A DESIGN FOR INTERCHANGABLE SIMULATION AND IMPLEMENTATION

Klaus Birkelund Jensen   Brian Vinter

August 25, 2015

Niels Bohr Institute

OUTLINE

1. **Introduction, background and motivation**
   Some context to understand why ISI was developed.
2. **The current state of storage simulation**
   What techniques are we using today, and what are the advantages and disadvantages?
3. **Our approach to interchangeability**
   What is interchangeability in simulation and implementation?
4. **Scalability results**
   What makes the ISI approach viable for large scale (storage) simulation?
5. **Summary**

INTRODUCTION AND MOTIVATION

MOTIVATION

To understand how large scientific data sets can be stored efficiently.

Efficiency in

- Performance
- Resources usage
- Locality
- **Energy consumption**

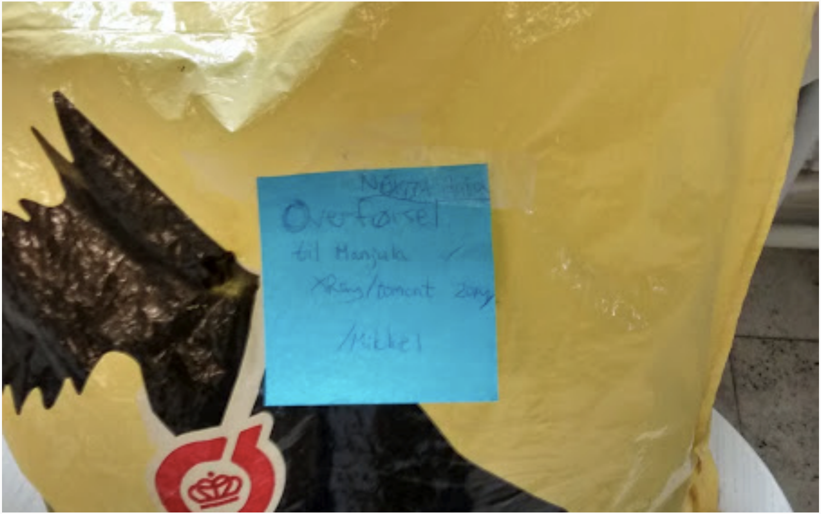
We focus on energy consumption.

ABOUT ME

Former systems operator at HPC/UCPH. Did storage and compute.

- Nordic T1 facility (storage & compute for ATLAS and ALICE)
- Multi PB disk, multi PB tape, thousands of compute cores.

Now, PhD student on the CINEMA project, working on storage techniques.


Overførsel
til Manjula
/Mikkel

THE PROBLEMS

The energy bill associated with storage is an ever larger part of the data center budget.

Most common technique to reduce energy consumption and maximize performance:

- Hierarchical Storage Management (HSM)
  The notion of managing data according to popularity, age, size etc. Move *passive* data to cheaper lower tier storage (usually tape).

faster ←

SSDs ⟷ HDDs ⟷ Tape

→ *cheaper*

THE PROBLEMS

HSM uses many reasonably good techniques including (but not limited to):

- LRU-caching and aging
- Manual tagging of data (i.e. “please do NOT move my data!”).
- Generally, *on-demand* retrieval. No prediction.

THE PROBLEMS

HSM is too general to efficiently store what we define as *known data sets*.

We focus on scientific and industrial tomography imaging.

Imaging data exhibits known workloads and structure.

We should acknowledge and exploit that.

THE PROBLEMS

In the data center, durability and reliability is most commonly provided by large RAID systems, but *erasure codes* are rapidly gaining traction.

In RAID, all drives must spin simultaneusly. There are solutions to this in the literature, including:

- *Power-aware* RAID (or *gear shifting*).
- *Intelligent* data placement (e.g. locality optimized).

They are all general in nature.

The principle of "optimizing for the common case" has always been a good strategy.

"This data was just used —
let's keep it around for a week... *or so*"

The principle of “optimizing for the common case” has always been a good strategy.

“This data was just used —
let’s keep it around for a week… *or so*”

But, the *common case* isn’t at all *common* when working with well-defined scientific data.

The principle of “optimizing for the common case” has always been a good strategy.

“This data was just used —
let’s keep it around for a week… *or so*”

But, the *common case* isn’t at all *common* when working with well-defined scientific data.

“This data has just been acquired —
the physicist won’t use it for months… *if ever*”

SOLUTIONS

What is possible if we exploit what is *known*?

- Raw data can be moved directly to tape
- Stream filtering

But how to quantify any possible benefits?

Simulation of storage hierarchies, workloads and data acquisition and consumption.

BUILDING A STORAGE SYSTEM

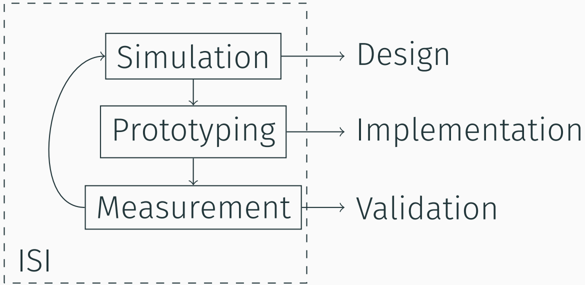
Developing a large-scale storage system where the design isn’t exactly known in advance, could go something like this:

1. *Simulate* a model and identify a design.
2. *Implement* a prototype from the design.
3. *Measure* the prototype and *validate* it and the model against predictions.
4. *Repeat.* Feed the results of the validation back into the simulator and/or model and repeat from step 1.

The process is sound, but can we improve it?

Interchangeability of simulation and implementation

Eliminating the *simulation–prototyping–measure* cycle.



STORAGE SIMULATION

Simulate the system model using *Discrete Event Simulation* (DES).

A DES is a priority queue of events, handled sequentially. Each event has a time stamp, updates the model and adds new events to the queue when handled.

Main loop of a DES.

Algorithm 1 Discrete Event Simulation

procedure DES-LOOP(Q)
    while $Q \neq \emptyset$ **do**
        $e \leftarrow$ DEQUEUE(Q)
        $T \leftarrow$ CLOCK(e) ▷ update world clock
        PROCESS(e) ▷ process event and add new
    end while
end procedure

An *event* is processed by a handler. Typically a huge function with a single switch-statement.

Parallel DES (PDES), generalizes this by allowing multiple processes to have a local priority queue.

ROSS (Rensselaer's Optimistic Simulation System) is an *optimistic* PDES.

- Extremely high performance
- Runs on millions of cores
- Relies on *Reverse Computation*

ROSS (Rensselaer's Optimistic Simulation System) is an *optimistic* PDES.

- Extremely high performance
- Runs on millions of cores
- Relies on *Reverse Computation*

In summary: a savage beast

INTERCHANGEABLE SIMULATION AND IMPLEMENTATION

Model the system components as the individual processes they are.

The process logic directly implements a prototype.

Requires an environment supporting millions of independently communicating processes:

- Language based: Go, Erlang, occam-π
- Library based: ZeroMQ

Substantial reduction in time spent going from modeling to prototype.

INTERCHANGEABLE SIMULATION AND IMPLEMENTATION

Do measurement at the same points that does simulation.

No (explicit) priority queues. Communication is done directly between interacting entities.

Communicate instead of dictating events.

DISCRETE VS. REAL-TIME

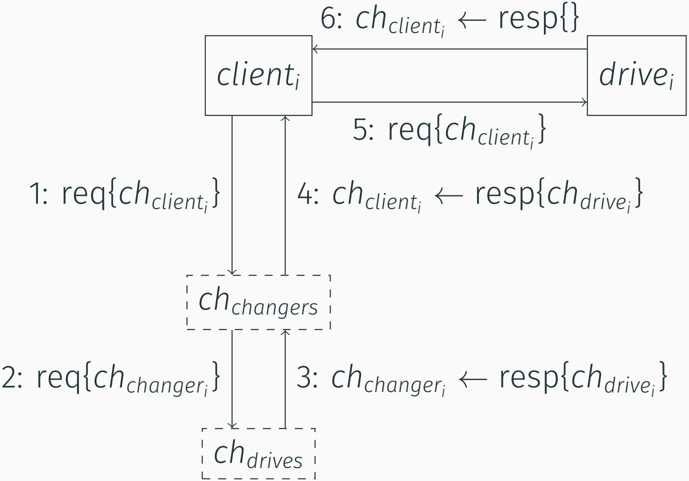
Simulated durations are calculated in the processes that does the work.

Interchangeability allows components to be swapped around and possibly mixing discrete time for some components with real-time for other components.

SIMULATING A RATHER HUGE TAPE LIBRARY

- 90 days of constant I/O
- Three types of entities: clients, tape drives and changers
- Fixed ratio of 16 : 8 : 1
- Up to 250,000 processes simulated.

I/O COMMUNICATION PATH



GO

Open source concurrent programming language, created and primarily developed by Google.

Designed to be highly productive and easy to learn.

Follows the *principle of least surprise*.

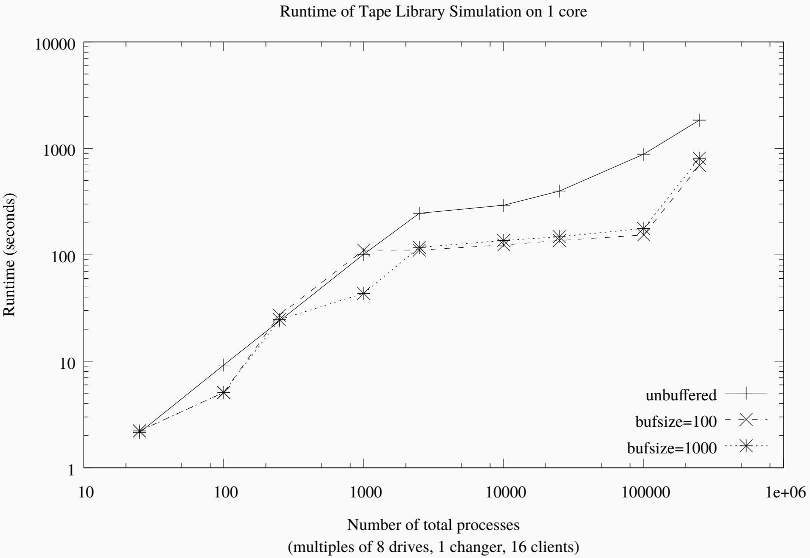
Key features:

- CSP and π-calculus style channels and processes as low level language features.
- Garbage-collected
- Compiled
- Statically typed

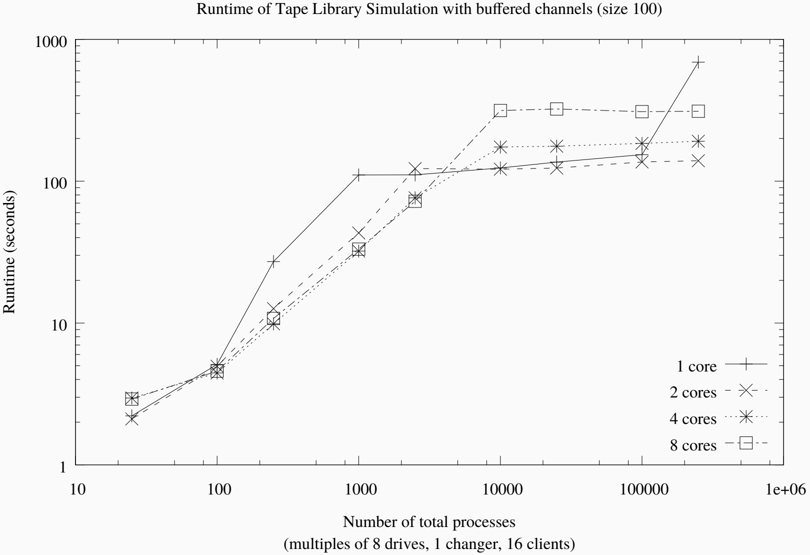
```go
func client(lib *library) {
  ch := make(chan response, *chanBufSize)

  for {
    lib.changers <- request{mount, ch, clock}

    resp = <-ch
    clock = clock.Add(resp.t)

    waitTime += resp.t

    t += resp.t

    resp.ch <- request{read, ch, clock}

    resp = <-ch
    clock = clock.Add(resp.t)
    t += resp.t
    ioTime += resp.t
  }
}
```

SCALABILITY RESULTS

RESULTS (SEQUENTIAL)



RESULTS (PARALLEL)



UNBUFFERED CHANNELS

Processes	1 core	2 cores	4 cores	8 cores
25	**2.14**	4.14	4.04	4.00
100	9.22	**4.96**	5.82	6.15
250	23.90	10.77	**10.71**	13.39
1000	101.09	37.75	**32.00**	37.77
2500	245.64	80.37	**70.10**	75.45
10000	292.40	365.42	585.33	**243.83**
25000	**397.34**	419.40	652.45	528.45
100000	881.00	**726.77**	902.13	1788.21
250000	1839.43	**1307.85**	1392.19	3671.10

BUFFERED CHANNELS

Processes	1 core	2 cores	4 cores	8 cores
25	2.22	**2.11**	2.96	2.91
100	5.11	4.96	**4.44**	4.58
250	27.09	12.63	**9.86**	10.78
1000	110.72	43.19	**32.16**	33.19
2500	110.83	122.83	76.30	**72.19**
10000	123.91	**121.59**	174.01	315.17
25000	136.47	**123.72**	176.50	322.98
100000	153.77	**136.59**	184.69	309.25
250000	691.15	**139.34**	191.30	311.50

SUMMARY AND FUTURE WORK

- Rapid transition from simulation/modeling to prototype
- Communicate instead of dictating events
- No reverse computation
- Scales well with at least Go

- Further refinement and packageing of the ISI patterns.
- Look into locality management of Goroutines.

Thank you

Questions?