# The Speedup of Discrete Event Simulations by Utilizing CPU Caching

Wennai Wang and Yi Yang

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/50397

## 1. Introduction

Discrete event simulation (DES) has been widely used for both theoretical and application researches in various fields. The low-cost numerical experiments by DES can be carried out repeatedly to investigate the collective behavior or dynamics of a complex system which usually consists of a huge number of elements or is too expensive to be established realistically [3]. As for engineering applications, new algorithms, measures, or even standards, often require a great deal of DES experiments before actual deploying, especially in the field of communication networks. However, the feasibility of DES method relies not only on the correctness of computational model of the target system but also the spend in the time of computation. An experiment that costs over days of computer computation will limit the application area of DES considerably.

For a DES system designed for communication network simulation, which is implemented by the event-driven method, computation tasks waiting for processing are usually represented by their corresponding events. These events need to be maintained in-sequence by a systematic scheduler. The causality that a scheduler must preserve is usually guaranteed by sorting and dispatching according to the time of event. An event is to be suspended unless its time value is smaller than or equal to the present time. When a great number of tasks or events are in pending, the cost of event scheduling becomes the top contributor to the computational time. For a large scale communication network simulation, for example, scheduling can consume over 40% of the total computational time [12]. To accelerate scheduling, several practical algorithms have been proposed and adopted for some widely used simulators such as NS2 [9] and OMNet++ [14]. These algorithms can be categorized, according to the data structure used, into linear list, partitioning list, heap and tree. Among them, the partitioning list has attracted much attention.

The partitioning list algorithm proposed by R. Brown is called Calendar Queue (CQ) [4], where a data structure, named bucket, is used to partition events into a sequence of sub-lists.

CQ can reduce the time for event searching and decrease the complexity down to O(1). Since then, some improved algorithms, such as Dynamic CQ (DCQ) [1], SNOOPy CQ [13], and sluggish CQ [15], have been proposed in order to improve the adaptability of CQ to the event distribution in the time domain. However, the way of event enqueue and dequeue remains unchanged. For these algorithms, the complexity of O(1) is only valid for bucket searching and it doesn't take into account event locating within a bucket. In addition, the overhead for bucket managing inevitably results in some negative effects on the performance of computation. K. Chung, J. Sang and V. Rego compared some earlier-day scheduling algorithms and found that, for a token-ring network simulation, both CQ and DCQ are no better than those based on heap or tree structure [6].

In the past decades, researchers have put their interesting into parallel and distributed approaches [7] to speedup simulation by a cluster of computers or processors. Parallel and distributed event scheduling need to cope with challenges such as time synchronization and task balance and remain a gap to extensive adoption. An alternative way to speedup DES simulation, we think, is to exploit the potentials offered by the modern processors. As an instance of attempts, recently H. Park and P.A. Fishwick proposed a graphics processing units (GPU) based algorithm which shows 10-fold speedup [11]. However, until now the caching mechanism of processor has not been considered in existing event scheduling algorithms. Herein we provide a cache aware algorithm and verify its improvement on the performance by extending the conventional NS2.

The motivation to utilize caching of processor or CPU is straightforward. Both cache based method and cache aware method have been applied successfully in some heavy data-retrieving algorithms, including IP routing table lookup [5] and Radix tree implementation [2], where a huge amount of data are frequently accessed. This situation happens for a large scale DES as well, and CPU caching is benefit to speed up event scheduling.

The chapter is organized as follows. Section 2 gives a analysis of event driven mechanism of the NS2 simulator, an estimation on the number of event of the typical application, and a description of the conventional CQ algorithm. A cache aware algorithm is presented in Section 3, followed by a complexity analysis on enqueue and dequeue operations. In section 4, experiments to verify CPU cache awareness are provided, aiming at the evaluation of performance and its relationship with the size of event queue. Section 5 summarizes the chapter.

## 2. NS2 and Calendar Queue algorithm

For the typical event-driven DES system, simulation events waiting for processing are buffered in a queue and sorted according to their simulation time or timestamp. A simple and natural choice is to introduce a linear list to manage the queue of event. The number and distribution of events in the time domain hence dominate the performance of queue management. Such performance depends on not only simulation complexity, but also the implementation of a simulator. In order to make the issue tractable, the following analyses are developed on the widely used network simulator NS2 [9].

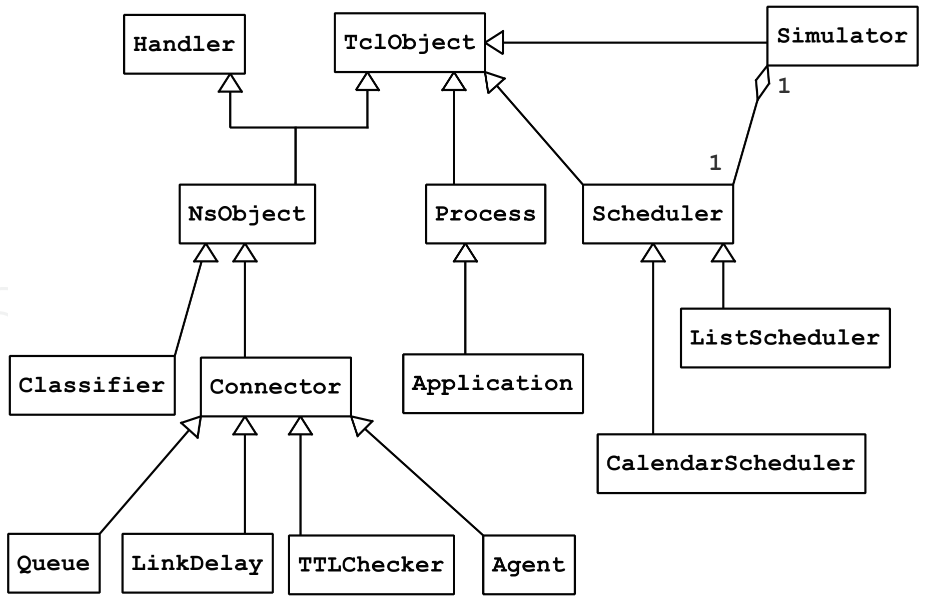

**Figure 1.** The hierarchical structure of main classes of NS2 in UML schema

## 2.1. Event handling mechanism

Aiming at network simulating at packet level, NS2 has been developed as a platform consisting of a comprehensive network elements by the object-oriented programming method. Four classes of fundamental objects, named `Node`, `Link`, `Agent`, and `Application`, are modeled for packets transporting, generating and terminating. While, functionalities such as configuration helping, routes computing, experiment managing and so on, are put into a global container class and named after `Simulator`.

Unlike `Agent` and `Application`, both `Node` and `Link` are compound classes which consist of some other elemental objects. For example, a simplest `Node` contains two routing objects, one for addressing/routing at network layer and the other for multiplexing/demultiplexing at transport layer, both derived from the same parent class `Classifier`. The physical link is modeled by `Link` which consists in series of three objects: `Queue`, `LinkDelay`, and `TTLChecker`. The elemental classes including `Agent`, `Queue`, `LinkDelay`, and `TTLChecker` are all derived from the parent class `Connector`. Together with `Classifier`, `Connector` is derived from `NsObject` and `Handler` in turn, as depicted in Fig.1.

In the class `NsObject`, two abstract functions named `recv()` and `send()` are defined for simulating packet receiving and sending, respectively. The class `Handler` encapsulates a function named `handle()` for specific event handling. These functions are overridden in the derived classes for different purposes. For instance, the implementation of `recv()` in `Classifier`, ie. `Classifier::recv()`, is to forward the received packet to the next object predefined by `Simulator` according to routing logics. Details of the implementation mechanism of NS2 can be referred to [10].

During network simulating, a packet handling causes usually one or more elemental objects to change their state. In most cases, the change results in at least one new event which relates to another function `Handler::handle()` of the next specified object. This event-driven invocation is managed systematically by a global sole object of `Scheduler`, which is instantiated and kept by another global object of `Simulator`. The derived classes based on

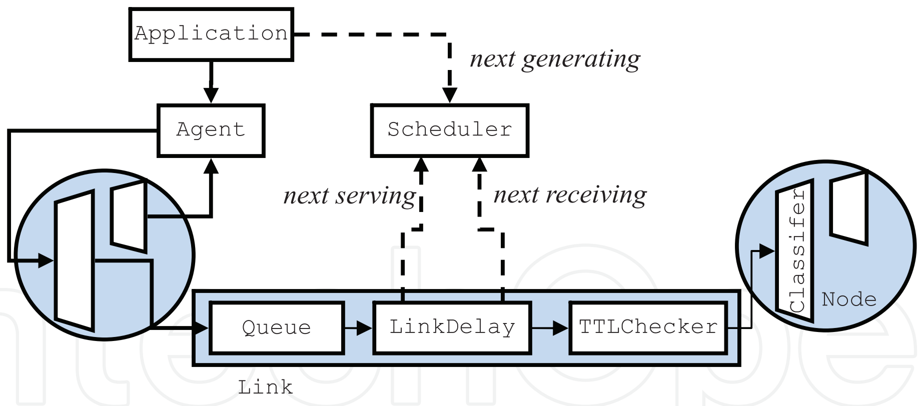

**Figure 2.** The configuration node and link of NS2 and typical events written in italic

the `Scheduler` include `ListScheduler`, which implements the linear list algorithm, and `CalendarScheduler`, which implements CQ and DCQ.

The event generation and consumption can be illustrated by tracing a packet transportation along a simulation link. As it can be imaged, the end of packet transferring at a output port of node will trigger a next serving event which will be sent back to the output queue, i.e. `Queue`, in the future time of simulation. While time delay due to propagation through the `LinkDelay` will create a packet arriving event which will be forwarded the input interface, i.e. `TTLChecker`, of the next node later. These two kinds of events lead to invocations of the class `Queue` and `TTLChecker` are buffered into the event queue of `Scheduler`, as showed in Fig.2.

Packet generating is controlled by the object `Application`, which has been developed and divided into two types which represent the stochastic generator and application simulator, respectively. The former creates an event that will trigger packet generating of `Application` recursively. The latter generates packet according to the mechanism of the simulated application. The following discussion will focus on the former for simplicity.

## 2.2. Population of standing events

As mentioned above, there are 3 types of events in NS2, including packet generating, timer and "at-event". Packet is used to model packet's transmission, timer to build protocol state-machine and packet's generator, and "at-event"to control computation. `Application` and its derivations are designed to generate a flow of packets, where one or more timer(s) is/are used to trigger the generation in a recursive manner. `Queue` and `LinkDelay` are designed for packet forwarding and transmitting through a link connecting a pair of neighbor nodes, respectively. Two timers are generated when a `LinkDelay` handles a received packet, one for triggering packet receiving at the next node later on, the other for calling `Queue` back when the `LinkDelay` is available for transmitting next packet buffered in the `Queue`. Hopping of a packet along a link, hence, induce two consequential events.

Let $h_k$ denotes the number of hops or links over an end-to-end path indexed by $k$. The number, represented by $g_k$, of packets are generated by the corresponding `Application` during One Way Delay (OWD) of the path. The total number of `Application`'s instances, $n$, is assumed as the same as the number of paths. Then, in the context of NS2, one packet-typed event induces $2h_k$ timer-typed events if the packet does not drop during forwarding and transmitting. Therefore, the number $N$ of pending events is as follows,

$$N = \sum_{k=1}^{n} g_k \times (2h_k + 1), \tag{1}$$

where, $k \in [1..n]$, is also the index of `Applications` of $n$ concurrent traffic flows. For a network with 100 nodes and a full mesh typed traffic pattern, $n$ equals 9900.

For a typical traffic with bandwidth demand 1 Mbps and packet size 1000 bytes, the packet generation rate $r$ is 125 packet/s. Assuming the propagation delay along each link is fixed as 2 ms and each path consists of 6 hops in average, we have $h_k$ = 6 and OWD is greater than 12 ms. Then,

$$g_k = r \times OWD = 125(packet/s) \times 12(ms) = 1.5. \tag{2}$$

It is easy to figure out that the number of events is approximately,

$$N = 9900 \times 1.5 \times 13 \approx 2 \times 10^5. \tag{3}$$

Here, for simplicity, we neglect the effects of packet transmitting and queuing at an output port. As one knows, transmitting delay is inverse proportion to the bandwidth of output link, and it contributes an increment to OWD. Queuing can bring about packet drops and introduce a decrement to $g_k$ during network congestion. However, as it is the result of overloading, network congestion implies much more pending events. Therefore, the above estimated $N$ can be seen taken roughly as the low bound for event scheduling.

If $N$ events are buffered in single one linear list, the average times of memory accessing for sorted insertion is about $N/2$ ($10^5$). Such heavy volume of accessing can lead to much too high time overhead. For networks with burst pattern traffics and higher traffic demands, the speed of event scheduling becomes a critical factor for fast simulation.

## 2.3. The Calendar Queue scheduling

The default event scheduler of NS2 is based on Calendar Queue (CQ) algorithm, which works similar to a desktop calendar [4]. In CQ, a bucket is used to stand for the day of year and store a sub-list of events. Events happening on the same day, despite the deference of year, are stored in the same bucket as shown in Fig.3.

For simplicity, 5 events, $E_0$ to $E_4$, are illustrated in Fig.3, whose timestamps are defined as 0, 1, 1, 3 and 16 in second. Given that the depth or size of bucket, $T_B$, is 2 seconds in time and one year consists of 3 buckets, the length of year, $T_y$, is 6 seconds. Therefore, events $E_0$ to $E_2$ are located in bucket $B(0)$, $E_3$ in $B(1)$, and $E_4$ in $B(2)$ of the third year.

The bucket location or index is determined by the following arithmetic computation,

$$n_B = \lfloor (t_e \bmod T_y)/T_B \rfloor, \tag{4}$$

where, $t_e$ is the timestamp of an event. $E_0$, $E_1$ and $E_2$ are inserted in the bucket $B(0)$ since computations according to Eq.(4) result in the same value 0. Again, the computation of the time, 16, of $E_4$ gives a value 2, indexing the bucket $B(2)$.

The computation of Eq.(4) is independent on the number of events in the queue, the complexity is therefore of O(1) [4]. However, event enqueue in a single bucket works as the

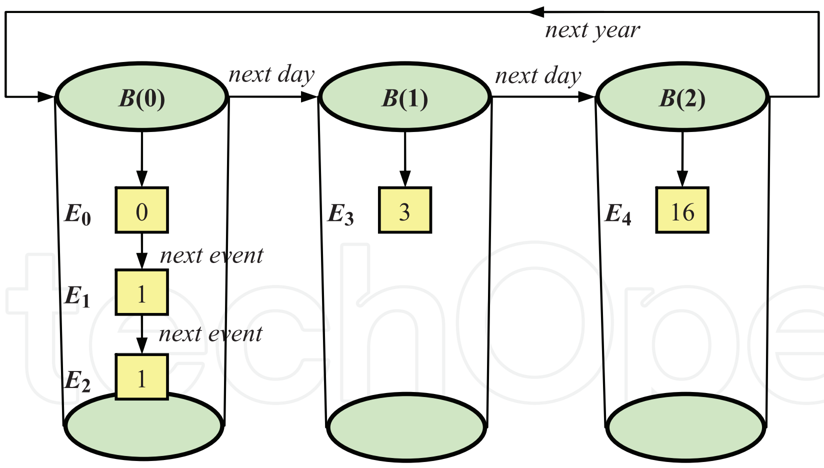

**Figure 3.** The structural relationship between event and bucket in CQ

same way as a linear list. In the case as showed in Fig.3, the insertion of $E_2$ will require two extra steps of comparing. The final complexity is dominated by the sorted insertion in the bucket unless the number of bucket is greater than the total number of events and the distribution of event in time is uniform.

Actually, bucket plays as a container and its size determines events partitioning. The fixed structure of bucket becomes an obstacle for performance improvement. It has been shown that a skewed nonuniform distribution of events in the time domain can degrade the performance of CQ [12]. Since then, several improved scheduling algorithms [1, 13, 15] have been proposed to solve the problem of adaptability to the distribution.

In addition to the structure of bucket, there are two issues that CQ and the related improved algorithms can not cope with readily. One is concurrent events insertion and the other is the earliest event fetching. The former can be seen from the insertion of $E_2$ after $E_1$ as shown in Fig.3. Although they have equal timestamp, logically $E_2$ occurs later than $E_1$ and one more comparison is needed to insert $E_2$. If the number of concurrent events is huge, say 9900 as cited in Eq.(3), a large number of comparisons are needed. The latter can be illustrated via $E_4$'s fetching. After $E_3$ departing, a CQ scheduler will carry out 6 times of bucket access across two years, then reach $E_4$. This is more complex than a linear list, in which the earliest event is always located at the head.

As is seen, CQ and the related improved algorithms do not take the modern structure of processor into considerations, especially high speed CPU caches. It has been demonstrated that a cache aware algorithm can speedup greatly an application that involves heavy data-retrieving [2, 5]. In the following section, we provide a fast cache aware scheduling algorithm to accelerate the simulation of large scale communication networks.

## 3. The cache aware scheduling

### 3.1. Data structure for event partitioning

Similar to CQ, the algorithm with cache awareness belongs to the category of partitioning list. Two lists work in a correlated manner, one for DES events and named event queue (*Qe*), and the other for indexing the sub-list of DES events and named digest queue (*Qd*). *Qe* is organized the same as a linear list, while *Qd* is implemented in an array structure acts as a

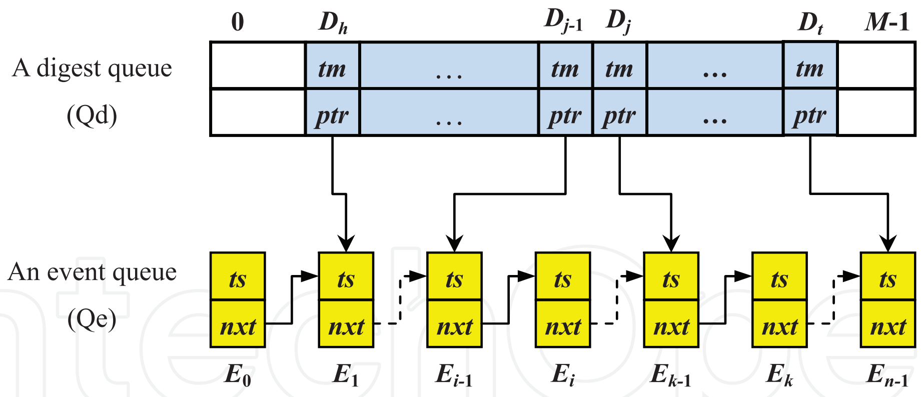

**Figure 4.** Event digests and their relationship with DES events

ring-typed buffer. The element of *Qd*, called digest, is used to index the tail of a sub-list of *Qe*, as depicted in Fig.4.

A digest $D_j$ contains a time delimiter *tm* and an event pointer *ptr* to the corresponding DES event of a sub-list. In Fig.2, $D_0$ is the digest of the sub-list ($E_0$, $E_1$), and $D_j$ is the digest of the sub-list ($E_i$, ..., $E_{k-1}$). The time delimiter of digest is defined as follows,

$$tm = \lfloor g \times ts \rfloor \tag{5}$$

where the coefficient $g$ is used to control the size of sub-list, and *ts* is the timestamp of the corresponding event. If $g$ = 1, in unit of timestamp, a sub-list consists of only those events with the same timestamp. For $g > 1$, events happen within $g$ interval are indexed by a digest pointing to the latest of them. In effect, the configurable coefficient $g$ is analog to bucket size of CQ. The difference is that two successive digests need not have their *tm* contiguous. In other words, $D_{j+1} \rightarrow tm$ - $D_j \rightarrow tm$ is allowed to be greater than $1 \times g$. This makes our sub-list more adaptive to event distribution in the time domain.

The value of *ptr* of a digest is the address of corresponding DES event. The function of this address is two-fold, one for heading of the next sub-list, the other for tailing of the current sub-list. The reason why the *ptr* points to the tail of a sublist is to avoid repeating comparisons for concurrent events with the same time-stamp. This situation happens more frequently in case simulation parameters, ex. bandwidth or link delay, are configured with the same value.

### 3.2. Enqueue and dequeue operations

As described in Algorithm 1, inserting a new DES event, or ENQUEUE, is a little more complex operation than a linear list. There are 6 significant steps in ENQUEUE operation, including,

- to calculate the time maker, $x$, of the new DES event, according to Eq.(4);
- to find the target sub-list by comparing $x$ with each digest of *Qe*;
- to sort and insert the new event into the sub-list by `InsertList()`;
- to replace the digest if the new event is inserted at the tail, otherwise;
- to update the digest if digest queue is full, otherwise;
- to create a digest and insert it into digest ring buffer by `InsertDigQueue()`.

**Algorithm 1:** ENQUEUE operation over a non-empty digest queue

```
input  : A new event $new$, a DES event list headed by $H$, and a digest ring headed by $h$
         and tailed by $t$.
output: Updated event list and digest ring.

$x \longleftarrow \lfloor g \times new \rightarrow ts \rfloor$;
$list \longleftarrow H$;
for $j \longleftarrow h$ to $t$ do /*To find the target sub-list*/
    if $D[j] \rightarrow tm > x$ then
        break;
    end
end
if $j$ != $h$ then
    $j \longleftarrow j - 1$;
    $list \longleftarrow D[j] \rightarrow ptr$;
end
InsertList ($list, new$);
if $D[j] \rightarrow tm == x$ then /*To check the time maker of the digest*/
    if $list \rightarrow ts <= new \rightarrow ts$ then /*To replace the end of sub-list*/
        $(D[j] \rightarrow ptr) \longleftarrow new$;
        return;
    end
else if $t + 1 ==h$ then /*To update the digest*/
    $(D[j] \rightarrow ptr) \longleftarrow new$;
    $(D[j] \rightarrow tm) \longleftarrow x$;
else /*To create a digest and insert it at ($j$+1)-th of the digest ring */
    InsertDigQueue ($new, x, j + 1$);
end
```

Since an array structure is used for the digest queue, the conventional ring-buffer access method can be adopted to implement `InsertDigQueue()`. The function `InsertList()` is, however, the same as that for a linear list.

The event departure or DEQUEUE operation is simply as a linear list, as shown in Algorithm 2. The function `FetchList()` is to fetch the head of DES event list, and the following block of codes to update the variable *h* is to remove the first of digest queue. The extra processing on the head is used to keep the correctness of digesting.

**Algorithm 2:** DEQUEUE operation when digest queue is not empty

```
input  : An event queue $Qe$, and a digest ring headed by $h$.
output: The earliest event $evt$.

$evt \longleftarrow$ FetchList($Qe$);
if $D[h] \rightarrow ptr == evt$ then
    $h \longleftarrow h + 1$;
end
```

### 3.3. Cache awareness and computational complexity

The difference between a cache aware algorithm and CQ is the way to partition events into sub-lists. For the cache aware algorithm, an array of digest is designed to index sub-lists and works as a ring buffer. In the ENQUEUE operation, both sub-list searching and digest inserting (`InsertDigQueue()`) execute at first over the array.

Generally speaking, a fixed array of data is with great possibility allocated by the memory management of an OS to be continuously distributed in the memory space. This locality is feasible for fast accessing by a processor with caches. Caches are fast memories in which a block of data around the request are loaded from the main memory. The caching operation is benefit to the future requests if they locate in the loaded block. The ratio, $\beta$, of access latency of main memory to cache, is usually greater than 10. Therefore, an optimum data structure and its accessing can be designed to utilize the benefit. Two types of approach have been used in designing, one is to access cache directly by instructions based on the hardware structure, the other is to design a deliberate data structure and access them as locally as possible. The former promises better performance but less applicable for different hardware structures, while the latter is cache awareness and has a better compatibility for general purposes.

Buckets of CQ/DCQ embedded in NS2 are allocated by an array and can assure locality. But this locality can not make use of the benefit of caching, because the bucket determination for event searching is computed directly and has no relationship with buckets' allocation.

For the cache aware algorithm, the target sub-list is searched via the digest queue $Qd$. If whole of $Qd$ can be loaded into caches, the searching can speed up $\beta$ times. Therefore, the complexity of sub-list searching is equivalent to $O(\frac{m}{2\beta})$. For an average distribution of event in time, the size of each sub-list bing $\frac{n}{m}$, the complexity of `InsertList()` is then $O(\frac{n}{2m})$. This leads to the complexity of $O(\frac{m}{2\beta}) + O(\frac{n}{2m})$ for an ENQUEUE operation. The optimum condition is as follows,

$$m = \sqrt{\beta n}. \tag{6}$$

Consequently, the lowest complexity of ENQUEUE is $O(\sqrt{\frac{n}{\beta}})$, twice as that of CQ configured with $m = \sqrt{\beta n}$ buckets. However, for the case of a sub-list consists of concurrent events with the same timestamp, the cache aware algorithm has an advantage over CQ since that the tail of sub-list is indexed. Sublist searching of the algorithm is of O(1) while that of CQ goes up of $O(\sqrt{\frac{n}{\beta}})$.

As for DEQUEUE operation, the cache aware algorithm is equivalent to remove the head of a list. The complexity is O(1) and it is independent on the distribution of event in time. As a conclusion, the algorithm is better than CQ.

## 4. Implementation and performance evaluations

### 4.1. Description of implementation

The cache aware algorithm has been implemented within an extended class `CacheScheduler` which is derived from the existing `ListScheduler`. Two functions, `insert()` and `deque()`, are overridden, and some helper functions appended. Modifications are based on the NS with version 2.33 and should be compatible with the most of other versions because no change is required except a config modification for Simulator is required to select CacheScheduler rather than CalendarScheduler in default.

Two fixed array are defined in the class CacheScheduler, one named key_ with type of unsigned int for digest's time, the other named event_ with type of Event * for digest's pointer. Two variable members, head_ and tail_, are defined to index the ring-typed buffer. The coefficient $g$ in Eq.(5) is represented by the third variable member pricision_, and the size of digest by the last size_.

The overridden function deque() is defined in c++ as following,

```
Event* CacheScheduler::deque() {
    Event *e = queue_;
    if (e)} {
        queue_ = e->next_;
        if (event_[head_] == e) {
            event_[head_] = 0;
            key_[head_] = 0;
            head_ = ++head_ % size_;
        }
    }
    return (e);
}
```

where, the variable member queue_ points to the head of DES event list. The condition (event_[head_] == e) is for checking whether it should remove the head of digest queue or not. The c++ codes of the function insert() is showed as following,

```
Event* CacheScheduler::insert(Event *e) {
    Event **p;
    unsigned int idx, key;
    double t = e->time_;

    key = t * pricision_;
    idx = findDigest(t, key, p);

    for( ;* p != 0; p = &(*p)->next_ )
        if ( t < (*p)->time_ ) break;

    e->next_ = *p;
    *p = e;

    insertDigest(t, key, idx, *p);
}
```

where, the functions findDigest() and insertDigest() operate over the array key_ and event_. The function findDigest() results the pointer p of sublist of DES event after which the new should be inserted, and the index `idx` of the sublist in the digest queue. The c++ codes of `findDigest()` is defined as,

```
01    unsigned int CacheScheduler::findDigest(double t, \\
02                   unsigned int key, Event **& p) {
03        unsigned int idx = head_;
04
05        for (; idx != tail_; idx++, idx %= size_)
06            if (key_[idx] >= key) break;
07
08        if (idx == head_) { // is it at head
09            p = &queue_;
10            return idx;
11        }
12
13        if (idx == tail_ || event_[idx]->time_ > t) {
14            // go to the tail of previous sub-list
15            idx = idx + size_ - 1;
16            idx %= size_;
17        }
18
19        p = (Event**)event_[idx];
20        return idx;
21    }
```

Codes in the above from Line 05 to 06 are to find out the sublist that the event timestamped with *key* should be contained. Codes from Line 08 to 10 are to handle the special case of the first sublist, and codes from Line 13 to 17 are introduced to compensate tail pointing mechanism used in sublist digesting, considering that the head of a sublist is equivalent to the tail of its previous one.

The insertion function `insertDigest()` is little more complex as listed in the following,

```
01  void CacheScheduler::insertDigest(double t, unsigned int key, \\
02          unsigned int idx, Event *e) {
03      unsigned int tmp = (tail_+1) % size_; // target of tail moving
04
05      if (tmp == head_) {             // buffer is full
06          ...
07      } else if (head_ == tail_) {    // buffer is empty
08          ...
09      } else if (key_[idx] == key) { // replace only
10          if (event_[idx]->time_ <= t)
11              event_[idx] = e;
12      } else if (!e->next_) {         // append at tail
13          key_[tail_] = key;
14          event_[tail_] = e;
15          tail_ = tmp;
16      } else {                          // in the middle
17          idx = ++idx % size_;
18          if (key_[idx] > key) {        // insert a new element
19              if (tail_ > idx) {        // right moving
20                  for (tmp = tail_; tmp > idx; tmp--) {
21                      key_[tmp] = key_[tmp-1];
22                      event_[tmp] = event_[tmp-1];
23                  }
24                  tail_ = ++tail_ % size_;
25              } else {                  // left moving
26                  head_ --;
27                  idx --;
28                  for (tmp = head_; tmp < idx; tmp++) {
29                      key_[tmp] = key_[tmp+1];
30                      event_[tmp] = event_[tmp+1];
31                  }
32              }
33          }                             // end of moving
34          key_[tmp] = key;
35          event_[tmp] = e;
36      }
37  }
```

There are 5 conditional branches in the function `insertDigest()`, the 1st (omitted at Line 06) and 2nd (omitted at Line 08) handle the buffer of digest with full and empty, respectively. The 3rd case is that the new digest has the same digest time as an existing one, os replacing is required. The 4th is identified in order to avoid the complex operations as defined in the 5th case. The last case involves element movings over an array and can lead to much more memory accesses. Such processing is, however, executing on a memory area that can be allocated continually or locally. The time overhead of `insertDigest()` can be hence reduced by the benefit of CPU caching, as discussed in the section 3.

The variables `precision_` and `size_` dominate the dynamics of cache aware scheduling. The bigger the `precision_` is, the longer the sub-list of DES event tends to be. The optimal value depends on the population of standing events and distribution in the time domain. However, `size_` can be determined according to the size of CPU caches.

## 4.2. Experiment environment and results

For simplicity in performance evaluation, simulation experiments are carried out over a random network with 100 nodes, each node connects to 6 others being selected randomly. Simulation configurations are coded in a Tools Command Language (TCL) script. Every link of the network is configured with fixed bandwidth 155 Mbps and fixed propagation delay 2 ms. A Constant Bit Rate (CBR) generator with demand bandwidth 10 Mbps and packet size 1000 bytes is assigned for each traffic flow and kept active during simulation time from 0.01 to 2.0 seconds. The number of flows varies from 99, i.e. one node is chosen to generate packets to the rest, to 9900, i.e. a full mesh-typed flow pattern is arranged. Before putting into computations, the simulated network is examined and replaced by a regenerated topology until it is fully connected. In order to make the experiment more generalized, the simulated network topology is built randomly. The number of network nodes and the pattern of traffics are adjustable.

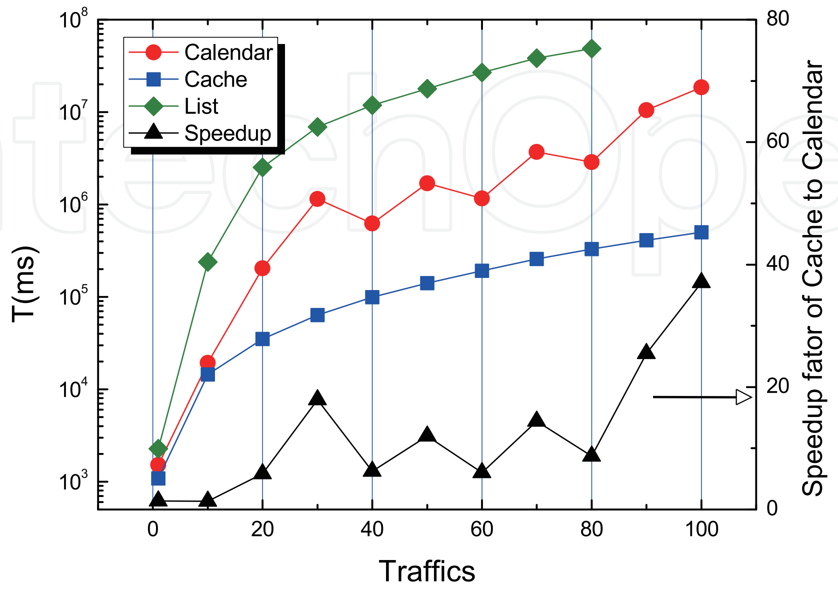

**Figure 5.** Computational time and speedup factor over CQ VS traffics (in unit of 99 flows)

| Num. of traffic flows | List (ms) | Calendar (ms) | Cache (ms) | Speedup factor |
|---|---|---|---|---|
| $1\times99$ | 2,273 | 1,517 | 1,086 | 1.40 |
| $10\times99$ | 238,797 | 19,245 | 14,455 | 1.33 |
| $20\times99$ | 2,523,570 | 203,784 | 34,991 | 5.82 |
| $30\times99$ | 6,916,930 | 1,144,140 | 63,792 | 17.94 |
| $40\times99$ | 11,862,000 | 621,804 | 99,360 | 6.26 |
| $50\times99$ | 17,934,800 | 1,692,130 | 140,996 | 12.00 |
| $60\times99$ | 26,835,100 | 1,159,060 | 192,093 | 6.03 |
| $70\times99$ | 38,388,600 | 3,712,680 | 257,185 | 14.44 |
| $80\times99$ | 48,637,600 | 2,880,920 | 330,402 | 8.72 |
| $90\times99$ | - | 10,459,600 | 410,934 | 25.45 |
| $100\times99$ | - | 18,573,400 | 501,229 | 37.06 |

**Table 1.** Computational times of three scheduling algorithms and speedups of the proposed (Cache) to CQ

Table 1 and Fig.5 show the computational time spending for simulations $T$ versus the number of traffic flows, for CQ (Calendar), the cache aware (Cache) and the linear list (List) schedulers. Experiments are carried out on a personal computer with a Intel(R) Pentium(R) 4 typed CPU with 2.93 GHz in frequency and 1024 kB in cache size. The computational time is evaluated by invoking TCL predefined command `clock` at 0.0 and 2.0 seconds in the simulation time. It can be seen from Fig.5 that the cache aware algorithm is always faster than CQ and the maximum speedup factor reaches 37.

Fig.6 shows the computational time varying with the size of digest queue, for the case configured with a cache aware scheduler and 30 sources, i.e. $99 \times 30 = 2970$ flows. Since a ring-buffer management is used in the cache aware algorithm, the actual capacity of the queue is one smaller than its literal size. The condition with the queue sized 1 means to disable the digest queue, and the cache aware algorithm is degraded and equivalent to the linear list. The condition with the queue sized 2 allows only one digest which always points to the tail of DES event list. Experiment results on the size of digest queue are also listed in Tab.2.

| size | 1 | 2 | 3 | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|---|---|
| T(ms) | 4,052,043 | 4,131,210 | 1,929,621 | 281,398 | 64,113 | 63,186 | 64,140 | 63,109 |

**Table 2.** Computational time (T) varies with the size of digest queue

From Fig.6 and Tab.2, it can be concluded that a cache aware algorithm makes use of the benefit of CPU caches when the size of digest queue is greater than 16. It is can be seen also that the speedup efficiency relies on the size of cache-line rather than the size and topology of the simulated network. The detailed analysis is given in the next section.

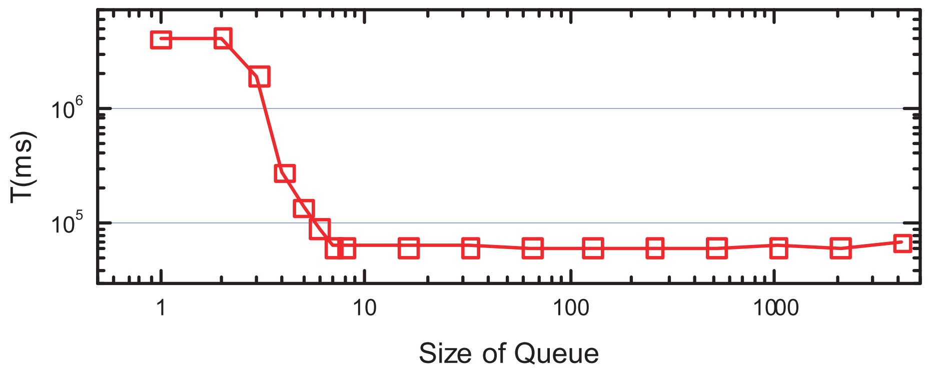

**Figure 6.** Computation time VS digest queue size

## 4.3. Effectiveness of CPU caching

The result of the experiment indicates that the cache aware algorithm can speedup DES event scheduling for large scale networks. Continuously allocated digest queue in memory is benefit to fully use the feature of caching. Meanwhile, the digest queue acts as a classifier to partition DES event list into sub-lists with a shorter length, and hence reduce the searching time of ENQUEUE operations.

The digest in the cache aware algorithm is managed by two variables, `key_` and `Event_`, each 4 bytes in length. Hence, a digest queue with size 16 occupies 128 bytes in the main memory. This is exactly equivalent to 128-byte sized L2 cache line of the processor [8] used in experiments. The dropping approaching the size 16 shown in Fig.6 coincides with the CPU L2 caching capability. Therefore, the algorithm is cache-line awareness for the CPU chip employed in our experiments. This means that computation time reduction remains nearly constant when the size of digest queue increases over that of cache line, i.e., 8 elements or 64 bytes, roughly 10 as showed in experiments. In this condition, missing in single a cache line causes a regular main memory access so that the contribution of CPU caching is saturated.

The algorithm provided in this chapter is not the best that can fully utilize the capability of CPU resources since that the design of digest queue and operations have no knowledge of CPU's microstructure. However, a further improvement that depends on the specific CPU products will bring a serious issue in compatibility. The trade-off between performance and compatibility needs extensive investigations and relates to the application field of discrete event simulations.

## 5. Conclusion

Discrete event simulation method has been employed in various research fields for exploring a complex or large system numerically. The applicability of this kind of simulation method relies in large on the computation speed. In this chapter, fast event scheduling approaches for simulations of large scale networks is discussed. The typical simulation procedures are described according to NS2 platform, followed by a brief analysis of the conventional Calendar Queue algorithm. Based on the list partitioning method, a digest queue on the fixed array structure is introduced to partition and index the sub-list of DES events. The double-list structure can utilize the benefit of caching mechanism of modern CPU chips.

Details of enqueue and dequeue algorithm are given and the complexity analysis is presented. Also, developments based on the open-source NS2 software are showed for the algorithm implementation. In order to verify the benefit of cache awareness to speedup simulation, we report computational experiments over an on-shelf personal computer. It is shown that the performance of a cache aware algorithm is considerably better than the conventional Calendar Queue. Experiment results show that the cache aware algorithm makes simulation faster by up to a factor of 37. The improvement in computation efficiency is applicable for any ordinary network topology and traffic pattern since the randomized topologies and patterns are employed in our experiments.

For simplicity, the algorithm's design and implementation are focused on the network simulation with all traffics assumed over UDP/IP protocol stacks. As for the simulation involves in TCP connection, events used for protocol machine controlling, such as those for time-out processing, bring numbers of canceling operations into event scheduling. The event cancelation in the queue based on the ring-typed buffer should result in more computational complex than that of convectional algorithms. A better solution, we think, is to put these "at-events" being significant for the local network node into a separate object from the global event scheduler. This separation is not only benefit in speedup to the scheduler but also to parallel enabler.

On the other hand, the advancement of multi-core architecture of CPU deserves to be verified whether the proposed algorithm can utilize caches of multi-core chip as well. Moreover, it's valuable to study multi-core technology to achieve parallel speedup in the meantime.

## Author details

Wennai Wang and Yi Yang
*Key Lab of Broadband Wireless Communication and Sensor Network Technology, Nanjing University of Posts and Telecommunications, China*

## 6. References

[1] Ahn, J. & Seunghyun, O. [1999]. Dynamic calendar queue, *Annual Simulation Symposium*, pp. 20–25.

[2] Askitis, N. & Sinha, R. [2007]. Hat-trie: a cache-conscious trie-based data structure for strings, *Proc. of the thirtieth Australasian conference on Computer science - Volume 62*, Australian Computer Society, Inc., Darlinghurst, Australia, pp. 97–105.

[3] Banks, J. [1999]. Introduction to simulation, *Proceedings of the 31st conference on Winter simulation: Simulation—a bridge to the future - Volume 1*, WSC '99, ACM, New York, NY, USA, pp. 7–13.
URL: *http://doi.acm.org/10.1145/324138.324142*

[4] Brown, R. [1988]. Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem, *Communications of the ACM* 31(10): 1220–1227.

[5] Chiueh, T. & Pradhan, P. [1999]. High performance ip routing table lookup using cpu caching, *INFOCOM*, pp. 1421–1428.

[6] Chung, K., Sang, J. & Rego, V. [1993]. A performance comparison of event calendar algorithms: an empirical approach, *Softw. Pract. Exper.* 23: 1107–1138.

[7] Fujimoto, R. [1999]. Parallel and distributed simulation., *Winter Simulation Conference'99*, pp. 122–131.

[8] Hinton, G., Sager, D., Upton, M., Boggs, D., Carmean, D., Kyker, A. & Roussel, P. [2001]. The microarchitecture of the pentium 4 processor, *Intel Technology Journal* 5(1): 1–13.

[9] Issariyakul, T. & Hossain, E. [2008]. *Introduction to Network Simulator NS2*, 1 edn, Springer Publishing Company, Incorporated.

[10] Fall, K. & Varadhan, K. [2009]. The ns manual.
URL: *http://www.isi.edu/nsnam/ns/ns-documentation.html*

[11] Park, H. & Fishwick, P. A. [2011]. An analysis of queuing network simulation using gpu-based hardware acceleration, *ACM Trans. Model. Comput. Simul.* 21(18): 1–22.

[12] Siangsukone, T., Aswakul, C. & Wuttisittikulkij, L. [2003]. Study of optimised bucket widths in calendar queue for discrete event simulator, *Proc. of Thailand's Electrical Engineering Conference (EECON-26)*, Phetchaburi, pp. 6–7.

[13] Tan, K. L. & Thng, L. [2000]. Snoopy calendar queue, *Winter Simulation Conference*, pp. 487–495.

[14] Varga, A. & Hornig, R. [2008]. An overview of the omnet++ simulation environment, *Simutools '08: Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, ICST, Brussels, Belgium, pp. 1–10.

[15] Yan, G. & Eidenbenz, S. [2006]. Sluggish calendar queues for network simulation, *Proc. of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'06)*, Monterey, CA, USA, pp. 127–136.