Repeatable Timing in Software and Networks

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

ESWEEK 2011 Tutorial on
Time-Predictable and Composable Architectures for Dependable Embedded Systems
ESWeek
Taipei, Taiwan, Oct. 9, 2011

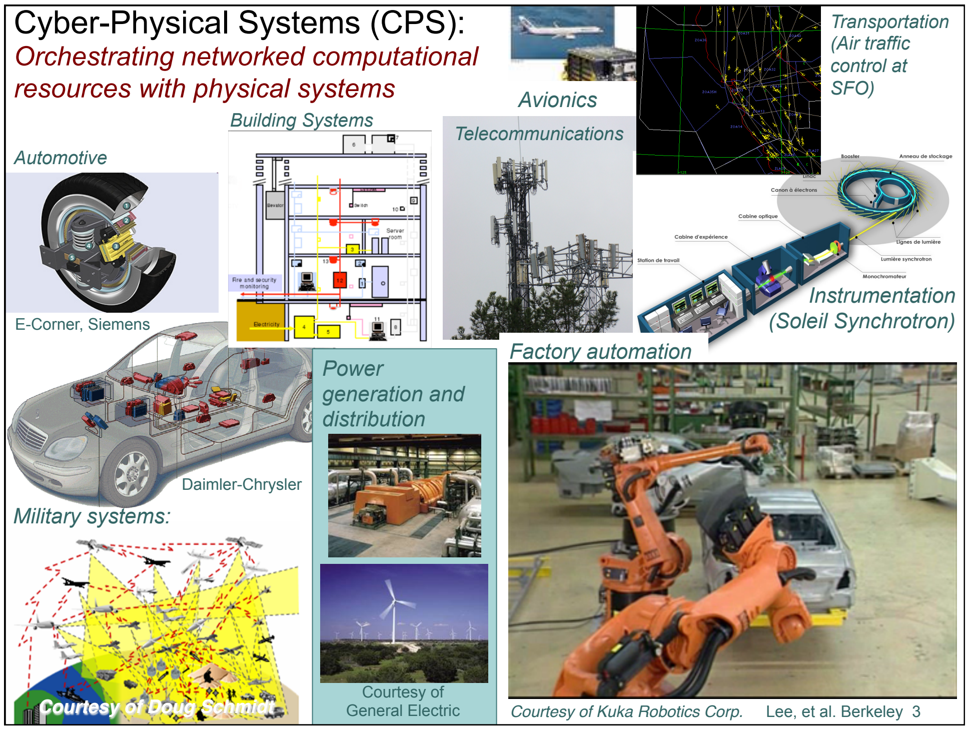
Key Collaborators on work shown here:

- *Stephen Edwards*
- *Sungjun Kim*
- *Isaac Liu*
- *Hiren Patel*
- *Jan Reinke*
- *Sanjit Seshia*
- *Mike Zimmer*

Abstract

All widely used software abstractions lack temporal semantics. The notion of correct execution of a program written in every widely-used programming language today does not depend on the temporal behavior of the program. But temporal behavior matters in almost all systems. Even in systems with no particular real-time requirements, timing of programs is relevant to the value delivered by programs, and in the case of concurrent programs, also affects the functionality. In systems with real-time requirements, such as most cyber-physical systems, temporal behavior affects not just the value delivered by a system but also its correctness.

In this talk, we will argue that time can and must become part of the semantics of programs for a large class of applications. To illustrate that this is both practical and useful, we will describe two recent efforts at Berkeley in the design and implementation of timing-centric software systems. On the design side, we will describe PTIDES, a programming model for distributed real-time systems. PTIDES rests on a rigorous semantics of discrete-event systems and reflects the realities in distributed real-time, where measuring the passage of time is imperfect. PTIDES enables deterministic time-sensitive distributed actions. It relies on certain assumptions about networks that are not trivial (time synchronization with bounded error and bounded latency), but which have been shown in some contexts to be achievable and economical. PTIDES is also robust to subsystem failures, and, perhaps most interestingly, provides a semantic basis for detecting such failures at the earliest possible time. On the implementation side, we will describe PRET machines, which redefine the instruction-set architecture (ISA) of a microprocessor to include temporal semantics.



Cyber-Physical Systems (CPS):

Orchestrating networked computational resources with physical systems

Automotive

E-Corner, Siemens

Building Systems

Avionics

Telecommunications

Transportation (Air traffic control at SFO)

Instrumentation (Soleil Synchrotron)

Daimler-Chrysler

Power generation and distribution

Courtesy of General Electric

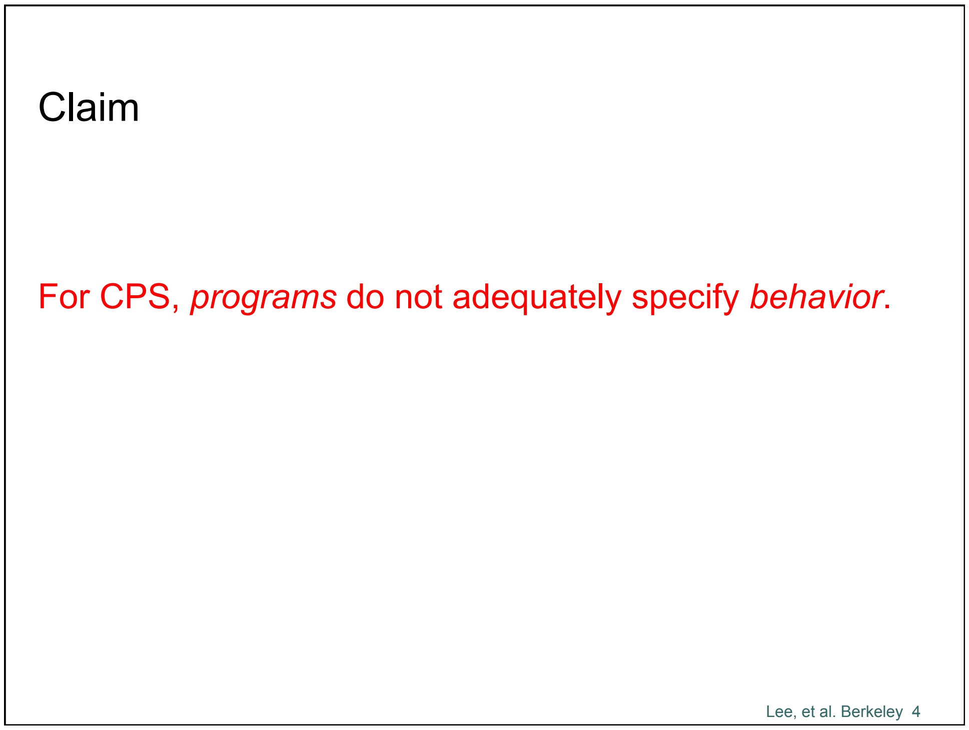
Factory automation

Military systems:

Courtesy of Doug Schmidt

Courtesy of Kuka Robotics Corp.

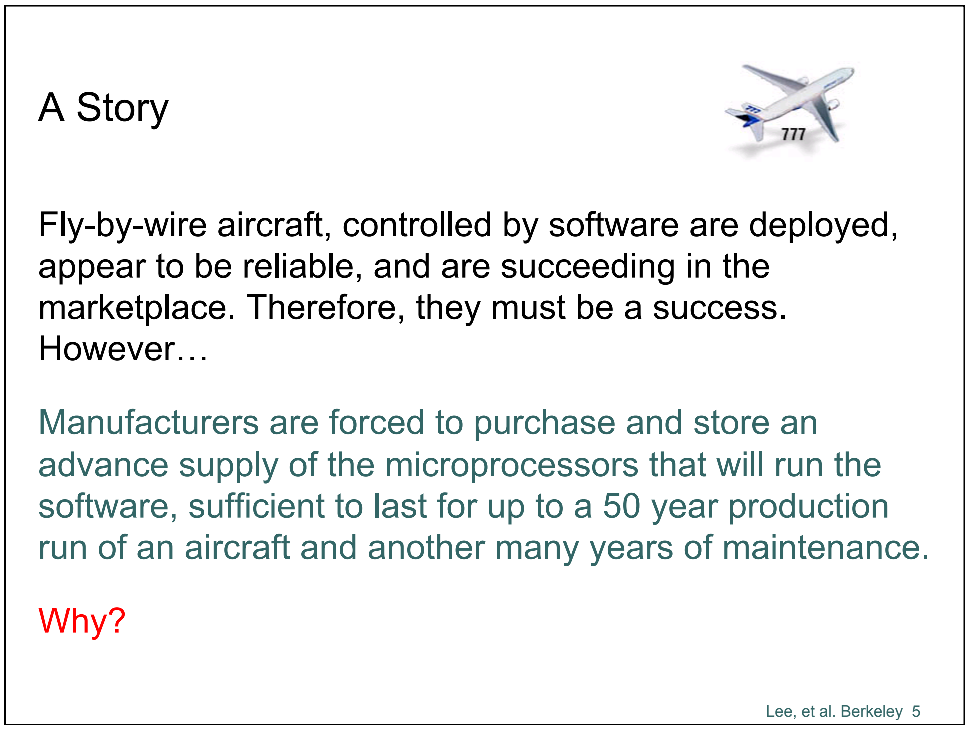
Lee, et al. Berkeley 3



Claim

For CPS, *programs* do not adequately specify *behavior*.

Lee, et al. Berkeley 4



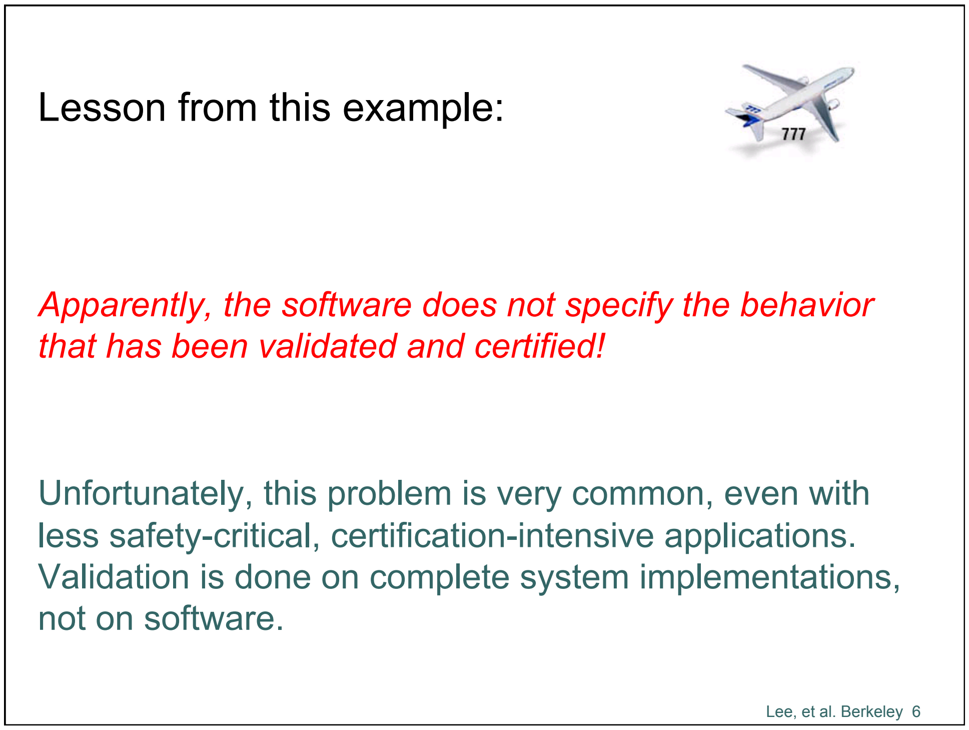
A Story

Fly-by-wire aircraft, controlled by software are deployed, appear to be reliable, and are succeeding in the marketplace. Therefore, they must be a success. However…

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

Why?

Lee, et al. Berkeley 5



Lesson from this example:

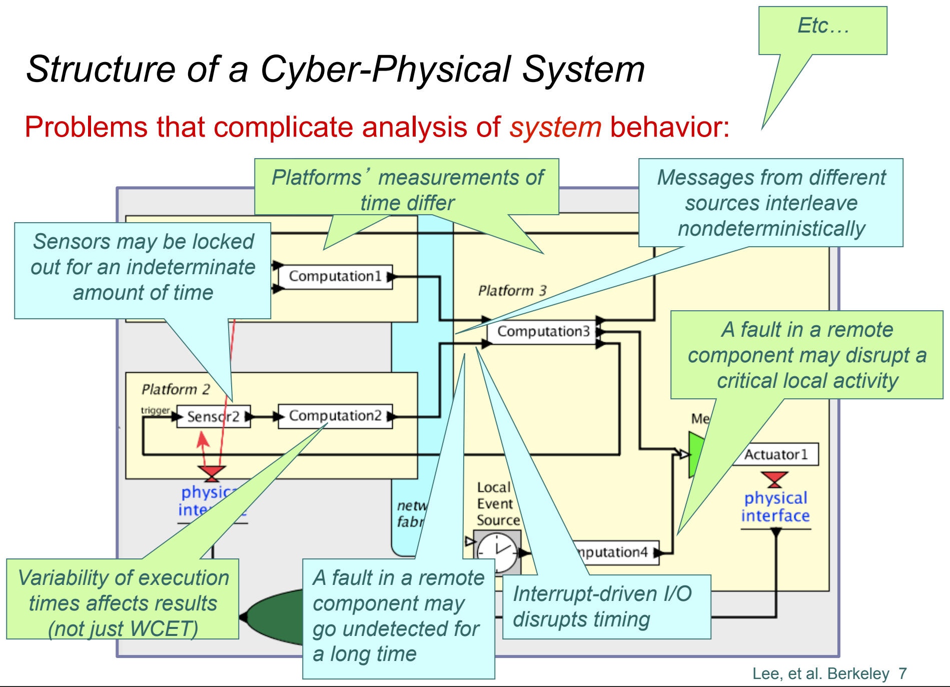
Apparently, the software does not specify the behavior that has been validated and certified!

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

Lee, et al. Berkeley 6

Structure of a Cyber-Physical System

Problems that complicate analysis of *system* behavior:

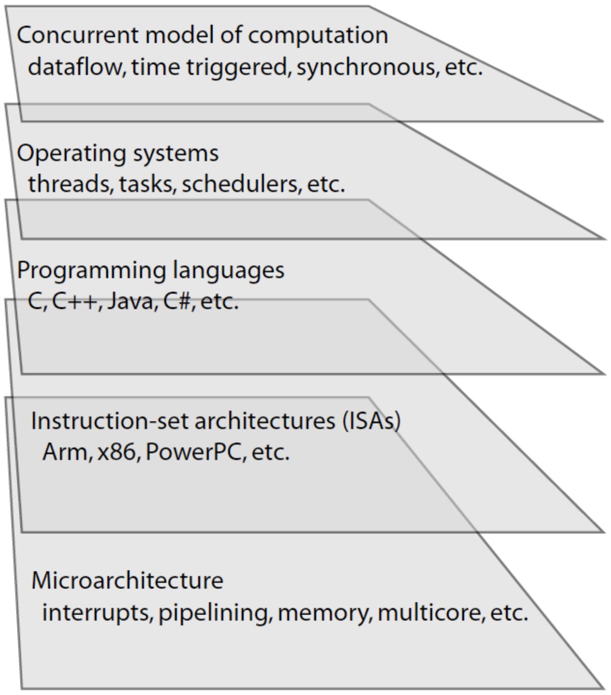

Etc…
Platforms' measurements of time differ
Messages from different sources interleave nondeterministically
Sensors may be locked out for an indeterminate amount of time
A fault in a remote component may disrupt a critical local activity
Variability of execution times affects results (not just WCET)
A fault in a remote component may go undetected for a long time
Interrupt-driven I/O disrupts timing
Computation1
Platform 3
Computation3
Platform 2
trigger
Sensor2
Computation2
physical interface
network fabric
Local Event Source
Computation4
Actuator1
physical interface

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.

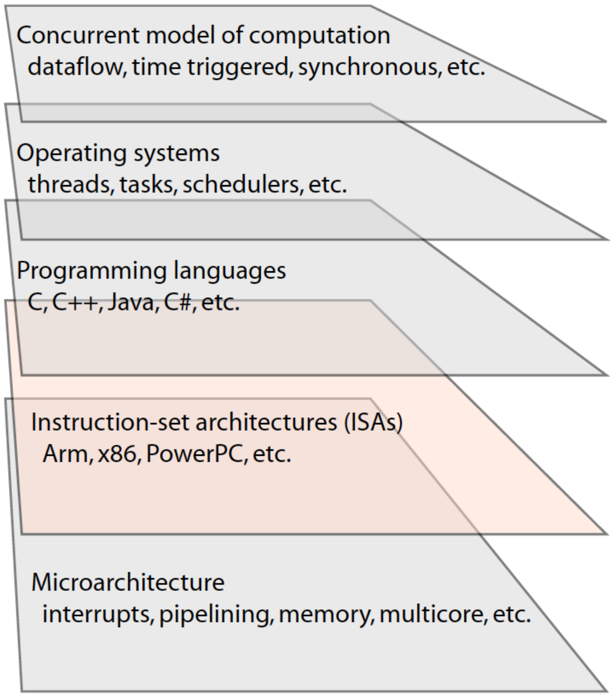
Layers of Abstraction


Concurrent model of computation
dataflow, time triggered, synchronous, etc.
Operating systems
threads, tasks, schedulers, etc.
Programming languages
C, C++, Java, C#, etc.
Instruction-set architectures (ISAs)
Arm, x86, PowerPC, etc.
Microarchitecture
interrupts, pipelining, memory, multicore, etc.

Layers of Abstraction

The time it takes to realize an instruction in an ISA is irrelevant to correctness of the realization.

Regaining temporal semantics at higher levels becomes difficult.


Concurrent model of computation
dataflow, time triggered, synchronous, etc.
Operating systems
threads, tasks, schedulers, etc.
Programming languages
C, C++, Java, C#, etc.
Instruction-set architectures (ISAs)
Arm, x86, PowerPC, etc.
Microarchitecture
interrupts, pipelining, memory, multicore, etc.

Execution-time analysis, by itself, does not solve the problem!

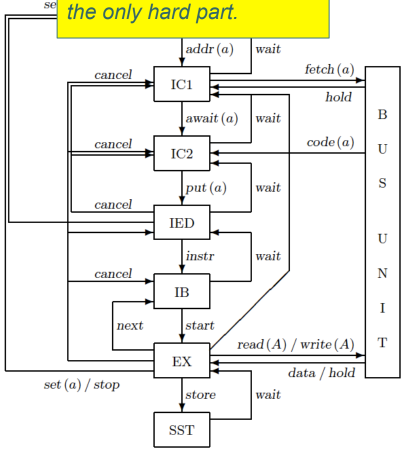
Our first goal is to reduce the problem so that this is the only hard part.

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.



Wilhelm, et al. (2008). "The worst-case execution-time problem - overview of methods and survey of tools." ACM TECS 7(3): p1-53.

Part 1: PRET Machines

- **PRE**cision-**T**imed processors = **PRET**
- **P**redictable, **RE**peatable **T**iming = **PRET**
- **P**erformance *with* **RE**peatable **T**iming = **PRET**

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

+

= PRET

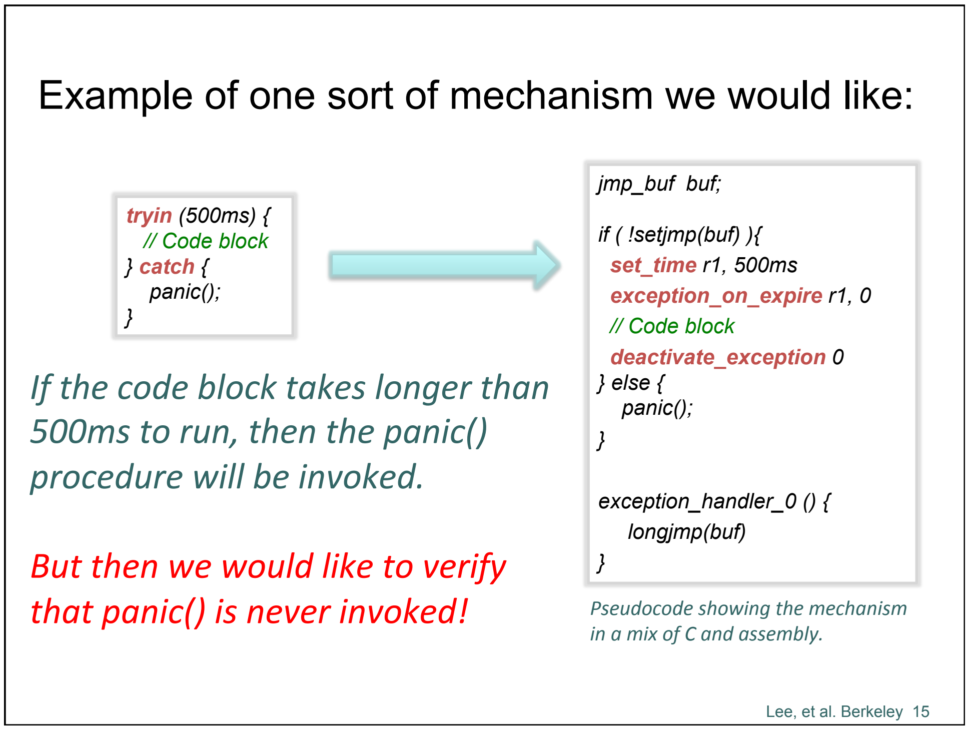
Computing *With time*

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

Related Work that has Influenced Our Thinking

1. Akesson et al., Book Chapter, 2010: *Composability and predictability for Independent application development*.
2. Barre, Rochange, and Sainrat, ARCS 2008: *A predictable simultaneous multithreading scheme for hard real-time*.
3. El-Haj-Mahmoud, Al-Zawawi, Anantaraman, and Rotenberg, CASES 2008: *Virtual multiprocessor: an analyzable, high-performance architecture for real-time computing*.
4. Mische, Uhrig, Kluge, and Ungerer, ICCD 2008: *Exploiting spare resources of in-order SMT processors executing hard real-time threads*.
5. Pitter and Schoeberl, ACM TECS 2010: *A real-time Java chip-multiprocessor*.
6. Rosen, Andrei, Eles, and Peng, RTSS 2007: *Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip*.
7. Ungerer et al. IEEE Micro 2010: *Merasa: Multicore execution of hard real-time applications supporting analyzability*.



Example of one sort of mechanism we would like:

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

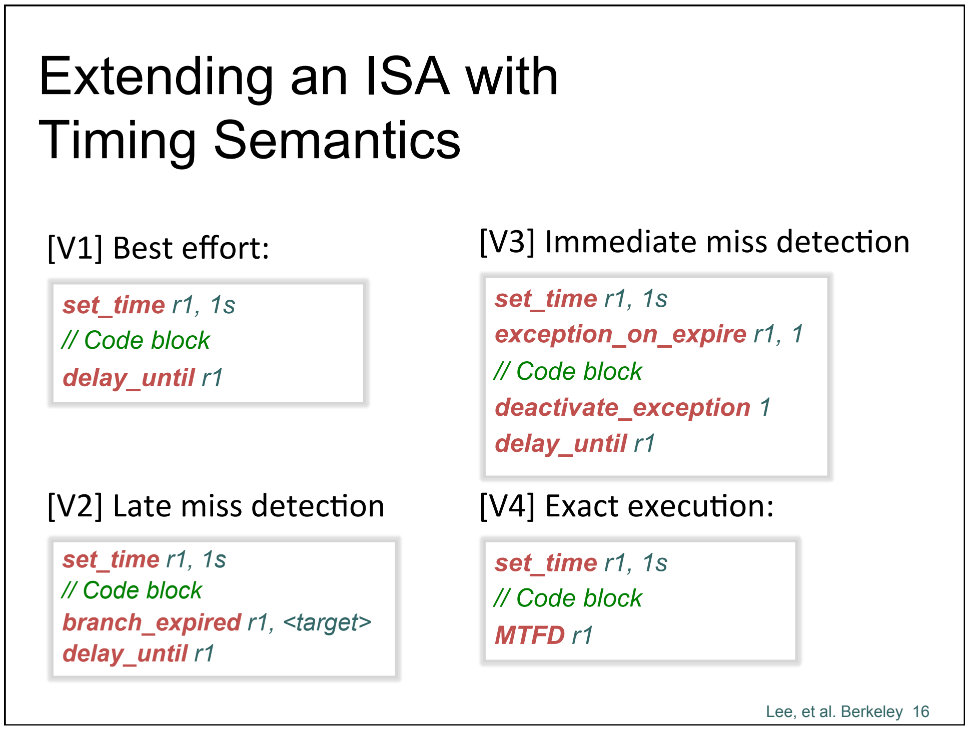
```
jmp_buf  buf;

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 the mechanism in a mix of C and assembly.

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!

Lee, et al. Berkeley 15



Extending an ISA with Timing Semantics

[V1] Best effort:

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

[V2] Late miss detection

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

[V3] Immediate miss detection

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

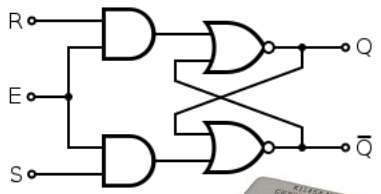
[V4] Exact execution:

```
set_time r1, 1s
// Code block
MTFD r1
```

Lee, et al. Berkeley 16

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.

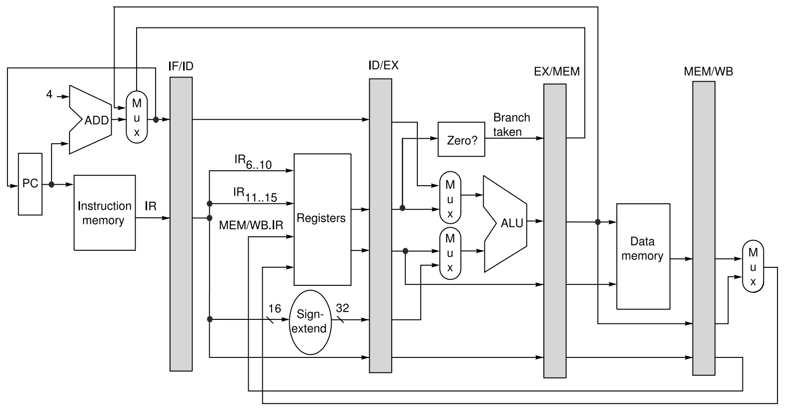
```
// Perform the convolution.
for (int i=0; i<10; i++) {
  x[i] = a[i]*b[j-i];
  // Notify listeners.
  notify(x[i]);
}
```

To deliver repeatable timing, we have to rethink the microarchitecture

Challenges:

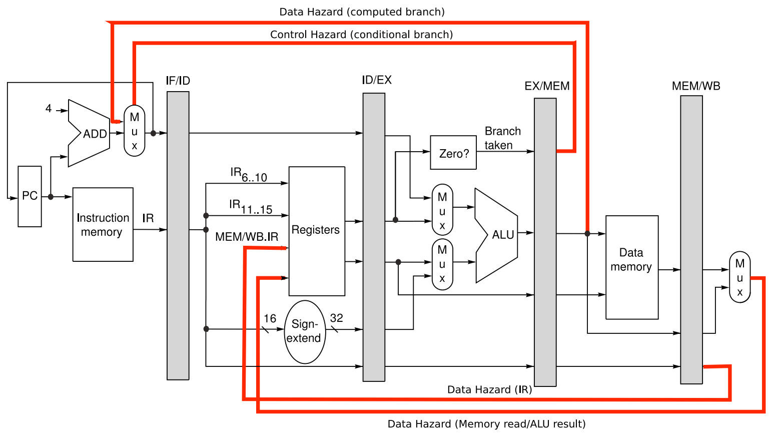
- Pipelining
- Memory hierarchy
- I/O (DMA, interrupts)
- Power management (clock and voltage scaling)
- On-chip communication
- Resource sharing (e.g. in multicore)

First Problem: Pipelining



Hennessey and Patterson, Computer Architecture: A Quantitative Approach, 4th edition, 2007.

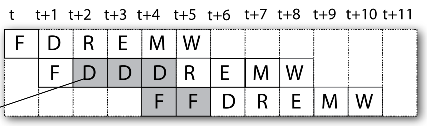
Pipeline Hazards



Hennessey and Patterson, Computer Architecture: A Quantitative Approach, 4th edition, 2007.

Pipeline Interlocks vs. Pipeline Interleaving

Traditional pipeline:

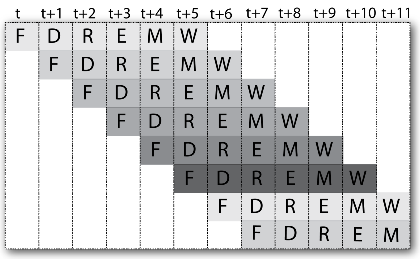


	t	t+1	t+2	t+3	t+4	t+5	t+6	t+7	t+8	t+9	t+10	t+11
T0: cmp %g2, 9	F	D	R	E	M	W						
T0: bg, a 40011b8		F	D	D	D	R	E	M	W			
T0: add %i1, %i2, %l3					F	F	D	R	E	M	W	

Stall pipeline

Dependencies result in complex timing behaviors

Thread-interleaved pipeline:



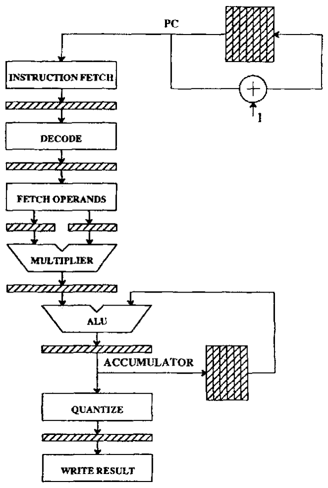
	t	t+1	t+2	t+3	t+4	t+5	t+6	t+7	t+8	t+9	t+10	t+11
T0: cmp %g2, 9	F	D	R	E	M	W						
T1: add %o0, %g1, %g2		F	D	R	E	M	W					
T2: sub %g1, %g2, %g1			F	D	R	E	M	W				
T3: bn 430011a0				F	D	R	E	M	W			
T4: ld [%fp + -12], %g1					F	D	R	E	M	W		
T5: cmp %g1, 4						F	D	R	E	M	W	
T0: bg, a 40011b8							F	D	R	E	M	W
T1: cmp %g1, 4								F	D	R	E	M

Repeatable timing behavior of instructions

Lee, et al. Berkeley 21

Pipeline Interleaving
(Aka Hardware threads, related to hyperthreading)

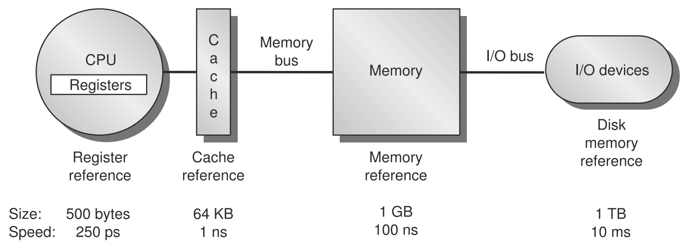
- History:
  - CDC 6600
  - Denelcore HEP
  - …
  - Sandbridge Sandblaster
  - XMOS
- Tradeoffs:
  + Simpler hardware (faster clocks)
  + Repeatable timing
  + Interference-free multithreading
  - Slower single-thread performance



Lee and Messerschmitt, Pipeline Interleaved Programmable DSPs, ASSP-35(9), 1987.

Lee, et al. Berkeley 22

Second Problem: Memory Hierarchy



Hennessey and Patterson, Computer Architecture: A Quantitative Approach, 4th edition, 2007.

- Register file is a temporary memory under program control.
  - *Why is it so small?* *Instruction word size.*
- Cache is a temporary memory under hardware control.
  - *Why is replacement strategy application independent?*

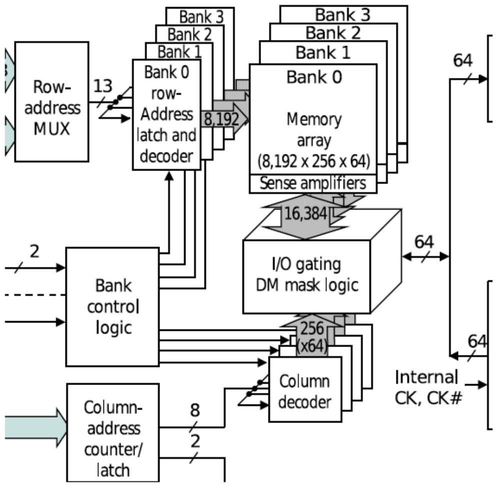
Separation of concerns.

PRET principle: any temporary memory is under program control.

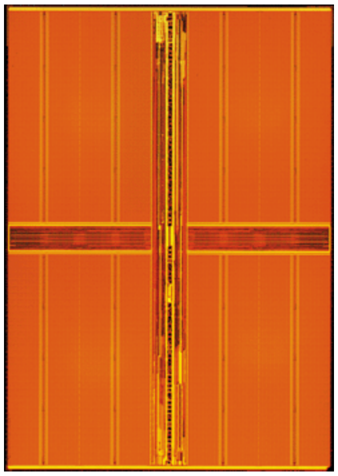
What about the main memory?

Access times depend on the history of accesses

See talk Monday at CODES-ISSS by Jan Reineke: PRET DRAM Controller: On the Virtue of Privatization



Micron corp.



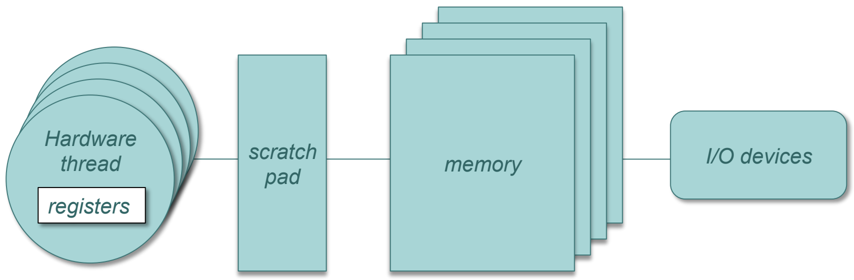
DDR2: Four pipelined banks
DDR3: Eight pipelined banks
DDRn: 2^n pipelined banks?

Our Current PRET Architecture

PTArm, a soft core on a Xilinx Virtex 5 FPGA (Isaac Liu)

Note inverted memory compared to multicore!

Fast, close memory is shared, slow remote memory is private!

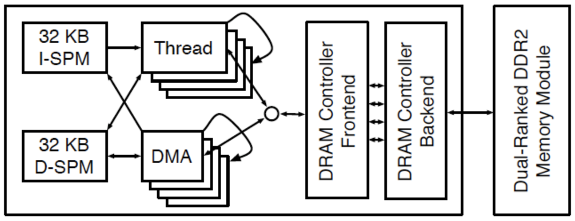


Interleaved pipeline with one set of registers per thread

SRAM scratchpad shared among threads

DRAM main memory, separate banks per thread

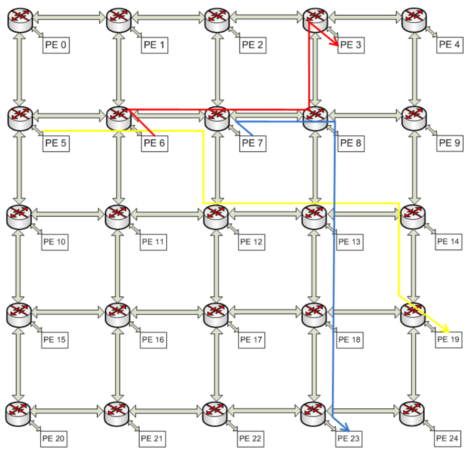
Memory Architecture in PTArm goes further



- Dual ported instruction/data scratchpads
- Load/stores can go to scratchpads or main memory
- DMA
  - One DMA unit per hardware thread
  - Thread can initiate DMA scratchpad-main transfers
  - Thread continues executing from scratchpad
  - Thread blocks on access to either DRAM or the affected region of the scratchpad until DMA is complete.
- DRAM refreshes are software controlled

See talk Monday at CODES-ISSS by Jan Reineke: PRET DRAM Controller: On the Virtue of Privatization

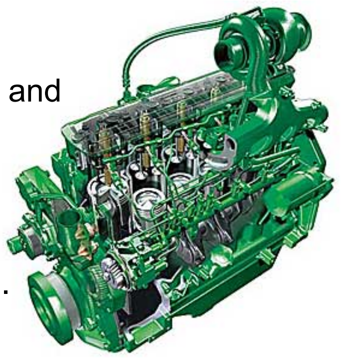
Multicore PRET



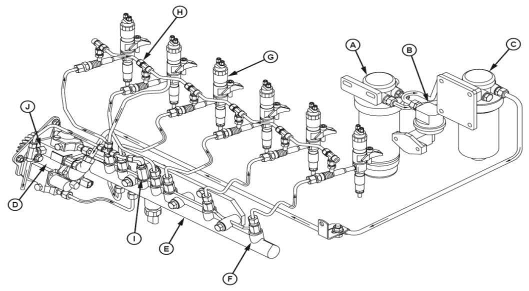
In today's multicore architectures, one thread can disrupt the timing of another thread *even if they are running on different cores and are not communicating*!

Our preliminary work shows that control over timing enables conflict-free routing of messages in a network on chip, making it possible to have non-interfering programs on a multicore PRET. (Dai Bui)

Application: Real-Time Computational Fluid Dynamics Simulation (Isaac Liu)



In collaboration with National Instruments and Matthew Viele (Colorado State) we have implemented on a multicore PRET a real-time simulation of a common-rail fuel injection system, for hardware-in the loop testing of control system designs.



Status of the PRET project

- Results:
  - PTArm implemented on Xilinx Virtex 5 FPGA (Isaac Liu).
  - UNISIM simulator of the PTArm facilitates experimentation.
  - DRAM controller with repeatable timing and DMA support.
  - PRET-like utilities implemented on COTS Arm.
  - PRET utilities implemented on Microblaze/pcore
- Much still to be done:
  - Realize MTFD, interrupt I/O, compiler toolchain, scratchpad management, etc.

A Key Next Step: Parametric PRET Architectures

```
set_time r1, 1s
// Code block
MTFD r1
```

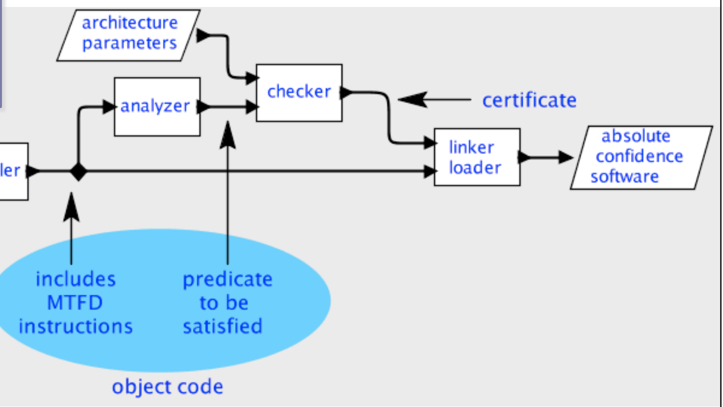
ISA that admits a variety of implementations:

- 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

```
set_time r1, 1s
// Code block
MTFD r1
```


architecture parameters
checker
certificate
analyzer
linker loader
absolute confidence software
source code
compiler
includes MTFD code blocks
includes MTFD instructions
predicate to be satisfied
object code

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.

Progress Towards Parametric PRET (Jan Reineke)

Execution time of program P under input i in timing model M.

All t_j functions are monotone in the parameters mu_j.

$$ET^{\mathcal{M}}_{P,i}(\lambda_1, \ldots, \lambda_m, \mu_1, \ldots, \mu_m) = \sum_{j=1}^{m} \lambda_j \cdot t_j(P, i, \mu_j).$$

“Linear” parameters

“Monotone” parameters

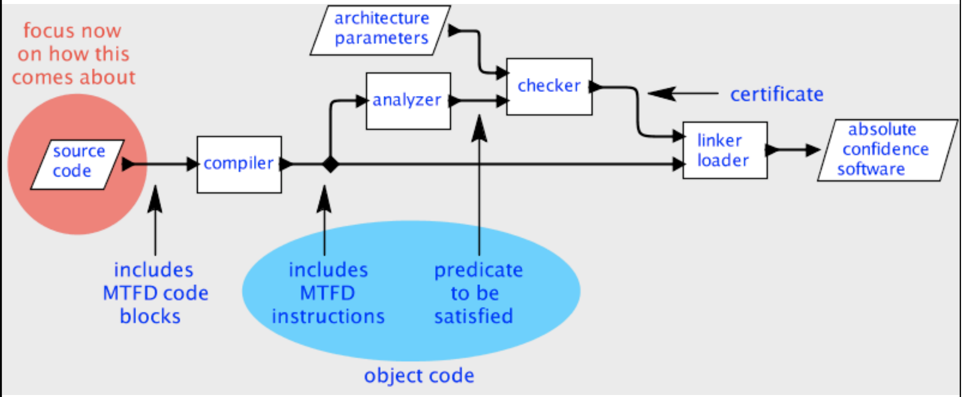
“Linear” Parameters scale part of execution time linearly

PRET Publications

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

- S. Edwards and E. A. Lee, "**The Case for the Precision Timed (PRET) Machine**," in the *Wild and Crazy Ideas* Track of DAC, June 2007.
- B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards and E. A. Lee, "**Predictable programming on a precision timed architecture**," CASES 2008.
- S. Edwards, S. Kim, E. A. Lee, I. Liu, H. Patel and M. Schoeberl, "**A Disruptive Computer Design Idea: Architectures with Repeatable Timing**," ICCD 2009.
- D. Bui, H. Patel, and E. Lee, "**Deploying hard real-time control software on chip-multiprocessors**," RTCSA 2010.
- Bui, E. A. Lee, I. Liu, H. D. Patel and J. Reineke, "**Temporal Isolation on Multiprocessing Architectures**," DAC 2011.
- J. Reineke, I. Liu, H. D. Patel, S. Kim, E. A. Lee, **PRET DRAM Controller: Bank Privatization for Predictability and Temporal Isolation** (to appear), CODES +ISSS, Taiwan, October, 2011.
- S. Bensalem, K. Goossens, C. M. Kirsch, R. Obermaisser, E. A. Lee, J. Sifakis, **Time-Predictable and Composable Architectures for Dependable Embedded Systems**, Tutorial Abstract (to appear), EMSOFT, Taiwan, October, 2011

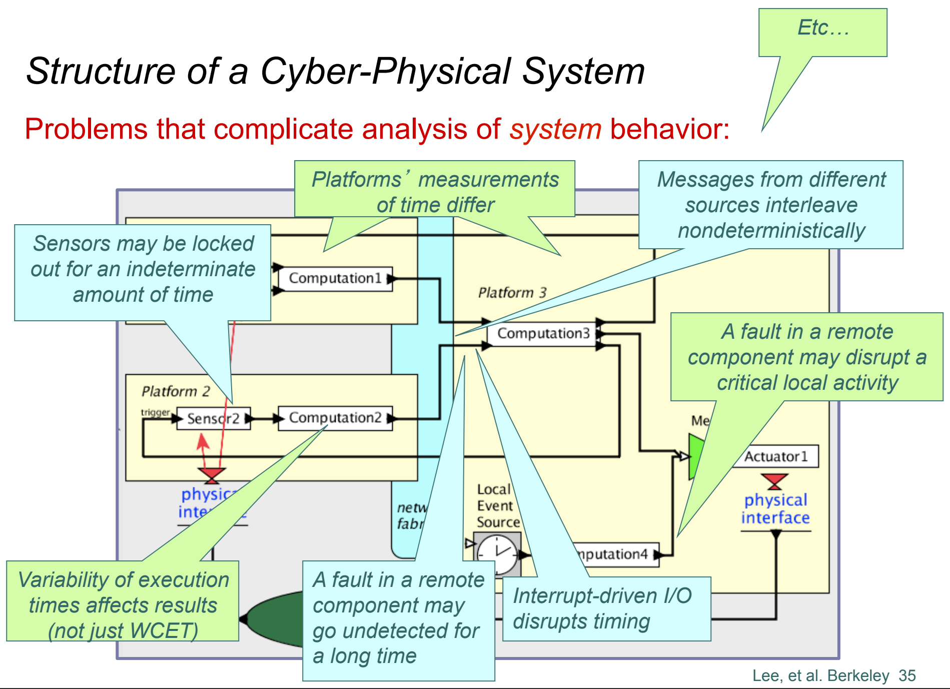
Part 2: How to get the Source Code?


focus now on how this comes about
architecture parameters
checker
certificate
analyzer
source code
compiler
linker loader
absolute confidence software
includes MTFD code blocks
includes MTFD instructions
predicate to be satisfied
object code

The input (most 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.

Structure of a Cyber-Physical System

Problems that complicate analysis of *system* behavior:



Etc…

Platforms' measurements of time differ

Messages from different sources interleave nondeterministically

Sensors may be locked out for an indeterminate amount of time

A fault in a remote component may disrupt a critical local activity

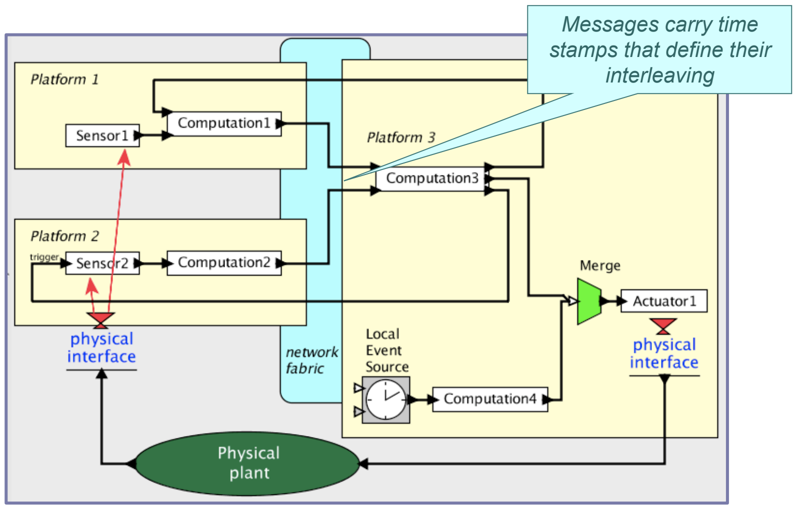
Variability of execution times affects results (not just WCET)

A fault in a remote component may go undetected for a long time

Interrupt-driven I/O disrupts timing

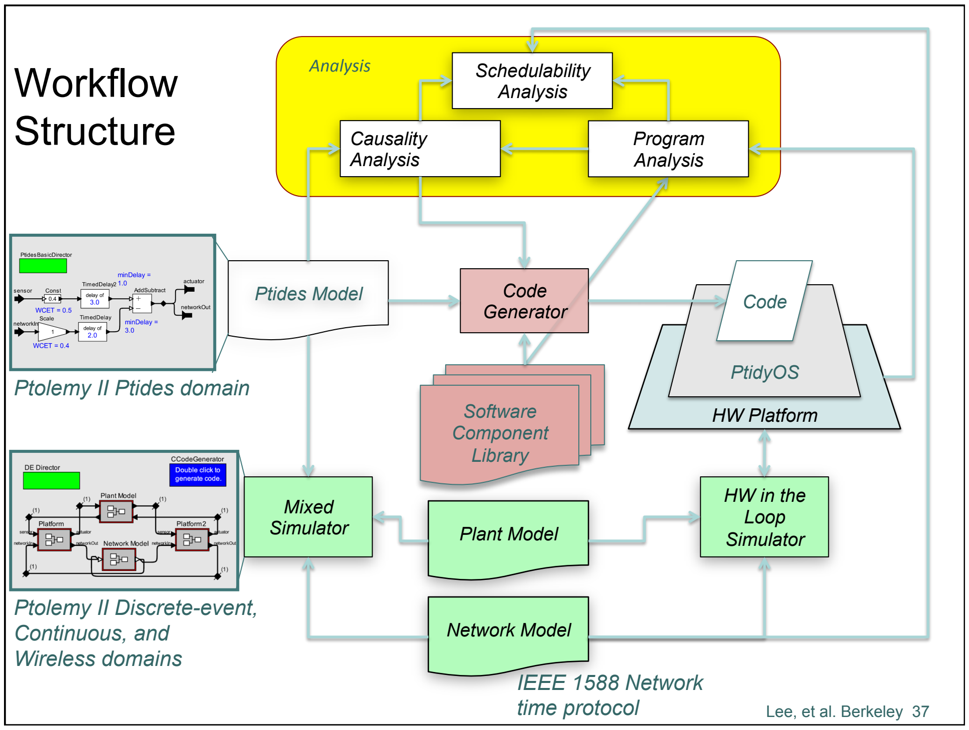
Lee, et al. Berkeley 35

Our Solution:
Ptides: A programming model based on time-stamped messages.

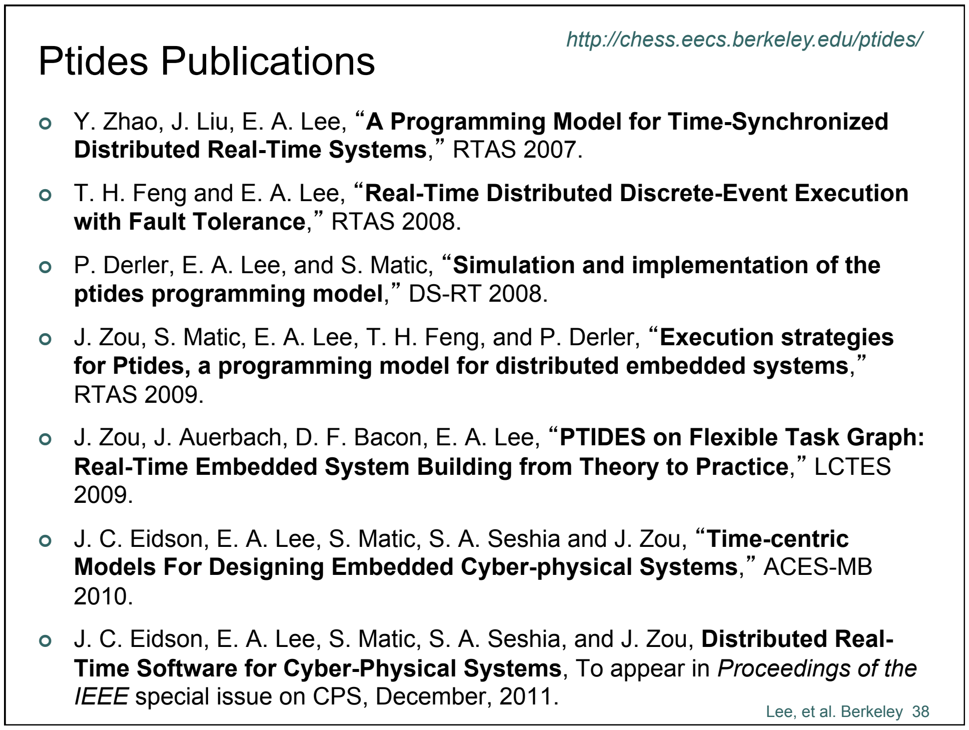


Messages carry time stamps that define their interleaving

Lee, et al. Berkeley 36



Workflow Structure



Ptides Publications

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

- Y. Zhao, J. Liu, E. A. Lee, "**A Programming Model for Time-Synchronized Distributed Real-Time Systems**," RTAS 2007.
- T. H. Feng and E. A. Lee, "**Real-Time Distributed Discrete-Event Execution with Fault Tolerance**," RTAS 2008.
- P. Derler, E. A. Lee, and S. Matic, "**Simulation and implementation of the ptides programming model**," DS-RT 2008.
- J. Zou, S. Matic, E. A. Lee, T. H. Feng, and P. Derler, "**Execution strategies for Ptides, a programming model for distributed embedded systems**," RTAS 2009.
- J. Zou, J. Auerbach, D. F. Bacon, E. A. Lee, "**PTIDES on Flexible Task Graph: Real-Time Embedded System Building from Theory to Practice**," LCTES 2009.
- J. C. Eidson, E. A. Lee, S. Matic, S. A. Seshia and J. Zou, "**Time-centric Models For Designing Embedded Cyber-physical Systems**," ACES-MB 2010.
- J. C. Eidson, E. A. Lee, S. Matic, S. A. Seshia, and J. Zou, **Distributed Real-Time Software for Cyber-Physical Systems**, To appear in *Proceedings of the IEEE* special issue on CPS, December, 2011.

Consequences of Precise Control over Timing

- Latency of software subsystems in CPS is controllable, enabling understanding of *system* dynamics.
- Resource sharing can become deterministic, making it less costly to implement (dispensing with interlocks) and eliminating interference.
- Network usage can be controlled, eliminating buffer overflow, interference, and message-dependent deadlock.
- Systems can be leaner (less overprovisioning).
- Systems will be safer (no unlikely confluences of events lurking in the background).
- Systems can be more secure (no timing side-channel attacks)
- What you test is what you ship!

Lee, et al. Berkeley 39

Conclusions

Overview References:
*•Lee. **Computing needs time**. CACM, 52(5):70–79, 2009*
•Derler, Lee, Sangiovanni-Vincentelli,
***Modeling Cyber-Physical Systems**,*
To appear in Proc. of the IEEE December, 2011.

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