Precision-Timed (PRET) Machines

Stephen A. Edwards

Department of Computer Science,
Columbia University

www.cs.columbia.edu/~sedwards

sedwards@cs.columbia.edu

with Edward A. Lee (Berkeley)
A Major Historical Event

In 1980, Patterson and Ditzel did not invent reduced instruction set computers (RISC machines).

Another Major Historical Event

In 2006, Lee and Edwards did not invent reduced precision-timed computers (PRET machines).


http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-149.html
We do not consider how fast a processor runs when we evaluate whether it is “correct.”

This is Sometimes Useful For

- Programming languages
- Virtual memory
- Caches
- Dynamic dispatch
- Speculative execution
- Power management (voltage scaling)
- Memory management (garbage collection)
- Just-in-time (JIT) compilation
- Multitasking (threads and processes)
- Component technologies (OO design)
- Networking (TCP)
But Time Sometimes Matters

Kevin Harvick winning the Daytona 500 by 20 ms, February 2007.
Certification

- Is rather expensive
- Software is *not* certified
- Entire system is certified
- Slight change, e.g., in the microprocessor, requires recertification
- Solution: stockpile parts; trust nobody
Worst-Case Execution Time

Virtually impossible to compute on modern processors.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Nearby instructions</th>
<th>Distant instructions</th>
<th>Memory layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipelines</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch Prediction</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Caches</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
WCET on a Motorola ColdFire

- Two coupled pipelines (7-stage)
- Shared instruction/data cache
- Artificial example supplied by Airbus
- Twelve independent tasks
- Simple control structures
- Cache/Pipeline interaction lead to large integer linear programming problem

The Problem

Digital hardware provides extremely precise timing

and architectural complexity discards it.
Our Solution: P⁴
Predictable Performance, the Precision Panacea

<table>
<thead>
<tr>
<th>Current</th>
<th>Alternative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caches</td>
<td>Scratchpads</td>
</tr>
<tr>
<td>Pipelines</td>
<td>Thread-interleaved pipelines</td>
</tr>
<tr>
<td>Function-only ISAs</td>
<td>ISAs with timing</td>
</tr>
<tr>
<td>Functional languages</td>
<td>Languages with timing</td>
</tr>
<tr>
<td>Data Races</td>
<td>Deterministic concurrency</td>
</tr>
<tr>
<td>Best-effort communication</td>
<td>Fixed-latency communication</td>
</tr>
</tbody>
</table>
Basic Idea

Q: How do you make software run at a precise speed?
Basic Idea

Q: How do you make software run at a precise speed?

A: Give it access to a clock.
One Usual Way: Timers

Period timer interrupt triggers scheduler

Large period reduces overhead

Linux uses a 10 ms clock

Result: OS provides 10 ms resolution at best

Higher precision requires more overhead

10 ms 20 ms 30 ms 40 ms 50 ms 60 ms
Or NOPs/cycle counting

Code from Linux arch/i386/kernel/timers/timer_none.c

delay_none:
  0:  push  %ebp
  1:  mov   %esp,%ebp
  3:  sub   $0x4,%esp
  6:  mov   0x8(%ebp),%eax
  9:  jmp   10

  10: jmp  20

  20: dec   %eax
  21: jns   20

  23: mov   %eax,−4(%ebp)
  26: leave
  27: ret

Tricky

Clock speed + cache behavior + branch behavior + ?

This example worries about cache alignment

Very much an assembly-language trick

1000s of lines of code in Linux needed for busy wait
Related Work: Giotto


The RTOS style: specify a collection of tasks and modes. Compiler produces schedule (task priorities).

Precision limited by periodic timer interrupt.

```plaintext
mode forward() period 200 {
  actfreq 1 do leftJet(leftMotor);
  actfreq 1 do rightJet(rightMotor);
  exitfreq 1 do point(goPoint);
  exitfreq 1 do idle(goIdle);
  exitfreq 1 do rotate(goRotate);
  taskfreq 2 do errorTask(getPos);
  taskfreq 1 do forwardTask(getErr);
}
```
Related Work: STI

Software Thread Integration [Dean, RTSS 1998]

Insert code for a non-real-time thread into a real-time thread.

Pad the rest with NOPs

Often creates code explosion

Requires predictable processor
Related Work: VISA

VISA [Meuller et al., ISCA 2003]

Run two processors:

- Slow and predictable
- Fast and unpredictable

Start tasks on both.

If fast completes first, use extra time.

If fast misses a checkpoint, switch over to slow.
A First Attempt

MIPS-like processor with 16-bit data path as proof of concept

One additional “deadline” instruction:

`dead timer, timeout`

Wait until `timer` expires, then immediately reload it with `timeout`. 
# Programmer's Model

## General-purpose Registers

<table>
<thead>
<tr>
<th>$0$ (= 0)</th>
<th>$1$</th>
<th>$2$</th>
<th>$13$</th>
<th>$14$</th>
<th>$15$</th>
</tr>
</thead>
</table>

## Timers

<table>
<thead>
<tr>
<th>$t0$</th>
<th>$t1$</th>
<th>$t2$</th>
<th>$t3$</th>
</tr>
</thead>
</table>

## Program counter

| $pc$ |
### Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add</code></td>
<td>Rd, Rs, Rt</td>
</tr>
<tr>
<td><code>addi</code></td>
<td>Rd, Rs, imm16</td>
</tr>
<tr>
<td><code>and</code></td>
<td>Rd, Rs, Rt</td>
</tr>
<tr>
<td><code>andi</code></td>
<td>Rd, Rs, imm16</td>
</tr>
<tr>
<td><code>be</code></td>
<td>Rd, Rs, offset</td>
</tr>
<tr>
<td><code>bne</code></td>
<td>Rd, Rs, offset</td>
</tr>
<tr>
<td><code>j</code></td>
<td>target</td>
</tr>
<tr>
<td><code>lb</code></td>
<td>Rd, (Rt + Rs)</td>
</tr>
<tr>
<td><code>lbi</code></td>
<td>Rd, (Rs + offset)</td>
</tr>
<tr>
<td><code>mov</code></td>
<td>Rd, Rs</td>
</tr>
<tr>
<td><code>movi</code></td>
<td>Rd, imm16</td>
</tr>
<tr>
<td><code>nand</code></td>
<td>Rd, Rs, Rt</td>
</tr>
<tr>
<td><code>nandi</code></td>
<td>Rd, Rs, imm16</td>
</tr>
<tr>
<td><code>nand</code></td>
<td>Rd, Rs, Rt</td>
</tr>
<tr>
<td><code>nor</code></td>
<td>Rd, Rs, Rt</td>
</tr>
<tr>
<td><code>nor</code></td>
<td>Rd, Rs, imm16</td>
</tr>
<tr>
<td><code>or</code></td>
<td>Rd, Rs, Rt</td>
</tr>
<tr>
<td><code>ori</code></td>
<td>Rd, Rs, imm16</td>
</tr>
<tr>
<td><code>sb</code></td>
<td>Rd, (Rt + Rs)</td>
</tr>
<tr>
<td><code>sbi</code></td>
<td>Rd, (Rs + offset)</td>
</tr>
<tr>
<td><code>sll</code></td>
<td>Rd, Rs, Rt</td>
</tr>
<tr>
<td><code>slli</code></td>
<td>Rd, Rs, imm16</td>
</tr>
<tr>
<td><code>srl</code></td>
<td>Rd, Rs, Rt</td>
</tr>
<tr>
<td><code>srl</code></td>
<td>Rd, Rs, imm16</td>
</tr>
<tr>
<td><code>sub</code></td>
<td>Rd, Rs, Rt</td>
</tr>
<tr>
<td><code>subi</code></td>
<td>Rd, Rs, imm16</td>
</tr>
<tr>
<td><code>dead</code></td>
<td>T, Rs</td>
</tr>
<tr>
<td><code>deadi</code></td>
<td>T, imm16</td>
</tr>
<tr>
<td><code>xnor</code></td>
<td>Rd, Rs, Rt</td>
</tr>
<tr>
<td><code>xnori</code></td>
<td>Rd, Rs, imm16</td>
</tr>
<tr>
<td><code>xor</code></td>
<td>Rd, Rs, Rt</td>
</tr>
<tr>
<td><code>xori</code></td>
<td>Rd, Rs, imm16</td>
</tr>
</tbody>
</table>
## Behavior of *Dead*

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Instruction</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4</td>
<td><code>deadi $t0, 8</code></td>
<td>8</td>
</tr>
<tr>
<td>-3</td>
<td>&quot;</td>
<td>2</td>
</tr>
<tr>
<td>-2</td>
<td>&quot;</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>&quot;</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td><code>add $r1, $r2, $r3</code></td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td><code>deadi $t0, 10</code></td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>&quot;</td>
<td>5</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>7</td>
<td>&quot;</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td><code>add $r1, $r2, $r3</code></td>
<td>9</td>
</tr>
</tbody>
</table>

8 cycles
Idioms: Straightline Code

deaddi $t0, 42
\text{;}

deaddi $t0, 58
\text{;}

deaddi $t0, 100

First block will take at least 42 cycles.

Second block: at least 58 cycles.
Idioms: Loops

L1:

```
  :  
  :  
deadi $t0, 42  
  :  
bne $r1, $r2, L1
```

Put a deadline in a loop:

Each iteration will take at least 42 cycles.
Case Study: Video

80 × 30 text-mode display

25 MHz pixel clock

Pixel shift register in hardware; everything else in software
Case Study: Video

Two nested loops:

- Active line
- Character

Two timers:

- $t1$ for line timing
- $t0$ for character output

78 lines of assembly
Replaces 450 lines of VHDL (1/5th)
Case Study: Serial Receiver

Sampling rate under software control

Standard algorithm:

1. Find falling edge of start bit
2. Wait half a bit time
3. Sample
4. Wait full bit time
5. Repeat 3. and 4.

movi $3, 0x0400 ; final bit mask (10 bits)
mov $5, 651 ; half bit time for 9600 baud
shli $6, $5, 1 ; calculate full bit time

wait_for_start:
  bne $15, $0, wait_for_start

got_start:
  wait $t1, $5 ; sample at center of bit
  movi $14, 0 ; clear received byte
  movi $2, 1 ; received bit mask
  movi $4, 0 ; clear parity
  dead $t1, $6 ; skip start bit

receive_bit:
  dead $t1, $6 ; wait until center of next bit
  mov $1, $15 ; sample
  xor $4, $4, $1 ; update parity
  and $1, $1, $2 ; mask the received bit
  or $14, $14, $1 ; accumulate result
  shli $2, $2, 1 ; advance to next bit
  bne $2, $3, receive_bit

check_parity:
  be $4, $0, detect_baud_rate
  andi $14, $14, 0xff ; discard parity and stop bits
Implementation

Synthesized on a Xilinx Spartan-3 FPGA

Coded in VHDL

Runs at 25 MHz

Unpipelined

Uses on-chip memory
Conclusions

- Embedded applications need timing control
- For high precision, we need hardware and software support
- Predictable Performance should be our mantra
- A first cut: Prototype MIPS-like processor runs at 25 MHz
- *Dead* instruction waits for timeout, then reloads synchronously
- Text-mode video display 1/5 the size of VHDL
- Serial controller even more simple