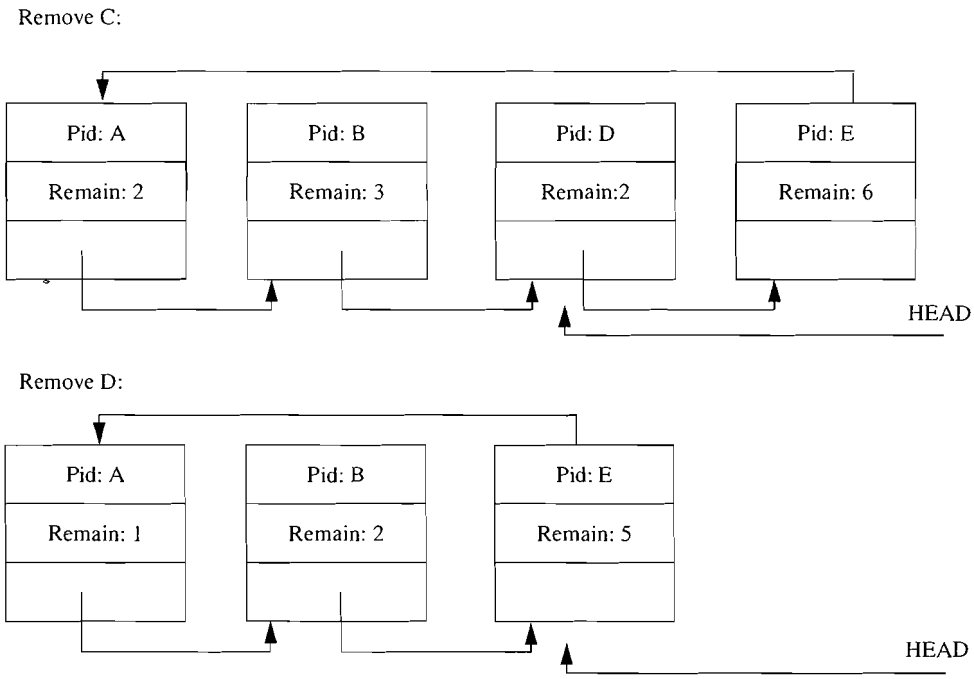


Figure 5: State of the pool after two departures

3.1 Derivation of the Generalized Formula

Having made the case that serious complications arise in a natural generalization of the formula from the old algorithm, primarily because of changes in head position, pool size, and relative positions, we propose a method that exploits the following:

- a fixed head,
- a fixed pool size,

- complete traversals of the pool from head to tail.

The basic idea is to use this somewhat artificially simplified pool, where the head and pool size are determined by the initial state, and used for all subsequent computations. With an approach that greatly simplifies computational updates, we only need to subtract the extra quanta to get an exact solution. As will be shown, the subtraction of quanta can be expressed mathematically, and can thus be easily accounted for. Further, the scheduling of all departures from the pool can be affected solely from initial state information. As was explained before, that the formula will be derived assuming that no new arrival occur between the scheduled n departures.

Using the same example as before, where each job record has an index, and indexing is used to indicate relative positions of jobs, we proceed to 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:

$$current\_time + [r_i \times pool\_size - (term1) - (term2)] \times q \tag{2}$$

where r_i = remaining time for job i, and (term 1) and (term 2) are explained in detail below. Observe that the formula reflects three constraints imposed on the pool:

$r_i \times$ pool_size	a complete traversal ($\rightarrow$ complete traversal condition),
(term 1)	extra steps (or ticks) due to counting elements that have already departed ($\rightarrow$ fixed queue condition),
(term 2)	extra steps due to traversal from a particular element to tail ($\rightarrow$ fixed head condition).

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:

$$\sum_{j=1}^{i-1}(r_i - r_j) = (i-1) \times r_i - \sum_{j=1}^{i-1} r_i. \tag{3}$$

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. The term $\sum_{j=1}^{i-1} r_i$ can be stored in a single variable. During each round, we would only need to add the last element, without needing to do the entire summation repeatedly.

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:

$$(pool\_size\_at\_round\_i) - (position\_of\_job\_i\_relative\_to\_head) \tag{4}$$

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

$$pool\_size\_at\_round\_i = (pool\_size - i + 1) \tag{5}$$

For relative positions, we take into account that:

- relative positions of jobs change as jobs are removed and scheduled for departure,
- relative positions of jobs stationed between the head and a departing job don't change, though the positions of others must change.

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:

$$relative\ position = p_i - \phi(p_i) \tag{6}$$

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 $\phi(p_i)$ is to compare the currently departing job's p_i with each of the relative positions of the jobs that just departed. Unfortunately, this kind of 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 $\phi(p_i)$.

Given a value p_i, the work required to find numbers smaller than p_i in a given set is almost the same as the work required to find the rank of p_i (i.e., its position) in the linear order of the set. They only differ in that the latter also counts the element itself. These two computational functions have the following relationship:

$$\phi(p_i) = rank(p_i) - 1 \tag{7}$$

A widely-used data structure, called augmented red-black tree or order-statistic tree[9], can support fast rank operations. A red-black tree has the following properties: 1) Every node is colored red or black. 2) The root is black. 3) A red node can only have black children. 4) Every path from the root to a leaf contains the same number of black nodes. These properties guarantee that a red-black tree with n nodes has a height of $O(\log n)$ and enables insertion or deletion in $O(\log n)$ time. In addition to the usual fields *key*, *color*, *parent*, $left$, and *right*, an augmented red-black tree has another field called *size* in each node. For a node x, the field $size(x)$ contains the number of nodes in the subtree rooted at x — a sum including the size of its left child, its right child, and 1 (for itself):

$$size(x) = size(left(x)) + size(right(x)) + 1 \tag{8}$$

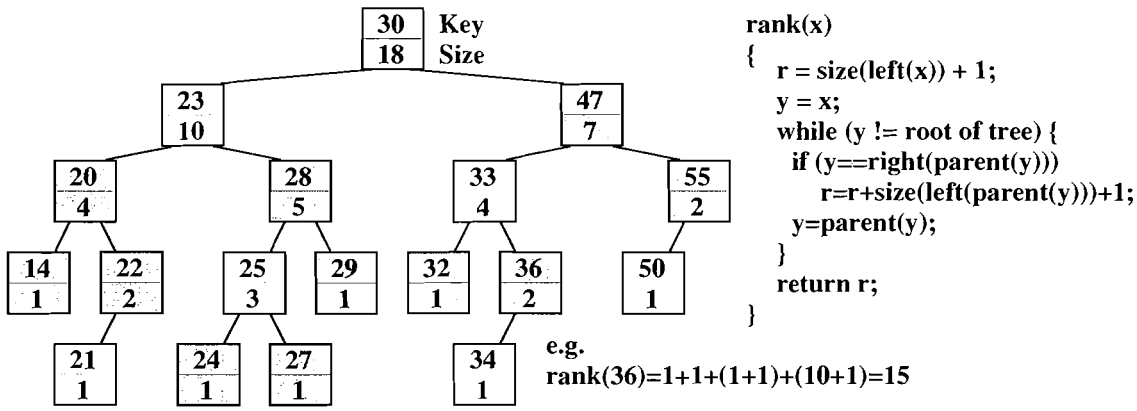


Figure 6: An augmented red-black tree and the rank operation

A detailed description of this data structure, relevant algorithms and time analysis can be found in [9]. Figure 6 shows an augmented red-black tree along with a function to determine the rank of an element. To find an element's rank, the algorithm traverses the path from the given element to the root, accumulating the size of any subtree appearing to the left of the path. For example, to get the rank of the element 36, the path to the root is $36 \rightarrow 33 \rightarrow 47 \rightarrow 30$ and there are three subtrees, rooted at nodes 34, 32, and 23, which appear to the left of the path. To get the final result, we accumulate the sizes of these three subtrees (i.e. 1, 1, 10) and then add 3 because the parents of these three subtrees (i.e. 36, 33, 30) are also smaller than 36. 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. Hence, computing $\phi(p_i)$ can also be done in $O(\log n)$.

Combining (5) and (6), we obtain a final form for term 2:

$$(pool\_size - i + 1) - p_i + \phi(p_i). \tag{9}$$

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

$$current\_time + [r_i \times pool\_size - ((i-1) \times r_i - \sum_{j=1}^{i-1} r_i) - ((pool\_size - i + 1) - p_i + \phi(p_i))] \times q$$

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 replacing single departures with batch departures, we encounter a new problem. A single departure requires one departure-time computation and at most one event cancellation, thus causing little overhead, if any. In the batch departure case, however, we may arrive at a situation (at an extreme, decidedly) 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 which

SCHEDULE_BATCH_DEPARTURE

```
job ← head
for i ← 1 to pool_size          ◁ fill table
    do T[i] ← job
       job ← job.next
quicksort T in ascending order by remaining service time
sum_remainder ← 0
for i ← 1 to pool_size          ◁ schedule departures
    do r ← T[i].remain
       p ← T[i].index           ◁ relative position
       term1 ← (i − 1) * r − sum_remainder
       sum_remainder ← sum_remainder + r
       insert p into an augmented red-black tree
       φ ← rank(p) − 1          ◁ jobs departed with smaller p
       term2 ← (pool_size − i + 1) − p + φ
       dep_time ← current_time + [r * pool_size − term1 − term2] * q;
       if (dep_time < look_ahead(next_arrival))      ◁ look ahead before scheduling
           then schedule a departure event for T[i].process at dep_time
```

Figure 7: The pseudo code of new batch departure algorithm

change the state of the pool and thus invalidate the already scheduled departures. 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. Since the simulation has access to processes or random number streams generating arrivals, this is easily achieved. It is worth mentioning that the look-ahead concept is not a new idea. It has been widely used in distributed simulation systems to avoid deadlocks[10].

In process-oriented simulation, for example, a simulation kernel would have to provide the application layer with a primitive that looks ahead and returns the time of the next arrival. The batch-departure computation algorithm checks the time of the next arrival and terminates the loop when the departure time is found to be greater than the arrival time. This makes for an efficient computation that determines only what is needed through constant monitoring via look-ahead. Because such a use of look-ahead 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 in increasing order of remaining service times r_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. Figure 7 depicts the algorithm in details.

Both the computational algorithm and the naïve 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 A Detailed Example

Consider the following illustration of the use of the batch-departure formula, based on the pool shown in Figure 2. The traversal is done from left to right to obtain a table T containing the remaining service times r_i, the relative positions p_i and the departure order i:

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

Sorting T by remaining service time r_i, we further obtain:

Job PID:	C	D	A	B	E
Remaining time r_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, $i = 1$:

  $r_i \times \text{pool\_size} = 3 \times 5 = 15$

  $\text{term1} = (i-1) \times r_i - \sum_{j=1}^{i-1} r_i = 0 - 0 = 0$

  $\text{term2} = (\text{pool\_size} - i + 1) - p_i + \phi(p_i) = (5 - 1 + 1) - 3 + 0 = 2$

  $\text{departure} = \text{current\_time} + [15 - 0 - 2]q = \text{current\_time} + 13\ q$

- **Departure 2**: Job D, $i = 2$:

  $r_i \times \text{pool\_size} = 4 \times 5 = 20$

  $\text{term1} = (i-1) \times r_i - \sum_{j=1}^{i-1} r_i = 1 \times 4 - 3 = 1$

  $\text{term2} = (\text{pool\_size} - i + 1) - p_i + \phi(p_i) = (5 - 2 + 1) - 4 + 1 = 1$

  $\text{departure} = \text{current\_time} + [20 - 1 - 1]q = \text{current\_time} + 18\ q$

- **Departure 3**: Job A, $i = 3$:

  $r_i \times \text{pool\_size} = 5 \times 5 = 25$

  $\text{term1} = (i-1) \times r_i - \sum_{j=1}^{i-1} r_i = 2 \times 5 - 7 = 3$

  $\text{term2} = (\text{pool\_size} - i + 1) - p_i + \phi(p_i) = (5 - 3 + 1) - 1 + 0 = 2$

  $\text{departure} = \text{current\_time} + [25 - 3 - 2]q = \text{current\_time} + 20\ q$

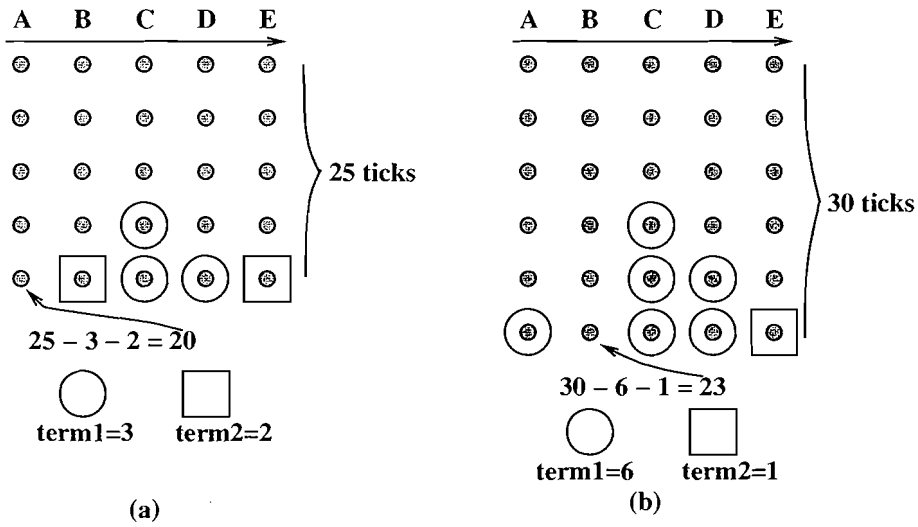


Figure 8: (a) Job A departs (b) Job B departs

- **Departure 4**: Job B, $i = 4$:

  $r_i \times \text{pool\_size} = 6 \times 5 = 30$

  $$\text{term1} = (i-1) \times r_i - \sum_{j=1}^{i-1} r_i = 3 \times 6 - 12 = 6$$

  $$\text{term2} = (\text{pool\_size} - i + 1) - p_i + \phi(p_i) = (5 - 4 + 1) - 2 + 1 = 1$$

  $$\text{departure} = \text{current\_time} + [30 - 6 - 1]q = \text{current\_time} + 23\ q$$

A graphical explanation of the computation is demonstrated in Figure 8. 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 8(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 8(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 8(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

When a batch of n jobs has been scheduled for departure, there are two ways of handling the change of pool state, i.e., of bringing the pool to a consistent state. This is true for $n = 1$ as well

```
ADJUST_POOL_BATCH_DEPARTURE
    discount ← the nth departure job's remain
    position ← the nth departure job's index
    job ← head
    for k ← 1 to pool_size
        do if job.index ≤ position
              then job.remain ← job.remain − discount
              else job.remain ← job.remain − (discount − 1)
        if job.index = position
            then newhead ← job.next
        tmp ← job
        if job.remain ≤ 0
            then delete job.process from pool
        job ← tmp.next
    pool_size ← pool_size − n
    head ← newhead
    job ← head
    for k ← 1 to pool_size
        do job.index ← k        ◁ new index
            job ← job.next
```

Figure 9: Update the state of the pool after n departures

as $n > 1$. The case of $n = 1$ is essentially the original single-departure computational algorithm. Each job leaves the pool as soon as simulation time coincides with its departure time, i.e., when the departure event is activated. Upon departing, the job updates the pool to the correct state at the simulation time of the event. In the case of $n > 1$, the state of the pool must be updated to reflect all n departures. 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. For example, only the last job to depart can affect the update, so that the others do not have to do any work.

For the batch departure case, we have to find a way of obtaining all necessary information in a single traversal of the pool, just as in the case of the single-departure algorithm. As explained earlier, different discounts have to be applied to different elements in the pool, depending on the position of the head with respect to these elements, as the traversal is done. The algorithm is detailed in Figure 9. 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. Note that there is no need to update the pool state when C first departs. 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*. Next, for each job in the pool, we subtract the quantity *discount* or *discount* $-$ 1 from its remaining time,

depending on whether its relative position lies before or after D. The jobs A, B, C, and D get the discount quantity 4, while job E gets the value 3. We can thus obtain updated values of remaining service-time when D leaves the system.

Job PID	A	B	C	D	E
Remaining time r_i	5	6	3	4	8
Relative position p_i	1	2	3	4	5
discount (–)	4	4	4	4	3
Updated remaining time r_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 $time = clock + 18q$. The new head will now point to job E.

Job PID	A	B	E
Remain r_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[11]. The input parameters used were:

- Quantum q.
- Exponentially distributed job interarrival times with mean $1/\lambda$.
- Exponentially distributed job departure times with mean $1/\mu$.
- 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:

naRR – the naïve round-robin algorithm.
orCA – the original single-departure computational algorithm.
inCA – the original computational algorithm, using indexing.
nuBD – the batch departure algorithm with one-departure at a time.
buBD – the batch departure algorithm 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. The use of averages does not represent the absolute performance of the algorithms but rather their relative performance given a particular parameter configuration.

In all our experiments we obtained a 95% confidence interval based on a Student-t, using n = 20. We routinely computed the standard errors in this process and found that with n = 20, the standard error was small. For example, with a mean of approximately 12 seconds, the variance was close to one. Variance stability was verified over several runs for all the algorithm versions. Two sample curves showing the 95% confidence intervals are shown in Figure 16. For clarity and to avoid clutter, all other figures don't include the confidence interval.

5.1 Benchmark 1: Single arrivals

These experiments were designed to evaluate the behavior of the different variations of the algorithms subjects to arrivals that occur one at a time.

Experiment 1 (Sensitivity to quantum)

The purpose of this experiment was to measure the performance of the algorithms as quantum size q is changed, since each algorithm uses different data structures to yield the same results. The parameters used were $1/\lambda = 50$ and $1/\mu = 40$, with $N = 20000$ jobs. All six variations of the algorithms indicated earlier were tested. The results are shown in Figure 10.

Experiment 2 (Sensitivity to traffic intensity ρ)

The purpose of this experiment was to measure the performance of the algorithms as the ratio $\rho = \lambda/\mu$ (traffic intensity) is varied. Here, μ and N were fixed at the values defined above, while λ was varied. As before, all variations of the algorithms were tested, except for naRR. The results are shown in Figure 11. The same data was used to generate Figure 12, where naRR is omited, to show a closer view of the other algorithms.

Interpretation of Results

The difference in performance between algorithms naRR and orCA was explained in [4]. In essence, the näive algorithm requires more time for small q because a larger number of events (and thus, context-switches) needs to be handled. Clearly, all the other computational algorithms offer much better behavior than naRR. As far as the variations of the computational algorithms and batch departure formula-based algorithms are concerned, we observe that all exhibit a near constant-time performance. This happens because the amount of computation required is independent of q. In particular, orCA and inCA exhibit almost the same performance behavior; they tend to offer better performance than nuBD and buBD, which again seem to be similar to one another in behavior.

In regard to traffic intensity effects, we observe that simulation runtime increases as ρ approaches 1; it's clear that more work is required to maintain the data structures when load increases. The same observations as before apply, i.e., orCA and inCA exhibit almost the same performance, and are better than nuBD and buBD. All five algorithms offer much better performance than naRR, and the performance difference decreases as ρ increases towards 1.

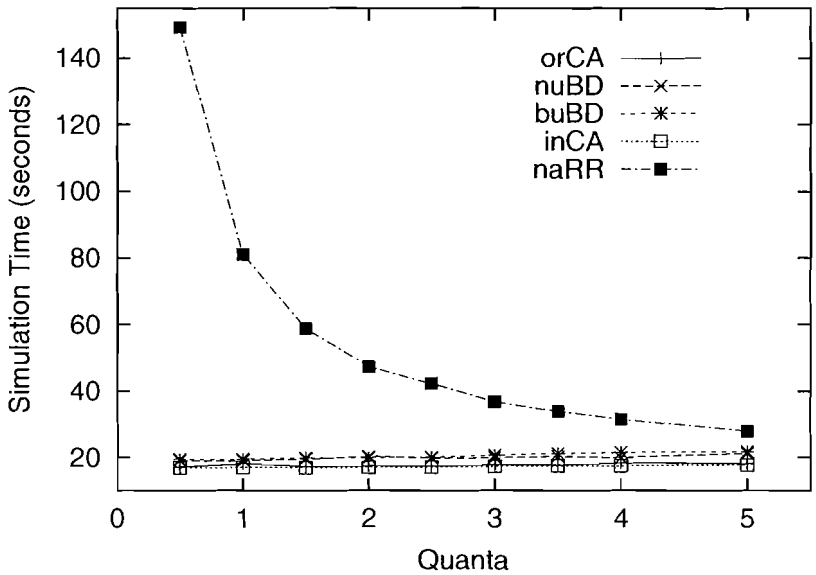


Figure 10: Simulation time vs. Quanta

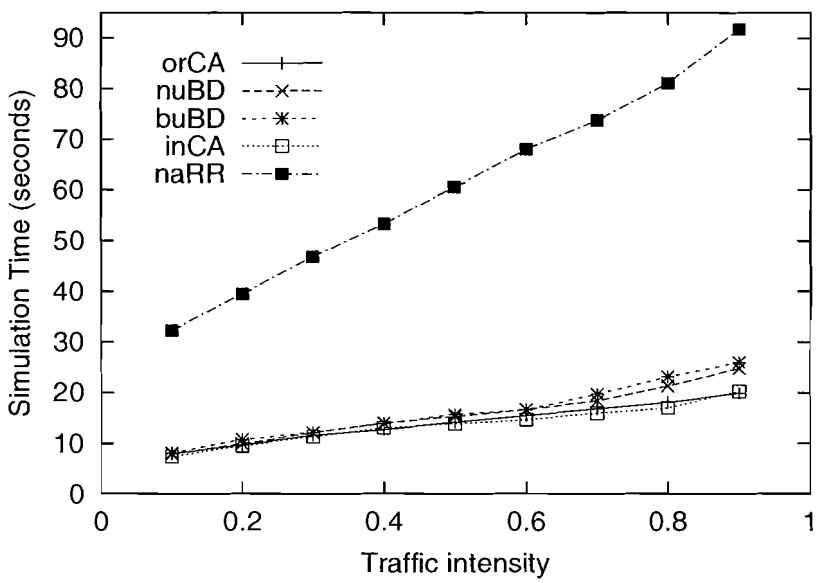


Figure 11: Simulation time vs. Traffic intensity (Single arrivals)

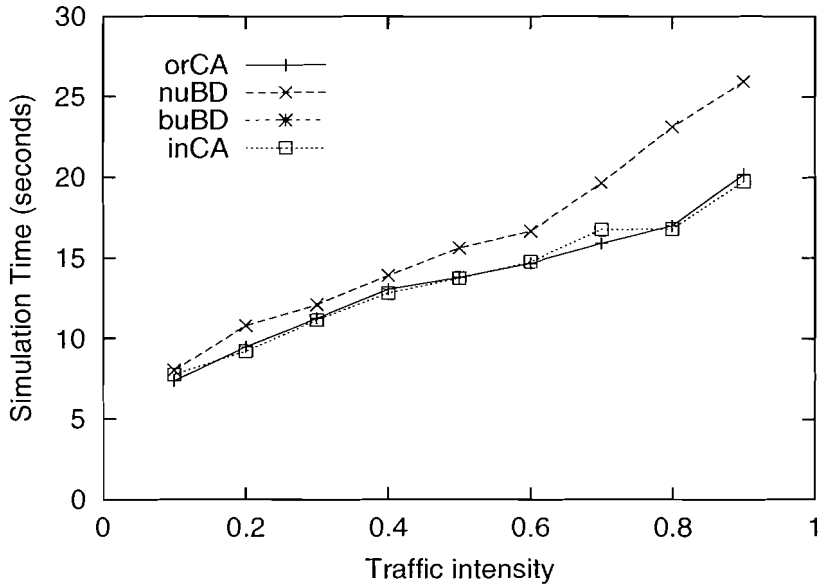


Figure 12: Simulation time vs. Traffic intensity (Single arrivals)

From our experiments we conclude that indexing and look-ahead do little to improve the performance of orCA in any significant way. In the case of indexing, a traversal of the pool is not required; in the case of look-ahead, cancellations can be avoided. It turns out, however, that the cost of the original implementation and the new implementation works out to be almost the same. Algorithm nuBD also appears to exhibit almost the same cost as algorithm buBD. This is because both algorithms affect the same number of computations, and the cost of handling departures one-at-a-time is roughly the same as the cost of processing departures in a batch.

5.2 Benchmark 2: Burst arrivals

These 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 stability in the system.

Experiment 3 (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 $1/\lambda$, $1/\mu = 200$ with $N = 10000$ jobs, while $1 + 1/\beta$ was varied. The experiment was repeated for different values of ρ, by varying $1/\lambda$, with results for $1/\lambda = 160, 320$ and 533 ($\rho = 0.8, 0.5$ and 0.3) shown in Figures 13, 14 and 15, respectively. The 95% confidence interval for the orCA and buBD curves, with $\rho = 0.5$ are shown in Figure 16.

Experiment 4 (Sensitivity to traffic)

The purpose of this experiment was to measure the performance of the algorithms with the batch size BA fixed, while ρ is varied. The systems evaluated include orCA, nuBD and buBD. The parameters used were $(1 + 1/\beta) = 30$, $1/\mu = 160$ with $N = 10000$ jobs, and $1/\lambda$ was varied. The results are shown in Figure 17.

Interpretation of Results

When comparing the original single-departure computational algorithm with the batch departure formula-based algorithm, it helps to consider the work each algorithm actually does. The single-departure computational algorithm: (1) traverses the pool to find the next-job to depart; (2) schedules that job for departure; (3) cancels the previously scheduled event, if necessary; (4) repeats the traversal for each distinct departure. In contrast, the batch departure formula-based algorithm: (1) traverses the pool to build a table; (2) sorts the table; (3) schedules as many consecutive departures as necessary before the next scheduled arrival; (4) repeats the process for the next batch of departures.

Seen in the context of the above explanation, no significant difference is observed between nuBD and buBD, so that both may be considered to be of equal performance, henceforth to be designated as BD. The following discussion regarding the batch departure formula refers to both these algorithms.

Since the algorithms were run on the same simulation testbed, observed differences in their performance can be attributed to the differences in the underlying algorithms. The orCA and BD implementations share most of their routines, with the exception of the algorithms for scheduling departures and updating the pool. Discarding all processing costs which can be assumed to be equal in both algorithms, it can be shown that the main cost incurred by the orCA algorithm is due to repeated traversals of the pool and cancellations (of many departures), while 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 the costs change with the pool size. Experimentally, the size of the pool is controlled by two variables — the traffic intensity and the size of arriving batches. The size of the pool grows with both large batch arrival size and 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 Experiments 1 and 2. In this region, the pool sizes are relatively small, and it is cheaper to traverse the pool many times instead of performing sorting operations. Traffic intensity affects the size of the pool, but the effect is not enough to change the relative performance of the algorithms. 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) $\rho = 0$ 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 3 and 4. 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 perform equally well.

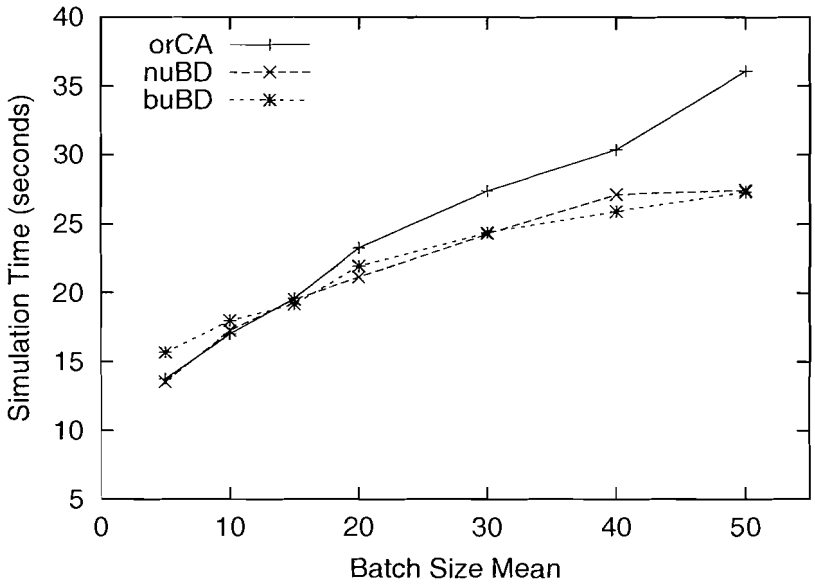


Figure 13: Simulation time vs. Batch Size ($\rho = 0.8$)

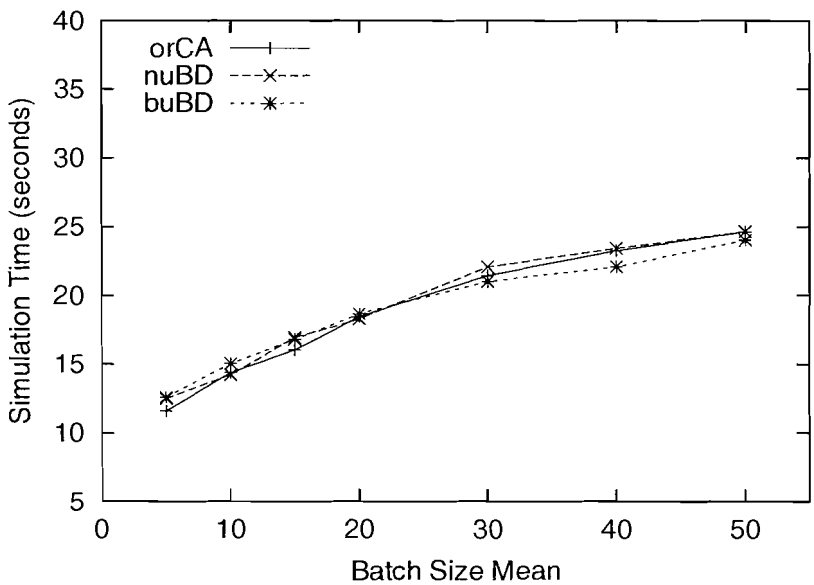


Figure 14: Simulation time vs. Batch Size ($\rho = 0.5$)

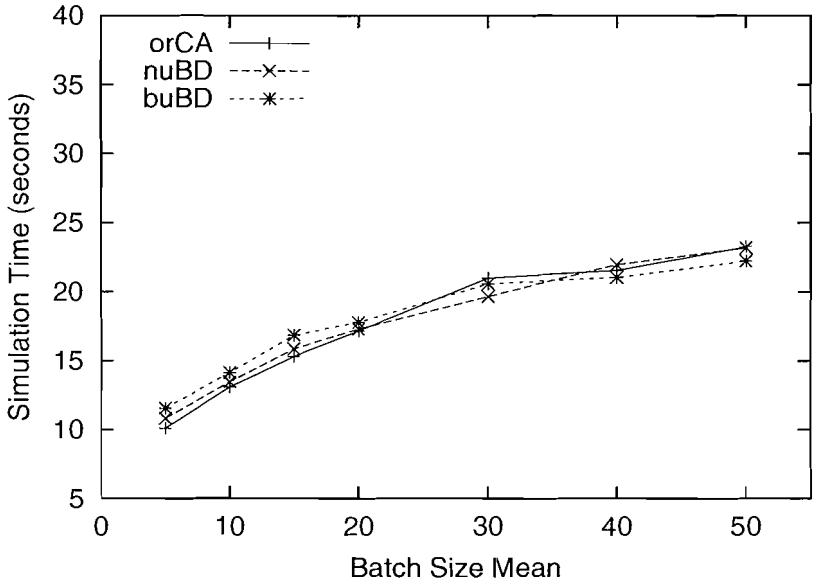
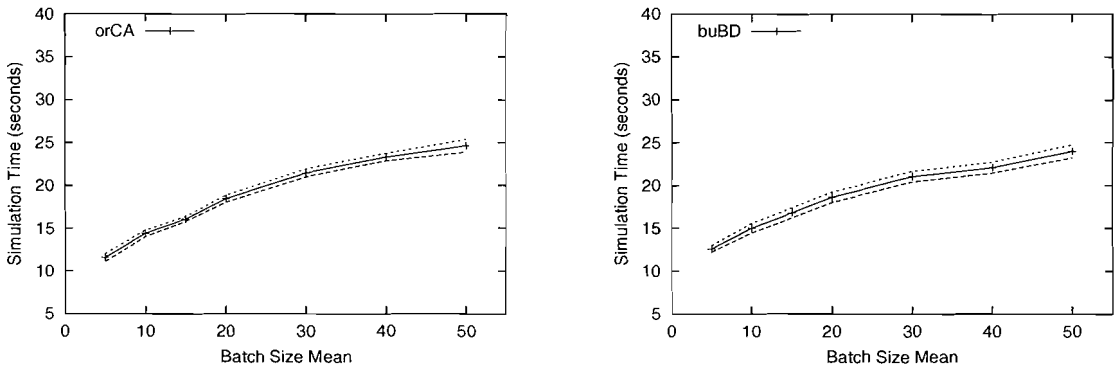


Figure 15: Simulation time vs. Batch Size ($\rho = 0.3$)



(a) (b)

Figure 16: 95% Confidence Intervals for $\rho = 0.5$

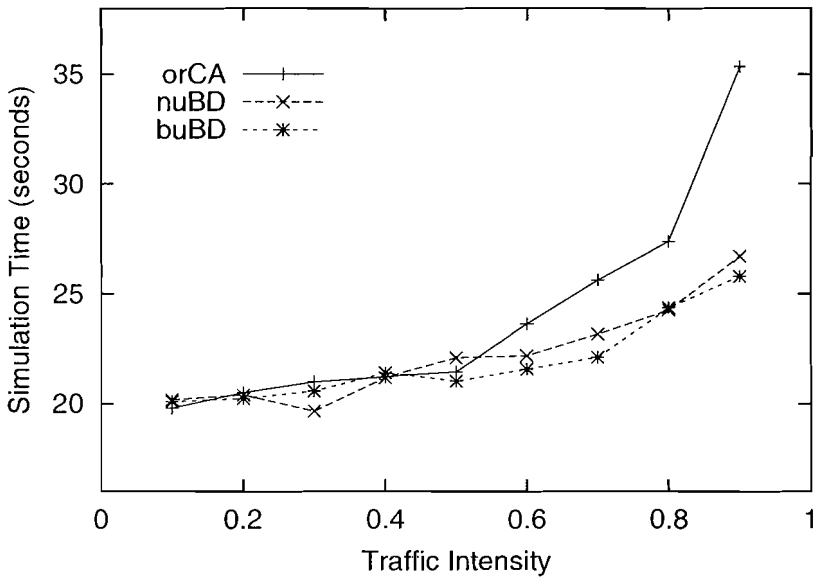


Figure 17: Simulation time vs. Traffic intensity (Burst arrivals)

- The third region involves medium to high traffic intensity (according to the figures, this corresponds to the experiments with $\rho = 0.5$ 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 the following table:

		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

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. In this region, another characteristic of orCA is apparent, i.e., more cancellations tend to occur when the traffic intensity is high, as more (potential) departures are scheduled, and these must be cancelled if an arrival occurs before the departures. Also, the difference between BA and orCA increases with increasing batch size, because of the effect of increasing pool size.

6 Conclusion

The underlying idea behind the new algorithms is that a reduction in the number of scheduled events will effect a corresponding reduction in simulation time. We built upon a previously proposed computational algorithm — based on a formula which predicts the next (potential) job departure — which schedules only one departure for each event and traversal of the pool. By generalizing this idea to batches, at the expense of some complexity, we conclude that it is possible to run efficient simulations that accommodate bursty traffic; multiple departures may be simultaneously scheduled during each traversal of the pool. Our experience with event reduction leads us to conclude that there may be a variety of scheduling algorithms where pre-computed schedules efficiently replace multiple scheduled events. Further, efficient algorithms effecting these schedules may be implemented within the domain layer (e.g., queueing domain versus particle-physics domain) in a portable way. The only requirement for this portability is the existence of primitives which allow access to pool-state and to the simulation calendar. 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. Based on empirical results, we conclude that the new burst arrival/batch departure algorithms perform better than the original single-departure computational algorithm when traffic intensity is high and batch sizes are large.

References

[1] H. D. Schwetman. Using CSIM to model complex systems. In *Proceedings of the Winter Simulation Conference*, pages 246–253, 1988.

[2] Roger McHaney. *Computer Simulation: A Practical Perspective.* Academic Press, San Diego, California, 1991.

[3] W. R. Franta. *The Process View of Simulation.* North-Holland, Amsterdam, 1977.

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

[5] M. H. MacDougall. Computer system simulation: An introduction. *Computing Surveys*, 2:191–210, Sep. 1970.

[6] M. H. MacDougall. *Simulating Computer Systems: Techniques and Tools.* The MIT Press, Cambridge, Massachusetts, 1987.

[7] H. D. Schwetman. Hybrid simulation models of computer systems. *Comm. ACM*, 21:718–723, Sep. 1978.

[8] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.

[9] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms, Chapter 14: Augmenting Data Structures.* McGraw Hill, Boston, Mass., second edition, 2001.

[10] J. Sang, E. Mascarenhas, and V. Rego. Mobile-Process Based Parallel Simulation. *Journal of Parallel and Distributed Computing*, February 1996.

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