Time for High-Confidence Cyber-Physical Systems

Edward A. Lee
Robert S. Pepper Distinguished Professor
UC Berkeley

Invited Plenary Talk

Performance Metrics for Intelligent Systems
(PerMIS'12) Workshop
University of Maryland

Key Collaborators on work shown here:

• Steven Edwards
• Jeff Jensen
• Sungjun Kim
• Isaac Liu
• Slobodan Matic
• Hiren Patel
• Jan Reinke
• Sanjit Seshia
• Mike Zimmer
• Jia Zou
Cyber-Physical Systems (CPS): Orchestrating networked computational resources with physical systems

Automotive
- E-Corner, Siemens
- Daimler-Chrysler

Military systems:

Power generation and distribution
- Courtesy of General Electric

Building Systems

Telecommunications
- Courtesy of Doug Schmidt

Avionics
- Courtesy of Kuka Robotics Corp.

Transportation (Air traffic control at SFO)

Factory automation

Instrumentation (Soleil Synchrotron)
Claim

For CPS, programs do not adequately specify behavior.

Corollary: Performance of program execution may not be a good metric.
A Story

The Boeing 777 was Boeing’s first fly-by-wire aircraft, controlled by software. It is deployed, appears to be reliable, and is succeeding in the marketplace. Therefore, it must be a success. However...

Boeing was forced to purchase and store an advance supply of the microprocessors that will run the software, sufficient to last for the estimated 50 year production run of the aircraft and another many years of maintenance.

Why?
Lesson from this example:

Apparently, the software does not specify the behavior that has been validated and certified!

Unfortunately, this problem is very common, even with less safety-critical, certification-intensive applications. Validation is done on complete system implementations, not on software.
Structure of a Cyber-Physical System

Problems that complicate analysis of system behavior:

- Sensors may be locked out for an indeterminate amount of time
- Platforms’ measurements of time differ
- Messages from different sources interleave nondeterministically
- Variability of execution times affects results (not just WCET)
- A fault in a remote component may disrupt a critical local activity
- A fault in a remote component may go undetected for a long time
- Interrupt-driven I/O disrupts timing
- Etc…
A Key Challenge: Timing is not Part of Software Semantics

Correct execution of a program in C, C#, Java, Haskell, OCaml, etc. has nothing to do with how long it takes to do anything. All our computation and networking abstractions are built on this premise.

Programmers have to step outside the programming abstractions to specify timing behavior.
The first edition of Hennessy and Patterson (1990) revolutionized the field of computer architecture by making performance metrics the dominant criterion for design.

Today, for computers, timing is merely a performance metric.

It needs to be a correctness criterion.
Execution-time analysis, by itself, does not solve the problem!

Analyzing software for timing behavior requires:

- Paths through the program (undecidable)
- Detailed model of microarchitecture
- Detailed model of the memory system
- Complete knowledge of execution context
- Many constraints on preemption/concurrency
- Lots of time and effort

And the result is valid only for that exact hardware and software!

Fundamentally, the ISA of the processor has failed to provide an adequate abstraction.

Part 1: P Reed Machines

- **PREcision-Timed processors** = PRET
- **Predictable, REpeatable Timing** = PRET
- **Performance with REpeatable Timing** = PRET

```java
// Perform the convolution.
for (int i=0; i<10; i++) {
    x[i] = a[i]*b[i-j];
    // Notify listeners.
    notify(x[i]);
}
```
Dual Approach

- Rethink the ISA
  - Timing has to be a *correctness* property not a *performance* property.

- Implementation has to allow for multiple realizations and efficient realizations of the ISA
  - Repeatable execution times
  - Repeatable memory access times
Example of one sort of mechanism we would like:

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

If the code block takes longer than 500ms to run, then the `panic()` procedure will be invoked.

But then we would like to verify that `panic()` is never invoked!

```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)
}
```

Pseudocode showing how this might be implemented today. The result is very platform dependent.
Extending an ISA with Timing Semantics

[V1] Best effort:

\[
\text{set\_time } r1, 1s
\]

// Code block

\[
\text{delay\_until } r1
\]

[V2] Late miss detection

\[
\text{set\_time } r1, 1s
\]

// Code block

\[
\text{branch\_expired } r1, <\text{target}>\]

\[
\text{delay\_until } r1
\]

[V3] Immediate miss detection

\[
\text{set\_time } r1, 1s
\]

// Code block

\[
\text{exception\_on\_expire } r1, 1
\]

// Code block

\[
\text{deactivate\_exception } 1
\]

\[
\text{delay\_until } r1
\]

[V4] Exact execution:

\[
\text{set\_time } r1, 1s
\]

// Code block

\[
\text{MTFD } r1
\]
To provide timing guarantees, we need implementations that deliver repeatable timing.

Fortunately, electronics technology delivers highly reliable and precise timing…

… but the overlaying software abstractions discard it. Chip architects heavily exploit the lack of temporal semantics.

```c
// Perform the convolution.
for (int i=0; i<10; i++) {
    x[i] = a[i] * b[j-i];
    // Notify listeners.
    notify(x[i]);
}
```
To deliver repeatable timing, we have to rethink the microarchitecture

Challenges:

- Pipelining
- Memory hierarchy
- I/O (DMA, interrupts)
- Power management (clock and voltage scaling)
- On-chip communication
- Resource sharing (e.g. in multicore)
Our Current PRET Architecture

*PTArm*, a soft core on a Xilinx Virtex 5 and 6 FPGA

- Hardware thread registers
- Interleaved pipeline with one set of registers per thread
- scratch pad
- SRAM scratchpad shared among threads
- memory
- DRAM main memory, separate banks per thread
- I/O devices
Status of the PRET project

- **Results:**
  - PTArm implemented on Xilinx Virtex 5 FPGA.
  - UNISIM simulator of the PTArm facilitates experimentation.
  - DRAM controller with repeatable timing and DMA support.
  - PRET-like utilities implemented on COTS Arm.

- **Much still to be done:**
  - Realize MTFD, interrupt I/O, compiler toolchain, scratchpad management, etc.
A Key Next Step: Parametric PRET Architectures

ISA that admits a variety of implementations:

- Variable clock rates and energy profiles
- Variable number of cycles per instruction
- Latency of memory access varying by address
- Varying sizes of memory regions
- …

A given program may meet deadlines on only some realizations of the same parametric PRET ISA.
Realizing the MTFD instruction on a parametric PRET machine

The goal is to make software that will run correctly on a variety of implementations of the ISA, and that correctness can be checked for each implementation.
PRET Publications


Part 2: How to get the Source Code?

The input (mostly likely C) will ideally be generated from a model, like Simulink or SCADE. The model specifies temporal behavior at a higher level than code blocks, and it specifies a concurrency model that can limit preemption points. However, Simulink and SCADE have naïve models of time.
Ptides: Programming model for distributed real-time systems, using time-stamped messages.
A CPS Problem and Potential Pitides
Application: Printing Press

- Application aspects
  - local (control)
  - distributed (coordination)
  - global (modes)
- Open standards (Ethernet)
  - Synchronous, Time-Triggered
  - IEEE 1588 time-sync protocol
- High-speed, high precision
  - Speed: 1 inch/ms
  - Precision: 0.01 inch
  -> Time accuracy: 10us

Bosch-Rexroth

Goal: Orchestrated networked resources built with sound design principles on suitable abstractions
Example – Flying Paster

Source: http://offsetpressman.blogspot.com/2011/03/how-flying-paster-works.html
Source: http://offsetpressman.blogspot.com/2011/03/how-flying-paster-works.html

Flying Paster
What this needs is *correct* timing, not *high performance*.

Simulation

Renesas

XMOS

Contact (red), Top Dead Center (green), Cut (blue) and Arm (black)

Oscilloscope traces on GPIO pins
We have demonstrated platform independent timing for programs automatically generated from model.

Simulation

Renesas

XMOS

Contact (red), Top Dead Center (green), Cut (blue) and Arm (black)

Oscilloscope traces on GPIO pins

Lee, Berkeley 27
Ptides Publications


http://chess.eecs.berkeley.edu/ptides/
Implications for Performance Metrics

- Performance metrics for computing have been oversimplified:
  - Minimize execution time on standard benchmarks.
  - Minimize energy or power on standard benchmarks.

- Ideas for revised performance metrics:
  - Timing precision (or variability) at I/O connections.
  - Precision and reliability of time synchronization.
  - Minimize energy subject to meeting timing constraints.
  - Timing variability across platform implementations.

This will require new benchmarks!
Conclusions

Today, timing behavior is a property only of *realizations* of software systems.

Tomorrow, timing behavior will be a semantic property of *programs* and *models*.

Raffaello Sanzio da Urbino – The Athens School

Overview References: