Architectures for Ubiquitous Computing

David May

Bristol University

Zero Power Computing

David May

Bristol University

Why Zero Power?

Power embedded computers from the environment
... or from batteries which last for the product lifetime

Put 1000 computers on a chip
... and 1,000,000 in a server

Ubiquitous Computing

Aim is to design quickly: computer-based devices are becoming fashion items

But:

- Design cost is increasing, especially verification
- Manufacturing set-up cost is increasing

Need for standard programmable and/or configurable platforms
... programmed at a high level

Event-driven systems

Low power doesn’t mean low performance

1 milliwatt-second = 1 watt-millisecond

Many systems will be idle most of the time, waiting for an event - a change of environment state

This may result in a massive amount of activity, involving thousands of processors

What does low power involve?

Low voltage circuits

Low power logic design

Dynamically switching off stuff when it isn’t in use

Minimal operations *including* data transfers

Event driven systems *including* software

Where does the power go?

Clocks

Cache access

Memory access

Register file access

... ?

An experiment

- Simple architecture
- Simple language
- Some modern compiler optimisations
- Efficient concurrency
- Efficient communication and input-output

Instruction set architecture

- How many bits in a word? ... instruction?
- How many instruction types?
- What data-types?
- What operations?
- How many registers?

Language

- skip, assignment, call, while, if
- input, output, parallel, alternative
- no jumps, no exits
- no pointers
- no aliases, no sharing

Compiler

- Exploit absence of aliases and sharing
- Exploit simple control structure
- Analyse dependencies and liveness
- Minimise register - memory transfers
- Optimise concurrency, communication and input/output

An architecture

All language implementations need dedicated registers

- PC
- Stack pointer
- Frame pointer
- ...

So we might as well optimise access to them instead of making them general purpose registers

Instruction coding

Suppose we use 16 registers of which 4 are dedicated (PC, FP, SP, GP).

The 12 remaining general purpose registers can be used for base addresses, arithmetic, parameter passing ...

It’s possible to encode a complete (small) instruction set using (only) 16-bit instructions - and it’s wordlength independent

With appropriate compiler optimisations, this seems to result in instruction profiles similar to 32-bit RISC instruction sets

Compiler

Prevent aliases: every object (variable, process, channel ...) has only one name

Parameters can be passed *copy-in, copy-out* or *by reference*

Prevent sharing: every object belongs to at most one process - so free variables can be accessed by copying or by reference

Keep track of which objects are live - optimise register usage

Event-driven systems

Processors switch off when they have nothing to do

Input-output systems switch off when they have nothing to do

Nothing iteratively watches for events (no polling)

Only a few transistors *need* to be active, watching for a state-change in the environment

... and this shouldn't need special power-management software

Experiment - Multiprocessing

8 hardware process contexts, feeding a short execution pipeline

16 internal communication channels

16 external communication ports for connection to other processors

n external configurable input-output ports to provide interfaces to physical devices (pins)

(A channel is two connected ports; it is *point-to-point*)

Why is multiprocessing useful?

Do more than one thing at once!

Overlap input-output and processing

Overlap communication and processing

Keep execution pipeline full

Hide latency of memory access

Hide latency of branches and calls

Synchronisation

Each process context is identified by a bit in an 8-bit value.

A set of processes is an 8-bit value.

INITP	context, pc	: supply new pc
INITS	context, sp	: supply new stack
END		: terminate
STARTP	set	: activate set of processes
WAITP	set	: wait for all of set to terminate

Communications

Split communications

- STARTIN - commit to accept data; continue
- STARTOUT - put data in buffer; continue
- ENDIN - take data or wait; ack
- ENDOUT - wait for data taken

Rules

- ENDIN completes after STARTIN
- ENDOUT completes after STARTOUT

Concurrent communication

Example:

```
{  {  in ? next                  STARTIN next
   &  out ! last                 STARTOUT last
   &  compute(this)              compute this
   }                             ENDOUT last; ENDIN next
;  last,this := this,next        last,this := this,next
}
```

The communication overlaps with the computation

Concurrent communication

Example:

```
{  {  input (next)              STARTP input next
   &  out (last)                STARTP output last
   &  compute(this)             compute this
   }                            WAITP input, output
;  last,this := this,next       last,this := this,next
}
```

Again, the communication overlaps with the computation - extra processes are used as generalised DMA units

Communication and processing

Example:

```
var xbuf[], ybuf[], zbuf[];
...
while true do
{ { input(xbuf) & compute(ybuf) & output(zbuf) }
; { input(zbuf) & compute(xbuf) & output(ybuf) }
; { input(ybuf) & compute(zbuf) & output(xbuf) }
}
```

This can be optimised using *barrier* instructions

Alternative

Example:

```
{ count < max : put ? (x) do
    { .. store data .. ; count := count + 1 }
| count > min : get ? (y)  do
    { .. get data .. ; count := count - 1 }
}
```

How can we implement *Alternative*?

Alternative

Example:

```
{ when temperature > alarm_level do
    { .. sound_alarm ! temperature .. }
| set_alarm ? alarm_level do
    { .. }
}
```

How can we implement *Alternative*?

Implementing Alternative

Identify set of guard events (potentially ready inputs)

Mask with guard conditions

Wait until (at least) one guard is ready *(with power off)*

Select one guarded body for execution

Jump to code of the selected guarded body

... most computers can't do this efficiently

Instructions for Alternative

Each port can have an associated register to hold the address of a corresponding guarded body

These are loaded by a LDEVNT instruction

Each port has an enable bit which is set/cleared depending on the guard condition by a SETEVNT instruction

When the guards are enabled, an EWAIT instruction is executed ... which (eventually) returns a ready guard

Optimising Alternative

Observe that *alternatives* are often iterated

... and guard conditions are changed within guarded bodies

We don't *need* to recompute all guards and re-enable/disable the ports on every iteration

We can move a guard condition evaluation - and resulting modification of the wait condition(s) - to the place(s) where the values change

Software input-output

Even at 100-200MHz, we can execute a lot of instructions in a microsecond!

So we should be able to keep up with all but the fastest input-output using high-level language programs executing input, output and alternative

And high-level language ‘DMA controllers’ should be able to sustain at least 10Mbytes/second

Pipelined communications

It's convenient to use a process to communicate a data structure:

- traversing
- encoding/decoding
- compression/decompression
- encryption/decryption

This process will perform a series of communications on the same channel; this can be optimised as

- a series of unsynchronised communications
- a final synchronising communication

Mobility

We want to move processes to other processors efficiently

To do this, we need *mobile* processes

As there are no aliases we can safely - and efficiently - communicate

- processes
- objects
- ports

We can move data objects and processes by copying or by passing a reference; moving ports needs a simple protocol

Speeding it up (1)

If we need to go faster, we can allow combinations of instructions to be executed together

- communicate + inc + branch
- memory + alu + branch
- shift + i/o + branch

Data can be streamed at one object (bit/byte/halfword/word) every two cycles

Multiply-accumulate runs at two operations every three cycles

Speeding it up (2)

If we need a cache, we can use a partitioned direct-map cache. These

- prevent interference between processes
- prevent interference between data structure access
- support streaming efficiently

- are relatively small

(but they have to be controlled by compilers)

Chips full of processors

By replicating a simple processor, we can create *programmable processor arrays*

These can support a variety of concurrent programming styles

We can migrate data and processes to minimise latency and power

We can dynamically re-use processing resources

How fast can a computation spread?

If we want to write parallel subroutines, we have to be able to distribute sub-computations rapidly

This issue has been ignored in all (?) parallel computer designs, which is why they are so hard to use

A good way to do this is to use a *worm*, which spreads by replicating itself

We can optimise this with *wormhole computing* - executing a program - which starts to forward itself - whilst it's still arriving

Summary

It is almost certain that we can build much more power efficient computers than anything we currently have

It’s already clear that we can achieve *zero-power* in the sense of simple computers powered from the environment

... but there’s lots more to do!