Building simulators

Simulation:

- Model the the system, using software, hardware, or both.
  - Model is a behaving the way the system should behave
  - Model can run programs, user can interact with the model
  - Perform experiments on this model
- Key issue: Abstraction.
- Model will not be the real thing
  - It won't have all details
  - It won't run at the same speed as the real machine, probably *much* slower than the real thing.

Simulation: an example

Take an airplane design

- Aerodynamics are tested on a scale model in a windtunnel
- Everything is scaled down.
- It is made from other material
- There are no passenger seats inside, abstracted away.

Simulation of computer systems works in a similar way:

- We do not model simply by scaling,
- Hardware / Software is going to be modelled by software (and possibly hardware)
- There is no such thing as scaling here.

Simulation: benefits

Simulation is cheap

- Cheaper than building a system for real.

Simulation can be done during an early stage of design

- First 4 months of development.

Simulation allow you to perform "what-if" experiments

- What happens if we double the speed of the processor-clock?
- What if we remove a cache?
- What if we use a different routing algorithm?

Simulation is a proof of concept.

Precision of Simulation?

Modelling: Abstraction from reality

- Simplifications
- Things have been made orthogonal

Laws (invariants):

- A model is never correct
- Detail may lead to higher accuracy
  - Model is as weak as weakest component
- Execution Speed = $\frac{1}{\text{Detail}}$

Find balance

- Between details and speed
- Between details in components of simulator

Level of Detail

The primary factor affecting the precision of discrete simulators is the level of detail

Choice in level of detail:	Architecture example
• Low	transistors/gates
• Medium	Instructions, memory trace
• High	abstract from application

Lets study the possibilities for a machine:

- Multiprocessor system with 4 processors and a bus running some fantastic program.

Low level

Lowest levels: make a complete implementation in logic (transistors, gates)

- Simulate this implementation at electronic level (voltages, currents, ...)

⇒ Execution time, tens of hours per clock-cycle
  - Gives information about timings within clock-cycle
  - Does not tell anything about timings of program

- Simulate this implementation at Digital level (0/1)

⇒ Quite a bit faster, minutes per instruction?
  - Does not tell anything about timings of program

Medium level

Model of the components:

- Memory:
  - Model it as an array of bytes (or integers)
- Bus:
  - Simple model: bus is in use or not
  - Could incorporate arbitration timings
- Cache:
  - Software implementation of set associative cache
- Processor:
  - Instruction interpreter

≈ 1000 instructions per second.

Bottleneck:

- Processor simulation
- Can be solved with a simple trick

High level

Model

- Memory:
  - Forget it.
- Bus:
  - Simple queueing model
  - or Forget the bus completely (PRAM)
- Cache:
  - Probability model 95%/5%
  - 100% Hit (PRAM)
- Processor:
  - (Pseudo) Random accesses, or pick random pieces of traces.

How accurate?

- Still tells something about behaviour, especially the contention on critical sections in software

Simulation models

Basic decision about the type of the simulation model:

- Functional Simulator
  - Causal relationships only; functional simulation.
- Continuous time
  - Time flows continuously, as in physical problems.
- Constant time
  - Time flows with constant steps dictated by a clock, as in a computer.
- Discrete time
  - Time makes irregular steps, as in queueing problems.

Functional Simulators

- Execute functionality of a system.
- Ignores timing constraints.
  - Cache is as fast as a memory.
  - Addition is as fast as square root.
- Useful if you just want to check whether a system is functionally correct, and you aren't too bothered about timings.
- Models all causal constraints
  - ⇒ Observes flow of time
  - ⇒ Forces that X must happen after Y and Z have happened.

Implementing a Functional Simulator

- Every action will cause another action to happen.
- If there are no concurrent activities:
  - One function in a program can call another function to simulate the activity of another component of the system.
- If there are concurrent activities:
  - Strategy above may work, if you can execute concurrent activities sequentially.
  - Eg, main program asks ten processors to execute an instruction; one at a time
  - Otherwise; you will need processes and a clock to synchronise them.

Continuous time

Continuous time:

- take small steps δt to approximate time that should flow continuously.
- $\lim \delta t \downarrow 0$ gives better approximations.

Mostly used to "solve" differential equations that define the model

- Gas flow, Computational Fluid Dynamics, known as CFD, Weather predictions, chip simulations.

Not solved, mostly a crude approximation

- Solving the equation would give a precise answer (and an analytical model).

Continuous time example, a chip

```
Initialise charges
Repeat
  For every square micrometer
    Calculate new charge
  EndFor
  time = time + dt ;
Until next week
```

Observe:

- Execution Speed $= O\left(\frac{1}{\delta t}\right)$
- Accuracy $= O\left(\frac{1}{\delta t}\right)$

Accuracy is limited by numerical stability and step size

Implementing continuous time

In general two sets of state variables

- One contains current values
- One to calculate new values
- Copy new to old every iteration

```
Repeat
  Calculate New state from Old
  Time = Time + dt
  Copy New to Old
Until ...
```

Optimisations:

- Use two sets alternating (no need to copy)
- When there are no dependencies a single state suffices

Implementing Continuous Time

Difficulty of continuous time:

- Find the right approximations.

E.g.,

- Charge distributes through a plane:

```
for all x, y do
   ncharge[x,y] =
      c*dt*ocharge[x,y-1] + c*dt*ocharge[x,y+1] +
      c*dt*ocharge[x-1,y] + c*dt*ocharge[x+1,y] +
      (1-c*4)*dt*ocharge[x,y] ;
```

This is often an integration by discrete steps.

- Errors accumulate quickly!
- Take a good approximation (Euler).

Constant time steps

Used to simulate systems with a clock:

- Low level computer hardware.
- Small steps δt model *exact* behaviour.
- No choice in setting δt.

Example of circuit simulation:

```
Repeat
  Stabilise circuit
  time = time + dt
Until time > end_time
```

Observe:

- No choice in δt.
- Execution Speed $= O\left(\frac{1}{\delta t}\right)$
- Accuracy is constant; worthless if circuit cannot stabilise in δt

Implementing Constant Time

Constant time: trivial

- Like continuous time, but there are no circular dependencies
  - (– direct feedback in your machine!)
  - (– Might have a two clock scheme)
- Work out dependencies
- update from left to right.

```
Repeat
  For all phases i of the clock
    Do all operations for clock phase i
Until ...
```

⇒ You do not need to maintain the clock!

Discrete time

Take time step until next interesting point.

- Nothing happens between time steps, time steps differ in length.

Mostly used to simulate queueing systems:

- Cars at a traffic light,
- Messages in a network,
- Customers in a bank, example:

```
Repeat
  Wait till something happens
  If customer arrives,
    queuelength=queuelength+1
  If teller ready,
    queuelength=queuelength-1
Until time > end_time
```

Observe that this is an exact simulation

Virtual Time

All simulation models have something with time.

- May flow continuous or in discrete steps, but there is such a thing as time

A simulator runs in its own time space

⇒ Simulator implements a virtual clock.

- A virtual clock ticks, and defines the simulation time.
- It runs typically slower than a real clock (real time).

An example implementation of the clock:

```
long virtual_time ;
```

Increment it when necessary.

Implementing Discrete simulators

Simplest way: maintain an *event list*.

- Event: *what* is to happen *when*
- Sorted on Time of event
- Execute event on first element of list
- May generate future events in the list
- Time is defined by first element

We will first discuss the event list in detail

- Example:
- Bank with customers.

Event List: Time 11:12, Queue: 0

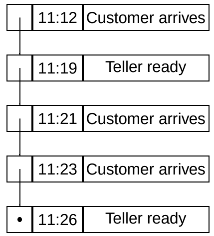
```
REPEAT
  Time = of head of list
  Take one from list
  IF customer THEN
    queuelength++
  If teller ready THEN
    queuelength--
    Time+7: teller ready
UNTIL list is empty.
```

Event List: Time 11:19, Queue: 1

```
REPEAT
  Time = of head of list
  Take one from list
  IF customer THEN
    queuelength++
  If teller ready THEN
    queuelength--
    Time+7: teller ready
UNTIL list is empty.
```

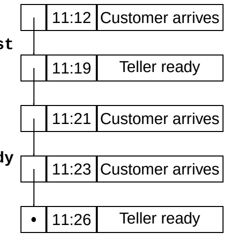
Event List: Time 11:21, Queue: 0

```
REPEAT
  Time = of head of list
  Take one from list
  IF customer THEN
    queuelength++
  If teller ready THEN
    queuelength--
    Time+7: teller ready
UNTIL list is empty.
```



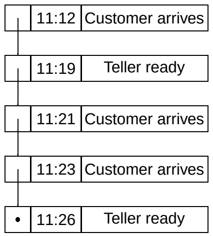
Event List: Time 11:23, Queue: 1

```
REPEAT
  Time = of head of list
  Take one from list
  IF customer THEN
    queuelength++
  If teller ready THEN
    queuelength--
    Time+7: teller ready
UNTIL list is empty.
```



Event List: Time 11:26, Queue: 2

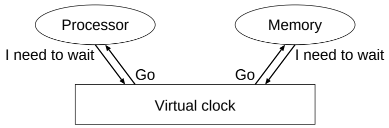
```
REPEAT
  Time = of head of list
  Take one from list
  IF customer THEN
    queuelength++
  If teller ready THEN
    queuelength--
    Time+7: teller ready
UNTIL list is empty.
```



Implementation can be painful

Writing it with a single loop becomes a pain when there are more than two components.

- Requires a concurrency model
- A central process manages the event list.
- Processes can add events to the event list.
- Events are executed and scheduled by passing them to a process.



The Virtual Clock

The virtual clock maintains the current time, event list and acts as the scheduler.

- Each process sends a request to the clock to be stopped for n timeunits.
- The virtual clock will send a message 'Ok, you have waited long enough' when n has passed.
- The virtual clock *schedules* the other processes.
  - The blobs are "Objects".
  - The square thing is a scheduler
  - The arrows indicate communication between objects and the scheduler.

Discrete Event Simulators in C

```
typedef struct event { struct event *next, int time ;
                process_context *c ; } event ;
static event *head = 0 ;

int main( void ) {
  event *tmp ;
  while( head != NULL ) {
    tmp = head ;
    now = tmp->time ;
    head = head->next ;
    (*tmp->c->func)( tmp->c ) ;
    free( tmp ) ;
  }
}
```


Conclusions

Advantage:

- You don't have to physically build the system

Before simulating:

- Think before you start:
  - What is the purpose (functionality? Timings?)
  - Can't you solve it analytically?
  - What level of detail is important?
  - What is the expected run time (number of computations)
  - What is the accuracy?
- All choices that must be made explicitly before making a simulator.
- Balance
  - Whole system is accurate as worst component

⇒ Makes no sense to have some very accurate parts (unless to convince yourself of the functionality)