Multicore Architecture

David May

The Past

INMOS started 1978: introduced the idea of a a communicating computer - *transputer* - as a system component

Key idea was to simplify system design by moving to a higher level of abstraction

A concurrent language based on *communicating processes* was to be used as a design formalism and programming language

Programming language *occam* launched 1983; transputer launched 1984

CSP, Occam and Concurrency

Sequence, Parallel, Alternative

Channels, communication using message passing

Event Driven

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

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 interconnects for specific applications

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

But no easy way to build subroutines and libraries!

Emerging need for a new platform

Post 2000, divergence between emerging market requirements and trends in silicon design and manufacturing

Electronics becoming fashion-driven with shortening design cycles; but state-of-the-art chips becoming more expensive and taking longer to design ...

Concept of a single-chip tiled processor array as a programmable platform emerged

Importance of I/O - mobile computing, ubiquitous computing, robotics ...

The Present

We can build chips with hundreds of processors

We can build computers with millions of processors

We can support concurrent programming in *hardware*

We can define and build digital systems in *software*

Architecture

Regular, tiled implementation on chips, modules and boards

Scale from 1 to 1000 processors per chip

System interconnect with scalable throughput and low latency

Streamed (virtual circuit) or packetised communications

Architecture

High throughput, responsive, input and output

Support compiler optimisation of concurrent programs

Power efficiency - compact programs and data, mobility

Energy efficiency - event driven systems

Interconnect

Support multiple bidirectional links for each tile - a 500MHz processor can support several 100Mbyte/second streams

Scalable bisection bandwidth can be achieved on silicon using crosspoint switches or multi-stage switches even for hundreds of links.

In some cases (eg modules and boards), low-dimensional grids are more practical.

A set of links can be configured to provide several independent networks -important for diverse traffic loads

Interconnect Protocol

Protocol provides *control* and *data* tokens; applications optimised protocols can be implemented in *software*.

A route is opened by a message *header* and closed by an *end-of-message* token.

The interconnect can then be used under software control to

- establish virtual circuits to stream data or guarantee message latency
- perform dynamic packet routing by establishing and disconnecting circuits packet-by-packet.

Processes

A processor can provide *hardware* support for a number of processes, including:

- a set of *registers* for each process
- a *scheduler* which dynamically selects which process to execute
- a set of *synchronisers* for process synchronisation
- a set of *channels* for communication with other processes
- a set of *ports* used for input and output
- a set of *timers* to control real-time execution

Processes - use

Allow communications or input-output to progress together with processing.

Implement ‘hardware’ functions such as DMA controllers and specialised interfaces

Provide latency hiding by allowing some processes to continue whilst others are waiting for communication with remote tiles.

The set of processes in each tile can also be used to implement a kernel for a much larger set of software scheduled tasks.

Process Scheduler

The process scheduler maintains a set of runnable processes, *run*, from which it takes instructions in turn.

A process is not in the *run* set when:

- it is waiting to synchronise with another process before continuing or terminating.
- it has attempted an input but there is no data available.
- it has attempted an output but there is no room for the data.
- it is waiting for one of a number of events.

The processor can power down when all processes are waiting

Process Scheduler

Guarantee that each of n processes has $1/n$ processor cycles.

A chip with 128 processors each able to execute 8 processes can be used as if it were a chip with 1024 processors each operating at one eighth of the processor clock rate.

Share a simple unified memory system between processes in a tile.

Each processor behaves as symmetric multiprocessor with 8 processors sharing a memory with no access collisions and with no caches needed.

Instruction Execution

Each process has a short instruction buffer sufficient to hold at least four instructions.

Instructions are issued from the instruction buffers of the *runnable* processes in a round-robin manner.

Instruction fetch is performed within the execution pipeline, in the same way as data access.

If an instruction buffer is empty when an instruction should be issued, a *no-op* is issued to fetch the next instruction.

Execution pipeline

Simple four stage pipeline:

1	decode	reg-write		
2		reg-read		
3	address	ALU1	resource-test	
4	read/write/fetch	ALU2	resource-access	schedule

At most *one* instruction per thread in the pipeline.

Concurrency

Fast initiation and termination of processes

Fast barrier synchronisation - one instruction per process

Compiler optimisation using barriers to remove join-fork pairs

Compiler optimisation of sequential programs using multiple processes (such as splitting an array operation into two half size ones)

Fork-join optimisation

```
while true
{ par { in(inchan,a) || out(outchan,b) };
  par { in(inchan,b) || out(outchan,a) }
}
```

```
par
{ while true
  { in(inchan,a); SYNC c; in(inchan,b); SYNC c }
|| while true
  { out(outchan,b); SYNC c; out(outchan,a); SYNC c }
}
```

Concurrent Software Components

```
while true
{ par { in(nextx) || in(nexty) || nextr := f(x, y) || out(r) };
  x, y, r := nextx, nexty, nextr
}
```

```
while true
{ par { in(nextx) || in(nexty) || nextr := f(x, y) || out(r) };
  par { move(nextx, x) || move(nexty, y) || move(nextr, r) }
}
```

Components can be composed to implement deterministic concurrent systems.

Communication

Communication is performed using *channels*, which provide full-duplex data transfer between *channel ends*

The channel ends may be

- in the same processor
- in different processors on the same chip
- in processors on different chips

A channel end can be used as a destination by any number of processes - *server* processes can be programmed

The channel end addresses can themselves be communicated

Communication

Channel communication is implemented in hardware and does *not* involve memory accesses

This supports fine grained computations in which the number of communications is similar to the number of operations.

Within a tile, it is possible to use the channels to pass addresses.

Synchronised communication is implemented by the receiver sending a short acknowledgement message to the sender.

Ports, Input and Output

Inputs and outputs using ports provide

- direct access to I/O pins
- accesses synchronised with a clock
- accesses timed under program control

An input can be delayed until a specified condition is met

- the time at which the condition is met can be *timestamped*

The internal timing of input and output program execution is decopled from the operation of the input and output interfaces.

Ports, Input and Output example

```
proc linkin(port in_0, in_1, ack, int token) is
var state_0, state_1, state_ack;
{ state_0 := 0; state_1 := 0; state_ack = 0; token := 0;
  for bitcount = 0 for 10 do
  { select
    { case in_0 ?= ¬state_0: state_0 => token := token>>1
      case in_1 ?= ¬state_1: state_1 => token:=(token>>1)|512
    };
    ack ! state_ack; state_ack := ¬state_ack
  }
}
```

Timed ports example

```
proc uartin(port uin, byte b) is
{ var starttime;
  in ?= 0 at starttime;
  sampletime := starttime + bittime/2;
  for i = 0 for 8
   t := t + bittime; (uin at t) ? >> b ;
  (uin at (t + bittime)) ? nil
}
```

Event-based scheduling

A process can wait for an event from one of a set of channels, ports or timers

An *entry point* is set for each resource; a *wait* instruction is used to wait until an event transfers control directly to its associated entry point.

A compiler can optimise repeated event-handling in inner loops - the process is effectively operating as a programmable state machine - the events can often be handled by (very) short instruction sequences

Events vs. Interrupts

A process can be dedicated to handling an individual event or to responding to multiple events.

The data needed to handle each event have been initialised prior to waiting, and will be instantly available when the event occurs.

This is in sharp contrast to an interrupt-based system in which context must be saved and the interrupt handler context restored prior to entering it - and the converse when exiting.

Summary

Concurrent programming can be efficiently supported in *hardware* using tiled multicore chips.

They enable systems to be defined and built using *software*.

Each process can be used

- to run conventional sequential programs
- as a component of a concurrent computer
- as a hardware emulation engine or input-output controller

Event-driven hardware and software enable energy efficient systems.

XMOS XS1-G4

Four tiles	1600 MIPS; 32 processes
Switch	4 links per tile; 16 external
SRAM	64k bytes per tile
Synchronisers	7 per tile
Timers	10 per tile
Channel ends	32 per tile
Ports	1,4,8,16,32-bit
Links	16 at 400Mbits/second

The Future

Some people will bet on scalable shared memory systems - if they don’t care about cost, power and performance.

Some people will bet on complex heterogeneous architectures and compilers that do magical optimisations - if they don’t know that compilers take much longer to develop than hardware

Some people will bet on ‘abstraction layers’ to allow legacy software to be ported to parallel machines - if they haven’t yet discovered why their mobile phone takes so long to boot.

Realisation

The full potential of concurrency can be delivered directly to the user.

We can use processors with tightly integrated communications as system components - we now have the technology and the need for them

The language and formalism already exist - based on *concurrent processes*

We need to learn how to use them to build scalable concurrent computers and embedded system components

Concurrent Languages

Focus on data, control and resource dependency

Contrast:

Conventional programming languages: over-specified sequencing

Hardware design languages: over-specified parallelism

Need a single language to trade-off space and time (by designer or compiler); also need a semantics to do this automatically.

Concurrent computers and processors

Millions of processes/computer; 100s of processors/chip

General purpose embedded components with behaviour defined by concurrent software

These will enable rapid design of innovative consumer products - and chipless, fabless electronics companies

There is a potential to use new technologies such as plastics

Concurrency

Emphasis on *process structures* will replace emphasis on data structures

A *paradigm shift* in computer science and engineering - a universal computer is an infinite array of finite processors, not a finite array of infinite processors

Our design languages should reflect exactly those features common to *both* hardware and software

It’s time to educate a generation of concurrent thinkers!