Controlling Timing vs. Measuring Timing

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

Keynote

Workshop on Suite of Embedded Applications and Kernels (SEAK)
Design Automation Conference (DAC)
June 7, 2015.
San Francisco, CA

Special Thanks to:
• David Broman
• Isaac Liu
• Hiren Patel
• Jan Reineke
• Michael Zimmer
Abstract

Embedded systems differ from many general purpose and scientific computing applications in that repeatability is often more important than performance. The goal in an embedded system is not usually to get a task done as soon as possible, but rather to get a task done reliably, on time, and with minimal energy consumption. Benchmarking, however, is typically focused on performance, not repeatability. In this talk, I will argue that when the primary goal is repeatability, design decisions can be very different, and benchmarking needs to be done differently. I will describe the Berkeley PRET project, which shows that embedded processors can be designed to deliver repeatable timing, and with appropriately adjusted measures, with no loss in performance.
The Context for this Talk: Cyber-Physical Systems or The Internet of Important Things (IoIT)

Leveraging Internet technology in cyber-physical systems.

Challenges:

- **Isolated networks** are reliable, predictable, and controllable. But they lose the benefits of connectedness.
- **Safety** is the most critical design requirement.
- **Security** is essential, particularly w.r.t. how it impacts safety.
- **Privacy** (protection of data) is required.

This Bosch Rexroth printing press is a cyber-physical factory using Ethernet and TCP/IP with high-precision clock synchronization (IEEE 1588) on an isolated LAN.
A Key Characteristic of Such Systems

Desired behavior is well defined:

“Print high quality books and remain running 24/7.”

In view of this, getting higher performance from a microprocessor does not necessarily help.
In fact, it can hurt by causing unexpected interactions with other parts of the system.

*Repeatability becomes more important than performance.*
Cyber-Physical Systems (CPS):
Orchestrating networked computational resources with physical systems

Automotive

Military systems:

Building Systems

Power generation and distribution

Biomedical

Factory automation

Instrumentation (Soleil Synchrotron)
Schematic of a simple CPS
Repeatability requires

Determinacy

or

The same inputs yield the same outputs.
Is Determinism Achievable? Sources of Nondeterminism

- Physical noise
- Imperfect actuation
- Parts failures
- Unknowable execution times
- Unknowable delays
- Uncontrollable scheduling
In the face of such nondeterminism, does it make sense to talk about deterministic models for cyber-physical systems?
Models vs. Reality


You will never strike oil by drilling through the map!

Solomon Wolf Golomb (1932) mathematician and engineer and a professor of electrical engineering at the University of Southern California. Best known to the general public and fans of mathematical games as the inventor of polyominoes, the inspiration for the computer game Tetris. He has specialized in problems of combinatorial analysis, number theory, coding theory and communications.
But this does not, in any way, diminish the value of a map!
The Kopetz Principle

Many (predictive) properties that we assert about systems (determinism, timeliness, reliability, safety) are in fact not properties of an implemented system, but rather properties of a model of the system.

We can make definitive statements about models, from which we can infer properties of system realizations. The validity of this inference depends on model fidelity, which is always approximate.

(paraphrased)
Deterministic Models of Nondeterministic Systems

Physical System

Model

Synchronous digital logic

Image: Wikimedia Commons
Deterministic Models of Nondeterministic Systems

Physical System

Model

<table>
<thead>
<tr>
<th>Integer Register-Register Operations</th>
</tr>
</thead>
</table>

RISC-V defines several arithmetic R-type operations. All operations read the \( rs1 \) and \( rs2 \) registers as source operands and write the result into register \( rd \). The \( funct \) field selects the type of operation.

<table>
<thead>
<tr>
<th>31</th>
<th>27</th>
<th>26</th>
<th>22</th>
<th>21</th>
<th>17</th>
<th>16</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>rd</td>
<td>rs1</td>
<td>rs2</td>
<td></td>
<td></td>
<td>funct10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dest</td>
<td>src1</td>
<td>src2</td>
<td>ADD/SUB/SLT/SLTU</td>
<td>OP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dest</td>
<td>src1</td>
<td>src2</td>
<td>AND/OR/XOR</td>
<td>OP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dest</td>
<td>src1</td>
<td>src2</td>
<td>SLL/SRL/SRA</td>
<td>OP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dest</td>
<td>src1</td>
<td>src2</td>
<td>ADDW/SUBW</td>
<td>OP-32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dest</td>
<td>src1</td>
<td>src2</td>
<td>SLLW/SRLW/SRAW</td>
<td>OP-32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


Instruction Set Architectures (ISAs)
Deterministic Models of Nondeterministic Systems

Physical System

Model

```java
/** Reset the output receivers, which are the inside receivers of the output ports of the container. * @throws IllegalActionException If getting the receivers fails. */
private void _resetOutputReceivers() throws IllegalActionException {
    List<IOPort> outputs = ((Actor) getContainer()).outputPortList();
    for (IOPort output : outputs) {
        if (_.debugging) {
            _.debug("Resetting inside receivers of output port: " + output.getName());
        }
        Receiver[] receivers = output.getInsideReceivers();
        if (receivers != null) {
            for (int i = 0; i < receivers.length; i++) {
                if (receivers[i] != null) {
                    for (int j = 0; j < receivers[i].length; j++) {
                        if (receivers[i][j].instanceOf FSMReceiver) {
                            receivers[i][j].reset();
                        }
                    }
                }
            }
        }
    }
}
```

Single-threaded imperative programs

Lee, Berkeley
Deterministic Models of Nondeterministic Systems

Physical System

Model

\[ \dot{x}(t) = \dot{x}(0) + \frac{1}{M} \int_{0}^{t} F(\tau) d\tau \]

Differential Equations

Image: Wikimedia Commons
A Major Problem for CPS:
Combinations of these Models are Nondeterministic

/** Reset the output receivers, which are the inside receivers of
 * the output ports of the container.
 * @exception IllegalActionException If getting the receivers fails.
 */
private void _resetOutputReceivers() throws IllegalActionException {
    List<IOPort> outputs = ((Actor) getContainer()).outputPortList;
    for (IOPort output : outputs) {
        if (_debugging) {
            _debug("Resetting inside receivers of output port: ",
                      + output.getName());
        }
        Receiver[][] receivers = output.getInsideReceivers();
        if (receivers != null) {
            for (int i = 0; i < receivers.length; i++) {
                if (receivers[i] != null) {
                    for (int j = 0; j < receivers[i].length; j++) {
                        if (receivers[i][j] instanceof FSMReceiver) {
                            receivers[i][j].reset();
                        }
                    }
                }
            }
        }
    }
}

Image: Wikimedia Commons

\[
\dot{x}(t) = \dot{x}(0) + \frac{1}{M} \int_{0}^{t} F(\tau) d\tau
\]
A Key Challenge: 
Timing is not Part of Software Semantics

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

Programmers have to step *outside* the programming abstractions to specify timing behavior.

Programmers have no map!
The Model

\[ \dot{x}(t) = \dot{x}(0) + \frac{1}{M} \int_{0}^{t} F(\tau) d\tau \]
The Reality

Computational Platform → Network Fabric → Computational Platform

Physical plant
The Model is not much more deterministic than the reality.

The modeling languages have disjoint, incompatible semantics.
System dynamics emerges from the physical realization

... leading to a “prototype and test” style of design
Computer Science has not completely ignored timing...

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.
Benchmarks are inherently about *performance metrics* and not about *correctness criteria*
Correctness criteria

We can safely assert that line 8 does not execute

(In C, we need to separately ensure that no other thread or ISR can overwrite the stack, but in more modern languages, such assurance is provided by construction.)

```
void foo(int32_t x) {
    if (x > 1000) {
        x = 1000;
    }
    if (x > 0) {
        x = x + 1000;
        if (x < 0) {
            panic();
        }
    }
}
```

We can develop absolute confidence in the software, in that only a hardware failure is an excuse.

But not with regards to timing!!
The hardware out of which we build computers is capable of delivering “correct” computations and precise timing…

The synchronous digital logic abstraction removes the messiness of transistors.

… but the overlaying software abstractions discard the timing precision.

```
// Perform the convolution.
for (int i=0; i<10; i++) {
    x[i] = a[i]*b[j-i];
    // Notify listeners.
    notify(x[i]);
}
```
PRET Machines – Giving Programs the Capabilities their Hardware Already Has.

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

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

http://chess.eecs.berkeley.edu/pret

Lee, Berkeley
Major Challenges
and existence proofs that they can be met

- Pipelines
  - fine-grain multithreading
- Memory hierarchy
  - memory controllers with controllable latency
- I/O
  - threaded interrupts, with bounded effects on timing
PRET Publications

PRET ISA Realizations:
- PRET1, Sparc-based  
  - [Lickly et al., CASES, 2008]
- PTARM, ARM-based  
  - [Liu et al., ICCD, 2012]
- FlexPRET, RISC-V-based  
  - [Zimmer et al., RTAS, 2014]

PRET Principle:
- The case for PRET  
  - [Edwards & Lee, DAC 2007]
- PRET ISA extensions  
  - [Edwards at al., ICCD 2009]
- Temporal isolation  
  - [Bui et al., DAC, 2011]
- Design challenges  
  - [Broman et al., ESLsyn, 2013]
- Cyber-physical systems  
  - [Lee., Sensors, 2015]

PRET Applications:
- Control systems  
  - [Bui et al., RTCSA 2010]
- Computational fluid dynamics  
  - [Liu et al., FCCM, 2012]

PRET for Security:
- Eliminating side-channel attacks  
  - [Lie & McGrogan, Report 2009]

PRET Memory Systems:
- DRAM controller  
  - [Reineke et al., CODES+ISSS 2011]
- Scratchpad management  
  - [Kim et al., RTAS, 2014]
- Mixed criticality DRAM controller  
  - [Kim et al., RTAS 2015]
Major Challenges, Yes, but Leading to Major Opportunities

- Improved determinism
- Better testability
- Reduced energy consumption
- Reduced overdesign (cost, weight)
- Improved confidence and safety
- Substitutable hardware
Three Generations of PRET Machines at Berkeley

- PRET1, Sparc-based (simulation only)
  - [Lickly et al., CASES, 2008]

- PTARM, ARM-based (FPGA implementation)
  - [Liu et al., ICCD, 2012]

- FlexPRET, RISC-V-based (FPGA + simulation)
  - [Zimmer et al., RTAS, 2014]
Our Second Generation PRET

*PTArm*, a soft core on a
Xilinx Virtex 5 FPGA (2012)

- Hardware thread
- 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
How does this perform on benchmarks?

It depends on how you read the results!

Conventional reading: terribly!!

Single-thread performance on adpcm Mälardalen benchmark is 3.2x slower.

Lee, Berkeley

How does this perform on benchmarks?

An alternative reading:

Aggregate performance on adpcm Mälardalen benchmark is 20% better. Pipeline bubbles are eliminated. Requires concurrent workloads

Lee, Berkeley

How does this perform on benchmarks?

An an even better reading:

Pipeline is simpler, hence be clocked faster.

Timing can be controlled, so synchronization overhead can be eliminated.

How does this perform on benchmarks?

A still better reading:

Timing can be controlled, so clock rate (and hence energy consumption) can be reduced to the bare minimum.

Hardware costs can be much lower than a conventional design!

Lee, Berkeley

Our Third-Generation PRET: Open-Source FlexPRET (Zimmer et al., 2014)

- 32-bit, 5-stage thread interleaved pipeline, RISC-V ISA
  - **Hard real-time HW threads:** scheduled at constant rate for isolation and predictability
  - **Soft real-time HW threads:** share all available cycles (e.g. HW thread sleeping) for efficiency
- Deployed on Xilinx FPGA (area comparable to Microblaze)

![Diagram showing thread scheduling]

- Whenever cycle available (arbitrary interleaving)
- Every 3 cycles (unless done)

Digilent Atlys (Spartan 6) and NI myRIO (Zync)
FlexPRET

Hard-Real-Time (HRT) Threads
Interleaved with Soft-Real-Time (SRT) Threads

HRT threads have deterministic timing.
SRT threads share available cycles

SRAM scratchpad shared among threads

DRAM main memory provides deterministic latency for HRT threads.
Conventional behavior for the rest.

Michael Zimmer
Interrupt-Driven I/O is notorious for disrupting timing

Interrupts have no effect on HRT threads, and bounded effect on SRT threads!
FlexPRET Shows:

- Not only is there no performance cost for appropriate workloads, but there is also no performance cost for inappropriate workloads!
- Pipelining, memory hierarchy, and interrupt-driven I/O can all be done without losing timing determinacy!

**Conventional use of benchmarking instead shows no benefit!**
Software
Example of one sort of mechanism we can achieve (with some difficulty!) today:

```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!

```
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 to do this today.
Extending an ISA with Timing Semantics

[V1] Best effort:

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

[V2] Late miss detection

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

[V3] Immediate miss detection

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

[V4] Exact execution:

```plaintext
set_time r1, 1s
// Code block
MTFD r1
```
But Wait…

The whole point of an ISA is that the same program does the same thing on multiple hardware realizations.

*Isn’t this incompatible with deterministic timing?*
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 many implementations of the ISA, and that correctness can be checked for each implementation.
How to Make PRET Widespread? Real-Time Units (RTUs)

- Offload timing-critical functions to the RTU
  - Compare with dedicated hardware
- Software peripherals
  - Bit-banging for custom protocols
- Software API: OpenRT?
  - Richer interface for smart sensors/actuators
PRET Enables *Deterministic Interaction* Between the Cyber and the Physical

\[
\dot{\mathbf{x}}(t) = \dot{\mathbf{x}}(0) + \frac{1}{M} \int_0^t \mathbf{F}(\tau) d\tau
\]
But what about interaction over the Network?

We have also developed *deterministic models* for distributed real-time software, using a technique called PTIDES.
PTIDES: Programming Temporally Integrated Distributed Embedded Systems

See

http://chess.eecs.berkeley.edu/ptides

(or invite me back next year)
One Last Comment…

Model Fidelity

- In **science**, a good model matches well the behavior of the physical world.
- In **engineering**, a good physical implementation matches well the behavior of the model.

*In engineering, model fidelity is a two-way street!*  

*For a model to be useful, it is necessary (but not sufficient) to be able to construct a faithful physical realization.*
A Model
A Physical Realization
Model Fidelity

- To a *scientist*, the model is flawed.

- To an *engineer*, the physical realization is flawed.

I’m an engineer…
PRET offers better models with less flawed physical realizations.
Determinism?

For a model to be useful, it is necessary (but not sufficient) to be able to construct a faithful physical realization.

bullet The real world is highly unpredictable.
bullet So, are deterministic models useful?
  ● Is synchronous digital logic useful?
  ● Are ISAs useful?
  ● Single-threaded imperative programs?
  ● Differential equations?
Determinism?

Deterministic models do not eliminate the need for robust, fault-tolerant designs.

In fact, they *enable* such designs, because they make it much clearer what it means to have a fault!
**PRET Publications**

**PRET ISA Realizations:**
- PRET1, Sparc-based
  - [Lickly et al., CASES, 2008]
- PTARM, ARM-based
  - [Liu et al., ICCD, 2012]
- FlexPRET, RISC-V-based
  - [Zimmer et al., RTAS, 2014]

**PRET ISA Realizations:**
- PRET2, Sparc-based
  - [Lickly et al., CASES, 2008]
- PTARM, ARM-based
  - [Liu et al., ICCD, 2012]
- FlexPRET, RISC-V-based
  - [Zimmer et al., RTAS, 2014]

**PRET Applications:**
- Control systems
  - [Bui et al., RTCSA 2010]
- Computational fluid dynamics
  - [Liu et al., FCCM, 2012]

**PRET for Security:**
- Eliminating side-channel attacks
  - [Lie & McGrogan, Report 2009]

**PRET Principle:**
- The case for PRET
  - [Edwards & Lee, DAC 2007]
- PRET ISA extensions
  - [Edwards et al., ICCD 2009]
- Temporal isolation
  - [Bui et al., DAC, 2011]
- Design challenges
  - [Broman et al., ESLsyn, 2013]
- Cyber-physical systems
  - [Lee., Sensors, 2015]
Conclusions

For IoT and embedded applications, repeatability may be more important than performance.

Today, over emphasis and/or misuse of benchmarking can be an obstacle to innovation in this dimension.