XMOS Architecture
XC Language

David May

XMOS™

Embedded processing

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

XC

Sequence - like C

Parallel with disjointness checks made by compiler - no race conditions caused by shared variables

Synchronising channel communication - no race conditions caused by buffers

Control of non-deterministic events using *select* and *guards*
to control communication, input-output and access to shared state

XC

Error containment

Array bound checks, array slices

Failed processes *stop*, preventing propagation of incorrect behaviour and data

Communication by copying or by *moving* - and *movables* have exactly one owner

XC

Timers

Input and Output - ports, clocked ports and timed ports

In conjunction with the XMOS processor, *timing* of execution and Input/Output is deterministic

Potential to treat missed deadlines as hard errors

Potential to allocate communication resources to meet throughput requirements of channels and ports

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 systems, low-dimensional grids are more practical.

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

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.

Threads

Each processor provides *hardware* support for a number of threads, including:

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

Threads are used for latency hiding or to implement ‘hardware’ functions such as DMA controllers and specialised interfaces

Thread Scheduler

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

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

- it is waiting to synchronise with another thread 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 threads are waiting - *event-driven* processing

Thread Scheduler

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

A chip with 128 processors each able to execute 8 threads 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 the threads 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 thread has a short instruction buffer sufficient to hold at least four instructions.

Instructions are issued from the instruction buffers of the *runnable* threads 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.

Most *no-ops* eliminated by compiler instruction scheduling.

Concurrency - aim

Fast initiation and termination of threads

Fast barrier synchronisation - ideally one instruction per process

Compiler optimisation using barriers to remove join-fork pairs

Compiler optmisation of sequential programs using multiple threads (such as splitting an array operation into two concurrent half-size ones).

Join-fork 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 }
}
```

Communication

Communication is performed using *channels*, which provide bidirectional data transfer between hardware *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 threads - *server* processes can be programmed

The channel end addresses can themselves be communicated

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

Concurrent Software Components

```
while (true)
{ par { a :> nextx; b :> nexty; nextr = f(x, y); c <: r };
  par { x = nextx; y = nexty; r =nextr }
}
```

or using *moves* (::>, <::, ::=):

```
while (true)
{ par { a ::> nextx; b ::> nexty; nextr = f(x, y); c <:: r };
  par { x ::= nextx; y ::= nexty; r ::= nextr }
}
```

Components can be put together in *deterministic* concurrent systems.

Synchronised communication

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

It is impossible to scale interconnect throughput unless communication is pipelined.

This means that the use of end-to-end synchronisations is minimised; a compound communication (transaction) achieves this:

```
transaction inarray(chan in, int data[]; int size)
{ for (int i = 0; i < size; i++)
    in :> data[i];
}
```

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 decoupled from the operation of the input and output interfaces.

Ports, Input and Output example

```
void linkin(port in_0, port in_1, port ack, int &token)
{ int state_0, state_1, state_ack;
  state_0 = 0; state_1 = 0; state_ack = 1; token = 0;
  for (int bitcount = 0; bitcount < 10; bitcount++)
  { select
    { case in_0 when pinseq(!state_0) :> state_0 :
        token = token>>1
      case in_1 when pinseq(!state_1) :> state_1 :
        token = (token>>1) | 0x100
    };
    ack <: state_ack; state_ack = !state_ack
  }
}
```

Timed ports example

```
void uartin(port uin, char &b)
{ int starttime, sampletime;
  uin when pinseq(0) :> void @ starttime;
  sampletime = starttime + bittime/2;
  for (i = 0; i < 8; i++)
  { t := t + bittime; (uin @ t) :> >> b };
  uin @ (t + bittime) :> void
}
```

Event-based scheduling

A thread 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 threadis effectively operating as a programmable state machine - the events can often be handled by (very) short instruction sequences

Events vs. Interrupts

A thread 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 hardware thread 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 links
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

www.xmos.com