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.
Claim

For CPS, programs do not adequately specify behavior.
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?

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.
Problems that complicate analysis of system behavior:

- Sensors may be locked out for an indeterminate amount of time.
- Messages from different sources interleave nondeterministically.
- Platforms’ measurements of time differ.
- 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.
- Variability of execution times affects results (not just WCET).
- Interrupt-driven I/O disrupts timing.

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 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.
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: PRET Machines

- 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]);
}
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*.
5. Pitter and Schoeberl, ACM TECS 2010: *A real-time Java chip-multiprocessor*.
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!**

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
```
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.

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


Pipeline Hazards

Pipeline Interlocks vs. Pipeline Interleaving

Traditional pipeline:
- T0: cmp %g2, 9
- T0: bg, a 40011b8
- T0: add %i1, %i2, %i3

Thread-interleaved pipeline:
- T0: cmp %g2, 9
- T1: add %o0, %g1, %g2
- T2: sub %g1, %g2, %g1
- T3: bn 430011a0
- T4: ld [%fp + -12], %g1
- T5: cmp %g1, 4
- T0: bg, a 40011b8
- T1: cmp %g1, 4

Dependencies result in complex timing behaviors

Repeatable timing behavior of instructions

Pipeline Interleaving
(Aka Hardware threads, related to hyperthreading)

- History:
  - CDC 6600
  - Denelcor 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.
Second Problem: Memory Hierarchy

- 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

Micron corp.

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

Our Current PRET Architecture

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

Our Current PRET Architecture

- Hardware thread
- scratchpad
- memory
- I/O devices

Interleaved pipeline with one set of registers per thread
SRAM scratchpad shared among threads
DRAM main memory, separate banks per thread

Note inverted memory compared to multicore!
Fast, close memory is shared, slow remote memory is private!

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

Lee, et al. Berkeley 26
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

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.

Progress Towards Parametric PRET
(Jan Reineke)

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

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

All $t_j$ functions are monotone in the parameters $\mu_j$.

“Linear” Parameters scale part of execution time linearly.

"Linear" parameters

"Monotone" parameters
Part 2: How to get the Source 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:

- Sensors may be locked out for an indeterminate amount of time.
- Messages from different sources interleave nondeterministically.
- Platforms’ measurements of time differ.
- 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.

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

Messages carry time stamps that define their interleaving.
Workflow Structure

Ptides Publications


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

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: