A COMPUTER SYSTEMS SIMULATOR

Ken Barker* and Graham Birtwistle*
Computer Science Department
University of Calgary

Abstract

We discuss the structure and content of a computer systems simulator being written in Demos, a Simula extension. The simulator is intended to bridge the gap between out and out performance simulators and simulators for software design. It provides skeleton descriptions of hardware components, and job profiles which can be tailored to particular applications. Use of the simulator is illustrated by examples of disk definitions, segmentation and paging.

INTRODUCTION

Simulation models of computer systems have been used to advantage for many years now. With recent advantages in queueing theory, many rough and ready performance issues are most conveniently undertaken using a queueing network package rather than a simulator. At the other end of the problem spectrum, simulation has been used to develop algorithms for software processes in real time systems, local area networks and long haul networks.

Two spectacular successes were scored by Belsnes [1] and Salih [2]. Belsnes simulated the link level and packet level protocols for X25 in Simula. Since the network node hardware was a NORD 10 mini which supports Simula, and Simula supports the notion of a process, it proved possible to use the simulation code for the actual implementation. Salih's work, also Simula based, mapped existing real time software into a detailed simulation with a Simula process corresponding to each real time process. Exact instruction timings were built into the model and Salih came up with an improved design that was not only 15% faster in the most critical region, but also more 'correct' (a possible deadlock was detected).

In practise, 'software' simulators are too slow running to be used as they stand to study performance issues. This is because all components, even relatively unimportant ones, are presented in full detail. Instead they, together with processes with fine grains of time, should be broad brushed out.

In this paper we discuss the structure and content of a simulator we are developing which is intended to (at least partially) overcome this difficulty. It provides a framework which can be used almost as it stands to study performance issues. But its main purpose is to allow for the insertion of detailed algorithmic descriptions within a fast running framework.

In our applications so far, we have mainly been interested in studying the impact of various memory management policies on job throughput. In our experiments, job profiles have reamined unchanged whilst we have experimented with various underlying architectures: partitioned, segmented and paged, with or without virtual memory. Our outline of the simulator is illustrated with segments of code taken from disk scheduling, segmentation and paging models. The paging model is being used for experimentation with strategies for overcoming thrashing in a multi-programming environment. Indeed, it was this study which prompted us to work backwards and design the simulator!

The work most closely related to our simulator is Unger's Oasis [3], which is now a Demos [4,5] extension.

However, Oasis is oriented towards system kernel process descriptions and hence Oasis programs require much detail. But there is a link, and we felt that our simulator ought to have a related name; it has been modestly christened 'Green' (short for Greenpiece).

STRUCTURE OF GREEN

Green has been written as an extension to Demos, itself a Simula based package. Demos builds in the simulation clock, event list scheduling, various synchronisations, and several data

*This work was supported by a grant from NSERC.

Proceedings of the 1982
Winter Simulation Conference
Highland * Chao * Madrigal, Editors

collection, random number, tracing and reporting routines. Two versions of Green will be made available. A debug version which uses standard Demos trace and reporting facilities. Once the model has been accepted, the unchanged source should be recompiled under the P_Green package which ignores traces and overrides the Demos data collection standards. Instead, separate files are created for each stream, and items are individually written-out for later analysis in the manner outlined in [6].

Green provides tools for the class of computer systems with an arbitrary number of io devices, and a block of memory shared by several identical cpu's. The simple program listed below represents a hardware configuration of two cpu's, eight disk packs accessed via the same channel and with twenty terminals logged on. Each user thinks for a while and then submits a request. Each request loops m times and each loop contains a cpu request followed by an io request to a randomly chosen disk. The user then repeats this work cycle.

```
(1)   BEGIN EXTERNAL CLASS GREEN;
(2)   GREEN
(3)     BEGIN
(4)       INTEGER N;
(5)       REF(IO_GROUP)DISK;
(6)       REF(RDIST)THINK;
(7)       REF(IDIST)BURST, LOOPS, PACK;
(8)       JOB CLASS USER;
(9)       BEGIN
(10)        INTEGER M, K;
(11)        HOLD(THINK.SAMPLE);
(12)        M := LOOPS.SAMPLE;
(13)        FOR K := 1 STEP 1 UNTIL M DO
(14)        BEGIN
(15)          EXECUTE(BURST.SAMPLE);
(16)          DISK.DOIO(PACK.SAMPLE);
(17)        END;
(18)        REPEAT;
(19)      END***USER***;
(20)      DISK  :- NEW IO_GROUP("DISK", 8,
                   NEW UNIFORM ("U", 0.000,
                   0.075), 0.010);
(21)      THINK :- NEW NEGEXP("THINK", 0.05);
(22)      BURST :- NEW RANDINT("BURST", 10000,
                   50000);
(23)      LOOPS :- NEW RANDINT("LOOPS/REQ", 1, 10);
(24)      PACK  :- NEW RANDINT("PACK", 1, 8);
(25)      CPU   :- NEW PROCESSOR("CPU", 0.0000001);
(26)      FOR N := 1 STEP 1 UNTIL 20 DO
(27)        NEW USER("USER").SCHEDULE(NOW);
(28)      RUN_SIMULATION_FOR(1000.0);
(29)    END;
(30)  END;
```

Line 1 declares our intended use of Green: its contents become available to the block stretching over lines 3-29 by its occurrence as block prefix on line 2. In lines 4-7 we declare the static quantity names used in the program; N is a loop counter, DISK names the 8 io devices. THINK, LOOPS, BURST, and PACK are names on distributions used to simulate the user think time, number of cpu visits per request, number of instructions executed in the current burst, and the particular disk for which this io request is intended.

Lines 8-19 detail the class of jobs we are simulating. Each user thinks for a while (line 11) and then submits a request (lines 12-17). EXECUTE simulates both the queueing for and completion of a cpu burst. The DOIO command simulates queueing for and completion of an IO request on one of the eight DISKS. The REPEAT command causes the class actions to be repeated from line 11.

In lines 20-25 we initalise the static quantities in the model and in lines 26-27 create 20 interactive jobs. The simulation run length is fixed in line 28.

When the simulation is over, a report on cpu and device usage is automatically presented (should the user be in debug mode).

Green supplies primitives for single or grouped io devices. Grouped devices share the same channel. Where an io can reasonably be modelled by a wait followed by a transfer, use the built-in definition below:

```
RES CLASS IO_DEVICE(LATENCY, TRANSFER);
  REF(RDIST)LATENCY; REAL TRANSFER;
BEGIN
  PROCEDURE DOIO;
  BEGIN
    ACQUIRE(1);
    HOLD(LATENCY.SAMPLE+TRANSFER);
    RELEASE(1);
  END***DOIO***;
  IF LATENCY == NONE THEN error;
  IF TRANSFER <= 0.0 THEN error;
END***IO_DEVICE***;
```

which represents an io device which services requests in priority order (FCFS if priorities are all the same).

For clustered disks on the same channel, use the built in class IO_GROUP whose outline is:

```
RES CLASS IO_GROUP(N, MOVE, LATENCY,
                   TRANSFER);
  INTEGER N; REF(RDIST)MOVE, LATENCY; REAL
                   TRANSFER;
BEGIN
  PROCEDURE DOIO(PACK); INTEGER PACK;
  BEGIN
    D[PACK].ACQUIRE(1);
      ACQUIRE(1);      ! channel;
      RELEASE(1);
      HOLD(MOVE.SAMPLE);
      ACQUIRE(1);      ! channel again;
      HOLD(LATENCY.SAMPLE+TRANSFER);
      RELEASE(1);
    D[PACK].RELEASE(1);
  END***DOIO***;
  REF(RES)ARRAY D[1:8];
END***IO_GROUP***;
```

Typical initialisation and call sequences were shown the program. Should neither of these built-in patterns suffice, then the user can define his own io disk allocator. For example, here is an allocator patterned on the SCAN algorithm (the logic and code are taken from [7])

```
     RES CLASS SCAN(LATENCY, TRANSFER, NR_CYLS);
       REF(QDIST)LATENCY;
       REAL TRANSFER; INTEGER NR_CYLS;
       VIRTUAL:  PROCEDURE HEADMOVE;
     BEGIN
       INTEGER HEAD_POS;
       BOOLEAN DIRECTION, BUSY;
       REF(CONDITION)UP, DOWN;

       PROCEDURE DOIO(DEST); INTEGER DEST;
       BEGIN
         IF BUSY THEN
         BEGIN
           IF HEAD_POS < DEST OR
              HEAD_POS = DEST AND UP
              THEN UPQ.WAIT(DEST)
              ELSE DOWNQ.WAIT(-DEST);
         END;
         BUSY := TRUE; HEADPOS := DEST;
         HOLD(HEADMOVE+LATENCY.SAMPLE+TRANSFER);
         BUSY := FALSE;
         IF UP THEN
         BEGIN
           IF UPQ.LENGTH > 0 THEN UPQ.SIGNAL ELSE
           BEGIN
             UP := FALSE;
             DOWNQ.SIGNAL;
           END;
         END ELSE
         BEGIN
           IF DOWNQ.LENGTH > 0 THEN DOWNQ.SIGNAL
                ELSE
           BEGIN
             UP := TRUE;
             UPQ.SIGNAL;
           END;
         END;
       END***DOIO***;
     END***SCAN***;
```

The exact definition of HEADMOVE from HEAD_POS to DEST has been left open (it has a virtual specification). The definition uses the Green CONDITION primitive in which jobs can be blocked queued according to a priority which is passed as a parameter in the calls on WAIT. Dormant processes are wakened by calls on SIGNAL.

Job patterns, like io devices are usually straightforward. Their bodies reflect the routings of that class of job as it moves from station to station with DOIO's calls on io devices and EXECUTE bursts on a cpu. When more detail is required, we drop from Green into the Simula host. As an example, the job profile below represents a class of jobs in a non-virtual memory segmented system and displays a 'get_segment' algorithm. Free segments are kept in SEGQ ordered according to their start address:

```
     JOB CLASS JOB_SEGMENT;
     BEGIN REF(SEGMENT)SEG;
       PROCEDURE LOAD;
       BEGIN
         REF(SEGMENT)S; BOOLEAN FOUND;
       RETRY:
         S :- S.FIRST;      ! first free segment;
         WHILE(S =/= NONE AND NOT FOUND) DO
         BEGIN
           IF S.LENGTH > SIZE THEN
           BEGIN
             SEG :- NEW SEGMENTS(S.START, SIZE);
             S.START := S.START+SIZE
             S.LENGTH := S.LENGTH-SIZE;
             FOUND := TRUE
           END ELSE
           IF S.LENGTH = SIZE THEN
           BEGIN
             SEG :- S;
             SEG.OUT;
             FOUND := TRUE;
           END ELSE S :- S.SUC;
         END;
         IF NOT FOUND THEN
         BEGIN
           BLOCKED.WAIT;
           GOTO RETRY
         END;
         SEG_FOUND:
           PRIORITY := 1;      ! high priority for new
                                    job;
           DISK_E.DOIO(SIZE);! read in the segment;
           PRIORITY := 0;
         END***LOAD***;

         PROCEDURE FREE.....;
         .....

         LOAD;
         FOR K := 1 STEP 1 UNTIL n DO
         BEGIN
           EXECUTE(burst);
           DISK.DOIO(n);
         END;
         FREE;
       END***JOB_SEGMENT***;
```

The final built-in device definition is that of a cpu. In our definition we have striven for ease of use and built-in many options - time slicing, paging, segmentation - not all of which will be wanted in any one model. The options are supplied with default definitions which ensure that unless they are overridden by the user, the (unrepresented) characteristics will not manifest themselves. The basic framework is:

```
     ENTITY CLASS PROCESSOR(SPEED); REAL SPEED;
       VIRTUAL: PROCEDURE GET_PAGE, GET_SEG, OTHER;
     BEGIN
       REF(JOB)J;
       select next job J(user or system) from
          READYQ;
       compute its burst length, B;
       HOLD(B);
       IF normal termination THEN J.SCHEDULE(NOW)
                  ELSE
       IF time sliced        THEN replace I in
          READYQ ELSE
       IF page fault         THEN GET_PAGE(J)
                  ELSE
       IF segment fault      THEN GET_SEG(J)
                  ELSE
          OTHER;
       REPEAT;
     END***CPU***;
```

If there are several cpu's they contend for jobs in READYQ. When a job has been selected, the length of its next burst is computed from data within J itself, and the burst is carried out. As a by product of computing the length of the burst, we also get an indication of why the job came off the cpu. The indicator is tested in the IF statements. In paged systems, a routine GET_PAGE has to be supplied to cover the case when a page fault arises. Similarly for segmented systems. The time slice is infinite unless set by a call

```
    SET_TIME_SLICE(t);
```

which sets its value to t(t > 0.0).

Note that our style of cpu modelling ignores external interrupts completely - internal interrupts (time slice, page fault, segment fault) are predicted ahead of time. This mode of simulating trades a tolerable loss accuracy for a real gain in speed (a factor of two or so).

THRASHING THRASHING

We illustrate modifications to the cpu definition with an example taken from a continuing investigation into the problem of controlling thrashing. Avoiding thrashing is one of the problems to be faced when designing a multiprogramming system. If too many programs are allowed to compete for a share of memory, then one or several will be unable to obtain sufficient pages to run comfortably and will suffer from very frequent page faulting. As a result, the system throughput is degraded due to an excessive number of page transfers. When thrashing sets in, the only cure is to remove one or several jobs from the main memory. It is not satisfactory to choose a constant level of multiprogramming, since this may not give sufficient io overlap if it is set low enough to prevent thrashing. If the multiprogramming level is allowed to vary, the response to the system on the onset of thrashing will be to satisfy immediate demands by stealing from yet furtherjobs (unless care is taken). This only exacerbates the problem.

Thrashing is an interesting problem because of the interplay between an outer algorithm for controlling the load level and an inner algorithm for distributing pages fairly amongst jobs in memory. The two algorithms must mesh together.

Our examples and data are taken from [8] which contains a full problem description. We illustrate the use of Green on the first of their algorithms due to Wharton, which may be spectacularly ineffective. Wharton's (inner) algorithm gives each job a priority. A job demanding a page can take one from the pool if one is free, steal a page from another job in READYQ of lesser priority, or cannibalize a page from itself. If these attempts all fail (the job has no pages and a low priority) it is blocked.

Jobs are initially loaded with no pages and have to build up their working sets by demand. The interval between demands is predicted from a user supplied definition (NEXTPF) which here is a function of the number of pages now owned (RCP) and the job's working set (SIZE)

The Green code for a job and a cpu working to Wharton's algorithm is:

```
    JOB CLASS PROGRAM;
    BEGIN
      REAL PROCEDURE NEXTPF;
      BEGIN
        REAL K, F;    ! see [8], page 154;
        K := 0.003 * SIZE/TREQ;
        F := (2 ** (-16 * RCP/WSET) +K)/(K+1.0);
        NEXTPF := (1.0 - F)/(F * 1000.0);
      END***NEXTPF***;
      ......
      WHILE TLEFT > 0.0 DO
      BEGIN
        EXECUTE(burst);
        DISK.DOIO;
      END;
    END***PROGRAM***;

    CPU CLASS WHARTON;
    BEGIN
      PROCEDURE GET_PAGE(J); REF(JOB)J;
      BEGIN REF(JOB)V;
        IF MEM > 0 THEN MEM := MEM - 1 ELSE
        BEGIN
          READYQ.FIND(V, V.RCP > 0 AND
            V.PRIORITY < J.PRIORITY);
          IF V == NONE THEN J.INTO(BLOCKEDQ)
                      ELSE READ_PAGE(J);
        END;
      END***GETPAGE***;
    END***WHARTON***;

    SET_TIME_SLICE(9.040);
```

The core map below indicates why Wharton's inner algorithm on its own will not do: there is simply no way the top priority job can ever lose pages and eventually it will have in main memory every page it ever demanded. Each line represents a memory snapshot (at a time given on the right). Each job has its own character, and RCP, the number of pages it currently owns is displayed by printing that character RCP times. * represents a page in transit - being read in, or written out. The output shows Wharton's algorithm at its worst, showing how top priority jobs push lower priority jobs out of memory, thus reducing the effective level of multiprogramming and io overlap possible.

```
AAAAAAAAAAAAAAAA....................  1
AAAAAAAAAAAAAAAAAAAABBBBBBBCCCDDDDDD  2
AAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBB*  3
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBB  4
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  5
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  6
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  7
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  8
BBBBBBBBBBBBCCCCCCDDDDDDDDEEEEFFFGGG  9
BBBBBBBBBBBBBBCCCCCCCCCCCDDDDDDDDDDE 10
BBBBBBBBBBBBBBCCCCCCCCCCCDDDDDDDDDDD 11
BBBBBBBBBBBBBBBBCCCCCCCCCCDDDDDDDDD* 12
BBBBBBBBBBBBBBBBBBBBCCCCCCCCCCDDDDDD 13
BBBBBBBBBBBBBBBBBBBBCCCCCCCCCCDDDDDD 14
BBBBBBBBBBBBBBBBBBBBBB*CCCCCCCCCCCD* 15
BBBBBBBBBBBBBBBBBBBBBBBCCCCCCCCCCCCD 16
BBBBBBBBBBBBBBBBBBBBBBBBCCCCCCCCCCCC 17
BBBBBBBBBBBBBBBBBBBBBBBBBCCCCCCCCCCC 18
```

```
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBCCCCCCC  20
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBCCC*  21
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  22
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  23
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  24
CCCCCCCCCCDDDDDDDDDDEEEEEEEEEFFFFFF*  25
CCCCCCCCCCCCCDDDDDDDDDDDDDEEEEEEEEEE  26
CCCCCCCCCCCCCCDDDDDDDDDDDDEEEEEEEEEE  27
CCCCCCCCCCCCCCCDDDDDDDDDDDDDDDEEEEEE  28
CCCCCCCCCCCCCCCDDDDDDDDDDDDDDDEEEEEE  29
CCCCCCCCCCCCCCCCCDDDDDDDDDDDDDDDDEEE  30
```

[8] gives variations on the theme proposed by Horning and Lynch. Horning's method looked for victims randomly in memory and also proved ineffective. Lynch's algorithm superimposed an outer algorithm on Wharton's inner algorithm by throwing in a new system process called a 'drain'. This drain is activated regularly (say, every second drum revolution) and hunts for pages from the job on the cpu and all jobs in an io wait. These pages are returned to the system pool and the drain process is then passivated a while. Whilst Lynch's algorithm can be made to work satisfactorily at high cpu utilisations, it performs poorly when memory is already underutilised. This fault can be corrected by letting the drain stop if memory utilisation is low.

Our experiments indicate that it is difficult to tune the drain process against arbitrary job mixes - it has to be matched with an inner algorithm and one has to decide how often it should be activated, how many pages it should free per activation, and when it should be put to sleep. Recently we have discarded the drain process idea and simply thrown a job out when paging traffic (as evidenced by the number of jobs waiting for a page transfer) gets high. The computations are more obvious and results so far are encouraging.

CONCLUSIONS

We have presented the structure and content of our Green simulator, which has been used as an educational tool and in investigating the control of thrashing in a multi-programmed computer system. Green provides skeleton definitions of hardware components which can be used as defined for many applications. In more detailed simulations, the Green definitions can be extended, overriden or replaced by Simula code. The examples emphasize the necessity for embedding a simulator in a good general purpose programming language. By supplying two versions of Green, we can switch from test runs to production runs without modifying the source code at all. This technique has proved very useful and adapts to many other Simula (and Ada) products.

BIBLIOGRAPHY

1. D. Belsnes and K. A. Bringsrud: "X.25 DTE implemented in Simula", Proc. Eurocomp, London, 1978.

2. A. M. McQuade, A. Salih, and H. J. Gray, "Simulation of a telecommunications multiprocessor switching system", Simulation 31(5), 145-155, 1981.

3. B. W. Unger and G. Lomow, "The Oasis 4.0 Reference Manual", University of Calgary Research Report, 1982.

4. G. M. Birtwistle, "Discrete event modelling on Simula", Macmillan, 1979.

5. G. M. Birtwistle, "The Demos Implementation Guide and Reference Manual", University of Calgary Research Report: 81/70/22, 1981.

6. G. M. Birtwistle, "Advanced use of Simula", Proc. WSC81 Atlanta, 293-304, 1981.

7. C. A. R. Hoare, "Monitors on operating system structuring concept. CACM, 17(10), 549-557, 1974.

8. A. Alderson, W. C. Lynch, and B. Randell, "Thrashing in a multiprogrammed paging system", 152-167 in Hoare and Perrott (Eds.) "Operating Systems Techniques", Academic Press, 1972.