Communicating Process Architecture for Multicores

David May

Bristol University and XMOS

Introduction

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 - aims

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 - aims

High throughput, responsive input and output

Instruction set designed to support compiler optimisations

Power efficiency - compact programs and data, mobility

Energy efficiency - event driven systems

Interconnect

Processor needs multiple bidirectional links - a 500MHz processor can support several 100Mbyte/second streams

Manufacturing processes now have many layers of interconnect

Fully connected networks can be used on-chip

- Clos network for 512 links uses much less area than 128 tiles

In modules and systems, n-dimensional grids can be used

Routing - modules and systems

Simple hardware operating on first few bits of message

Incoming bits compared with tile address, bit-by-bit

If all pairs match, the tile is the destination

If not, the first bit number of the first non-matching pair is used to select an outgoing route via a lookup table

This is sufficient to perform efficient deadlock-free routing in all n-dimensional arrays.

Two-dimensional array example

processor	entry	processor	entry	processor	entry	processor	entry
0	rrdd	4	rldd	8	lrdd	12	lldd
1	rrdu	5	rldu	9	lrdu	13	lldu
2	rrud	6	rlud	10	lrud	14	llud
3	rruu	7	rluu	11	lruu	15	lluu

Each table entry selects either a right link (r), left link (l), up link (u) or down link (d).

Performance can be enhanced using multiple links on each path

Interconnect protocol

Communication protocol provides *control* and *data* tokens

Can be used by software to construct applications-optimised protocols

Allows the interconnect to be used under program control to

- establish virtual circuits to stream data
- transport a series of packets

Alternatively can be used for dynamic packet routing by establishing and disconnecting circuits packet-by-packet

Processes

Each processor includes hardware support for a number of processes, including:

- a set of registers for each process
- a process scheduler which dynamically selects which process to execute
- 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
- a set of clock generators to enable synchronisation of the input-output with external time domains

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 processes.

Processor instruction set

How many registers for each process?

- not so many that the file for all the processes is big and slow
- enough for the processes to operate efficiently

Another issue - they have to be addressed as instruction operands and we want short (16-bit) instructions

Even with 16 registers, 12 bits are required to specify three operands leaving only 4 opcode bits.

Processor instruction encoding

Provide dedicated registers to access program, stack and data regions in memory

Provide 12 *operand* registers for general purpose use - three register operands can then be encoded using 11 bits (as $12 \times 12 \times 12 < 2048$) leaving 5 opcode bits.

One or two opcodes can be used to allow 32-bit instructions

- instructions with up to 6 operands (for cryptography, DSP etc)
- extended immediate range for jumps and stack offsets
- lots of spare opcodes

Processor - Resources

Each processor manages physical resources: processes, synchronisers, channels, timers, locks and clock generators.

Processes *claim* and *free* resources using special instructions.

Resources interact directly with the process scheduler and instructions such as inputs and outputs can potentially result in a process pausing until a resource is ready and then continuing.

Information about the state of a resource is available to the scheduler within a single processor cycle.

Process Scheduler

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

A process is removed from the *run* set when:

- its registers are being initialised prior to it being able to run.
- 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.

Process Scheduler - aim

Share a unified memory system between processes in a tile.

Should be possible as 2 instructions are fetched each cycle and data accesses are typically 25-30% of instructions

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

Use spare cycles to increase average performance or save energy.

Process scheduler - use

Imagine a chip with 128 processors each able to execute 8 processes

It can be used as if it were a chip with 1024 processors each operating at one eighth of the processor clock rate.

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

Process Scheduler - Implementation

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.

Processes which are not in use or are paused are ignored.

The execution pipeline has a *memory access stage* which is available to *all* instructions.

Use of memory access stage

Any instruction which requires memory access performs it

Branch instructions fetch their branch target instructions unless they also require a data access (in which case they will leave the instruction buffer empty).

Any other instruction performs an instruction fetch for the process - or for another process.

Note: If a process's instruction buffer is empty when an instruction should be issued, a special *fetch no-op* is issued to fetch the next instruction.

Concurrency - aim

Fast initiation and termination of processes

Fast barrier synchronisation - ideally 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 }
}
```

Concurrency and Synchronisation

Hardware synchronisers allow synchronisation to be performed at 1 instruction/process

Instructions provided to

- get a synchroniser
- get and attach new processes to a synchroniser
- free a synchroniser and all of its attached processes
- transfer data directly between the registers of two processes

Synchronisation instructions interact with the scheduler via the synchronisers

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

The channel-end identifiers can be used anywhere in a system

The channels provide a uniform method of communication throughout a system with multiple tiles or multiple chips.

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.

Channels carry messages built from *tokens*

- data takens are just bytes
- control tokens are used to encode communication protocols

Messages

Each message starts with a *header* containing the identifier of the destination channel end.

This is usually followed by a series of data or control tokens, sent and received by software using special instructions.

Each message ends with an *end of message* control token.

Packet-based communication or virtual circuit communication can be *programmed*.

Channel Ends

A channel end can be used as a destination by any number of processes

They are served on a round-robin basis.

In this case the sender will normally send an identifier of a channel end which can be used to send a reply, or to establish bi-directional communication.

As the identifiers of channel ends can be used anywhere, they can themselves be communicated.

Synchronised Communications

As most messages consist of many individual data items, there is no need for all of the individual items to be acknowledged.

It is impossible to scale interconnect throughput unless communication is pipelined

This requires that the use of end-to-end synchronisations is minimised.

Synchronised communication is implemented by the receiver sending an acknowledgement to the sender, usually as a message consisting of a header and an end-of-message token.

Compound Communications

A convenient way to express sequences of communications on the same channel is with a *compound communication*.

```
proc inarray(chan c, []int a) is
?{ for i = 0 for 10 do c ? a[i] ?}

proc outarray(chan c, []int a) is
!{ for i = 0 for 10 do c ! a[i] !}
```

The synchronisations at the end of each of these compound communications ensure that each compound output is matched by exactly one compound input.

Ports, Input and Output

Ports provide interfaces to physical pins.

Inputs and outputs using ports provide

- direct access to the pins
- accesses synchronised with a clock
- accesses timed under program control

A condition can be set which causes an input to delay

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

Timers and Clocks

Each tile has a free-running clock and a set of timers which can be used to read the current time or to wait until a specified time.

Input and output operations can be synchronised with an internally generated clock or an externally supplied clock.

When an output port is driven from a clock, the data on the pin(s) changes state synchronously with the clock.

If several ports are driven from the same clock, they will appear to operate as a single port.

Time domains

The processes executed by a processor can handle external devices at several different rates determined by clocks supplied externally or generated internally.

The ports decouple the internal timing of input and output program execution from the operation of the input and output interfaces.

The processor can operate using its own clock, or could potentially be asynchronous.

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 7
   t := t + bittime; (uin at t) ? >> b ;
  (uin at (t + bittime)) ? nil
}
```

Events and alternative input - aim

Allow a process to wait for input from any of a set of channels or ports

Very rapid response so that repeated alternatives can be used in low-level input-output operations

Support compiler optimisations to minimise response times

Allow procedures expressing compound communications to be used as guards

Events - implementation

An event transfers control to an associated program *entry point* which is set for a specific resource by a *setvector* instruction

Event generation for the resource can then be enabled and disabled using *enable* instructions

Having enabled events on one or more resources, a *wait* instruction is used to wait until an event transfers control to its associated entry point.

All of the events which have been enabled by a process can be disabled using a single *clear events* instruction.

Events - Compound communications

It is important to allow calls to procedures defining compound communications to be used in guards:

```
select
{ case inarray(c, a) => P(a)
  case inarray(d, a) => Q(a)
}
```

This is done by a *setenv* instruction to initialise an *environment* register in the port

This is usually set to the stack pointer value at the time the event is enabled by the communicating procedure.

Events - optimisations

Optimise repeated alternatives in inner loops where a process is operating as a programmable state machine - the guarded components are usually short instruction sequences

Move setting of event vectors and other invariants outside the loop

Provide conditional versions of *event enable* instructions to optimise guard enabling

Provide conditional versions of *wait* to replace the loop-closing branch

Events vs. Interrupts

A process can be dedicated to handling an individual event or to an alternative handling 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

Communicating process architecture can be used to design and program multicore chips with performance scaling to thousands of processes, or virtual processors.

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

Communicating processes provide a natural way of expressing energy efficient event-driven programs.

Summary

The architecture is tuned to compiler and software needs - supporting direct execution of concurrent software

The processor instruction set enables optimisation of concurrent and event-driven programs.

The compact instruction representation, position independent code and high speed interconnect enables software mobility.

This reduces latency and power and supports dynamic re-use of processors at runtime.