
# EPCC-SS-2000-09

# InterSim: ns scheduler enhancement

**Dave Oldham**

## Abstract

Ns is a large and widely used discrete event simulator. Events are handled by a scheduler class contained with ns which ensures that events are executed in the correct order.
A proposed Lazy scheduler aims to minimize the computational effort used by the scheduler; by placing events in Buckets that have a time width less than the minimum delay time across the simulated network. The performance of this scheduler was investigated by studying the behavior of a calendar scheduler as the Bucket width became very small. It was found that whilst the schedulers performance does depend on the simulation being run the scheduler continues to run at peak efficiency for a greater than predicted number of Buckets. This is caused by uneven job balancing in the Buckets and suggests that the small Bucket width used by the Lazy scheduler will not inhibit its performance.
An investigation was also taken into the performance of a double-linked-list as opposed to the normal single-linked-list and was showed some improvement in performance.

# Content

# Figures



# Tables

# 1.0 Introduction

It is desirable to simulate network traffic before building a real network so that the network configuration can be designed to give maximum performance. One of the programs freely available that can do this is ns which was developed at Berkley University.

Ns is a large and complex program comprising of code written in both Tcl and C++. With the coming of age of the internet, the development of the grid and advances in networking technology serial simulation of network traffic is not sufficient to model the increasingly complex networks and there is a need for a parallel version of ns.

The most serious problem to overcome when getting ns to run in parallel is performing events in a time compatible way. By the very nature of parallel programs it is possible for processors working on a network simulation to be at different virtual times in the simulation. Hence a scheduler must insure that events are executed in a time compatible way.

In this report I have looked as two possible methods to improve the performance of ns scheduler as a prelude to a proposed parallel scheduler: the Lazy scheduler. Firstly, arranging the queue as a double-linked-list (as opposed to the normal single-linked-list) and secondly by arranging the queue into a series of Buckets.

This document is laid out with an opening section explaining some of the theory and **background** material necessary to understand the workings of ns scheduler. A **method** section outlining the schedulers that were written and their expected performance. A **results** section shows the performance of the schedulers and discrepancies between the predicted and measured performance are debated in the **discussion** section.

# 2.0 Background

A note on the text. Much of the terminology used in this report is summarized in the Glossary in Appendix A. When a term that is covered in the Glossary first appears it is highlighted in **bold**. Computer code and file names are included with `fixed-delimited font face`.
It is important to make the distinction between the 'queue' and the 'scheduler'. The queue is the area of memory in which jobs are stored whilst the scheduler is program used to control the queue.

There is only a subtle difference between the meaning of the words 'job' and 'event'; 'event' is used to describe the execution of a task by the network simulation. In contrast 'job' is used to describe 'events' whilst they are contained in the queue. E.g. the event performed by the simulation placed a job in the queue.

## 2.1 Networks

For the purpose of this report it is convenient to think of a network as comprising of a series of **nodes** and **links**. A **node** is processor or computer that can calculate the values of variables and perform tasks. **Links** are wires connecting nodes that can carry a signal from one node to another. This signal has to physically travel down a wire and so has a travel time or delay time associated with it. This **delay time** will depend on the length of the connection and in the event of a signal that has to travel through more than one link will be the sum of the delay times.

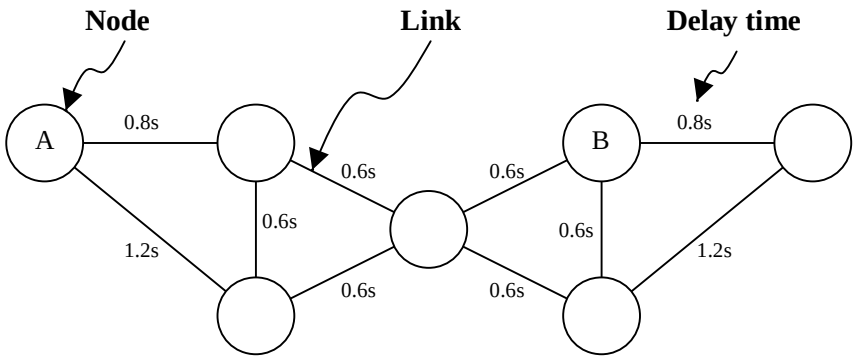



Fig 2.1: An example simple network.
Nodes are represented by circles and links by straight lines joining them, the delay time of a link is displayed next to the link. If we consider a signal sent from node A to node B we can see that it will have a delay of at least 2.0s.

## 2.2 Simulated Networks

To simulate network traffic a simulator must artificially reproduce the delay time taken for a message to pass a distance over a network. This is most commonly done by placing **events** (or **jobs**) in a **queue** with each job having an individual time stamp. The **time stamp** of a job is the **virtual time** at which it should be executed. Events are then executed in the order of their time stamps. The execution of an event may then create jobs that are placed in the queue with a time stamp that is in the future. See fig 2.2 for a detailed example. Please note in fig 2.2 we deal with an example of a scheduler that does not arrange jobs in the memory in any sort of order. This is done because it is easier to illustrate this type of scheduler in a diagram; we shall see the use of ordered schedulers in section 2.4.

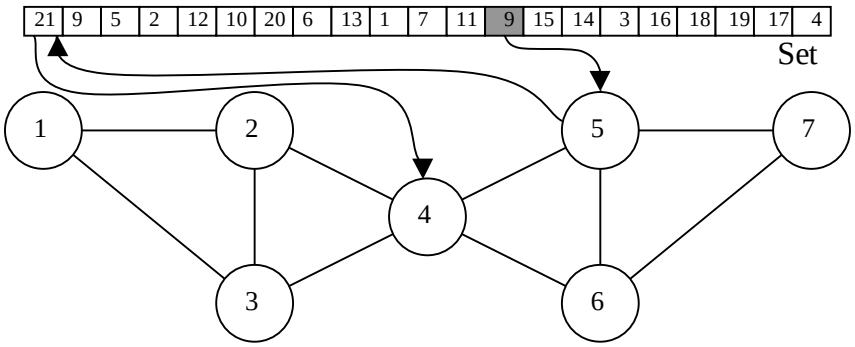



Fig 2.2: An example of an event in a network simulation.
The sequentially numbered array at the top of the diagram represents the set (since this is not ordered we do not call it a queue) while the numbered circles represent nodes and the lines joining them the links.
The job 9 has the lowest time stamp in the set and so is executed by the scheduler. This event tells node 5 to calculate a value k and pass the value of k onto node 4. The delay time between node 5 and node 4 is 1.2s (the time it would take for the signal to travel over the network). An event 21 is placed in the queue to take place on node 4 with a time stamp 1.2s after event 9. Event 21 then simply tells node 4 the value of k.

One of the main limits of this method is that each time an event is executed the program must first search through all the jobs in the queue and find the one with the lowest time stamp. This is computationally expensive and can easily be the most expensive calculation performed during an event. The aim of this project is to investigate methods to try and enhance the performance of the scheduler.

## 2.3 The ns Scheduler

ns is a heavily object orientated program and contains a class `Event` which contains all the information on individual events, including their time stamps. A scheduler is required to have the following member functions.

1. `Event* deque();`
   This should return a pointer to the event in the scheduler with the lowest time stamp.
2. `Void insert(Event*);`
   A pointer to an event is inputted and should be placed in the queue.
3. `Void cancel(Event*);`
   A pointer to an event is inputted that should then be removed from the queue.
4. `Event* lookup(int uid);`
   An integer is inputted. This is the unique I.D. of an event (stored in the Event class). The member function should then return a pointer to this event.

The percentage of the member functions called by schedulers are shown in fig 2.3.

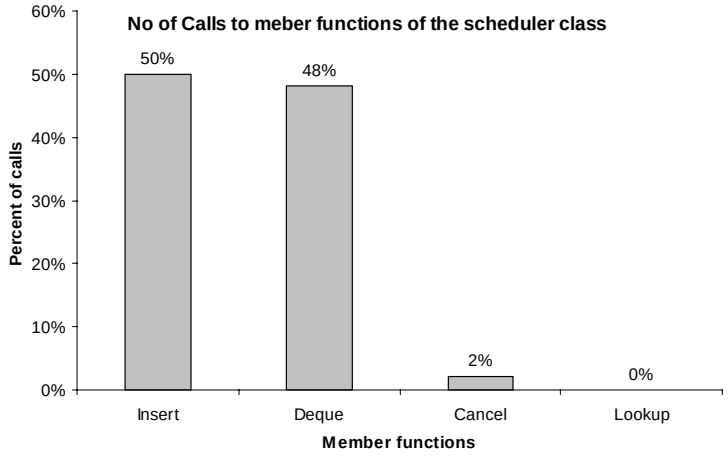



Fig 2.3: The percentage use of the member functions of scheduler e.g. 48% of the calls made to scheduler were to the member function Deque. This data was collected using a range of benchmarking programs writing by the epcc.

If we consider a simple scheduler where the events are placed in an unordered queue. Then we can see that to find the event in the queue with the lowest time stamp we shall have to search through all the events in the queue. See table 2.1.

| Scheduler function | Expected sort cost |
|---|---|
| Insert | O(1) |
| Cancel | O(1) |
| Deque | O(N) |
| Lookup | O(N/2) |

Table 2.1: The expected sort costs of the member functions of the scheduler class using a unordered scheduler.

## 2.4 The Linked-List Scheduler

A logical method to arrange the jobs in a queue is as a linked list. Here each job owns a pointer that points to the next job in the list with the lowest time stamp and a static pointer pointing to the first event in the queue. When a job is added to the queue the list must be searched through so that the correct point in the queue can be found to place the job. The pointer of the event before the job must then be changed to point to the new job and pointer of the new job must point to the next event. A similar procedure is needed for canceling: the queue must be searched to find the event before the canceled job and its pointer changed. To dequeue and perform an event the static pointer is used (there is no need to search through the queue) and then must be change to point to the next job. Searching through the jobs in the queue is an expensive task and by placing all the events in one queue in this way we run the scheduler in an inefficient way.

| Scheduler function | Expected sort cost |
|---|---|
| Insert | O(N/2) |
| Cancel | O(N/2) |
| Deque | O(1) |
| Lookup | O(N/2) |

Table 2.2: The expected sort costs of the member functions of the scheduler class using a linked list scheduler.

## 2.5 The ns Calendar Scheduler

The calendar scheduler reduces the computational effort needed to find a job in the queue by splitting the queue up into a series of bins called **Buckets**. Each Bucket contains jobs with time stamps in a specific time range. This has the advantage that each time the scheduler needs to either insert or cancel a job (for these member functions we know the time stamp of the event) then we need only search through the jobs in one Buckets and not all the jobs in the queue. See fig 2.4.

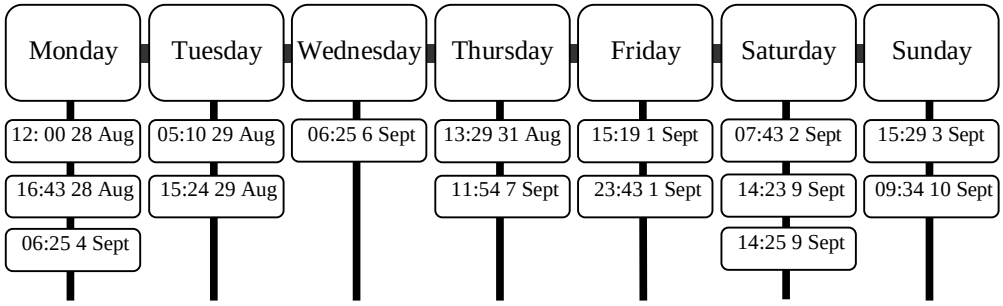



Fig 2.4: The structure of the calendar scheduler shown for the simple example of 'days of the week'. Here the **time span** of the calendar queue is 7 Days and the **Bucket width** is 1 Day. In the ns Calendar queue each Buckets contain jobs over a series of time spans. The time span, Bucket width and the number of Buckets are altered during the simulation to keep the scheduler working at optimum efficiency.

The **time span** of the calendar (commonly called the year) and the number of buckets active change during the simulation to get the highest performance out of the scheduler. There is no point have 100 Buckets if there are only 10 active jobs in the queue. Best performance is found to be when there are around 10 events in each Bucket (Brown 1988).

| Scheduler function | Expected sort cost |
|---|---|
| Insert | O(N/2{no of Buckets}) |
| Cancel | O(N/2{no of Buckets}) |
| Deque | O(1) |
| Lookup | O(N/2) |

Table 2.3: The expected sort cost of the member functions of the calendar scheduler class using a linked list scheduler.

## 2.6 Paralleling Network Simulation

Splitting up the tasks performed during a network simulation is not an easy task – mainly because of the problems of sorting events between processors and ensuring that processors remain at approximately the same virtual time.

One obvious architecture that can be used to parallise the network simulation is to divide the nodes up equally amongst the processors and use shared memory to store the queue. See fig 2.5.

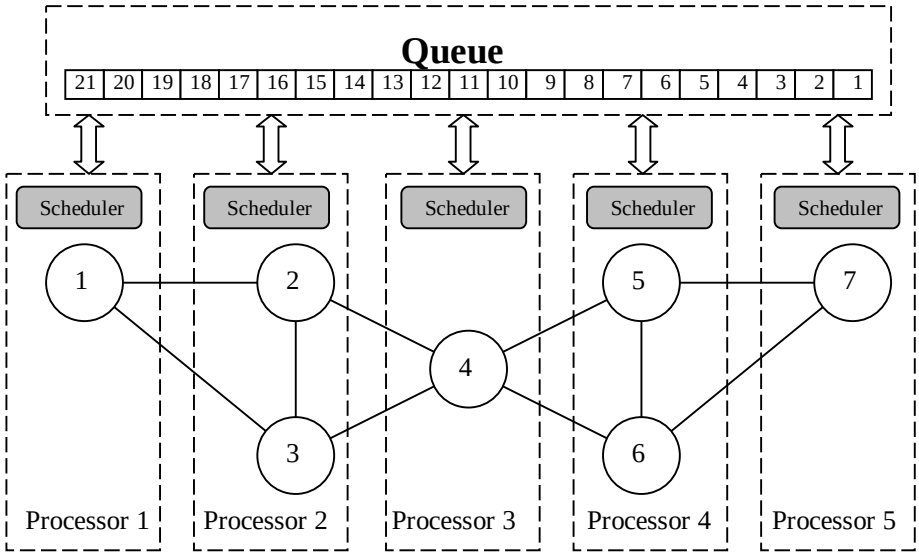



Fig 2.5: The simplest architecture to parallise network simulation. Nodes are divided up between processors and a single queue is used in shared memory. This could cause bottlenecking when large number of processors are used.

This method would give disappointing results when a large number of processors were used because of much for the processors time would be spent idle whilst they waited for the queue to become unlocked: a bottle-neck. Dividing the queue up between the processors can solve this problem. However this leads to the time compatibility problems discussed in section 2.7.

## 2.7 The time compatibility problem

One of the more serious problems encountered where trying to run a network simulation in parallel is in keeping the processors at the same virtual times. If one processor is allowed to move to a virtual time that is greater than the virtual time on other processors then it is possible for jobs to be scheduled in a **time incompatible** way. A time incompatible job is scheduled to take place with a time stamp that is less than virtual time of the processor i.e. it is scheduled to take place in the past. Time incompatible events will produce incorrect output since processors may have been using the wrong values of variables that would have been changed by the earlier 'missing' job.
There are two existing methods for dealing with the time incompatibility problem:

### 2.7.1 The Optimistic Solution

By assuming that time incompatible events are rare (this is why this is called the optimistic solution) we chose to wait until a time incompatible event occurs and then fix the problem by halting the simulation and undoing and redoing the jobs that have been affected.

The disadvantage of the optimistic approach is that when a time incompatible event does occur the computational cost is significant. If these events occur often then the speed of the simulation will be reduced.

### 2.7.2 The Conservative Solution

By deciding that time incompatible events occur often we halt the schedulers periodically until all the processors have reached the same virtual time. This periodic synchronizing of the processors should happen with a period less than the minimum delay time across the network. This ensures that a time incompatible event can not take place.

## 2.8 The Lazy Scheduler

We can make the following observation about time incompatible events…

We can be certain that two events are time compatible if: -

- The difference of their time stamps is less than the minimum time delay between any two node in the network.
- They are not at the same Node.

The pessimistic solution to the time incompatibility problem exploits this by ensuing that the difference of the virtual time of the processors does not exceed the minimum time delay over the network.

The proposed **lazy scheduler** takes advantage of this by dividing up the queue into a series of Buckets (like those found in the calendar scheduler) and sets the Buckets width to be just less than the minimum delay time over the simulated network. The events within the Buckets can then be dispatched in any order - as long as they are not on the same node. The queue is then divided over the nodes. So that each nodes has its own queue of Buckets. Each processor then takes a job that has to be executed on each node in turn. As long as all the jobs in one group of Buckets (Buckets with the same time range) are executed before the scheduler moves onto the next group of Buckets then the events will always be executed in a time compatible manner.

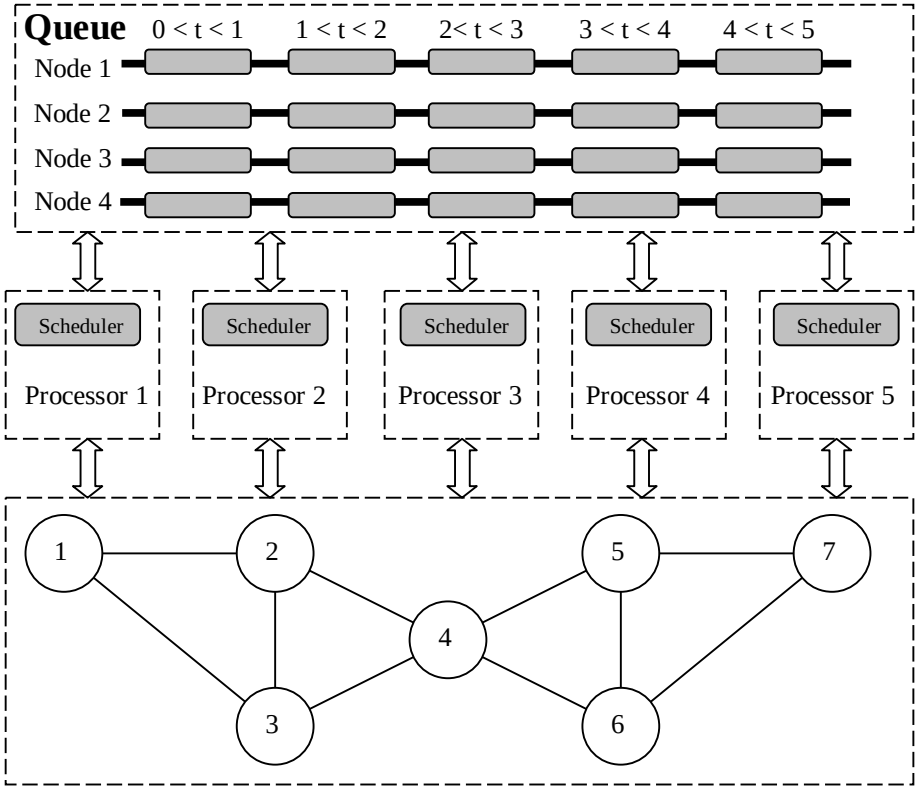



Fig 2.6: The Lazy scheduler. The queue is divided up into a series of Buckets, the width of which is just less than minimum delay time across the network (this is 1s in the example shown above). The queue is also then divided up in a series of queues one for each node and is stored in shared memory.

# 3.0 Method

The version of ns worked on during this report was ns-2.1b5. Changes had been made to this by both Kathy Nicholas and Martin Westhead.

## 3.1 The Linked-List scheduler

A scheduler was written for ns that arranged events in the queue as a linked-list. The pointer next_ in the event class was used to point to the event with the lowest time stamp. This was to all-intense and purposes identical to the ns linked list scheduler.

## 3.2 The Double Linked List

One of the limitations of the linked List scheduler is the searching through the queue that needs to be performed before a job can be canceled. The need to perform this search can be eliminated if a double linked list is used. This comes at the expense of the setup and maintenance of the second pointer in the `event` class

A second pointer was included in the `event` class called `last_` to point to the last job in the queue. Alterations were then made to the `insert`, `deque` and `cancel` member functions to produce a new scheduler. These included a series of if statements to test to see if the jobs were at the top or bottom of the queue (in which case they had to be treated differently).

The Double-Linked-List code can be seen in Appendix C

The expected sort cost of the member functions can be seen in table 3.1.

| Scheduler function | Expected sort cost |
|---|---|
| Insert | O(N/2) |
| Cancel | O(1) |
| Deque | O(1) |
| Lookup | O(N/2) |

Table 3.1: The expected sort cost of the member functions of the scheduler class using a double linked list scheduler.

## 3.3 Dave's Calendar Scheduler

A scheduler was written and inserted into `scheduler.cc` to perform a simple calendar operation. Buckets were divided equally amongst the time span of the simulation. See fig 3.1.

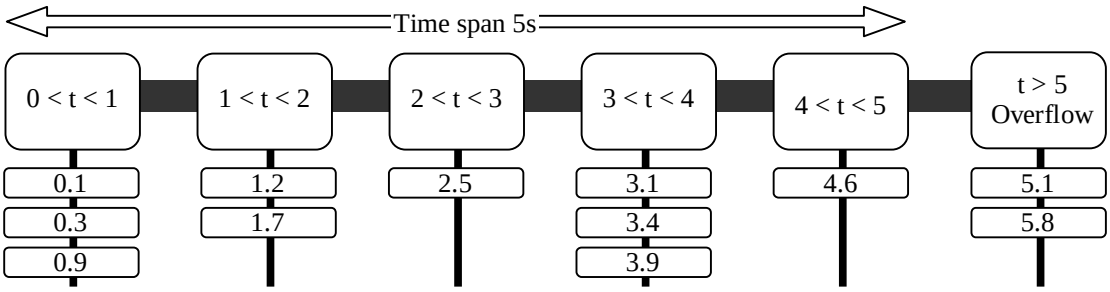



Fig 3.1: The structure of Dave's calendar scheduler shown with a simulation lasting 5 seconds and using 5 Buckets. The queue is split up into a series of Buckets each containing a linked list of events that cover a particular time span. This has the advantage that each time an event is inserted or cancelled only one Bucket needs to be searched.

The advantage of this method is that whenever a job is inserted or canceled from the queue only one Buckets needs to be sorted through. We recall that for the linked-list scheduler all the jobs in the queue needed searching and so we can see an instant improvement.

Since the events are spread over 5 Buckets in our example it follows that each time an event is placed in the queue or canceled one fifth of the events will need sorting compared with an ordinary linked list scheduler. The expected numbers of jobs that will need sorting by each member function are shown in table 3.2.

| Scheduler function | Expected sort cost |
|---|---|
| Insert | O(N/2{number of Buckets}) |
| Cancel | O(N/2{number of Buckets}) |
| Deque | O(1) |
| Lookup | O(N/2) |

Table 3.2: The expected sort cost of the member functions of the scheduler class using Dave's Calendar Scheduler.

The Dave's calendar scheduler code can be seen in Appendix C.

## 3.4 Counting the number of active jobs in each Bucket

One of the ways of understanding the behavior of a scheduler such as Dave's Calendar Scheduler is to look at the number of active jobs in the queue during the simulation and the number of jobs stored in the individual Buckets.

The code written to do this is included in the source code shown in section 3.3. This feature when activated (by uncommenting the necessary lines in the code) greatly inhibits the performance of the processor and was not uses during the performance testing simulations.

## 3.5 Making the schedulers visible from Tcl

ns is written in both C++ and tcl. Any classes in C++ need to be made visible to the tcl. This is done with the following code.

**Scheduler.cc**

```
Static class CalendarClass : public TclClass {
public:
        CalendarClass() : TclClass("Scheduler/DaveCal") {}
        TclObject* create(int /* argc */, const char*const* /* argv */) {
                return (new Calendar);
        } class_cal_sched;
```

## 3.6 Timing the schedulers

Testing the performance of a scheduler is not as simple as it sounds. There are a number of factors that will affect the speed at which ns runs. Firstly there is a random nature to the size and number of events run during a benchmark program and so if many large events are executed the simulation time will be greater than when the simulation contains fewer smaller events.

We can remove the effect of the number of jobs executed by dividing the simulation time by the total number of jobs dequeued during the simulation (which can be easily measured).
The affects of the variance in the size of the jobs can be assumed to cancel out for large numbers of events.

Code was inserted into `scheduler.cc` to time the simulation. This outputs the simulation time in nano seconds.

**Scheduler.cc**

```
void AtHandler::handle(Event* e)
{
        float exec_time;
        AtEvent* at = (AtEvent*)e;
        if (strcmp(at->proc_, "finish") == 0) {
                Scheduler::instance().exec_stop=gethrtime();
                exec_time=(float)((Scheduler::instance().exec_stop-
Scheduler::instance().exec_start));
                exec_time=exec_time/1000000000;
                printf ("Simulation time: %.3f\nEvents: %d\n", exec_time,
Scheduler::instance().event_count);
        }
        Tcl::instance().eval(at->proc_);
        delete[] at->proc_;
        delete at;
}
```

# 4.0 Results

## 4.1 The Double Linked List

The performance of the double linked list scheduler were compared with the ns linked list scheduler on the benchmarking program `ring_n20.tcl` run for 40s and with nine simulations run for both schedulers. The results are summarized in table 4.1.

| | Dave's Double Linked List | Ns Linked List |
|---|---|---|
| Average simulation time per event ($10^{-15}$s) | 12.91 | 13.14 |
| Standard deviation ($10^{-15}$s) | 0.19 | 0.38 |

Table 4.1: The performance of the linked list and the double linked list schedulers taken for 9 simulations run on ring-n20.tcl for 40s. The results do seem to show some improvement when using a double linked list. Full results are included in Appendix B.

## 4.2 Dave's calendar scheduler

The variation in performance of Dave's calendar scheduler with the number of Buckets used can be predicted by looking at the expected number of sorts needed per call to scheduler. By combining the data in fig 2.3 and table 3.2 we can see that the expected number of sorts needed per call to scheduler is given by…

$$\left\{\begin{matrix}\text{Expected sort}\\ \text{cost of the}\\ \text{scheduler}\end{matrix}\right\} = O\left((50\%)\frac{N}{2\{\text{Number of Buckets}\}} + (2\%)\frac{N}{2\{\text{Number of Buckets}\}}\right)$$

$$\left\{\begin{matrix}\text{Expected sort}\\ \text{cost of the}\\ \text{scheduler}\end{matrix}\right\} = O\left(\frac{0.52}{2}\frac{N}{\{\text{Number of Buckets}\}}\right) = O\left(0.26\frac{N}{\{\text{Number of Buckets}\}}\right)$$

This was then used to produce an expected speed up graph, which could in turn be compared with the measured speed up.

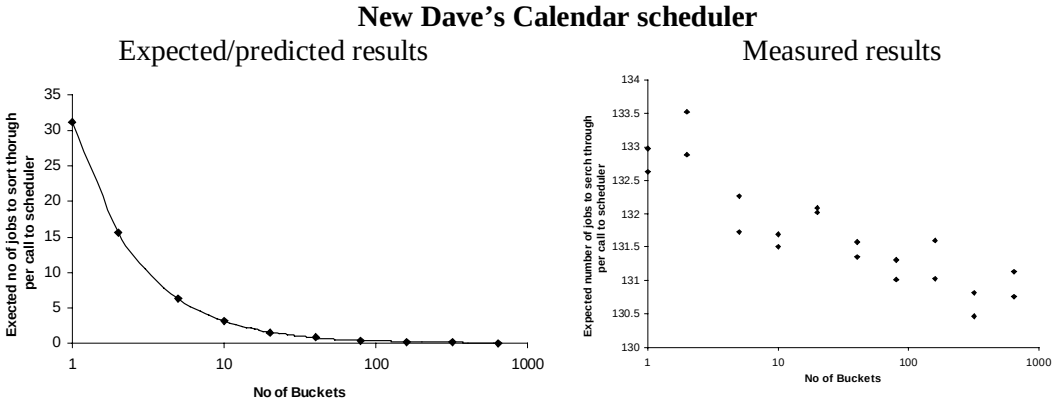



Fig 4.1: The expected and measured speed up curves for Dave's calendar scheduler run for 120s on ring-n20.tcl. Only low numbers of Buckets are shown here (up to 640). Notice that although some speed up does occur this is not as significant as predicted. This is caused by the uneven load balancing of jobs over the Buckets. See section 5.2.

## 4.3 Overall performance of the Schedulers

All the available schedulers were timed for 5 runs on `ring-n20.tcl` for 120s on Lomond and the results are shown in table 4.2 and fig 4.2 below. The complete results can be found in Appendix B .

| Scheduler | Average time per event ($10^{-15}$s per event) | Standard Deviation time per event ($10^{-15}$s per event) |
|---|---|---|
| Dave's Calendar Scheduler with 14422 Buckets | 12.67932553 | 0.04117853 |
| Dave's Double Linked List | 12.79151375 | 0.05339195 |
| ns linked List | 12.827549 | 0.04361929 |
| ns Calendar | 12.83103277 | 0.02824402 |
| Dave's Linked List | 12.89131725 | 0.12706934 |

Table 4.2: The performance of ns and Dave's schedulers run on ring-n20.tcl for 120s on Lomond.

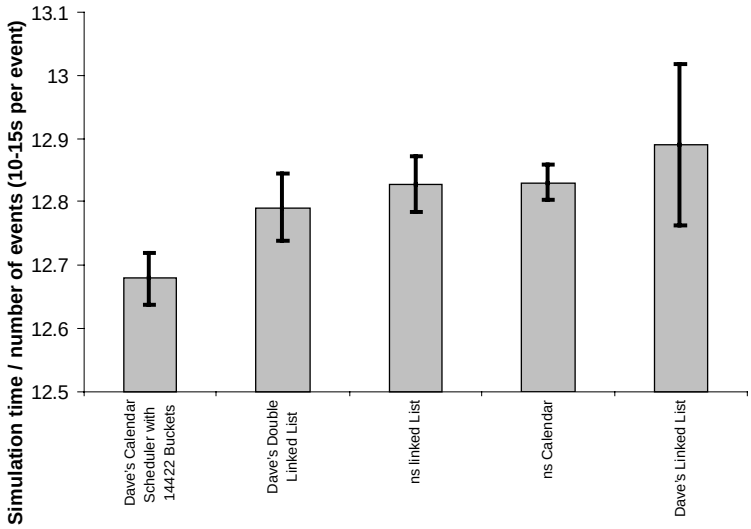



Fig 4.2: The performance of ns and Dave's schedulers run on ring-n20.tcl for 120s on Lomond. Five timings were made for each scheduler and the average is shown as the gray bar. The standard deviations of the five results are also shown as the black error bars. See section 5.3 for a possible explanation of why Dave's Calendar scheduler beats ns calendar scheduler.

# 5.0 Discussion

## 5.1 The Double Linked List

Table 4.1 and 4.2 shows the performance of the double linked list compared with the regular ns linked list. This seems to show some improvement in performance. This did rather surprise the author who expected that since only 2% of the calls to scheduler were to `cancel` compared with 50% to `insert` which has the same expect sort cost as `cancel`. That the set up cost of the double linked list would outweigh the improvements made to `cancel`.

The next step would be to use the Double linked list within a calendar scheduler and see if performance is improved. As we shall see in section 5.2 however, that the performance of the calendar scheduler it at its greatest when there are almost as many Buckets as events dequed (i.e. there are only around four jobs in each Bucket). In which case it is doubtful a double linked list would aid the scheduler's performance significantly.

## 5.2 Dave's Calendar Scheduler

The predicted and measured speed up of Dave's Calendar scheduler shown in fig 4.1 shows a lower than predicted speed up for small numbers of Buckets. It was decided to investigate this by looking at the number of jobs in the queue at any one time and how these jobs are distributed over the Buckets. The method use to do this is discussed in section 3.4.
An example of the results is shown below in fig 5.1.

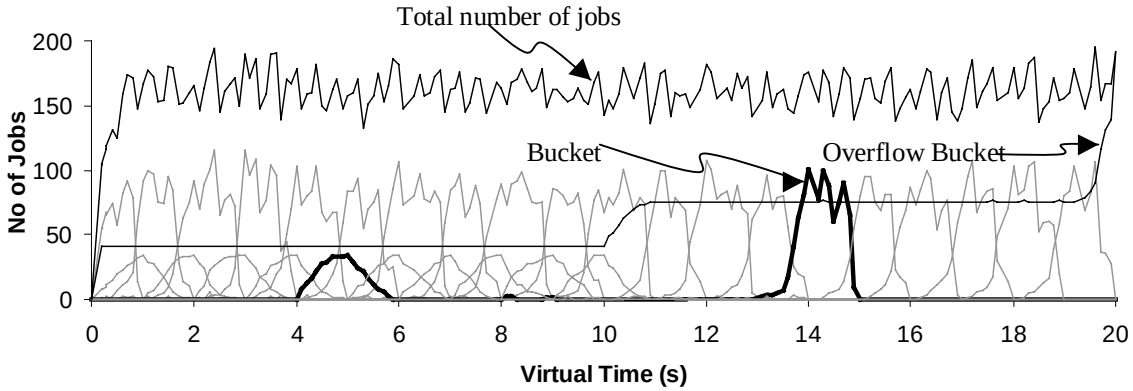



Fig 5.1: The number of jobs in each Bucket for a simulation run on ring-n20.tcl for 20s using 20 Buckets. An example Bucket (number 14) is shown in bold and the other Bucket in light grey. The total number of jobs and the number in the overflow Buckets are also shown.
We see that there is an uneven balancing of the jobs over the Buckets, with almost all the jobs in the queue at any one time being contained in the same Bucket.

The poor speed up is explained: the jobs are not being spread evenly over the Buckets and so the amount of jobs sorted through by the member functions of the scheduler is not being reduced by the amounts predicted.

However, Buckets are cheap. Each Bucket is just a pointer to the first event in a linked list, so the behavior of Dave's Calendar queue as the number of Buckets is increased was investigated. These results are shown in fig 5.2.

For a sufficiently high number of Buckets the computational cost of sorting through the Buckets will become so great that it will out weigh the benefits of putting jobs in Buckets and will inhibit the performance of Dave's Calendar Scheduler. There will be an optimum number of Buckets at which the performance of the scheduler is optimized.

By considering the total number of jobs or Buckets sorted by each member function of Dave's Calendar scheduler we can predict when this will happen.

Consider a simulation where there are T calls to scheduler, on average N jobs active in the queue and B Buckets are used.

The total number of jobs/Buckets sorted by each member function (see fig 2.3 and 3.2)is given by…

| Insert | Cancel |
|---|---|
| $(50\%)\mathrm{T}\frac{\mathrm{N}}{2\mathrm{B}}$ | $(2\%)\mathrm{T}\frac{\mathrm{N}}{2\mathrm{B}}$ |
| **Deque** | **Lookup** |
| B | 0 |

So expected performance of the simulation is …

$$O\left(0.26\frac{\mathrm{NT}}{\mathrm{B}}+\mathrm{B}\right)$$

We can then differentiate this and solve for B to find the optimum number of Buckets.

$$\left\{\begin{matrix}\text{Optimum}\\ \text{number of}\\ \text{Buckets}\end{matrix}\right\}=\sqrt{0.26\,\mathrm{NT}}$$

During a 120s simulation using `ring-n20.tcl` it is known that around 2,500,000 jobs are dequed (see Appendix B). Since 48% of the calls to scheduler are to deque then it follows that there are around 5,000,000 calls to scheduler, we also know that there are on average 160 jobs active in the queue (see fig 5.1). Putting these figures into our equation…

$$\left\{\begin{matrix}\text{Optimum}\\ \text{number of}\\ \text{Buckets}\end{matrix}\right\}=\sqrt{0.26\times 5{,}000{,}000\times 160}=14{,}422$$

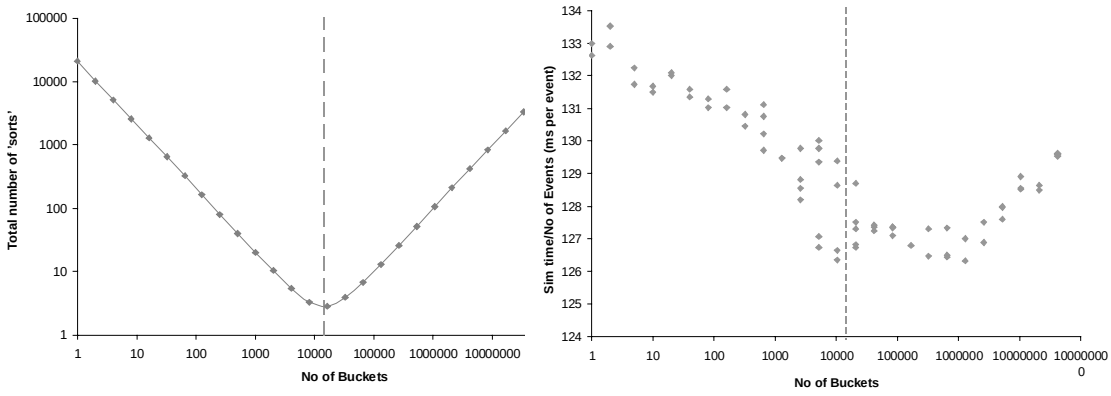



Fig 5.2: The predicted and measured performance of the Dave's Calendar scheduler with Buckets size. We see that performance improves up to 10240 Buckets where it stays reasonably stable until 1310720 Buckets after which performance starts to decrease. The gray dotted line shows the predicted optimum Bucket number 14,422.

The Scheduler is seen to keep it optimum performance up to 1,310,720 Buckets. For the simulation from which these results were obtained (120s on `ring-n20.tcl`) the total number of calls to scheduler was 5,000,000. This would suggest that the scheduler has its best performance when the total number of jobs inserted is a quarter of the number of Buckets (i.e. there are four jobs per Bucket).

Some work was done it investigate why Dave's Calendar scheduler kept working at a high efficiency up to a number of Buckets much higher than predicted (three orders of magnitude higher). This is almost certainly cause by the assumption made in calculating the expected sort cost that the jobs would be spread evenly over the Buckets. The number of jobs in the Buckets was measured for a simulation using 1,310,720 Buckets and gave surprising results. Many Buckets simply had no jobs in them at all whilst other had up to 60 jobs. However measurements were only made for a few sample Buckets and it can not be sure that this represents the general trend.

This is an important timing exercise since the proposed Lazy Scheduler would use Buckets with a very small time width and hence a large amount of Buckets and so it is important to see what the behavior of a calendar scheduler would be when extreme numbers of Buckets are used.

## 5.3 Overall performance of the Schedulers

The results shown in section 4.3 show that both the Double-Linked-List and Dave's Calendar Schedulers out perform all the ns schedulers tested. The poor performance of ns Calendar scheduler is probably caused by the relatively short time span over which the simulation was run (virtual time 120s). So the adjustment that had to be made to the ns calendar scheduler during the opening stages of the simulation when the number of jobs in the queue was rapidly increasing caused its poor performance. If longer simulations were run these effects would be less significant and the scheduler would perform better.

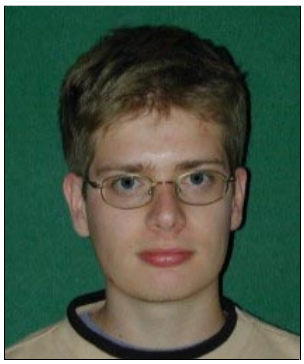
# 6.0 Conclusion

- The Double Linked List Scheduler has been shown to out perform the Linked List scheduler. However if a Calendar scheduler were written that used a Double Linked List is expected that it would only outperform the Linked List Calendar Scheduler for a low number of Buckets. At high numbers of Buckets (when there are less than one job per Bucket) then the double linked list would be of little advantage and its set up cost would inhibit the performance of the scheduler.

- The performance of Dave's Calendar Scheduler is shown to be at its most efficient when there are only a small number of events placed in each Bucket and so when the Bucket number is very large. This is good news for the Lazy scheduler which proposes using a large amount of Buckets.

- The good performance of Dave's Calendar scheduler compared with ns calendar scheduler suggest that it may be a better alternative to the ns scheduler when sort simulations or simulations where the number of jobs in the queue varies significantly. For these simulation ns calendar scheduler will be inhibited by the alterations made to the queue.

## 7.0 Biography



Dave Oldham studied Geophysics at the University of Edinburgh and graduated in 2000. He will be starting a PhD working on the TERRA project at the University of Liverpool Dept of Earth sciences in October 2000.

Project supervisor: Martin Westhead (epcc)

## 8.0 References

R. Brown, 1988, Calendar queues: a fast 0(1) priority queue implementation for the simulation event set problem.
This can be downloaded at (Sept 2000)…
http://www.acm.org/pubs/citations/journals/cacm/1988-31-10/p1220-brown/

# Appendix A: Glossary

- **Bucket(s)**
  A linked list within a calendar queue that covers a specific time range.
- **Calendar scheduler**
  An enhanced version of the ns scheduler that divides the queue up into a series of **Buckets** for a job to be inserted or canceled only the jobs in each Buckets need sorting.
- **Delay time**
  The time taken for a signal to travel over a network.
- **Event**
  A class within ns that stores the data on each individual job which must be executed in the simulated network.
- **Lazy Scheduler**
  A proposed scheduler that will use a range of Buckets to minimize the amount of sorting performed by scheduler.
- **Link**
  The part of the network which carries singles between nodes.
- **Linked List Scheduler**
  A queuing system where the jobs are kept in order with a pointer pointing to the next event in the queue.
- **Job**
  An event stored in the queue.
- **Node**
  The part of the network which does calculations and processes jobs.
- **Real Time**
  Time taken for a simulation to be run.
- **Scheduler**
  A queuing system used by a network simulator to organize events stored in the queue.
- **Time Incompatible Events**
  These are events that are scheduled to take place with a time stamp that is less than the virtual time of the simulator i.e. to take place in the past. They are a problem on parallel machines where uneven load balancing may mean that the processors have different virtual times.
- **Time Stamp**
  The virtual time at which an event should be executed.
- **Time Span**
  In the context of a calendar queue, this refers to the time range covered by all the buckets e.g. in the human Calendar the time span is one year.
- **Queue**
  The area in memory where the `event` class is stored.
- **Virtual Time**
  The time used by the clock within the simulator. This is different from real time in that say 1 hour of network or virtual time can be simulated in a few seconds of real time.

# Appendix B: Complete results

Dave's Double Linked List

| System time | Simulation time | Total number of events | sim/events(μs) |
|---|---|---|---|
| 2.64 | 106.893 | 838398 | 12.74967259 |
| 3.33 | 112.996 | 870916 | 12.97438559 |
| 2.71 | 107.38 | 839478 | 12.7912822 |
| 2.82 | 109.111 | 839471 | 12.99759015 |
| 2.82 | 107.412 | 842530 | 12.74874485 |
| 2.85 | 106.6 | 829073 | 12.85773388 |
| 2.8 | 112.294 | 871839 | 12.88013039 |
| 2.69 | 110.027 | 856122 | 12.85178982 |
| 3.04 | 115.717 | 866169 | 13.35963305 |

The performance of the double linked list scheduler taken for 9 simulations run on ring-n20.tcl for 40s on Lomond.

ns Linked List

| System time | Simulation time | Total number of events | sim/events(μs) |
|---|---|---|---|
| 2.67 | 113.3 | 883809 | 12.81951 |
| 2.69 | 110.091 | 860996 | 12.78647 |
| 3.18 | 115.687 | 875079 | 13.22018 |
| 2.89 | 107.502 | 833256 | 12.90144 |
| 2.79 | 114.68 | 860007 | 13.33478 |
| 2.83 | 118.429 | 852127 | 13.89805 |
| 2.75 | 109.089 | 843365 | 12.93497 |
| 3.18 | 112.156 | 869268 | 12.90235 |
| 2.82 | 116.143 | 860158 | 13.50252 |

The performance of the ns linked list scheduler taken for 9 simulations run on ring-n20.tcl for 40s on Lomond.

| No of Buckets | Simulation time ($10^{-9}$s) | No of events | Sim/events ($10^{-14}$s per event) |
|---|---|---|---|
| 1 | 336.678 | 2538573 | 132.6249 |
| 1 | 338.098 | 2542582 | 132.97428 |
| 2 | 340.125 | 2547356 | 133.5208 |
| 2 | 337.752 | 2541683 | 132.88518 |
| 5 | 340.584 | 2575135 | 132.2587 |
| 5 | 336.614 | 2555459 | 131.7235 |
| 10 | 331.057 | 2517484 | 131.50312 |
| 10 | 335.237 | 2545630 | 131.69117 |
| 20 | 331.419 | 2510463 | 132.01509 |
| 20 | 339.748 | 2572117 | 132.08886 |
| 40 | 333.673 | 2540368 | 131.34829 |
| 40 | 332.600 | 2527815 | 131.57608 |
| 80 | 342.410 | 2613632 | 131.00926 |
| 80 | 338.721 | 2579728 | 131.30105 |
| 160 | 335.535 | 2560845 | 131.02511 |
| 160 | 339.569 | 2580388 | 131.5961 |
| 320 | 328.554 | 2518269 | 130.46819 |
| 320 | 330.069 | 2523059 | 130.82096 |
| 640 | 332.086 | 2539685 | 130.75874 |
| 640 | 330.780 | 2522617 | 131.12573 |
| 1280 | 332.848 | 2570490 | 129.48815 |
| 1280 | 331.477 | 2560125 | 129.47688 |
| 2560 | 324.535 | 2501058 | 129.75909 |
| 2560 | 351.920 | 2567276 | 137.07915 |
| 2560 | 345.242 | 2518323 | 137.09203 |
| 5120 | 330.325 | 2545193 | 129.78387 |
| 5120 | 332.389 | 2556764 | 130.00379 |
| 5120 | 331.786 | 2564955 | 129.35354 |
| 10240 | 330.411 | 2553497 | 129.39549 |
| 10240 | 326.739 | 2540099 | 128.63239 |
| 20480 | 326.856 | 2539781 | 128.69456 |
| 5242880 | 323.665 | 2529717 | 127.94514 |
| 5242880 | 324.300 | 2541516 | 127.60101 |
| 5242880 | 324.841 | 2538210 | 127.98035 |
| 10485760 | 330.175 | 2569175 | 128.51402 |
| 10485760 | 330.013 | 2559747 | 128.92407 |
| 10485760 | 323.719 | 2518356 | 128.54378 |
| 20971520 | 329.732 | 2563302 | 128.63564 |
| 20971520 | 328.579 | 2557029 | 128.5003 |
| 41943040 | 335.573 | 2590465 | 129.54161 |
| 41943040 | 328.747 | 2536181 | 129.62285 |
| 41943040 | 332.659 | 2567170 | 129.58199 |
| 40960 | 327.215 | 2571623 | 127.24066 |
| 163840 | 330.869 | 2609170 | 126.81006 |
| 327680 | 326.356 | 2563754 | 127.29614 |
| 327680 | 322.299 | 2548589 | 126.46174 |
| 655360 | 322.287 | 2547673 | 126.5025 |
| 655360 | 325.349 | 2573114 | 126.44174 |
| 655360 | 325.678 | 2557969 | 127.31898 |
| 1310720 | 317.959 | 2516944 | 126.3274 |
| 1310720 | 325.258 | 2560843 | 127.01208 |
| 2621440 | 318.173 | 2507570 | 126.88499 |
| 2621440 | 324.606 | 2545858 | 127.50358 |
| 81920 | 328.182 | 2576969 | 127.35194 |
| 81920 | 328.559 | 2580554 | 127.32111 |
| 81920 | 323.738 | 2547117 | 127.09978 |
| 20480 | 326.723 | 2566558 | 127.30006 |
| 20480 | 325.418 | 2551842 | 127.52279 |
| 5120 | 320.197 | 2519901 | 127.06729 |
| 5120 | 323.099 | 2549062 | 126.75212 |
| 2560 | 329.203 | 2555742 | 128.80917 |
| 2560 | 328.976 | 2559337 | 128.53954 |
| 2560 | 327.344 | 2553164 | 128.21111 |
| 640 | 334.531 | 2568765 | 130.23029 |
| 640 | 324.491 | 2501543 | 129.71634 |
| 10240 | 323.226 | 2551881 | 126.66186 |
| 10240 | 321.046 | 2540651 | 126.36368 |
| 20480 | 324.648 | 2561398 | 126.74641 |
| 20480 | 324.144 | 2555675 | 126.83303 |
| 40960 | 327.286 | 2568250 | 127.43541 |
| 40960 | 324.101 | 2544949 | 127.35069 |
| 81920 | 325.950 | 2558954 | 127.37626 |

The performance of the ns Dave's Calendar scheduler run on ring-n20.tcl for 120s on Lomond.

| Dave's Double Linked List | | | | |
|---|---|---|---|---|
| system | simulation time ($10^{-9}$s) | Events | sys/events | sim/events ($10^{-9}$s per event) |
| 8.29 | 325.138 | 2529980 | 3.27671E-06 | 0.0001285 |
| 8.29 | 331.945 | 2584631 | 3.20742E-06 | 0.0001284 |
| 8.07 | 327.75 | 2574860 | 3.13415E-06 | 0.0001273 |
| 8.3 | 332.929 | 2606718 | 3.18408E-06 | 0.0001277 |
| 8.35 | 323.951 | 2538339 | 3.28955E-06 | 0.0001276 |
| | | av | 3.21838E-06 | 0.0001279 |
| | | sd | 6.49202E-08 | 0.0000005 |

| Dave's Linked List | | | | |
|---|---|---|---|---|
| system | simulation time ($10^{-9}$s) | Events | sys/events | sim/events ($10^{-9}$s per event) |
| 8.4 | 328.706 | 2561941 | 3.28E-06 | 0.0001283 |
| 7.37 | 340.811 | 2598117 | 2.84E-06 | 0.0001312 |
| 9.08 | 328.55 | 2559234 | 3.55E-06 | 0.0001284 |
| 8.13 | 325.299 | 2531140 | 3.21E-06 | 0.0001285 |
| 7.73 | 323.344 | 2522397 | 3.06E-06 | 0.0001282 |
| | | av | 3.19E-06 | 0.0001289 |
| | | sd | 2.63E-07 | 0.0000013 |

| ns linked List | | | | |
|---|---|---|---|---|
| system | Simulation time ($10^{-9}$s) | Events | sys/events | sim/events ($10^{-9}$s per event) |
| 8.09 | 333.158 | 2587450 | 3.12663E-06 | 0.0001288 |
| 8.48 | 325.496 | 2530946 | 3.35053E-06 | 0.0001286 |
| 8.69 | 324.467 | 2529810 | 3.43504E-06 | 0.0001283 |
| 7.91 | 326.376 | 2556730 | 3.0938E-06 | 0.0001277 |
| 8.15 | 330.539 | 2580307 | 3.15854E-06 | 0.0001281 |
| | | av | 3.23291E-06 | 0.0001283 |
| | | sd | 1.50723E-07 | 0.0000004 |

| ns Calendar | | | | |
|---|---|---|---|---|
| system | simulation time ($10^{-9}$s) | Events | sys/events | sim/events ($10^{-9}$s per event) |
| 8.21 | 327.652 | 2552567 | 3.22E-06 | 0.0001284 |
| 8.83 | 321.858 | 2509847 | 3.52E-06 | 0.0001282 |
| 7 | 324.962 | 2541072 | 2.75E-06 | 0.0001279 |
| 8.34 | 328.371 | 2557148 | 3.26E-06 | 0.0001284 |
| 8.49 | 326.682 | 2539210 | 3.34E-06 | 0.0001287 |
| | | av | 3.22E-06 | 0.0001283 |
| | | sd | 2.84E-07 | 0.0000003 |

| Dave's Calendar Scheduler with 14422 Buckets | | | | |
|---|---|---|---|---|
| system | simulation time ($10^{-9}$s) | Events | sys/events | sim/events ($10^{-9}$s per event) |
| 8.32 | 320.66 | 2521060 | 3.3002E-06 | 0.0001272 |
| 8.21 | 322.097 | 2541341 | 3.23058E-06 | 0.0001267 |
| 7.8 | 322.301 | 2550973 | 3.05766E-06 | 0.0001263 |
| 7.87 | 324.45 | 2565843 | 3.06722E-06 | 0.0001264 |
| 7.78 | 329.887 | 2592701 | 3.00073E-06 | 0.0001272 |
| | | av | 3.13128E-06 | 0.0001268 |
| | | sd | 1.27437E-07 | 0.0000004 |

The system and simulation times for a variety of schedulers run for 120s on ring-n20.tcl on Lomond. 5 runs are made for each scheduler and the average of the system and simulation per event are shown.

# Appendix C: Source code

## The Double-Linked –List scheduler

**Scheduler.cc**

```
void Double_Linked_list::insert(Event* e){

	//Get the time the event is due to be executed
	double execution_time = e->time_;

	//Find the event with the lowest execution time after this event
	Event **p_ptr;
	Event **q_ptr;
	//Find the event before e - this is stored in p_ptr
	for (p_ptr = &list_start_ptr; *p_ptr !=0; p_ptr = &(*p_ptr)->next_){
		if (execution_time < (*p_ptr)->time_)
			break;
		q_ptr=p_ptr;
	}
	//Set next_ at the new event to point to the event after it
	e->next_ = *p_ptr;

	//Set next_ at the event before to point to the new event
	*p_ptr = e;

	//The following is a series of if statements to find out if the event is the first
event in the queue, at the top of the queue, at the bottom of the queue or in the middle
of the queue - there has to be a better way of doing this...
	if (e->next_==NULL){
		//Bottom or Only event in the queue
		if (p_ptr == &list_start_ptr)
			//Only event
			e->last_=NULL;
		else
			//Bottom event
			e->last_=*q_ptr;
	}
	else{
		//Top or General event
		e->last_=(e->next_)->last_;
		(e->next_)->last_=e;
	}
};



void Double_Linked_list::cancel(Event *e){

	if (e->last_ == NULL){
		//First or top job
		list_start_ptr=e->next_;
	}
	else{
		//Bottom or General job
		(e->last_)->next_=e->next_;
	}
	if (e->next_ != NULL){
		//Top or General job
		(e->next_)->last_=e->last_;
	}
	//else ... for e->next_==NULL Bottom or first/only job and nothings needs doing

	//make the uid of the canceled event -ve so that ns knows it has been canceled
	e->uid_ = - (e->uid_);
};

Event* Double_Linked_list::deque(){
	//count--; //debug
```

```
        //printf ("d");//DEBUG
        Event *e = list_start_ptr;
        if (e)//always returns true: e is the first event in the queue
                {
                        list_start_ptr = e->next_; //updates the pointer to the first event
in the list
                        if (e->next_ != NULL)
                                //if this is not the last event
                                (e->next_)->last_=NULL;
                }
        return(e); //Returns the pointer to the event at the head of the queue
};

Event* Double_Linked_list::lookup(int uid){

        Event *e;
        //loop thorough all the events
        for (e = list_start_ptr; e != 0; e = e->next_)
                if (e->uid_ == uid)
                        break;
        return(e);
};
```

# Dave's Calendar scheduler

**Scheduler.cc**

```
////////////////////////////////////////////////////////////
/* Dave's Calendar scheduler                               */
////////////////////////////////////////////////////////////

// NO_BUCKETS – This includes an extra overflow bucket
// TIME_SCALE – The length of time over which the simulation is run


#define NO_BUCKETS 640
#define TIME_SCALE 120

class /*Daves*/ Calendar : public Scheduler {
public:
        inline Calendar(); //Constructor, initialize the list start pointers
        virtual void insert (Event*);//Add an event to the queue
        virtual void cancel(Event*);//Remove an event from the queue
        virtual Event* deque();//Execute the event with the lowest time stamp.
        Virtual Event* lookup(int uid);//Returns a pointer to the event with the uid
protected:
        Event *list_start_ptr[NO_BUCKETS+1];    //The first event in the queue
        //      int count[NO_BUCKETS+1]; //The no of jobs active in each bucket. This
should be commented out when performance is being tested.
};

//Intiailise the class
Calendar::Calendar(){
    int i;
    for (i=0;i<=NO_BUCKETS+1;i++){
                    //count[i]=0;//Commment out when testing performance
                    list_start_ptr[i]= NULL;
        }
    //count[i]=0;//comment out when testing performance
    cout << "Useing Dave's Calendar queue with " << i-2 << " buckets and ";
    i=TIME_SCALE;
    cout << i << "s time scale\n";
}

//This part of the code tells Tcl that Calendar scheduler exists
//and can be called from ns with the call 'DaveCal'.
Static class CalendarClass : public TclClass {
public:
        CalendarClass() : TclClass("Scheduler/DaveCal") {}
        TclObject* create(int /* argc */, const char*const* /* argv */) {
                return (new Calendar);
        }
} class_cal_sched;

void Calendar::insert(Event* e){

        int bucket_no; //the bucket the job will be placed in

        //Get the time the event is due to be executed
        double execution_time = e->time_;

        //Choose which Bucket to put the event in
        bucket_no = (int) ((execution_time * NO_BUCKETS ) / TIME_SCALE);
        //Check for bucket overflow
        if (bucket_no > NO_BUCKETS )
                        bucket_no = NO_BUCKETS;

        //Change the counter – note for some reason the zeroth element of the counter does
not work.
        //count[bucket_no+1]++; // Comment out when testing performance

        Event **p_ptr;
```

```
        //Find the event before e – this is stored in before_ptr
        for (p_ptr = &(list_start_ptr[bucket_no]); *p_ptr !=0; p_ptr = &(*p_ptr)->next_)
                if (execution_time < (*p_ptr)->time_)
                        break;

        //Set next_ at the new event to point to the event after it
                e->next_ = *p_ptr;
        //Set next_ at the event before to point to the new event
                *p_ptr = e;
};

void Calendar::cancel(Event *e){


        Event **q_ptr;
        int bucket_no;

        //Get the time the event is due to be executed
        double execution_time = e->time_;

        //Choose which Bucket to put the event in
        bucket_no = (int) ((execution_time * NO_BUCKETS)/TIME_SCALE);

        //Check for overflow
        if (bucket_no > NO_BUCKETS)
                        bucket_no = NO_BUCKETS;

        //count[bucket_no+1]--;//comment out when testing performance

        //Find the event before e
        for (q_ptr = &list_start_ptr[bucket_no]; *q_ptr !=NULL; q_ptr = &(*q_ptr)->next_)
                if (*q_ptr == e)
                                break;

        //Set the next_ of the event before to point to the event after e
        *q_ptr = (*q_ptr)->next_;

        //make the uid of the canceled event –ve so that ns knows it has been canceled
        e->uid_ = - (e->uid_);

};

Event* Calendar::deque(){

        Event *e;
        int i;
        //int j; //Remove for speed tests

        //Search through all the buckets in order
        for (i=0; i<=NO_BUCKETS; i++)
                if (list_start_ptr[i] != NULL)
                        {
                                e = list_start_ptr[i];

                                //Set up the new value of list_start_ptr
                                list_start_ptr[i] = e->next_;

                                //count[i+1]--; //Comment out to test speed
                                //This if stamtment prints out the no of event in each
bucket for a limited no of times (eg 0.49 – 0.51s). Be sure to comment if out of you want
to test the speed of the scheduler.
                                //if ( ((int)((e->time_)*100)%10)==0)
                                //{
                                //cout << e->time_;
                                //for (j=1;j<NO_BUCKETS;j++)
                                //cout << " , " << count [j];
                                //cout << "\n";
                                //}
                                break;
                        }
```

```
        return (e);
};

Event* Calendar::lookup(int uid){

        Event *e;
        int i;

        //loop thorough all the events in all the buckets
        for (i=0; i<=NO_BUCKETS; i++)
                for (e = list_start_ptr[i]; e != 0; e = e->next_)
                        if (e->uid_ == uid)
                                        break;
        return(e);
};


/* End of Dave's calendar scheduler code*/
```