Clouds, Things and Robots:

... the transputer revisited

David May

Background 1975-85

Ideas leading to CSP, occam and transputers originated in the UK around 1975.

1978: CSP published, Inmos founded
1983: occam launched
1984: transputer announced
1985: transputer launched and in volume production

This introduced the idea of a communicating computer - *transputer* - as a system component

Key idea was to provide a higher level of abstraction in system design - along with a design formalism and programming language

CSP, Occam and Concurrency

Sequence, Parallel, Alternative

Channels, communication using message passing, timers

Parallel processes, parallel assignments and message passing

Secure - disjointness checks and synchronised communication

Scheduling Invariance - *arbitrary interleaving* model

Initially used for software and programming transputers; later used for hardware synthesis of microcoded engines, FPGA designs and asynchronous systems

Transputers and occam

Idea of running multiple *processes* on each processor - enabling cost/performance tradeoff

Processes as *virtual processors*

Event-driven processing

Secure - runtime error containment

Language and Processor Architecture designed *together*

Distributed implementation designed *first*

Transputer overview

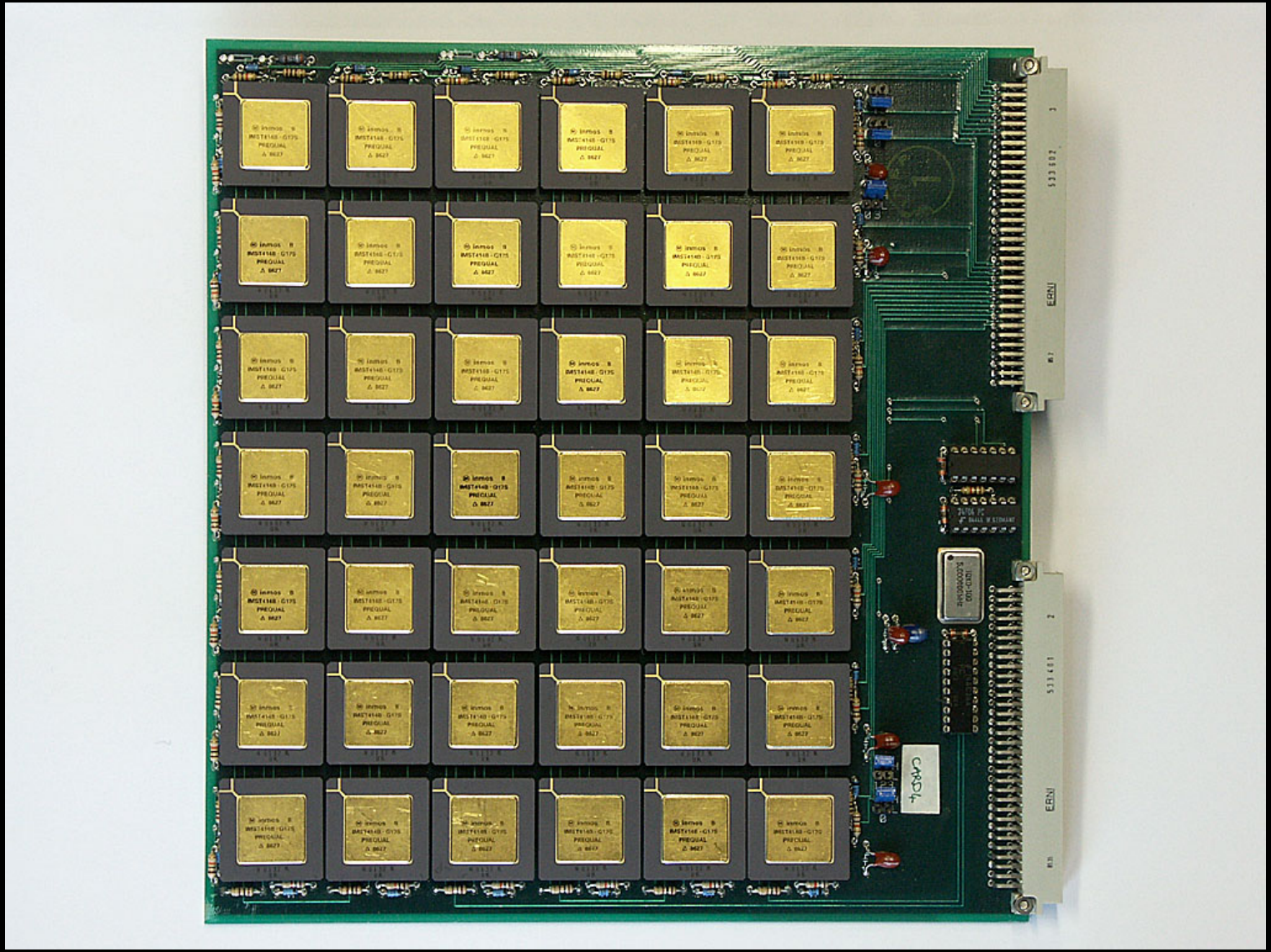
VLSI computer integrating 4K bytes of memory, processor and point-to-point communications links

First computer to integrate a large(!) memory with a processor

First computer to provide direct interprocessor communication

Integration of process scheduling and communication following CSP (occam) using microcode

Transputers 1986



What did we learn?

We found out how to

- support fast process scheduling (about 10 processor cycles)
- support fast interprocess and interprocessor communication ($<$ 2 microseconds)
- make concurrent system design and programming easy - using lots of processes
- implement specialised concurrent applications (graphics, databases, real-time control, scientific computing)

and we made some progress towards general purpose concurrent computing using recongfigurablity and high-speed interconnects

What did we learn?

We also found that

- we needed more memory (4K bytes not enough!)
- we needed efficient system-wide message passing
- we needed support for rapid generation of parallel computations

- 1980s embedded systems didn't need 32-bit processors or multiple processors
- most programmers didn't understand concurrency

General Purpose Concurrency

Key architectural ideas emerged:

- scale interconnect throughput with processing throughput
- hide latency with process scheduling (multi-threading)

Potentially these remove the need to design specialised processors and interconnects

Emerging software patterns: Task Farms, Pipelines, Data Parallelism ...

But no easy way to build subroutines and libraries!

Meiko and EPCC 1987

Built in Bristol using Inmos Transputers designed in Bristol

Routers

We built the first VLSI *router* - a 32 channel packet switch

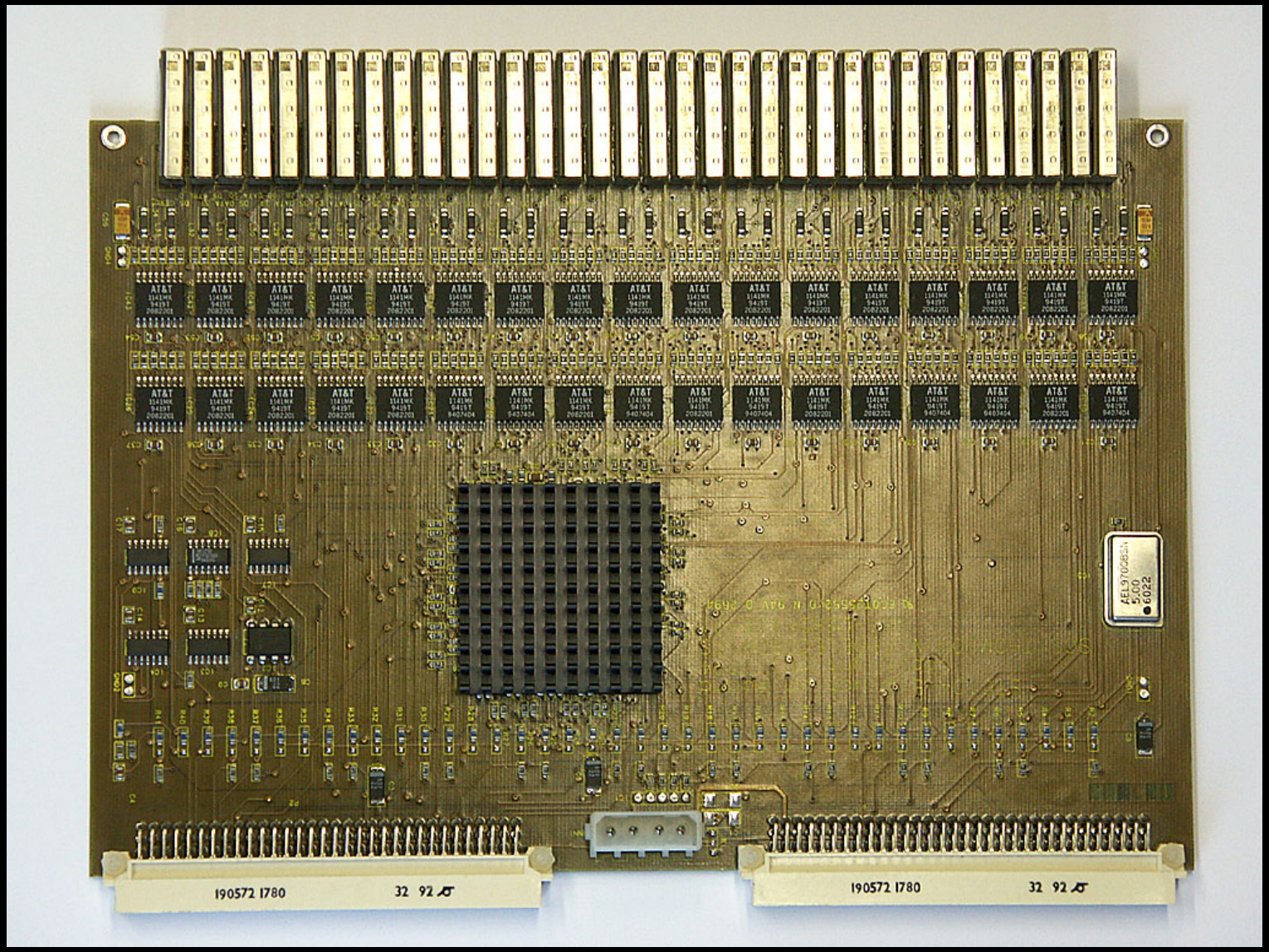
It was designed as a component for interconnection networks allowing latency and throughput to be matched to applications

Note that - for scaling - capacity grows as $p \times log(p)$; latency as $log(p)$

Low latency at low load is important for initiating processing; low (bounded) latency at high load is important for latency hiding

Network structure and routing algorithms must be designed together to minimise congestion (hypercubes, randomisation ...)

C104 router 1990



Bristol's embedded processors

Inmos - STMicroelectronics:	transputers, Chameleon, ST20, ST200
Infineon:	Tricore
Element14 - Broadcom:	Firepath
Pixelfusion - Clearspeed:	GPU
Picochip:	Picoarray
XMOS:	XCore
Graphcore:	IPU ... ?

The last 20 years

We have known how to do general purpose parallel computing since 1990

But the explosive growth of PCs has taken us in a different direction

Implementing Moore's 'law' is only possible with exponential market growth

But there has been exponential market growth so we have spent 20 years improving on 1980s technologies

... most superscalar techniques date from the 1960s (IBM 360/91)!

Clouds, Things and Robots

PCs are disappearing; we are now doing clouds, things and robots

We need scalable multiprocessors - on-chip and in-the-large

We need lots of interprocessor communication

We need lots of responsive input-output

We need to focus on software efficiency - stop relying on Moore's 'law'

We need predictability - of timing, power and energy

Why timing matters

In parallel processors, there are many potential sources of timing jitter - in processors, caches, memory and interconnect

Have to manage latencies in communication
... can’t manage latency if you don’t know how big it is

Have to provide synchronisation (barriers) in control flow or data flow
... these are delayed by the *slowest* participant

May need to use buffers to maintain rates when processing streams
... can’t determine buffer sizes if you don’t know the variance in timing

Time-determinism

Many parallel programs rely on synchronisation (barriers, reductions)

Execution must be time-deterministic - but many systems aren't!

p: probability of no unexpected delay when executing program P

Suppose n copies of P in execute in parallel, then synchronise

Probability that the synchronisation will not be delayed = p^n

- For n = 100 and p = 0.99, p^n = 0.37
- For n = 1000 and p = 0.99, p^n = 0.00004

Contention in interconnection networks gives rise to similar problems

The ‘Random Access’ Memory

Programmers think of computers as sequential machines accessing a random access memory - but in practice there is a complex hierarchy of caches

Let’s explore the effects of the memory hierarchy, and of *superscalar* execution - executing lots of instructions at the same time

Let’s write three similar programs, all of which sum the contents of a large array initialised by

```
for (i=0; i < maxi; i++)
  for (j=0; j < maxj; j++)
    a[i][j] = i;
```

Three programs (1)

```
int prog1()
{ int i;
  int j;
  int sum;
  sum = 0;
  for (i=0; i < maxi; i++)
  { j = 0;
    while (j < maxj)
    { sum = sum + a[i][j];
      j = j + 1;
    }
}
```

Three programs (2)

```
int prog2()
{ int i;
  int j;
  int sum;
  sum = 0;
  for (j=0; j < maxj; j++)
  { i = 0;
    while (i < maxi)
    { sum = sum + a[i][j];
      i = i + 1;
    }
}
```

Three programs (3)

```
int prog3()
{ int i;
  int j;
  int sum;
  sum = 0;
  for (j=0; j < maxj; j++)
  { i = 0;
    while (i < maxi)
    { sum = sum + a[i][j];
      i = a[i][j] + 1;
    }
}
```

Three programs - performance

In 1980, these three programs would have had the same performance

Today, program 3 is over 100 times slower than program 1

The processors rely on both *latency minimisation* and *latency hiding*
... they rely on the program containing ‘parallelism’

But no-one would think of these as ‘parallel’ programs

Parallel *composition* is an essential programming tool: need to minimise *variance* in runtime, not *average* runtime

Efficient processors for parallel computing are not the same as efficient processors for sequential computing

Sharing memory

Sharing the memory hierarchy makes things worse

It increases contention in the caches - associativity needs to scale with number of threads or cores

Cache coherency protocols introduce unexpected effects

And shared memory makes programming difficult

- non-deterministic behaviour
- non-deterministic timing

... a programmer has no idea what's happening 'behind the scenes'

Communication, input and output

Pipelined superscalar processors take a long time to respond

... so communication has to be offloaded to hardware devices
... along with input and output

The processor still has to interact with the hardware devices

But what if a programmed response to an external event is needed?

... and what about the numerous short messages in real programs?

High communication latency limits parallelism and makes programming difficult

Software and Algorithms

We need to architectures that are easy to use, so that we can focus on *software efficiency*

Efficient software can add a lot of value by increasing performance *and* energy-efficiency

And big data sets bring big opportunities for better software and algorithms:

Reducing the number of operations from $N \times N$ to $N \times log(N)$ has a dramatic effect when N is large

... for $N = 30\,billion$ this change is as good as 50 years of technology improvements!

Architecture revisited

General purpose parallelism - built on message passing

Predictable, low-latency communications
... short message latency < 100 instructions

The processors have to keep up with the interconnect and they don't have to be complex - just predictable

Programmable interfacing (horizontal execution; not pipelining)

Event-driven processing

Avoid heterogeneous *architecture* - although it's fine for the architecture to enable heterogeneous *implementations*

Universality

Turing: a Universal Machine can emulate any specialised machine

For Random Access Machines, the emulation overhead is constant

Is there an equivalent Universal Parallel Machine?

A key component is a Universal Network

Idea: A Universal Processor is an infinite network of finite processors

Another Idea: Use a non-blocking network

Universal Parallel Processors

Universal networks emulate specialised networks

Universal processors emulate specialised processors

Networks must have scalable throughput (bisection bandwidth)

Networks must have low latency ($log(p)$) under continuous load

Use network pipelining for continuous (stream) processing: optimal

Use *latency hiding* otherwise: optimal with $log(p)$ 'excess' parallelism

Program Structures

Parallel Random Access Machines

Data Parallelism; Systolic Arrays

Directed Dataflow Graphs

Task Farms and Server Farms

Sequential programs(!)

Recursive Embedding of any of the above

Communication Patterns

Communication and data access patterns are often known, especially in embedded processing (but also in HPC)

Communication can often be implemented as a series of permutation routing operations between known endpoints

For known patterns, compilers can allocate processors and network routes

For unknown patterns, use randomisation

For many-to-one, use hashing and combining (or replication)

Composition

Patterns can be composed and embedded within each other

Sometimes the entire program evolution is visible to a compiler

Sometimes the evolution is data-sensitive

The issues in allocating processors and network routes mirror those of allocating memory in sequential processing

... local, global, stack, heap

How fast can a computation spread?

Clos and Benes Networks

Clos networks implement *permutations* on their inputs - no contention

A *strict-sense* network can always allocate a new route

A *re-arrangeable* network needs fewer routers but may require re-arrangement of existing routes

Known permutation + re-arrangeable $=>$ Compile-time (or on-the-fly)

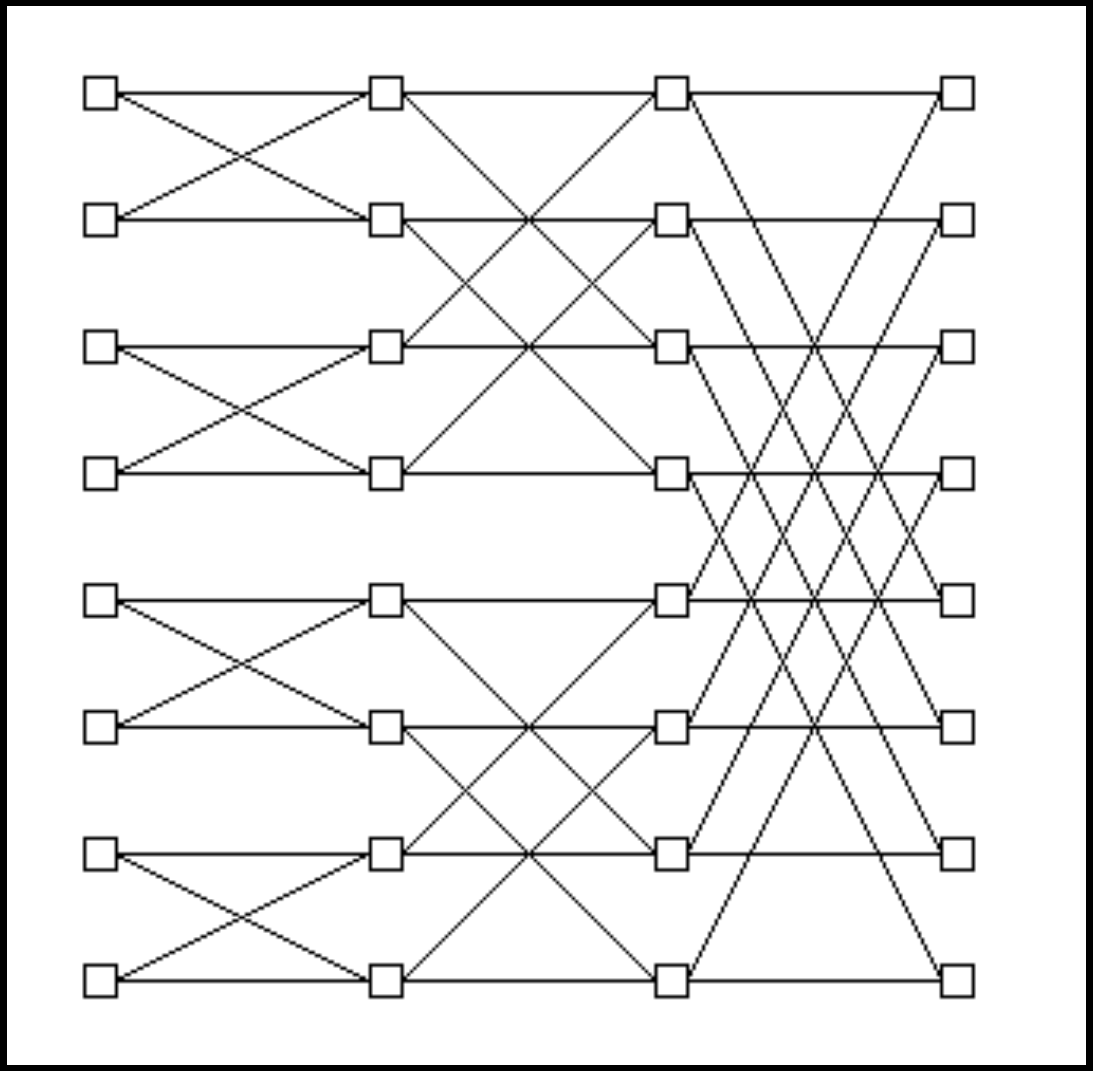
Unknown pattern + re-arrangeable $=>$ Run-time using randomisation

Benes networks are easily partitioned into sub-networks

Folded Benes Network

edge

core



Partitions and Synchronisation

Within a switch, *synchronised messages* are forwarded from both inputs before a following synchronised message is forwarded

This enables an entire partition to perform a series of distinct permutations; it is *in-order pipelined*

One-one communication: single permutation

Many-many communication: series of permutations
... compiling this series is an extension of network path allocation

One-many: series of permutations forming a tree from source

Sequential Programs

... implementing a large program with a set of small processors

Distribute data structures across processors

Distribute procedures, functions and objects across processors

Accesses to data are less than 10% of instructions; calls are less than 5%

No contention - network is under-loaded

Optimisations: concurrent accesses and concurrent calls; moving program to data

Programming Languages

Many recent programming languages need garbage collected memory

... this currently uses *stop-the-world* or *concurrent* collectors

Several languages need detection of array-bound violations and arithmetic overflows

... this currently uses libraries and is inefficient

All of this can be done efficiently in hardware ...

Memory management

A mark-sweep-compact garbage collector can be implemented in a small state-machine

Each state-transition makes at most one memory access

Transitions can be arbitrarily interleaved with instruction execution
... using memory cycles not needed by instructions

The instruction set architecture includes *get-memory* instructions

Array-bound violations checked in hardware; memory cleared during sweep

Processor management

Garbage-collected memory could potentially be extended to garbage-collected processes

The mark phase would involve sending marking messages along channels

A synchronisation would take place, followed by sweep, and another synchronisation

This would support some new (?) programming techniques such as concurrent recursive *find-first*, *achieve-bound*, *chose-best at deadline*

Fault containment

In a connected world, a small fault can have unlimited effects ...

And in a world of robots, a small fault can have physical effects ...

It's best to contain faults as soon as they arise

... array bound violations, arithmetic overflows, out-of-range shifts

Need a standard for integer arithmetic!

And by providing *move* as well as *copy*, we can make concurrency safe even when passing pointers

Caches

It is possible to make caches that are predictable

An easy way is to use separate caches for the program, stack and data of each processor/process

The program and stack caches can be direct map, The data cache should be fully associative with least-recently-used replacement

A better approach is to use a *partitioned cache* that enables software control over the partitioning

... this means that the cache can be divided to prevent interference between processes - and between accesses to different data objects

The need for Innovation

Align performance of processors, memory and interconnect (current processors optimised for speed, DRAMs optimised for density)

Support for real-time parallel computing and scalable input-output

Support for fault-containment

Support for modern programming languages, especially concurrent and parallel ones

A lot of improvements could be made based on existing architectures ... or we could do something new

Commodity parallel processing

Ideally, we want to build processing like memory

Choose an economical chip size ($70mm^2$ for DRAM, $100mm^2$ for logic)

... $100mm^2$ will hold hundreds of computers

Stack them up in 3D using through-silicon-vias.

Connect the stacked devices using silicon photonics and wavelength division multiplexing

General purpose components with behaviour defined by software

The open transputing architecture

Components for clouds (especially the edge), things and robots

Deterministic processing nodes with garbage collection

Low latency communication and input-output

Nodes for deterministic, non-blocking networks

Concurrent programming language and tools

Safe concurrency with error containment

... a higher level of abstraction in system design!