To Meet or Not to Meet the Deadline

Gage Eads
Stephen A. Edwards
Sungjun Kim
Edward A. Lee
Ben Lickly
Isaac Liu
Hiren D. Patel
Jan Reineke <speaker>

UC Berkeley
Columbia University
Columbia University
UC Berkeley
UC Berkeley
UC Berkeley
University of Waterloo
UC Berkeley

Ninth Biennial Ptolemy Miniconference
Berkeley, CA, February 16, 2011
Abstractions are Great
… if they abstract the right thing

Higher-level Model of Computation
C-level programming language
Instruction Set Architecture (ISA)
Hardware Realizations

Abstracts from execution time

Code Generation
Compilation
Execution
Current Timing Verification Process

C Program → Compiler → Binary → Architecture

WCET Analysis

✓ ✗
Current Timing Verification Process

- New Architecture ➔ Recertification
- Extremely time-consuming and costly

Airbus: 40 years supply of...
Agenda of PRET

- Higher-level Model of Computation
- C-level programming language
- Instruction Set Architecture (ISA)
- Hardware Realizations

Endow with control over timing
Predictable Execution Platform

Code Generation
Compilation
Execution

Reineke et. al, Berkeley 5
PRET Machines
Make Timing a Semantic Property of Computers

Precision-Timed (PRET) Machines

Timing precision with performance: Challenges:

- Memory hierarchy (scratchpads?)
- Deep pipelines (interleaving?)
- ISAs with timing (deadline instructions?)
- Predictable memory management (Metronome?)
- Languages with timing (discrete events? Giotto?)
- Predictable concurrency (synchronous languages?)
- Composable timed components (actor-oriented?)
- Precision networks (TTA? Time synchronization?)

See our posters!
Agenda of this Talk

- Higher-level Model of Computation
- C-level programming language
- Instruction Set Architecture (ISA)
- Hardware Realizations

Corresponding timing constructs
Endow with control over timing

Code Generation
Compilation
Execution
Adding Control over Timing to the ISA

Variant 1: “delay until”

Some possible capabilities in an ISA:

- [V1] Execute a block of code taking at least a specified time [Ip & Edwards, 2006]

Where could this be useful?

- Finishing early is not always better:
  - Scheduling Anomalies (Graham’s anomalies)
  - Communication protocols may expect periodic behavior
  - …
Adding Control over Timing to the ISA
Variants 2+3: “late” and “immediate miss detection”

- **[V2]** Do [V1], and then conditionally branch if the specified *time* was exceeded.

- **[V3]** Do [V1], but if the specified *time* is exceeded during execution of the block, branch immediately to an exception handler.
Applications of Variants 2+3
“late” and “immediate miss detection”

- [V3] “immediate miss detection”:
  - Runtime detection of missed deadlines to initiate error handling mechanisms
  - Anytime algorithms
  - However: unknown state after exception is taken

- [V2] “late miss detection”:
  - No problems with unknown state of system
  - Change parameters of algorithm to meet future deadlines
PRET Assembly Instructions
Supporting these Four Capabilities

**set_time** %r, <val>
- loads current time + <val> into %r

**delay_until** %r
- stall until current time >= %r

**branch_expired** %r, <target>
- branch to target if current time > %r

**exception_on_expire** %r, <id>
- arm processor to throw exception <id> when current time > %r

**deactivate_exception** <id>
- disarm the processor for exception <id>
Controlled Timing in Assembly Code

[V1] Delay until:

```assembly
set_time r1, 1s
// Code block
delay_until r1
```

[V2] Late miss detection

```assembly
set_time r1, 1s
// Code block
branch_expired r1, <target>
delay_until r1
```

[V3] Immediate miss detection

```assembly
set_time r1, 1s
exception_on_expire r1, 1
// Code block
deactivate_exception 1
delay_until r1
```

[V2] + [V3] could all have a variant that does not control the minimum execution time of the block of code, but only controls the maximum.
Application: Timed Loops

Fixed Period

```c
set_time r1, 1s
loop:
    // Code block
    delay_until r1
    r1 = r1 + 1s
b loop
```

Lower bound for each iteration

```c
set_time r1, 1s
loop:
    // Code block
    delay_until r1
    set_time r1, 1s
    b loop
```

The two loops above have different semantics:
Timed Loop with Exception Handling

Exact execution time (no jitter)

```
set_time r1, 1s
exception_on_expire r1, 0
loop:
  // Code block
  deactivate_exception 0
delay_until r1
  r1 = r1 + 1s
exception_on_expire r1, 0
b loop
```

This code takes exactly 1 second to execute each iteration. If an iteration takes more than 1 second, then as soon as its time expires, the iteration is aborted and an exception handler is activated.
Exporting the Timed Semantics to a Low-Level Language (like C)

```c
tryin (500ms) {
   // Code block
} expired {
   patchup();
}
```

```c
set_time r1, 500ms
// Code block
branch_expired r1, patchup
```

This realizes variant 2, “late miss detection.”

The code block will execute to completion. If 500ms have passed, then the patchup procedure will run.
Exporting the Timed Semantics to a Low-Level Language (like C)

```
jmp_buf buf;

if ( !setjmp(buf) ){
    set_time r1, 500ms
    exception_on_expire r1, 0
    // Code block
    deactivate_exception 0
} else {
    panic();
}

exception_handler_0 () {
    longjmp(buf)
}
```

This pseudo-code is neither C-level nor assembly, but is meant to explain an assembly-level implementation.
Variant with Exact Execution Times:

```c
tryfor (500ms) {
    // Code block
    panic();
}
catch {
    panic();
}
```

This is the same, except for the added `delay_until`
MTFD – Meet the F(inal) Deadline

- Variant [V1] ensure that a block of code takes \textbf{at least} a given time.
- Variants [V2, V3] allow to act upon deadline misses.
- [V4] “MTFD”: Execute a block of code taking \textbf{at most} the specified time.

Being arbitrarily “slow” is always possible and “easy”.

But what about being “fast”?

```
[V4] Exact execution:
set_time r1, 1s
// Code block
MTFD r1
delay_until r1
```
Current Timing Verification Process

C Program → Compiler → Binary → Architecture

WCET Analysis

✅ ✗
Current Timing Verification Process

- New Architecture ➔ Recertification
- Extremely time-consuming and costly
The Future (?) Timing Verification Process

- Timing is property of ISA
- Compiler can check constraints once and for all
- Downside: little flexibility in architecture development
The Future (?) Timing Verification Process: More Realistic?

- ISA leaves more freedom to implementations
- Compiler generates constraints on architecture to meet timing constraints
Conclusions

- Abstractions are great, if they are the right abstractions
- Real-time computing needs different abstractions