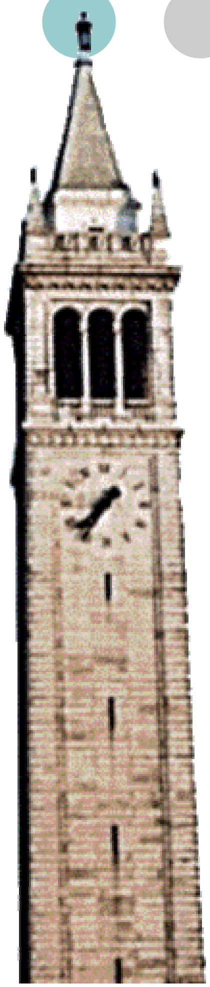


Verification of Quantitative Properties of Embedded Systems: Execution Time Analysis

Sanjit A. Seshia

UC Berkeley

EECS 249

Fall 2009

Source

Material in this lecture is drawn from the following sources:

- "The Worst-Case Execution Time Problem – Overview of Methods and Survey of Tools", R. Wilhelm et al., ACM Transactions on Embedded Computing Systems, 2007.
- Chapter 9 of "Computer Systems: A Programmer's Perspective", R. E. Bryant and D. R. O'Hallaron, Prentice-Hall, 2002.
- "Performance Analysis of Real-Time Embedded Software," Y-T. Li and S. Malik, Kluwer Academic Pub., 1999.
- "Game-Theoretic Timing Analysis", S. A. Seshia and A. Rakhlin, ICCAD 2008
  - Extended version is Technical Report EECS-2009-130

Worst-Case Execution Time (WCET) of a Task

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)

Worst-Case Execution Time (WCET) & BCET

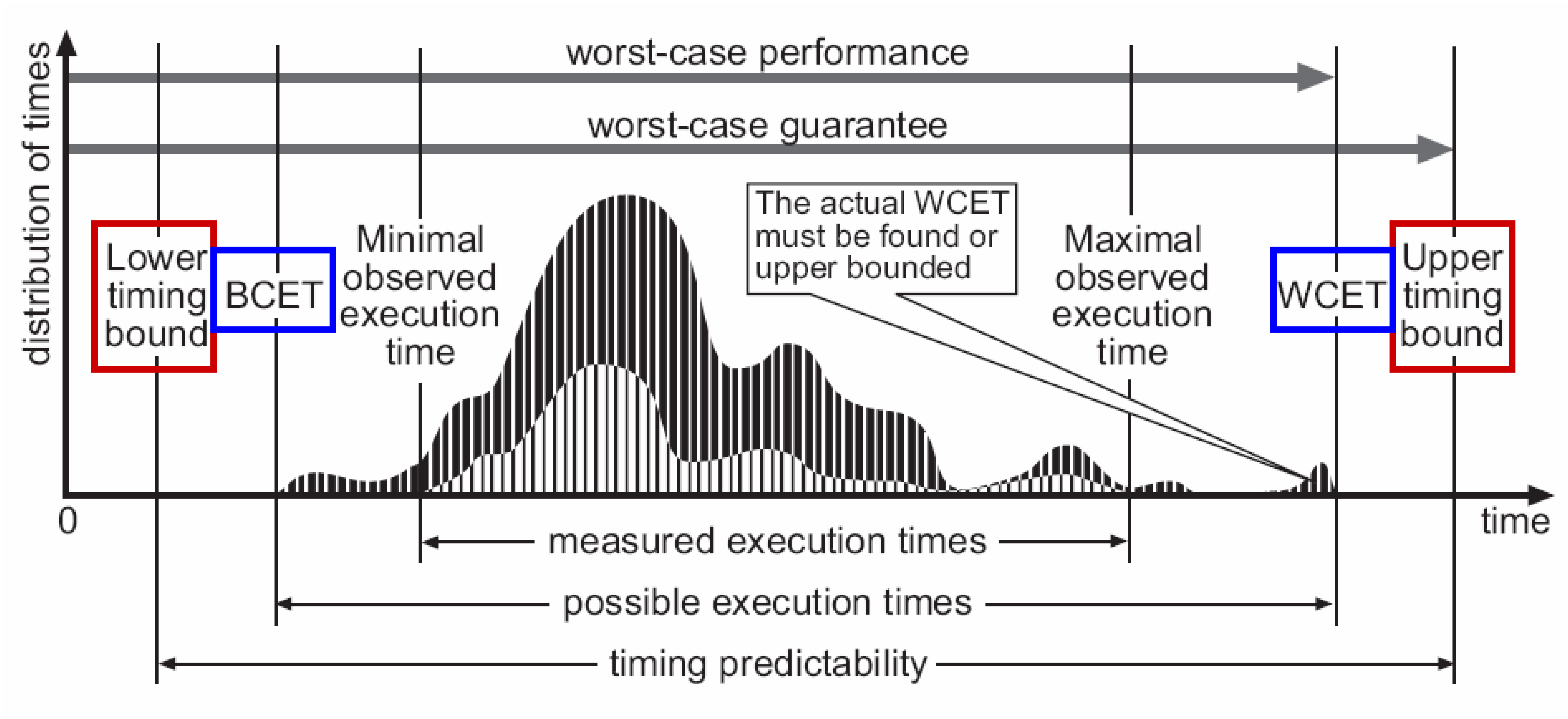


Figure from R.Wilhelm et al., ACM Trans. Embed. Comput. Sys, 2007.

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.

Why is this problem important?

The WCET is central in the design of RT 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*.

Typical WCET Problem Setting

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

Outline of the Lecture

- How to measure execution time
- Current Approaches to Execution Time Analysis
- Limitations
- The GameTime approach
- Demo of some tools

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
- …

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:

- 64 bit counter
- RDTSC instruction sets `%edx` to high order 32-bits, `%eax` 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]

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

Outline of the Lecture

- How to measure execution time
- **Current Approaches to Execution Time Analysis**
- Limitations
- The GameTime approach
- Demo of some tools

Components of Execution Time Analysis

- Program path (Control flow) analysis
  - Want to find longest path through the program
  - Identify feasible paths through the program
  - Find loop bounds
  - 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

```
for (Outer = 0; Outer < MAXSIZE; Outer++) {
/* MAXSIZE = 100 */
      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;
}
```

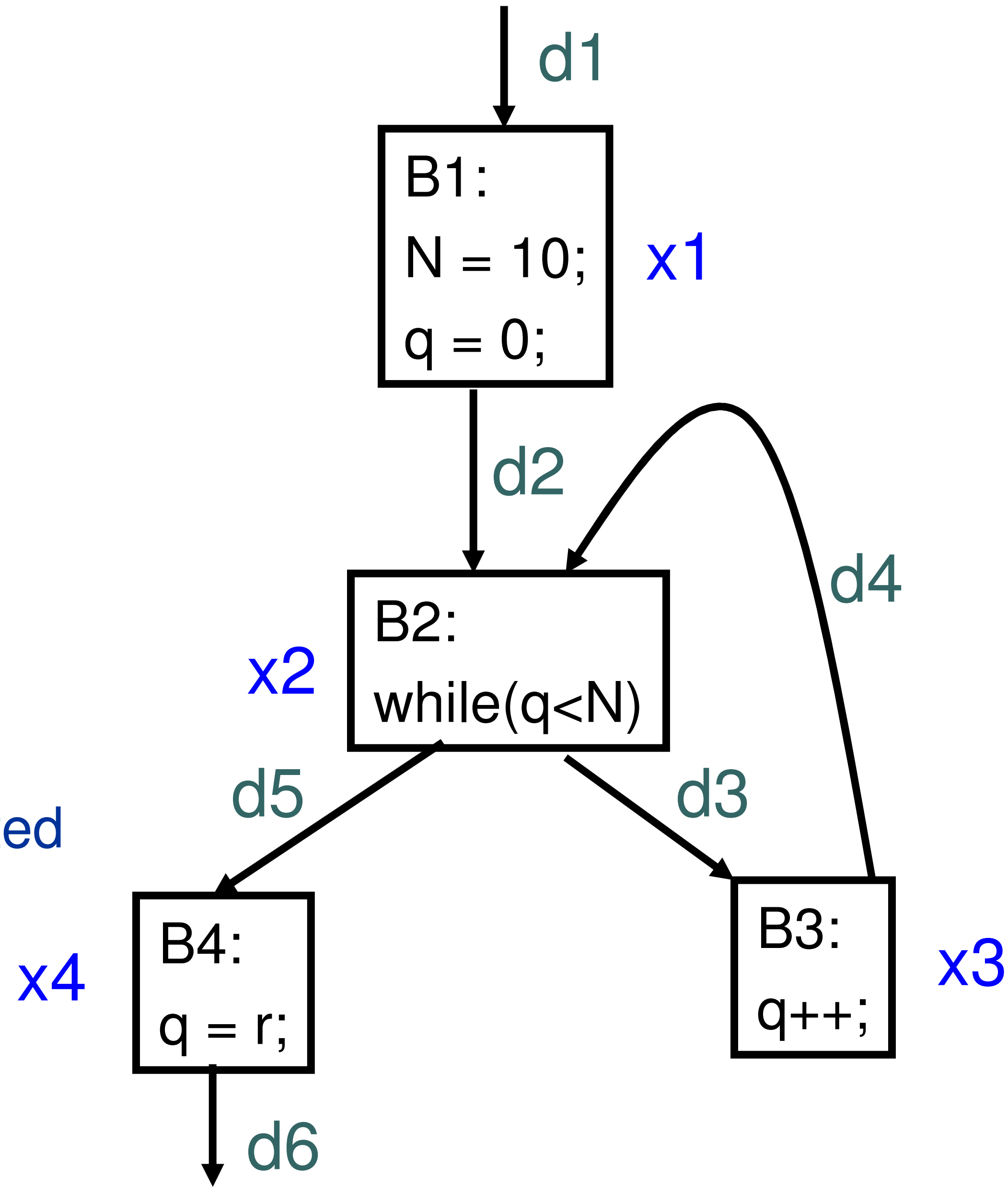
Example cnt.c from WCET benchmarks, Mälardalen Univ.

Program Path Analysis: Overall Approach

- Construct Control-Flow Graph (CFG) for the task
  - Nodes represent Basic Blocks of the task
  - Edges represent flow of control (jumps, branches, calls, …)
- The problem is to identify the longest path in the CFG
  - Note: CFG can have loops, so need to infer loop bounds and unroll them
  - This gives us a directed acyclic graph (DAG). How do we find the longest path in this DAG?

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

Program Path Analysis: Dependencies

```
#define CLIMB_MAX 1.0
...
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

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

Example, Revisited

xi → # times Bi is executed

dj → # times edge is executed

C_i → measured time taken by Bi

Want to

maximize $\sum_i C_i\, xi$

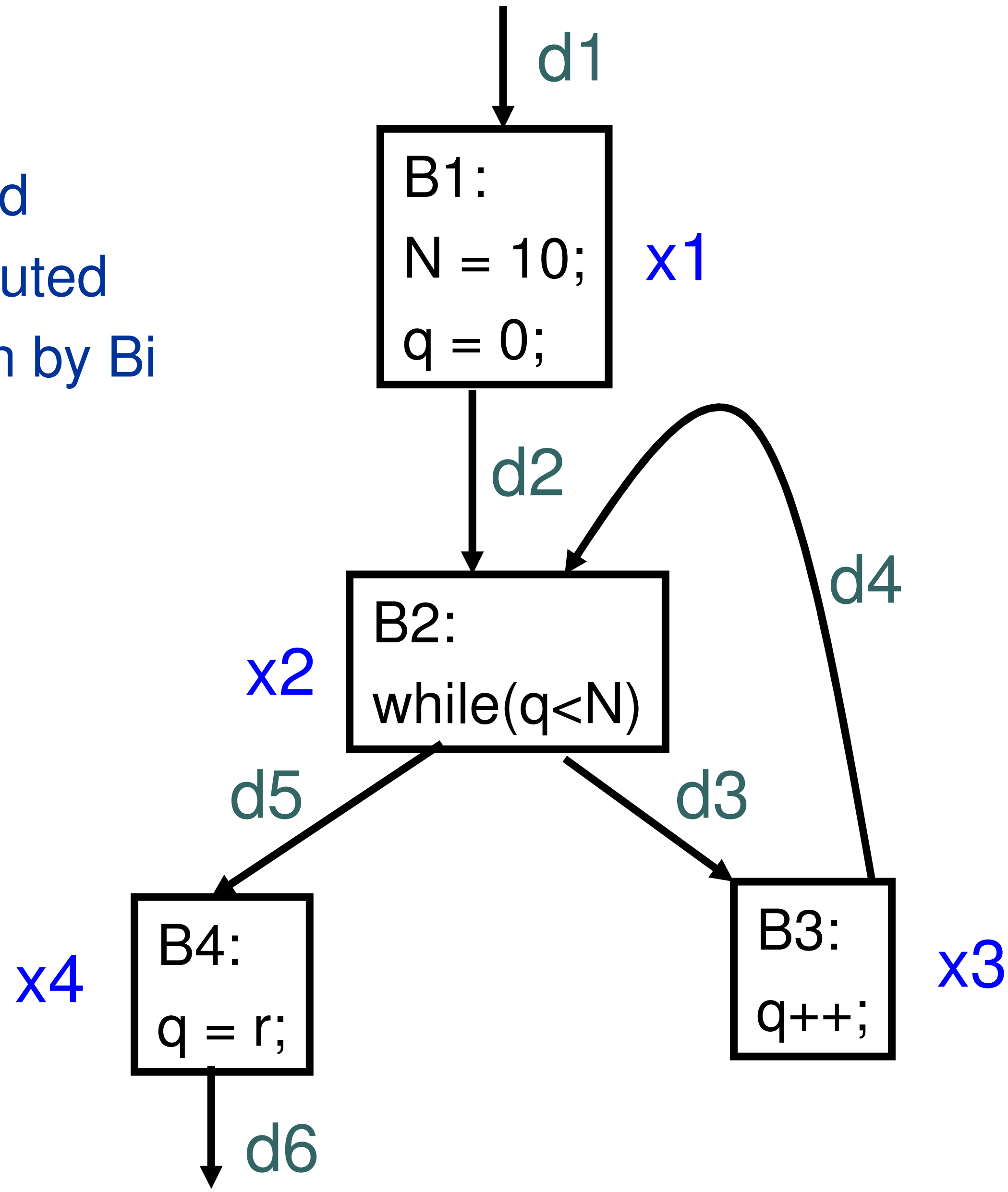
subject to constraints

x1 = d1 = d2

x2 = d2+d4 = d3+d5

x3 = d3 = d4 = 10

x4 = d5 = d6



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

Timing Analysis and Compositionality

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

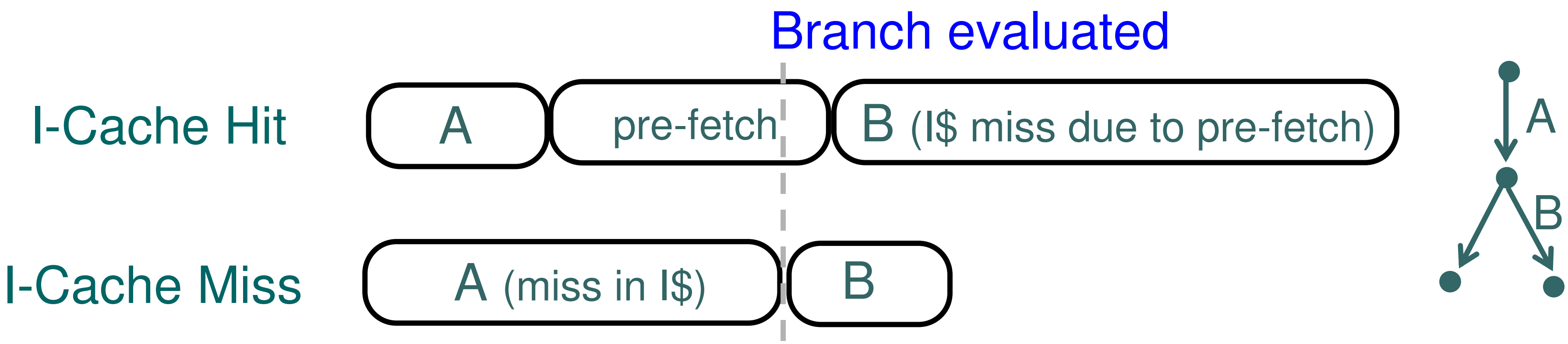
Is WCET(T) = WCET(A) + WCET(B) ?

NO!

WCETs cannot simply be composed ☹

→ Due to dependencies "through environment"

Timing Anomalies



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

Scenario 2: Instr 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.]

Outline of the Lecture

- How to measure execution time
- Current Approaches to Execution Time Analysis
- **Limitations**
- The GameTime approach
- Demo of some tools

Current WCET Methods: Limitations

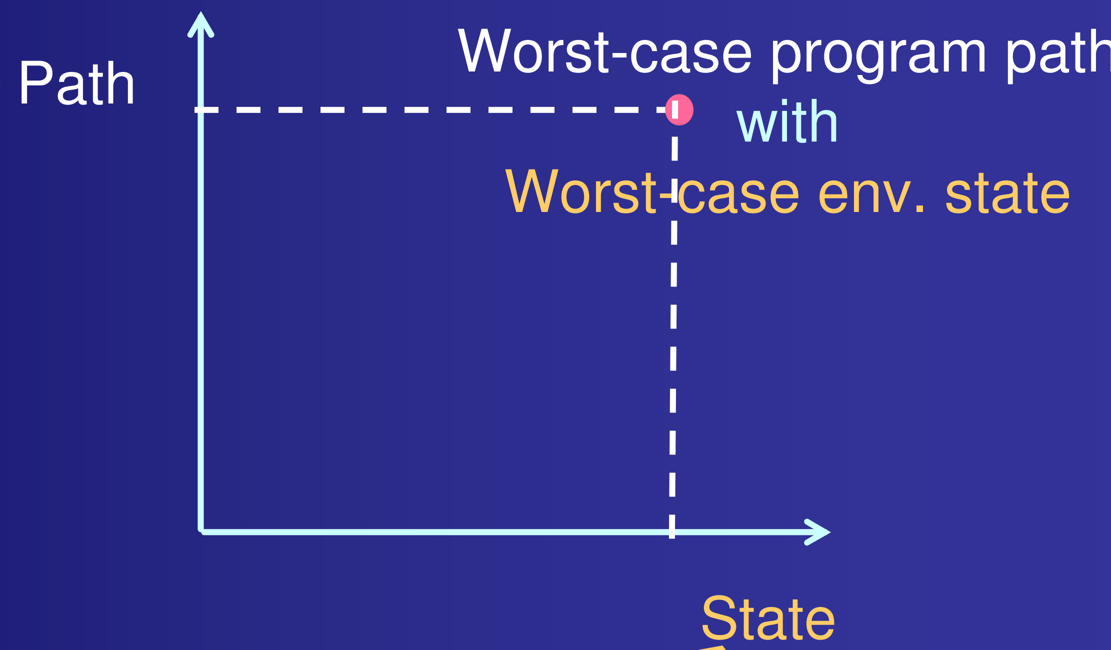
- **<u>Big Limitation:</u>** **Environment (Platform) Modeling**
  - **Where's my platform? Tools only work for selected processors/compilers for which detailed models are hand-constructed**
  - **Inaccurate & Tedious: platforms are becoming more complex, modeling takes months of human effort**
  - **Brittle, *not* portable: small changes to the platform can require completely re-doing the analysis**
  - See e.g., [E. A. Lee, TR'07], [Kirner and Puschner, *ISORC* 2008]

Beyond WCET: Other Execution Time Problems

- Average-case analysis
  - Given any program path (input), can we predict how long the program will take to execute, on average?
- Profile
  - Plot histogram of execution times of a program
  - Find top 10% of longest program paths
- ...

Two Dimensions of the WCET Estimation Problem

Challenge:
Exponentially-many program paths (in worst case)



Challenge:
How to find the worst-case state of the platform (environment)?

- Need accurate model of platform
- Need to find worst-case state

Classification of Current Tools

Static Analysis

- Abstract interpretation generates invariants to capture
  - worst-case environment states at control points
  - loop bounds
- Find time bounds on basic blocks (straight-line program fragments) from worst-case state
- Use implicit path enumeration (IPET) based on integer programming to compute WCET

➢ A very effective approach if an accurate platform model is available

Measurement-Based

- Run tests
  - Test suite generated randomly or heuristically, e.g., using genetic algorithms, or via systematic methods such as model checking
- Measure execution time
- Compute maximum over all observed times

➢ Under certain conditions, this could be done compositionally, but in general need end-to-end measurements

Some WCET Estimation Tools

Tool	Flow	Proc. Behavior	Bound Calc.
aiT	value analysis	static program analysis	IPET
Bound-T	linear loop-bounds and constraints by Omega test	static program analysis	IPET per function
RapiTime	n.a.	measurement	structure-based
SymTA/P	single feasible path analysis	static program analysis for I/D cache, measurement for segments	IPET
Heptane	-	static prog. analysis	structure-based, IPET
Vienna S.	-	static program analysis	IPET
Vienna M.	Genetic Algorithms	segment measurements	n.a.
Vienna H.	Model Checking	segment measurements	IPET
SWEET	value analysis, abstract execution, syntactical analysis	static program analysis for instr. caches, simulation for the pipeline	path-based, IPET-based, clustered
Florida		static program analysis	path-based
Chalmers		modified simulation	
Chronos		static prog. analysis	IPET

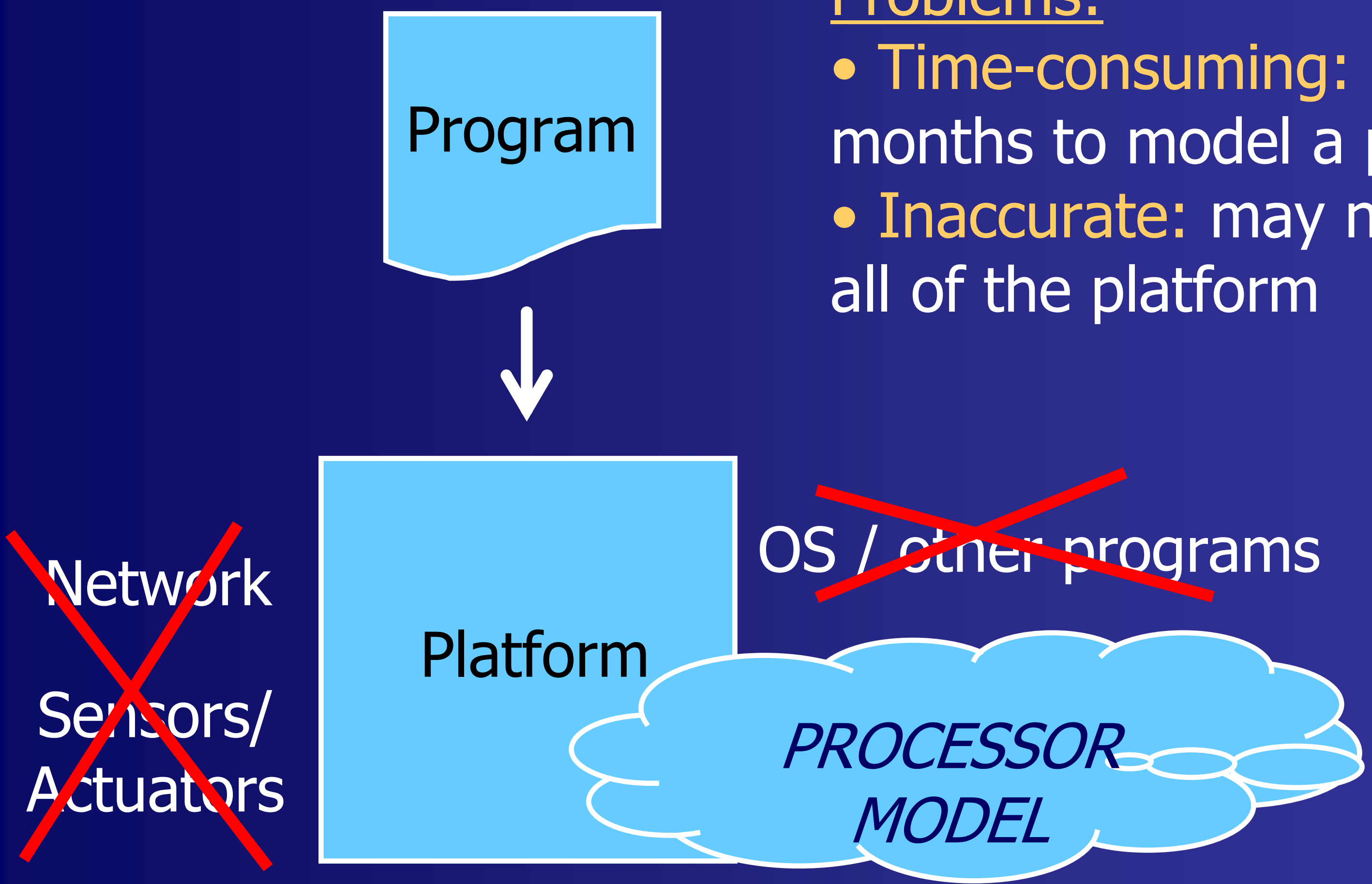
[R.Wilhelm et al., ACM Trans. Embed. Comput. Sys, 2007.]

Issues with Static Methods: Platform Modeling

Problems:

• Time-consuming: several months to model a processor

• Inaccurate: may not model all of the platform



Issues with Measurement-Based Tools

- How good is the test suite?
  - Good path coverage?
- Does the worst-case platform behavior occur?
- Is the measurement accurate?

Outline of the Lecture

- How to measure execution time
- Current Approaches to Execution Time Analysis
- Limitations
- **The GameTime approach (quick overview)**
- **Demo of some tools**

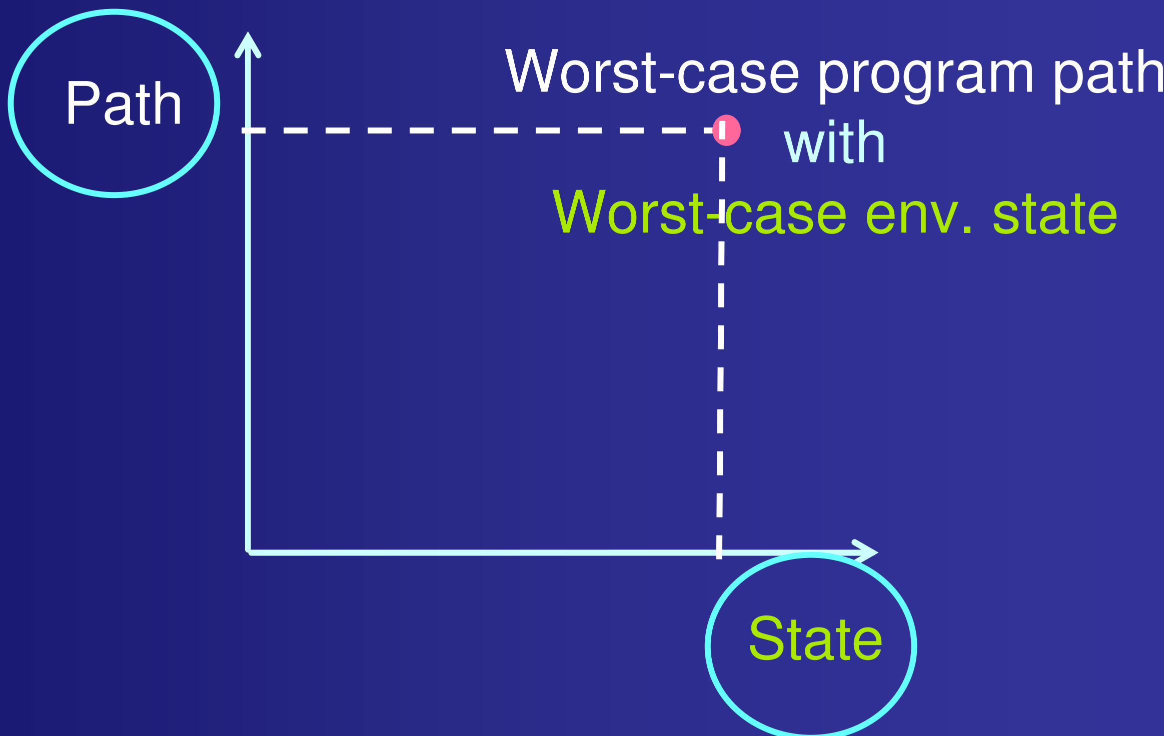
The GameTime Approach: Contributions

[Seshia & Rakhlin, ICCAD '08]

- **Model the estimation problem as a Game**
  - Tool vs. Platform
  - Robust to changes in the platform
- **Measurement-based**
  - Perform *end-to-end* measurements of execution time of selected (linearly many) paths on target platform
- **Learn Environment Model**
  - Learn a (graph) model of platform's behavior
- **Online algorithm: GameTime**
  - Theoretical guarantee: can find WCET with arbitrarily high probability under some assumptions
- **Leverages advances in "Verification Engines"**
  - Satisfiability modulo theories (SMT) solvers

Intuition for Our Approach

Controlled by "us"
(the estimation tool)



Game:
Tool
vs.
Platform

Controlled by the environment
(platform)

Components of the Game

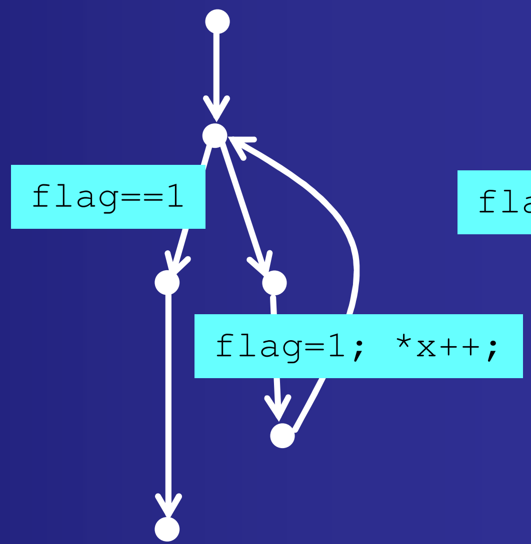
- Strategies (moves) of the Tool
- Strategies (moves) of the Platform (environment)

- Winning condition of the Game

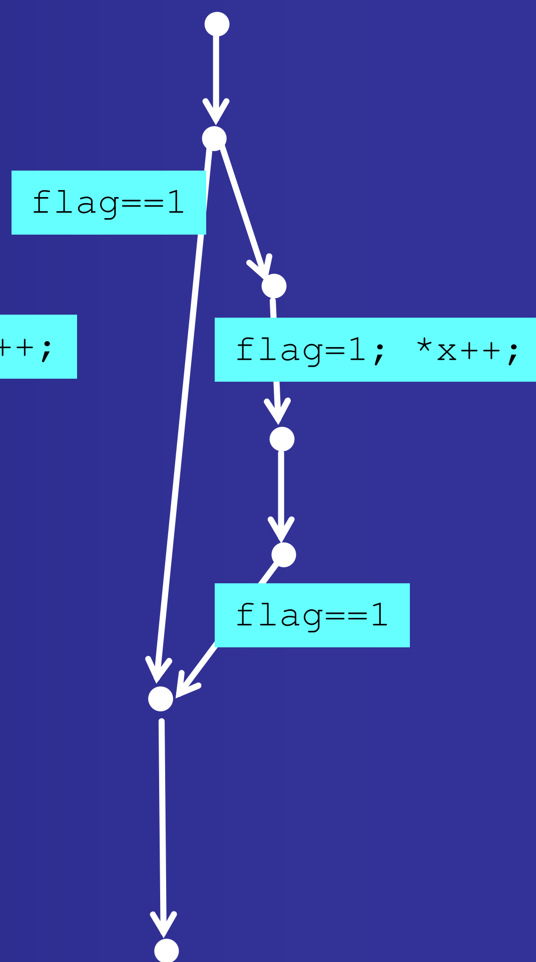
Program Model: Unrolled CFG

Model the program by its Control-Flow Graph (CFG)

- Unroll loops and recursive function calls, inline functions to get a Directed Acyclic Graph (DAG)
- Each source-sink path is a program execution that the tool can generate a test case for

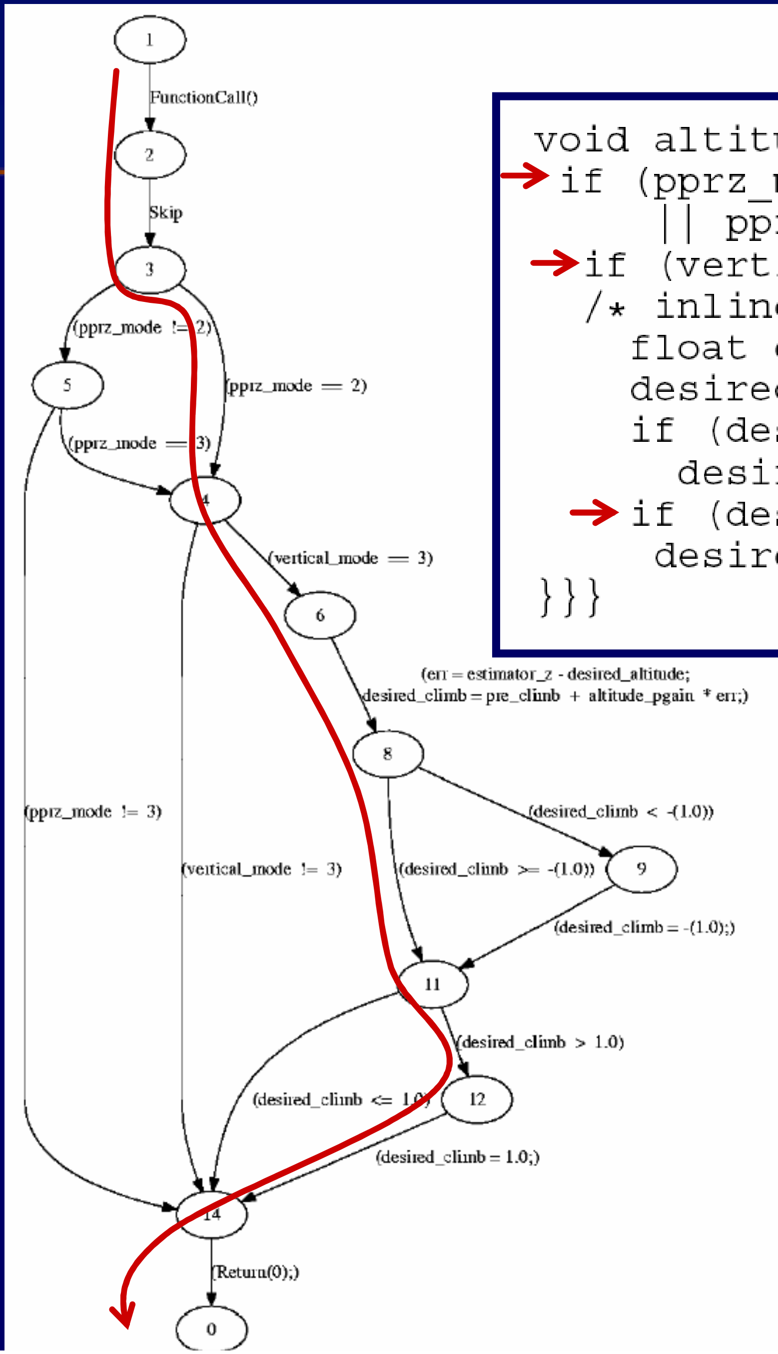


CFG for program containing loop with bound = 1



CFG unrolled to a DAG

Tool Strategies = Paths



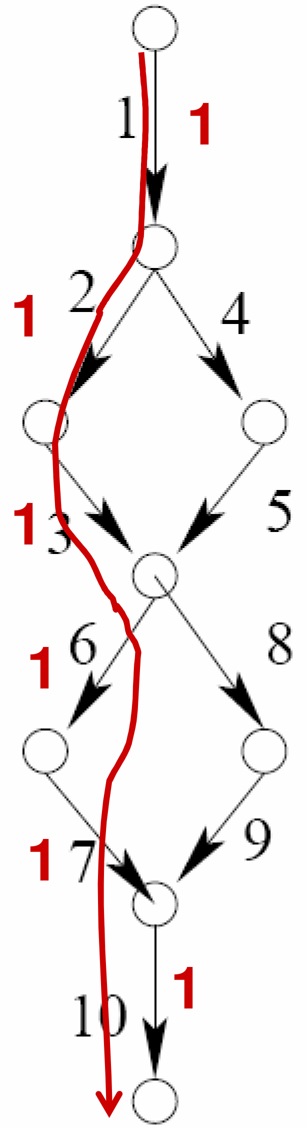

```
void altitude_control_task(void) {
 if (pprz_mode == PPRZ_MODE_AUTO2
     || pprz_mode == PPRZ_MODE_HOME) {
  if (vertical_mode == VERTICAL_MODE_AUTO_ALT) {
   /* inlined below: function  altitude_pid_run(); */
    float err = estimator_z - desired_altitude;
    desired_climb = pre_climb + altitude_pgain * err;
    if (desired_climb < -CLIMB_MAX)
      desired_climb = -CLIMB_MAX;
    if (desired_climb > CLIMB_MAX)
     desired_climb = CLIMB_MAX;
}}}
```

Tool strategy:
Paths in the control flow graph

Tool selects:
Inputs that drive the program down the chosen path

A Path is a Vector $x \in \{0,1\}^m$

(m = #edges)



x1 = (1,1,1,0,0,1,1,0,0,1)

x2 = (1,0,0,1,1,0,0,1,1,1)

x3 = (1,1,1,0,0,0,0,1,1,1)

x4 = (1,0,0,1,1,1,1,0,0,1)

Insight:
Only need to sample
a Basis
of the space of paths

Platform Model

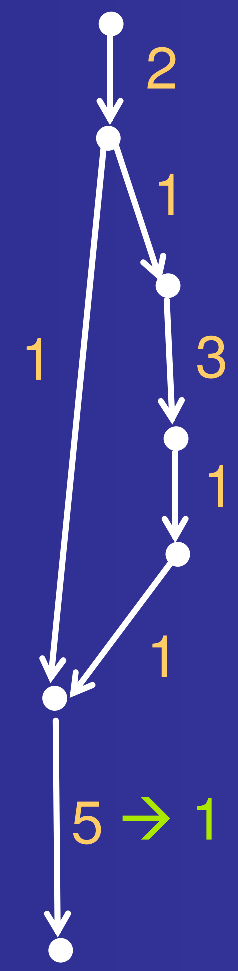
Models path-independent timing

Weights on edges of unrolled CFG

&

Path-specific perturbation

Models path-dependent timing



What we want to model:

- Impact of the platform on program execution time
- Lengths of all program paths

Platform's Strategies

Weights on edges of unrolled CFG $\mathbf{w} \in \mathbb{R}^{\mathbf{m}}$

&

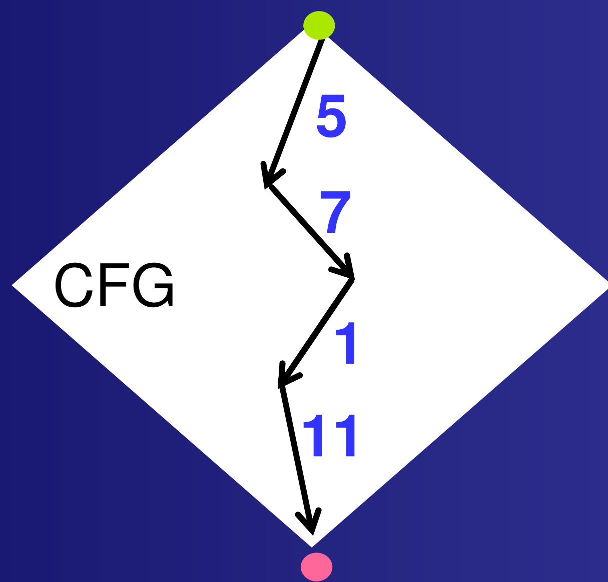
Path-specific perturbation $\boldsymbol{\pi} \in \mathbb{R}^{\mathbf{m}}$

The Game & Winning Condition

Played over several rounds $t = 1, 2, 3, \ldots, \tau$

At each round t:

Tool picks x_t



Platform picks w_t

Platform picks π_t

(-1, -1, -1, -1)

Tool observes $l_t = x_t \cdot (w_t + \pi_t)$

(5+7+1+11) - 4 = 20

At round τ : Tool predicts longest path x^*_τ

- Tool wins if its prediction is correct

Summary of Experimental Results

- GameTime is Efficient
  – 7×10^{16} total paths, vs. 183 basis paths
- Sampling basis paths tells us about longer paths we do not sample
  – Found paths 25% longer than sampled basis
- GameTime can accurately estimate the timing profile with few measurements
- GameTime does better than Random Testing
  – Found estimates twice as large
- GameTime *can even find larger WCET estimates* than conservative WCET estimation tools

Open Problems

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