# Lecture 22: Quantitative Analysis

Seyed-Hosein Attarzadeh-Niaki

Based on slides by Edward Lee



## Review

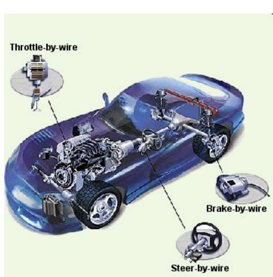
- (Embedded) Real-time OSs
  - Characteristics
  - Microkernels
  - Tasks and scheduling
  - Queues and intercommunication
  - Semaphores and synchronization
  - Other facilities
- RTOS standards

# Outline

- Worst-case execution time problem
- Programs as Graphs
- Challenges of Execution Time Analysis
- Current Approaches
- Limitations and Future Directions

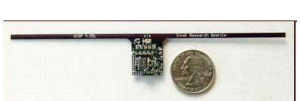


# Quantitative Analysis / Verification




Does the brake-by-wire software always actuate the brakes within 1$^{ms}$?
**Safety-critical embedded systems**

Can this new app drain my iPhone battery in an hour?
**Consumer devices**



How much energy must the sensor node harvest for RSA encryption?
**Energy-limited sensor nets, bio-medical apps, etc.**

## Time is Central to Cyber-Physical Systems

Several timing analysis problems:

- Worst-case execution time (WCET) estimation
- Estimating the distribution of execution times
- Threshold property: can you produce a test case that causes a program to violate its deadline?
- Software-in-the-loop simulation: predict execution time of particular program path

➢ Various forms of the same basic problem.



## Worst-Case Execution Time (WCET)

- The longest time taken by a software task to execute
  - Function of input data and environment conditions
- BCET = Best-Case Execution Time
  - shortest time taken by the task to execute

Consider this code: `*x = 10;`
on ARM Cortex-A9 MPCore dual core processor.

What's the WCET? BCET?

# Worst-Case Execution Time (WCET) & BCET

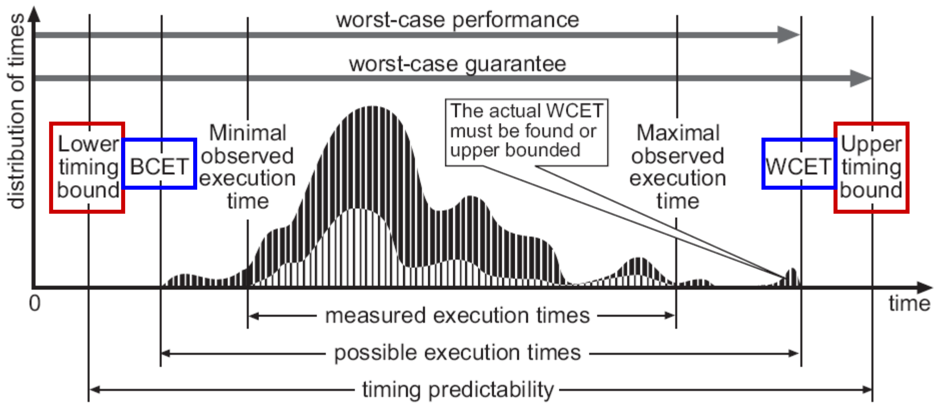
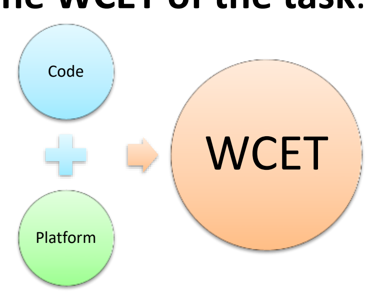



Figure from R.Wilhelm et al., ACM Trans. Embed. Comput. Sys, 2007.



# The WCET Problem

- Given
  - the code for a software task
  - the platform (OS + hardware) that it will run on
- **Determine the WCET of the task.**



# The WCET Problem

- Why is this important? Where is the WCET used?

The WCET is central to the design of Embedded Systems: Needed for Correctness (does the task finish in time?) and Performance (find optimal schedule for tasks)

- Can the WCET always be found?

In general, no, because the problem is *undecidable*.

- Is knowing WCET enough?

In general, no, because anomalies can occur when tasks end earlier than expected.



# Typical WCET Problem

- Task executes within an infinite loop

```
while(1) {
    read_sensors();
    compute();
    write_to_actuators();
}
```

This code typically has:
- loops with finite bounds
- no recursion

Additional assumptions:
- runs uninterrupted
- single-threaded

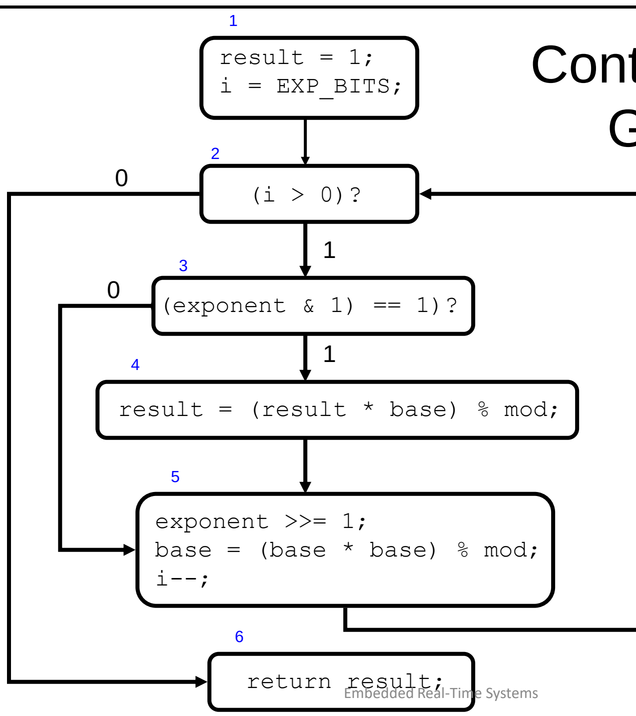
# Example Program: Modular Exponentiation

```c
#define EXP_BITS 32

typedef unsigned int UI;

UI modexp(UI base, UI exponent, UI mod) {
  int i;
  UI result = 1;

  i = EXP_BITS;
  while(i > 0) {
    if ((exponent & 1) == 1) {
      result = (result * base) % mod;
    }
    exponent >>= 1;
    base = (base * base) % mod;
    i--;
  }
  return result;
}
```

$$b^e = \begin{cases} (b^2)^{e/2} = (b^{e/2})^2, & e \text{ is even}, \\ (b^2)^{(e-1)/2} \cdot b = (b^{(e-1)/2})^2 \cdot b, & e \text{ is odd}. \end{cases}$$



# Control-Flow Graph




Each node is a basic block

# Components of Execution Time Analysis (traditional approaches)

- Program path (Control flow) analysis
  - Want to find longest path through the program
  - Find loop bounds
  - Identify feasible paths through the program
  - Identify dependencies amongst different code fragments
- Processor behavior analysis
  - For small code fragments (basic blocks), generate bounds on run-times on the platform
  - Model details of architecture, including cache behavior, pipeline stalls, branch prediction, etc.

➢ Outputs of both analyses feed into each other



# Program Path Analysis: Path Explosion

```
/* MAXSIZE = 100 */
for (Outer = 0; Outer < MAXSIZE; Outer++) {
   for (Inner = 0; Inner < MAXSIZE; Inner++) {
      if (Array[Outer][Inner] >= 0) {
         Ptotal += Array[Outer][Inner];
         Pcnt++;
      } else {
         Ntotal += Array[Outer][Inner];
         Ncnt++;
      }
      }
   Postotal = Ptotal;
   Poscnt = Pcnt;
   Negtotal = Ntotal;
   Negcnt = Ncnt;
}
```

How many paths?

$2^{10000}$

Example cnt.c from WCET benchmarks, Mälardalen Univ.

# Program Path Analysis: Determining Loop Bounds

```
#define EXP_BITS 32

typedef unsigned int UI;

UI modexp(UI base, UI exponent, UI mod) {
  int i;
  UI result = 1;

  i = EXP_BITS;
  while(i > 0) {
    if ((exponent & 1) == 1) {
      result = (result * base) % mod;
    }
    exponent >>= 1;
    base = (base * base) % mod;
    i--;
  }
  return result;
}
```

How many times around the while loop?

32

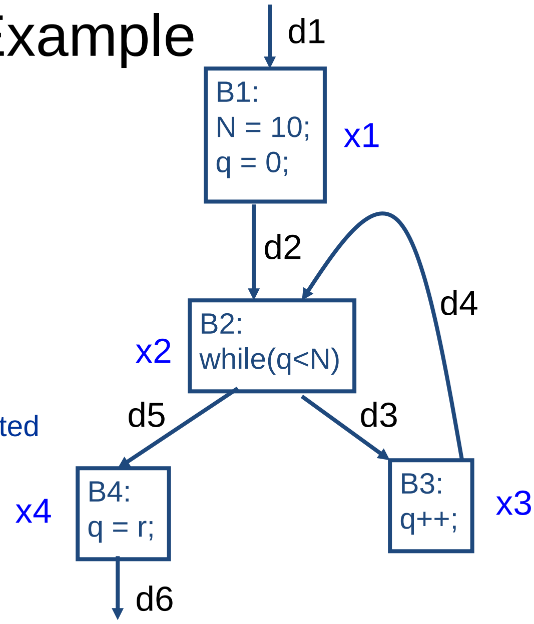


# Common Current Approach (high-level)

1. Manually construct processor behavior model
2. Use model to find “worst-case” starting processor states for each basic block → measure execution times of the blocks from these states
3. Use these times as upper bounds on the time of each basic block
4. Formulate an integer linear program to find the maximum sum of these bounds along any program path

# Example

```
N = 10;
q = 0;
while(q < N)
  q++;
q = r;
```



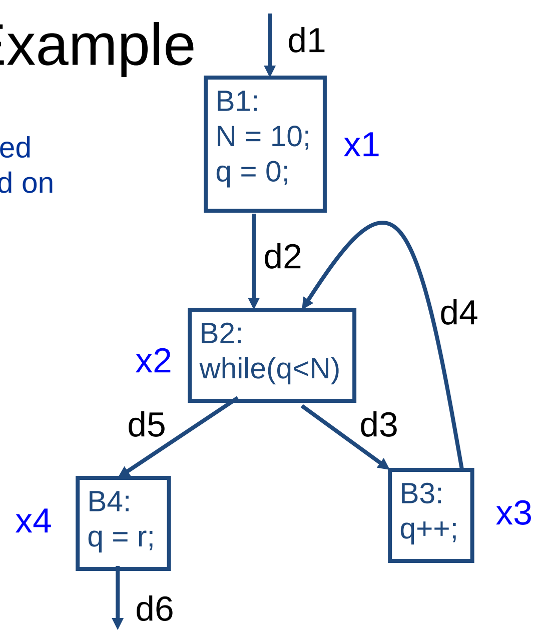

xi → # times Bi is executed
dj → # times edge is executed

Example due to Y.T. Li and S. Malik



# Example

xi → # times Bi is executed
dj → # times edge is executed
$C_i$ → measured upper bound on time taken by Bi




Want to
maximize $\sum_i C_i x_i$
subject to constraints
x1 = d1 = d2
d1 = 1
x2 = d2+d4 = d3+d5
x3 = d3 = d4 = 10
x4 = d5 = d6

Example due to Y.T. Li and S. Malik

# Integer Linear Programming

$x_i$ → # times $B_i$ is executed
$d_j$ → # times edge is executed
$C_i$ → measured upper bound on time taken by $B_i$

Want to maximize $\sum_i C_i x_i$
subject to constraints
$x1 = d1 = d2$
$d1 = 1$
$x2 = d2+d4 = d3+d5$
$x3 = d3 = d4 = 10$
$x4 = d5 = d6$

The problem to solve is:

Find integer values xi that maximize the sum (total execution time) subject to the (linear) constraint equations.

In general, this is an **Integer Linear Programming** (ILP) problem, which is known to be NP-hard, but for which there is good software.



# Program Path Analysis: Dependencies

```
void altitude_pid_run(void) {
  float err = estimator_z - desired_altitude;
  desired_climb = pre_climb + altitude_pgain * err;
  if (desired_climb < -CLIMB_MAX)
      desired_climb = -CLIMB_MAX;
  if (desired_climb > CLIMB_MAX)
      desired_climb = CLIMB_MAX;
}
```

Only one of these statements is executed (CLIMB_MAX = 1.0)

How many *feasible* paths?

Example from "PapaBench" UAV autopilot code, IRIT, France

## Program Path Analysis: Dependencies

```
void altitude_pid_run(void) {
  float err = estimator_z - desired_altitude;
  desired_climb = pre_climb + altitude_pgain * err;
  if (desired_climb < -CLIMB_MAX)
      desired_climb = -CLIMB_MAX;
  if (desired_climb > CLIMB_MAX)
      desired_climb = CLIMB_MAX;
}
```

Only one of these statements is executed (CLIMB_MAX = 1.0)

This adds an additional constraint of the form:
xi + xj <= 1

Example from “PapaBench” UAV autopilot code, IRIT, France



## But the problem gets even harder… Timing Analysis and Compositionality

Consider a task T with two parts A and B composed in sequence: T = A; B

Is WCET(T) = WCET(A) + WCET(B) ?

NOT ALWAYS!
WCETs cannot simply be composed ☹
→ Due to dependencies “through environment”

# Processor Behavior Analysis: Cache Effects

```
float dot_product(float *x, float *y, int n) {
  float result = 0.0;
  int i;
  for(i=0; i < n; i++) {
    result += x[i] * y[i];
  }
  return result;
}
```

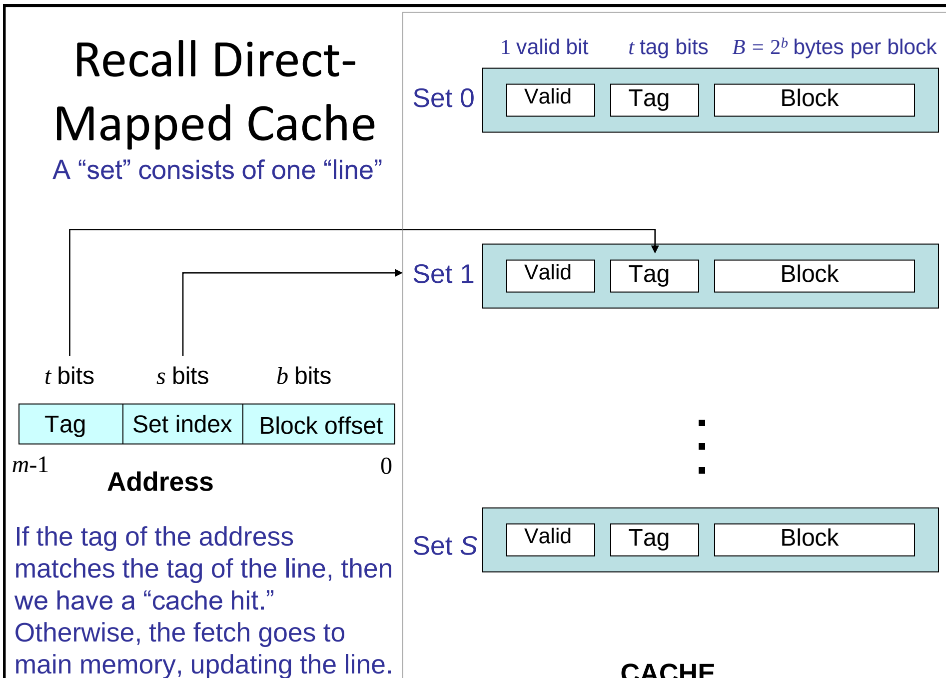
Suppose:
1. 32-bit processor
2. Direct-mapped cache holds two sets
   - 4 floats per set
   - x and y stored contiguously starting at address 0x0

What happens when **n=2**?



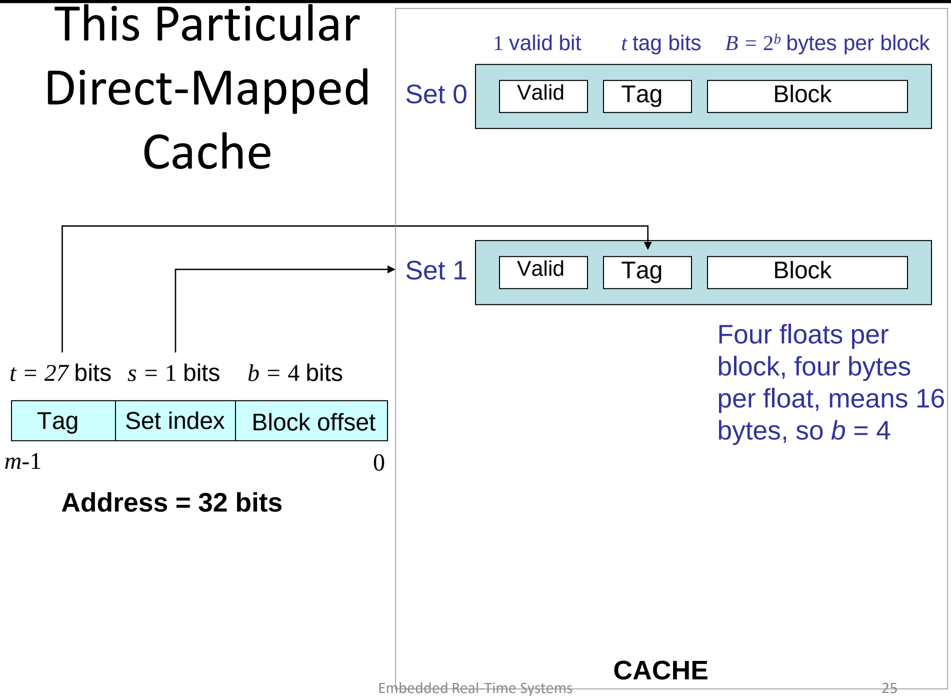
# Recall Direct-Mapped Cache

A "set" consists of one "line"




If the tag of the address matches the tag of the line, then we have a "cache hit." Otherwise, the fetch goes to main memory, updating the line.

# This Particular Direct-Mapped Cache






# Processor Behavior Analysis: Cache Effects

```
float dot_product(float *x, float *y, int n) {
  float result = 0.0;
  int i;
  for(i=0; i < n; i++) {
    result += x[i] * y[i];
  }
  return result;
}
```

What happens when **n=2**?

Suppose:
1. 32-bit processor
2. Direct-mapped cache holds two sets
   - 4 floats per set
   - x and y stored contiguously starting at address 0x0

x[0] will miss, pulling x[0], x[1], y[0] and y[0] into the set 0. All but one access will be a cache hit.

## Processor Behavior Analysis: Cache Effects

```
float dot_product(float *x, float *y, int n) {
  float result = 0.0;
  int i;
  for(i=0; i < n; i++) {
    result += x[i] * y[i];
  }
  return result;
}
```

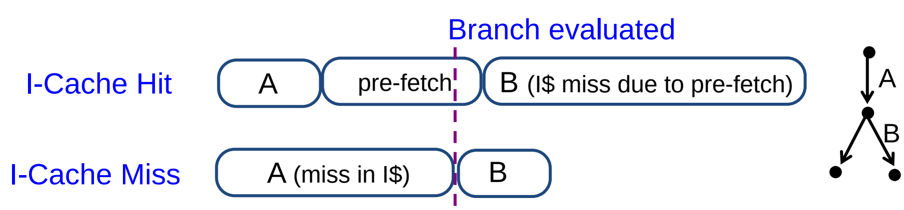
Suppose:
1. 32-bit processor
2. Direct-mapped cache holds two sets
   - 4 floats per set
   - x and y stored contiguously starting at address 0x0

What happens when **n=8**?

x[0] will miss, pulling x[0-3] into the set 0. Then y[0] will miss, pulling y[0-3] *into the same set,* evicting x[0-3]. Every access will be a miss!



## Timing Anomalies




Scenario 1: Block A hits in I-cache, triggers branch speculation, and prefetch of instructions, then predicted branch is wrong, so Block B must execute, but it’s been evicted from I-cache, execution of B delayed.

Scenario 2: Block A misses in I-cache, no branch prediction, then B hits in I-cache, B completes.

[ from R.Wilhelm et al., ACM Trans. Embed. Comput. Sys, 2007.]

## How to Measure Run-Time

Several techniques, with varying accuracy

- Instrument code to sample CPU cycle counter
  - relatively easy to do, read processor documentation for assembly instruction
- Use cycle-accurate simulator for processor
  - useful when hardware is not available/ready
- Use Logic Analyzer
  - non-intrusive measurement, more accurate
- …



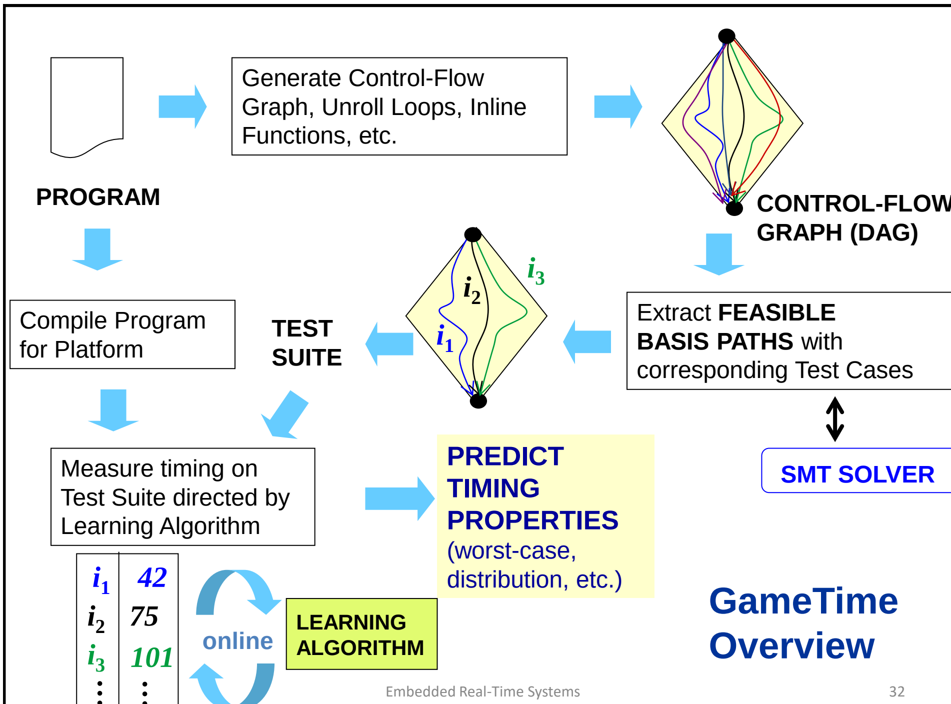
## Measurement Pitfalls

- Instrumentation incurs small overhead
  - measure long enough code sequence to compensate
- Multi-tasking effects: counter keeps going even when the task of interest is inactive
  - take multiple measurements and pick “k best” (cluster)
- Multicores/hyperthreading
  - Need to ensure that task is ‘locked’ to a single core
- Power management effects
  - CPU speed might change, timer could get reset during hibernation

# Some WCET Estimation Tools

- Commercial Tools: aiT, RapiTime, …
- University/Research Tools: GameTime, Chronos, …
  - GameTime: combines machine learning with automated theorem proving technology
  - Uses Programs as Graphs and integer linear programming ideas, but almost everything else different from other tools!





# Open Problems

- Architectures are getting much more complex.
  - Can we create processor models without the agonizing pain? [Yes, employ machine learning from systematic measurements → GameTime project]
  - Can we change the architecture to make timing analysis easier? [Yes, see PRET machine project]
- Analysis methods are "Brittle" – small changes to code and/or architecture can require completely re-doing the WCET computation
  - GameTime project addresses this:
    - Use robust techniques that learn about processor/platform behavior
    - Need to deal with concurrency, e.g., interrupts
- Need more reliable ways to measure execution time



# PRET Machines – Giving Software the Capabilities their Hardware Already Has.

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

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

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

*Computing*

\+

= PRET

*With time*

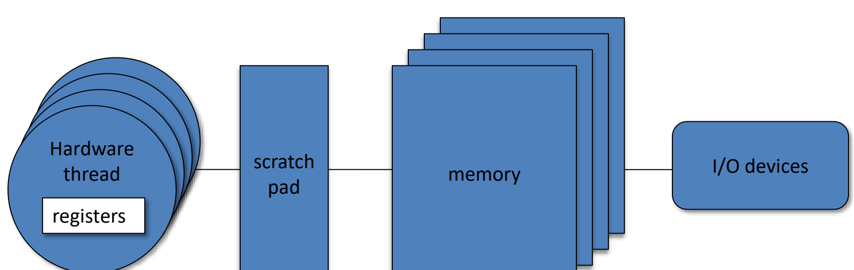
# Major Challenges
## and existence proofs that they can be met

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



# Second Generation PRET

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




*Interleaved pipeline with one set of registers per thread*

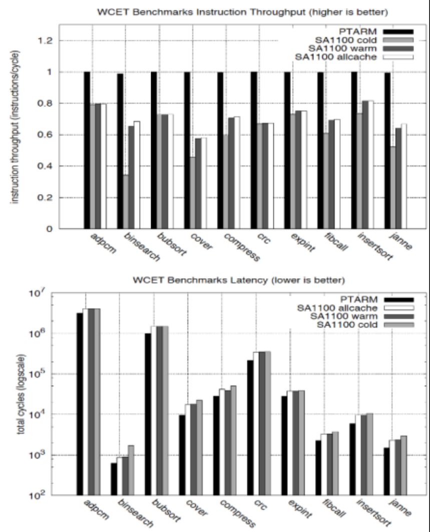
*SRAM scratchpad shared among threads*

*DRAM main memory, separate banks per thread*

# Performance Cost?

No!

- The PRET project has shown that you do not need to sacrifice performance to get control over timing.
- But in PtARM, you need enough concurrency to not lose performance.

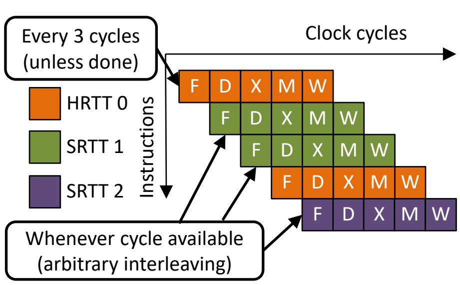



*[Isaac Liu, PhD Thesis, May, 2012]*



# Third-Generation PRET: Open-Source FlexPRET (Zimmer 2014/15)

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

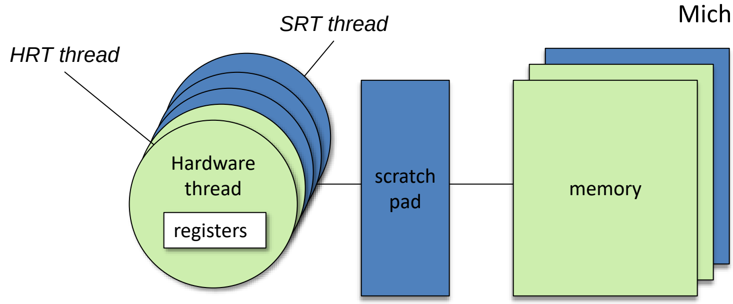



Digilent Atlys (Spartan 6) and NI myRIO (Zync)

# FlexPRET

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

Michael Zimmer




*HRT threads have deterministic timing. SRT threads share available cycles*

*SRAM scratchpad shared among threads*

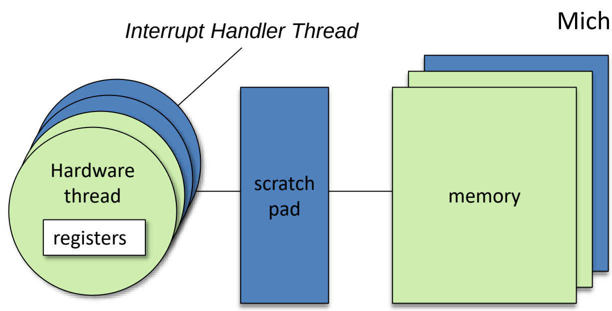
*DRAM main memory provides deterministic latency for HRT threads. Conventional behavior for the rest.*



# FlexPRET I/O

*Interrupt-Driven I/O is notorious for disrupting timing*

Michael Zimmer




Interrupts have *no effect* on HRT threads, and *bounded effect* on SRT threads!

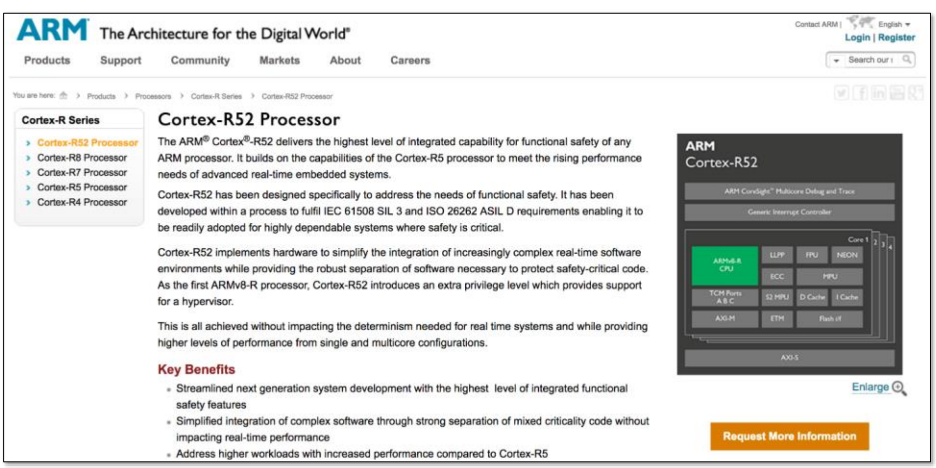
# FlexPRET Shows

- 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!



# Industry has shown some interest



# Optional Reading Material



## Cycle Counters

- Most modern systems have built in registers that are incremented every clock cycle
- Special assembly code instruction to access
- On Intel 32-bit x86 machines since Pentium:
  - 64 bit counter
  - RDTSC instruction (ReaD Time Stamp Counter) sets %edx register to high order 32-bits, %eax register to low order 32-bits
- Wrap-around time for 2 GHz machine
  - Low order 32-bits every 2.1 seconds
  - High order 64 bits every 293 years

[slide due to R. E. Bryant and D. R. O'Hallaron]

# Measuring with Cycle Counter

- Idea
  - Get current value of cycle counter
    - store as pair of unsigned's cyc_hi and cyc_lo
  - Compute something
  - Get new value of cycle counter
  - Perform double precision subtraction to get elapsed cycles

```
/* Keep track of most recent reading of cycle counter */
static unsigned cyc_hi = 0;
static unsigned cyc_lo = 0;

void start_counter()
{
  /* Get current value of cycle counter */
  access_counter(&cyc_hi, &cyc_lo);
}
```

[slide due to R. E. Bryant and D. R. O'Hallaron]



# Accessing the Cycle Counter

- GCC allows inline assembly code with mechanism for matching registers with program variables
- Code only works on x86 machine compiling with GCC

```
void access_counter(unsigned *hi, unsigned *lo)
{
  /* Get cycle counter */
  asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
      : "=r" (*hi), "=r" (*lo)
      : /* No input */
      : "%edx", "%eax");
}
```

- Emit assembly with rdtsc and two movl instructions

[slide due to R. E. Bryant and D. R. O'Hallaron]

# Completing Measurement

- Get new value of cycle counter
- Perform double precision subtraction to get elapsed cycles
- Express as `double` to avoid overflow problems

```
double get_counter()
{
  unsigned ncyc_hi, ncyc_lo
  unsigned hi, lo, borrow;
  /* Get cycle counter */
  access_counter(&ncyc_hi, &ncyc_lo);
  /* Do double precision subtraction */
  lo = ncyc_lo - cyc_lo;
  borrow = lo > ncyc_lo;
  hi = ncyc_hi - cyc_hi - borrow;
  return (double) hi * (1 << 30) * 4 + lo;
}
```

[slide due to R. E. Bryant and D. R. O'Hallaron]



# Timing With Cycle Counter

- Time Function P
  - First attempt: Simply count cycles for one execution of P

```
double tcycles;
start_counter();
P();
tcycles = get_counter();
```

  - What can go wrong here?

[slide due to R. E. Bryant and D. R. O'Hallaron]

# Dealing with Overhead & Cache Effects

- Always execute function once to "warm up" cache
- Keep doubling number of times execute P() until reach some threshold
  - Used CMIN = 50000

```
int cnt = 1;
double cmeas = 0;
double cycles;
do  {
  int c = cnt;
  P();                    /* Warm up cache */
  get_counter();
  while (c-- > 0)
    P();
  cmeas = get_counter();
  cycles = cmeas / cnt;
  cnt += cnt;
} while (cmeas < CMIN);  /* Make sure have enough */
return cycles / (1e6 * MHZ);
```



# Timing With Cycle Counter

- Determine Clock Rate of Processor
  - Count number of cycles required for some fixed number of seconds

```
double MHZ;
int sleep_time = 10;
start_counter();
sleep(sleep_time);
MHZ = get_counter()/(sleep_time * 1e6);
```

- Time Function P
  - First attempt: Simply count cycles for one execution of P

```
double tsecs;
start_counter();
P();
tsecs = get_counter() / (MHZ * 1e6);
```

[slide due to R. E. Bryant and D. R. O'Hallaron]

# Measurement Pitfalls

- Instrumentation incurs small overhead
  - measure long enough code sequence to compensate
- Cache effects can skew measurements
  - "warm up" the cache before making measurement
- Multi-tasking effects: counter keeps going even when the task of interest is inactive
  - take multiple measurements and pick "k best" (cluster)
- Multicores/hyperthreading
  - Need to ensure that task is 'locked' to a single core
- Power management effects
  - CPU speed might change, timer could get reset during hibernation