

The occam- π programming language and RMoX operating-system

a brief look of occam and CSP

occam- π and its applications

the rmoX operating-system

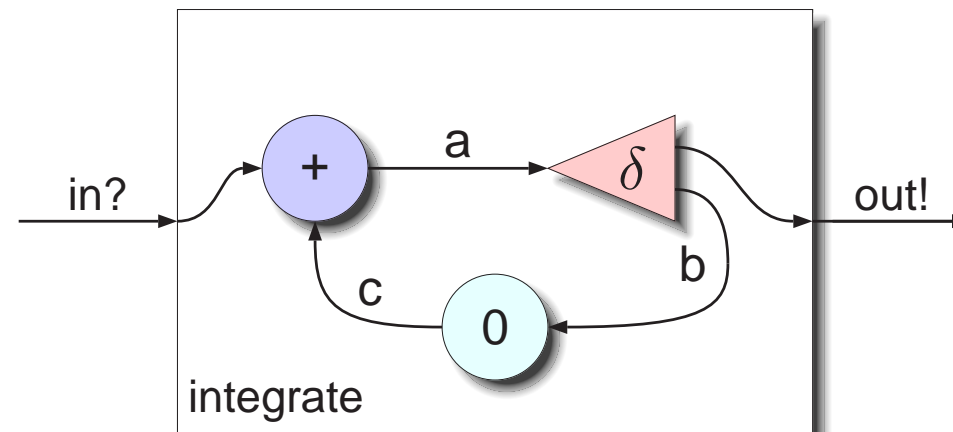


Fred Barnes, Systems Research Group
Computing Laboratory, University of Kent
F.R.M.Barnes@kent.ac.uk



The Communicating Processes Paradigm

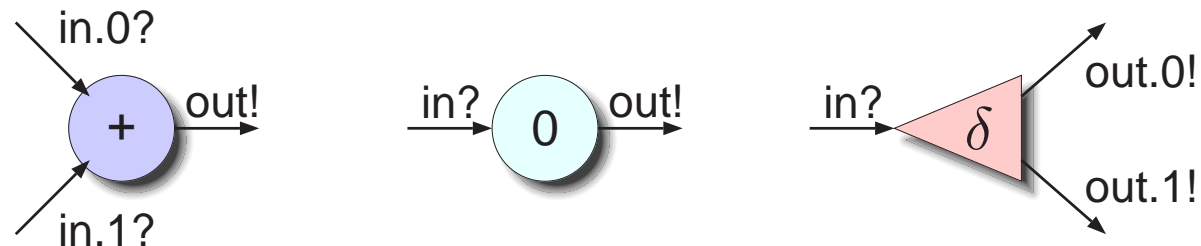
- Systems are built from layered networks of communicating parallel processes



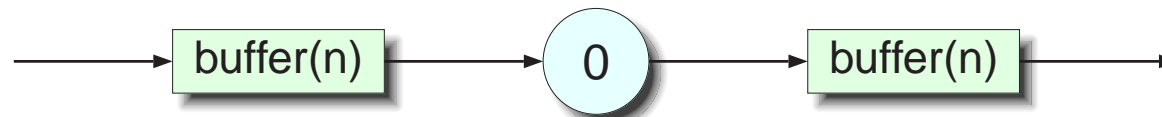
- Synchronous point-to-point communication via 'channels'
- The three sub-components here are 'plus', 'prefix' and 'delta'
 - that could also be process networks
- Implementation in occam; semantics from Hoare's CSP

The Communicating Processes Paradigm

- Individual processes are completely isolated
 - interacting with the environment only through channels visible in their interfaces



- The synchronous nature of channel communication means that different implementations may behave differently w.r.t. the environment
 - asynchronous behaviour is possible, however:

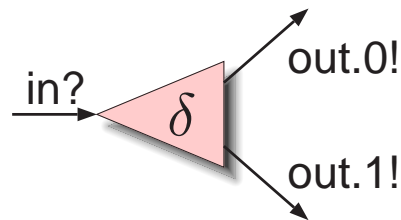


occam and CSP

- ▶ The occam language provides for 'clean' implementations of such processes
 - developed by David May (and others) at Inmos [1] (1983), last commercial revision was occam2.1 [2] (1995)
 - strict parallel-usage and aliasing checks give strong safety guarantees — i.e. freedom from race-hazard errors
 - remaining issues include: deadlock, livelock and starvation
- ▶ CSP is the underlying process algebra, used to formally reason about the behaviour of occam processes
 - developed by Tony Hoare [3, 4], standard text (currently) by Bill Roscoe [5]
- ▶ The mapping between CSP and occam is not an exact fit
 - but is sufficiently complete for reasoning about occam programs

occam and CSP

- As an example, consider the 'delta' process:



serial implementation:

```

WHILE TRUE
  INT x:
  SEQ
    in ? x
    out.0 ! x
    out.1 ! x
  
```

parallel implementation:

```

WHILE TRUE
  INT x:
  SEQ
    in ? x
    PAR
      out.0 ! x
      out.1 ! x
  
```

- May prefer the serial version for efficiency, but it alters the behaviour of the process w.r.t. the environment
- can reason about these behaviours in CSP, e.g. to know if/how a process will deadlock when placed in a certain environment

Failure, Divergence and Refinement

- ▶ Concepts from CSP that characterise processes by:
 - failures: sets of traces and environments that result in deadlock
 - divergences: sets of traces that could lead to livelock

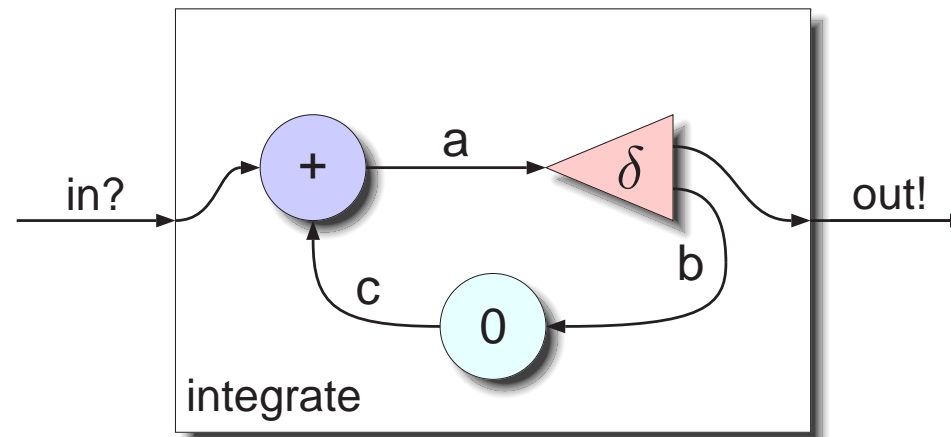
- ▶ Refinement is a relation between processes, expressed as:

$$P \sqsubseteq_{FD} Q \Leftrightarrow (failures_{\perp}(Q) \subseteq failures_{\perp}(P)) \wedge \\ (divergences(Q) \subseteq divergences(P))$$

- states that Q is more deterministic than P
- practical application: that Q may be substituted for P
- ▶ Turns out that the two versions of 'delta' have no refinement relation

Building Process networks

- The implementation of process networks such as ‘integrate’ is trivial:



```
PROC integrate (CHAN INT in?, out!)
```

```
  CHAN INT a, b, c:
```

```
  PAR
```

```
    plus (in?, c?, a!)
```

```
    prefix (0, b?, c!)
```

```
    delta (a?, b!, out!)
```

```
:
```

occam and occam- π

- Occam can express a wide range of behaviours, but some aspects are lacking
 - largely as a result of occam's original application for the Transputer
 - dynamic process creation
 - channel mobility
 - shared channels

- Primary aims of occam- π :
 - to provide the above features at a language level
 - not break the existing good safety guarantees of occam
 - make the language “programmer friendly”, i.e. something that people will actually want to use

occam- π Overview

- ▶ Extending the classical occam language with ideas of mobility and dynamic network reconfiguration, ideas from Milner's π -calculus [6]
 - we have found ways of implementing these extensions that involve significantly less resource overhead than imposed by the higher-level — but less structured, information and non-compositional — concurrency primitives of existing languages (such as Java) or libraries (such as POSIX threads)
- ▶ As a result, we can run applications with the order of millions of concurrent processes on modestly powered PCs
 - we have plans to extend the system, without sacrifice of too much efficiency and none of logic, to simple clusters of workstations, wider networks such as the Grid and small embedded devices

occam- π Overview

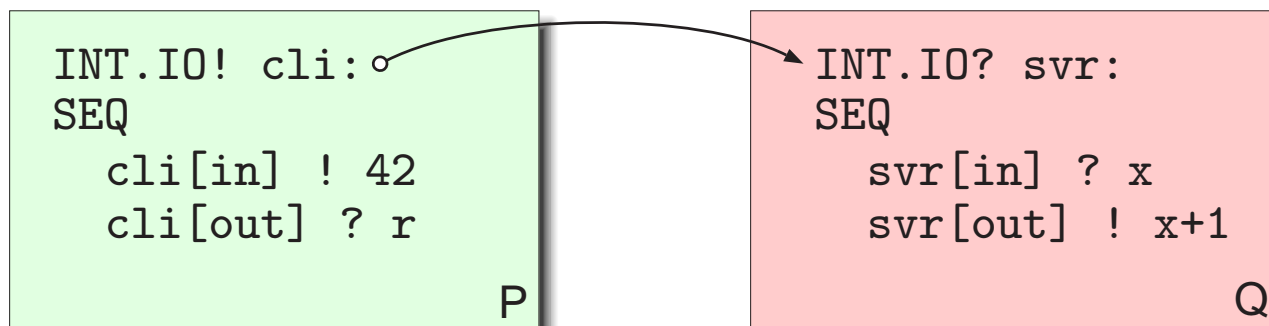
- ▶ In the interests of proveability, we have been careful to preserve the distinction between the original point-to-point synchronised communication of occam and the dynamic asynchronous multiplexed communication of the π -calculus; in this, we have been prepared to sacrifice the elegant sparsity of the π -calculus
 - we conjecture that the extra complexity and discipline introduced will make the task of developing, proving and maintaining concurrent and distributed programs easier

- ▶ occam- π features:
 - mobile data, channels and processes; dynamic process creation
 - shared channels, channel bundles, recursion, no race-hazards, no garbage, protocol inheritance, extended rendezvous, process priority, ...

Mobile Channel Bundles

- Defined as 'client' and 'server' ends of a "mobile channel-type", e.g.:

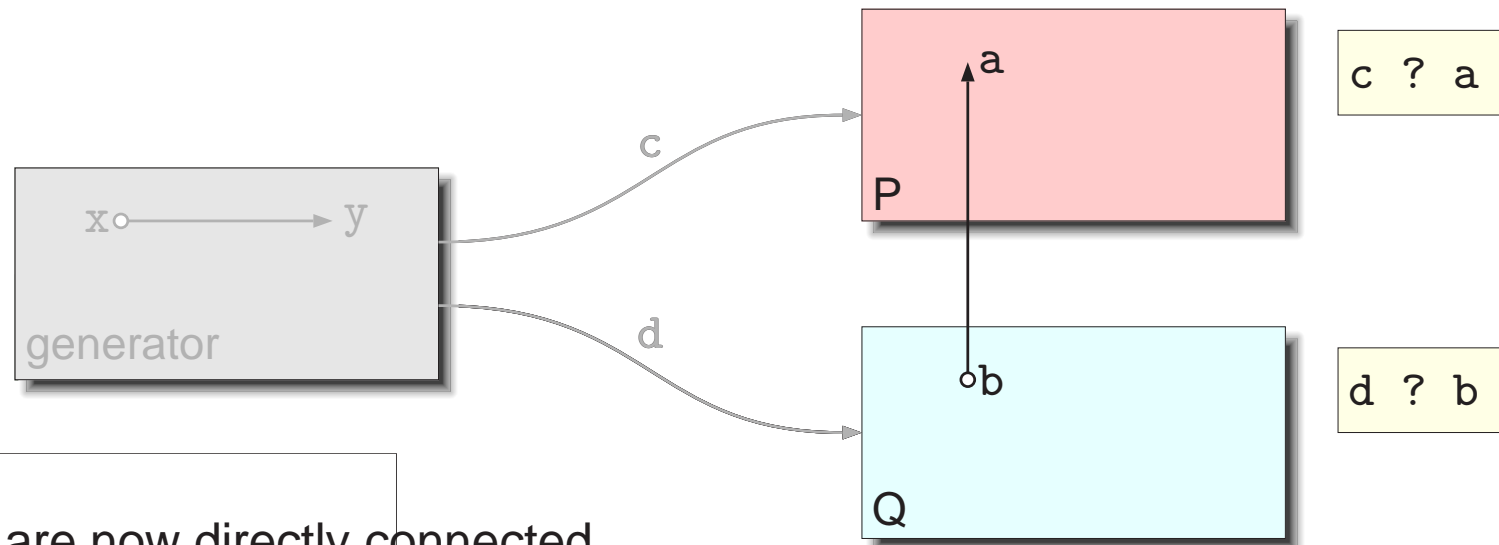
```
CHAN TYPE INT.IO
  MOBILE RECORD
    CHAN INT in?:
    CHAN INT out!:
:
```



- Directions specified in the type are server-relative
 - and may carry data both ways
- Allowing data to flow both ways is merely convenience — can get (mostly) the same effect using many individual mobile channels
- Grouping channels together, for related use, makes good sense however

Communicating Mobile Channels

- ▶ The main use of mobile channel bundles is to support the run-time reconfiguration of process networks



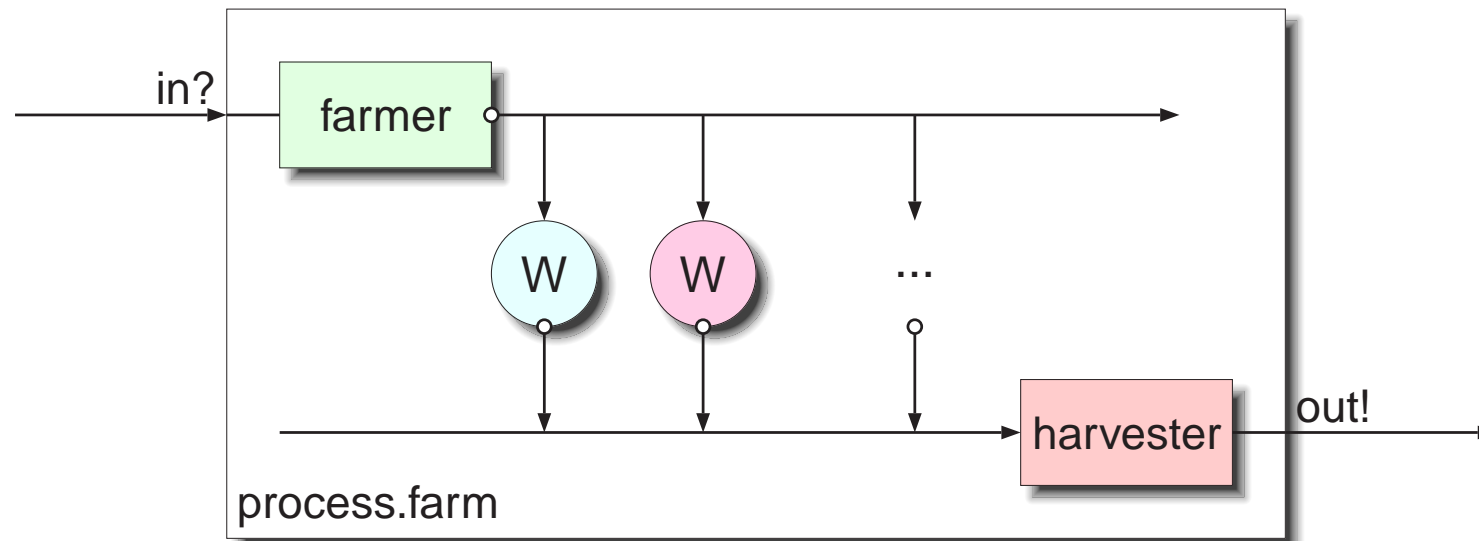
- ▶ P and Q are now directly connected
- ▶ Process networks may be arbitrarily reconfigured using mobile channels, but still only reconfiguring *static* networks
 - dynamic process creation makes this much more interesting

Dynamic Process Creation

- ▶ Three mechanisms provided in *occam- π* :
 - self-recursive procedures
 - *n*-replicated parallel
 - asynchronous process invocation (fork)
- ▶ All three can be modelled in CSP, but require dynamic memory for implementation
- ▶ The 'fork' is examined here
 - essentially a procedure instance that runs in parallel with the invoking process
 - parameter passing is strictly uni-directional and uses a communication semantics

Dynamic Process Creation

- ▶ The only way to communicate with a forked process is to pass to it a mobile channel-end
 - ordinary channels are non-communicable



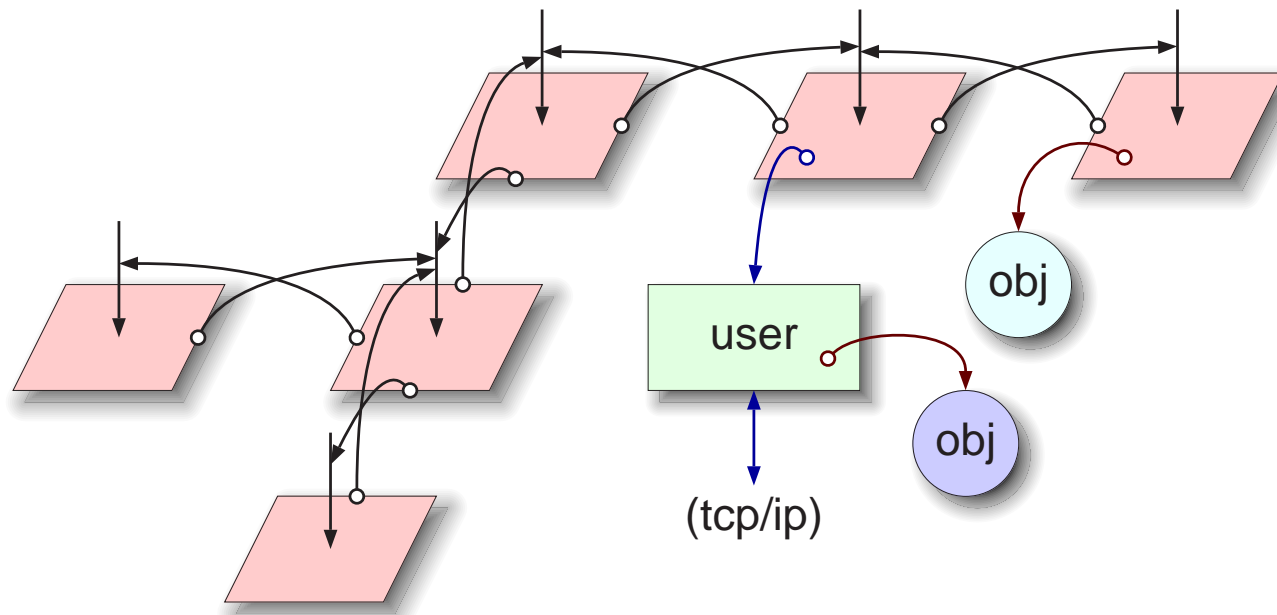
- ▶ Worker processes are created on demand

Dynamic Process Creation

- ▶ Process farms are one application of dynamic process creation, that turns out to be useful for many things
 - e.g. the occam(- π) web-server (<http://wotug.kent.ac.uk/ocweb/>)
 - more generally, almost any server that must handle multiple clients concurrently

- ▶ A more interesting example is the “occam- π adventure”
 - an interactive text-based multi-user game (MUD)
 - built as an interconnected matrix of ‘rooms’
 - ‘objects’ and ‘users’ connect to rooms using mobile channels, also used to link rooms
 - exercises channel mobility

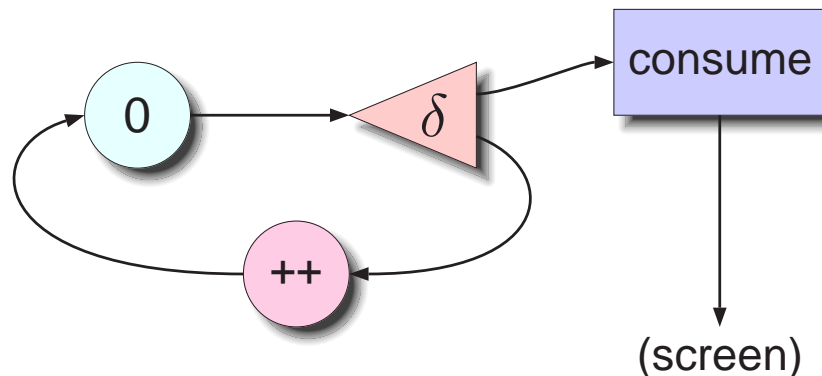
Dynamic Process Creation



- Game matrix is constructed dynamically
 - may add, remove and re-arrange rooms whilst running
- User processes connected to the room the user is in
- Objects lie around in rooms or are held by users

Performance

- Various benchmark programs are used to measure the performance of the KRoC/occam- π system
 - micro-benchmarks for measuring minimum overheads
 - a communicating process pairs benchmark for more realistic overheads
- Commstime (integer communication benchmark):

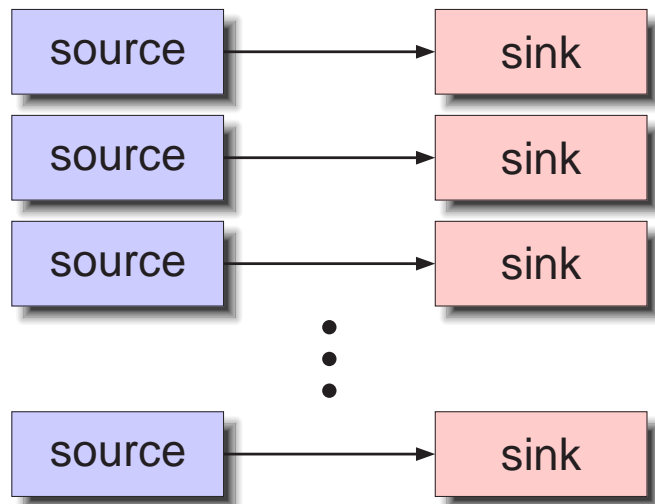


hardware	cxt-sw (ns)	start-stop (ns)
P3 800	60	30
P4 3400	30	0
P4 3200	15	4

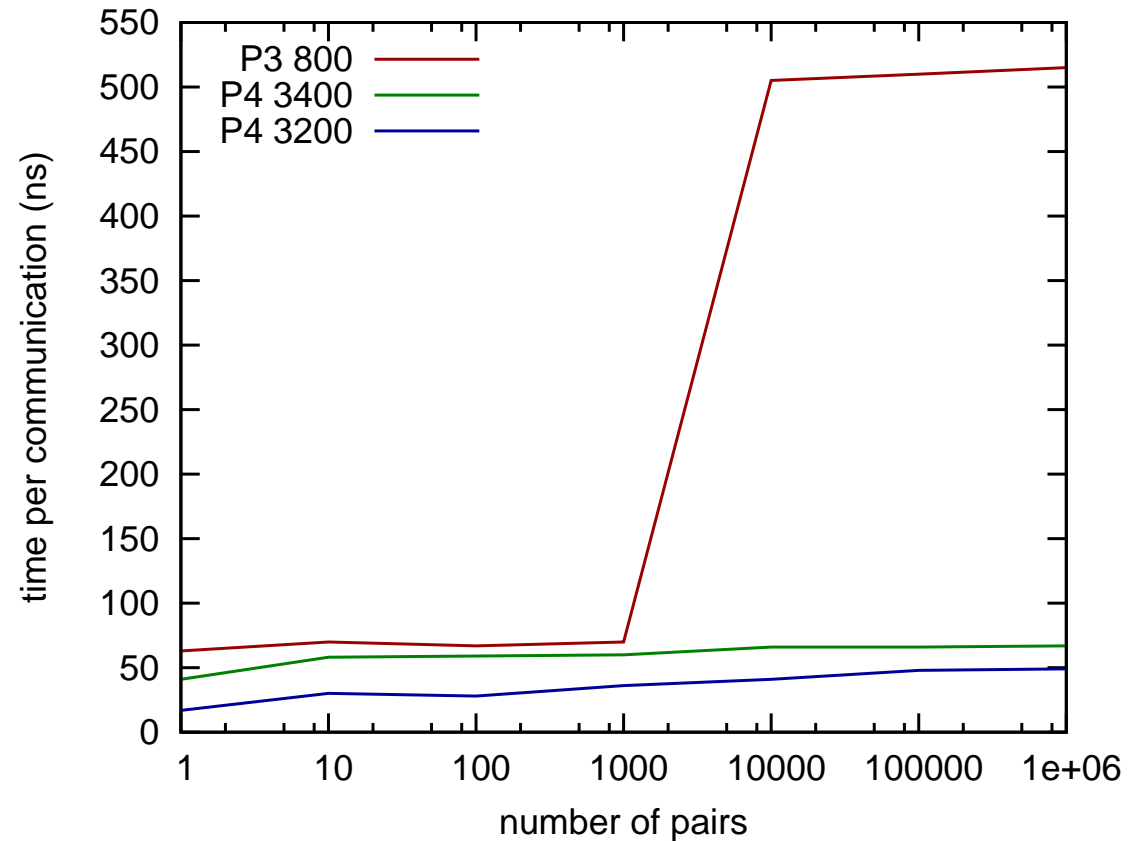
(3.2 GHz P4 has a slightly different internal architecture)

Performance

► Communicating process pairs benchmark:



up to 1,000,000 pairs



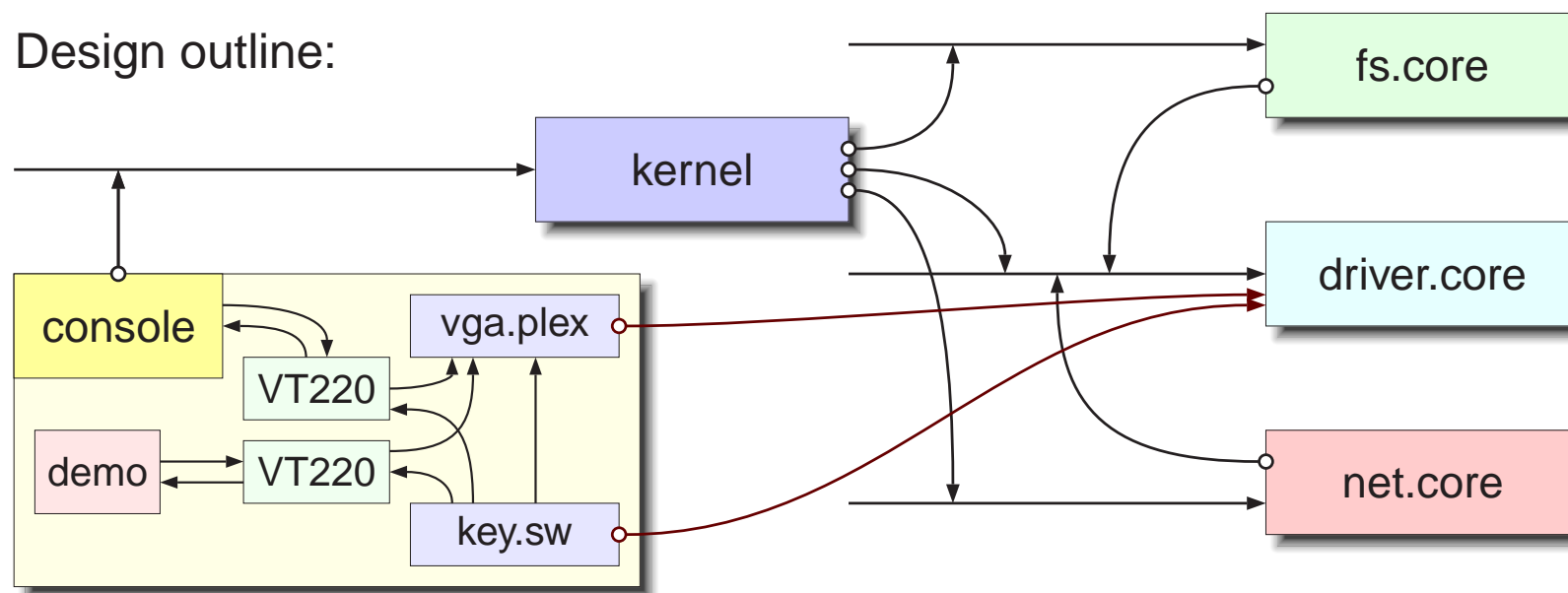
The RMoX Operating System

- On the whole, operating-systems are large, complex concurrent systems
- Real world operating-systems, even without attempting to exploit multiprocessor hardware, suffer from concurrency-related problems:
 - subtle race-hazard errors, that in some cases lead to serious security problems — e.g. local and remote exploits
 - or concurrency is “locked-down” to the point where it damages performance
- Multiprocessor architectures (typically SMP) just scale these problems
- The programming language, in most cases C, does not help:
 - synchronisation and locking must be programmed correctly, compiler cannot police usage
 - lack of appropriate encapsulation — e.g. a device-driver may inadvertently (or deliberately) interfere with another part of the system

The RMoX Operating System

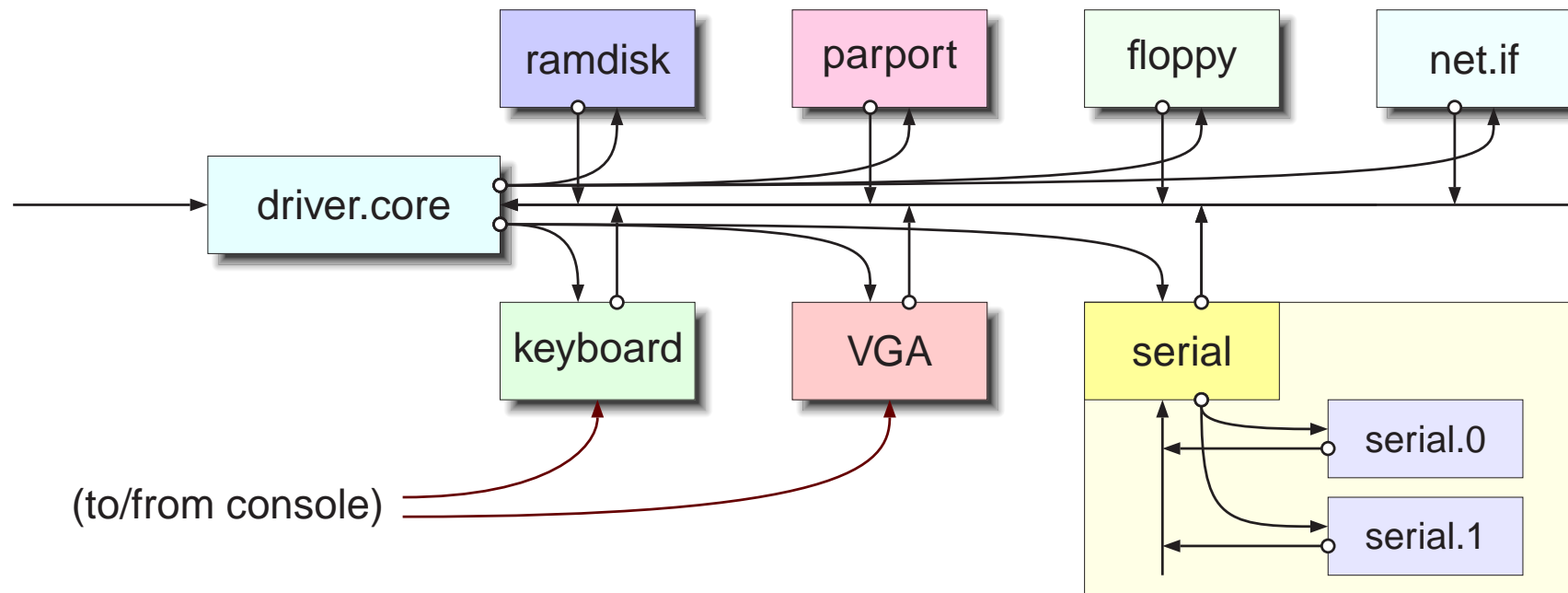
- Theory is: construct an operating system using occam- π and it will be:
- scalable: from small embedded devices to distributed multicomputers
 - safe: freedom from race-hazard and aliasing errors (incl. array-bounds, etc.)
 - fast: due to the extremely light concurrency overheads

- Design outline:



The RMoX Operating System

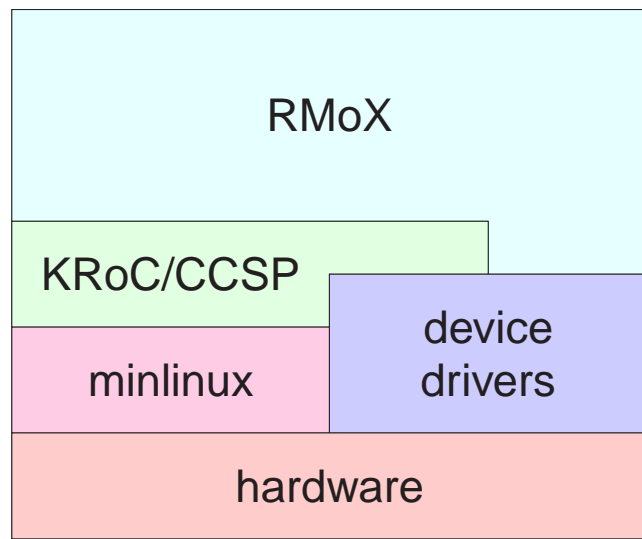
- The various “core” components utilise internal concurrency



- Drivers themselves may be concurrent internally
- Some drivers mostly complete, others under construction

The RMoX Operating System

► Putting it together:



► a stripped-down Linux is used, providing:

- boot-loader
- relatively simple access to hardware and interrupts
- basic memory management

► RMoX drivers use hardware directly

► CCSP schedules occam- π processes

► With suitable interfacing, RMoX will also run in a Linux environment (user-mode RMoX)

► We have an experimental version that runs directly on hardware

- running with Linux provides various useful features for development

Conclusions and Future Work

- Language not quite finished — but mostly complete
 - need a formal semantics for the behaviour of mobile channels
- For what is supported, performance is good
- Compiler and run-time system being developed
 - particularly for other platforms/architectures
 - investigating support for mostly transparent network distribution
- Also in development are demonstrator applications (e.g. servers) and graphics libraries
- Recently funded for a feasibility study into modelling and theories of nanite assemblies — gigabit switched 32-node 3.2 GHz P4 cluster ready to go

Acknowledgements

► People involved directly in this work:

- Fred Barnes, Peter Welch: principal investigators
- Christian Jacobsen, Adam Sampson, Joseph Sheridan: research students

► Additional thanks to:

- Damian Dimmich, Tony Hoare, Mario Schweigler, David Wood, Jim Woodcock
- The WoTUG/CPA and occam-com communities
- Various people around the world who have provided feedback and bug-reports for KRoC

References

- [1] David May. OCCAM. *ACM SIGPLAN Notices*, 18(4):69–79, April 1983.
- [2] Inmos Limited. occam 2.1 Reference Manual. Technical report, Inmos Limited, May 1995. Available at: <http://wotug.org/occam/>.
- [3] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.
- [4] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985. ISBN: 0-13-153271-5.
- [5] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997. ISBN: 0-13-674409-5.
- [6] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes – parts I and II. *Journal of Information and Computation*, 100:1–77, 1992. Available as technical report: ECS-LFCS-89-85/86, University of Edinburgh, UK.