Department of Computer Science
National Tsing Hua University

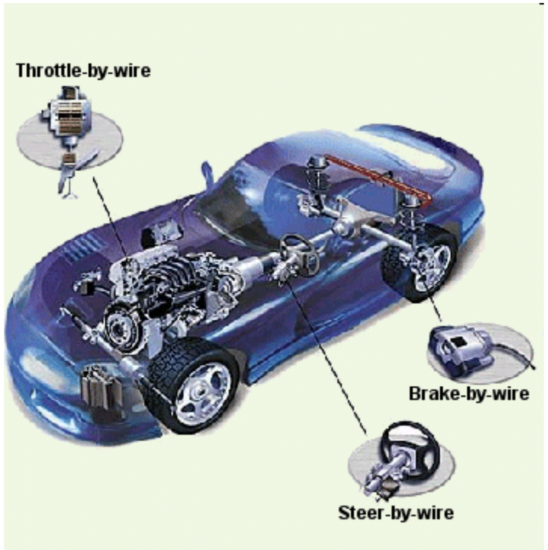
CS 5244: Introduction to Cyber Physical Systems

Unit 17: Execution Time Analysis (Ch.15)

Instructor: Cheng-Hsin Hsu

Acknowledgement: The instructor thanks Profs. Edward A. Lee & Sanjit A. Seshia at UC Berkeley for sharing their course materials

Quantitative Analysis / Verification

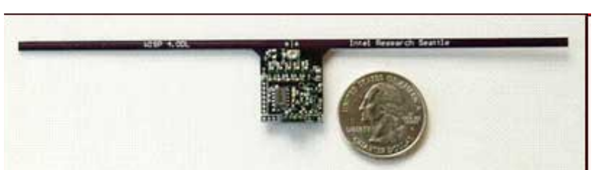


Does the brake-by-wire software always actuate the brakes within 1 ms?

Safety-critical embedded systems

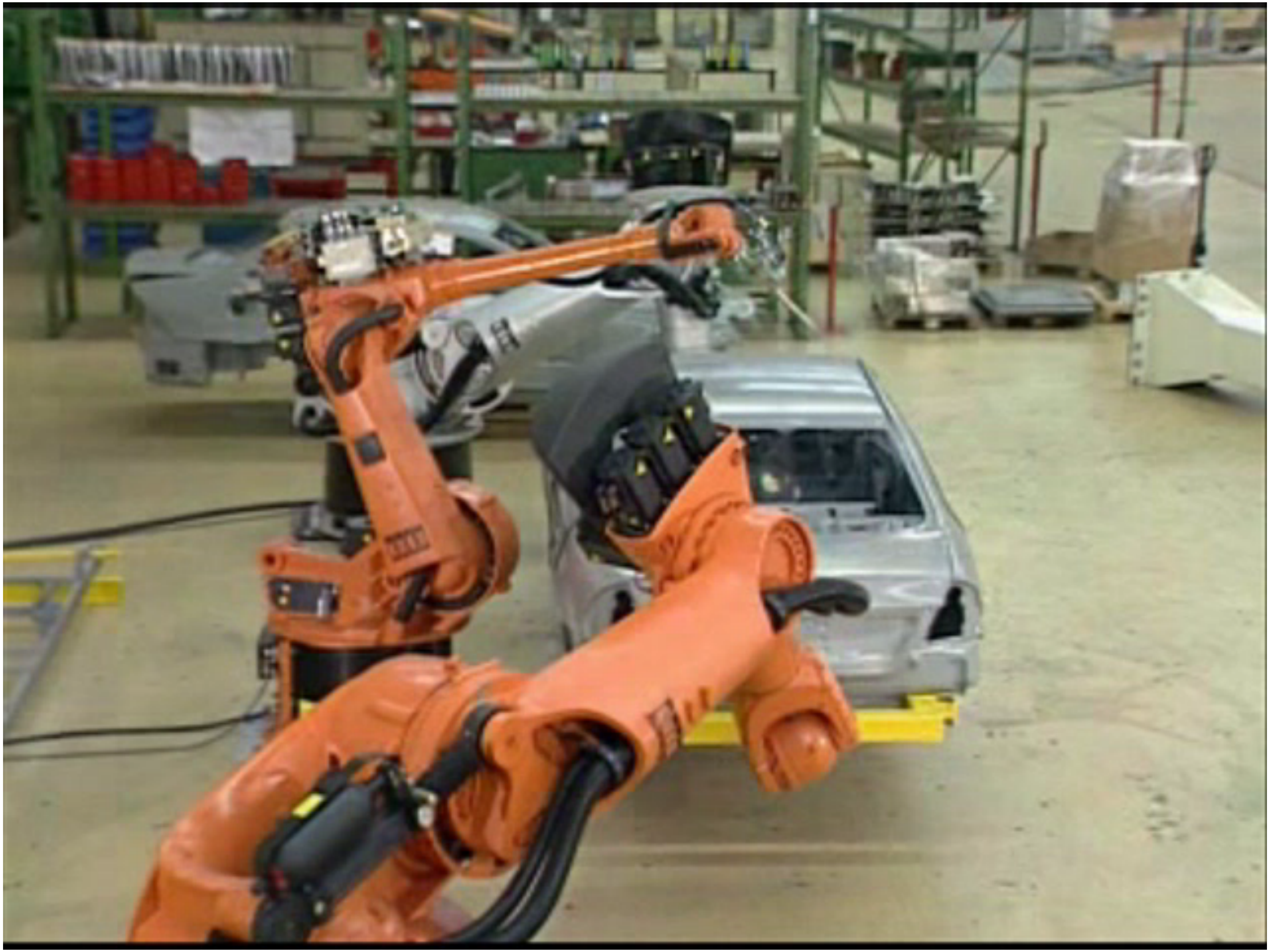
Can this new app drain my iPhone battery in an hour?

Consumer devices



How much energy must the sensor node harvest for RSA encryption?

Energy-limited sensor nets, bio-medical apps, etc.



Courtesy of Kuka Robotics Corp.

Time is Central to Cyber-Physical Systems

Several timing analysis problems:

- Worst-case execution time (WCET) estimation
- Estimating 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

ALL involve predicting an execution time property!

References

Material in this lecture is drawn from the following sources:

- **Chapter 15 of Lee and Seshia. See http://leeseshia.org**
- "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 journal version is "Quantitative Analysis of Systems Using Game-Theoretic Learning", ACM TECS.

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

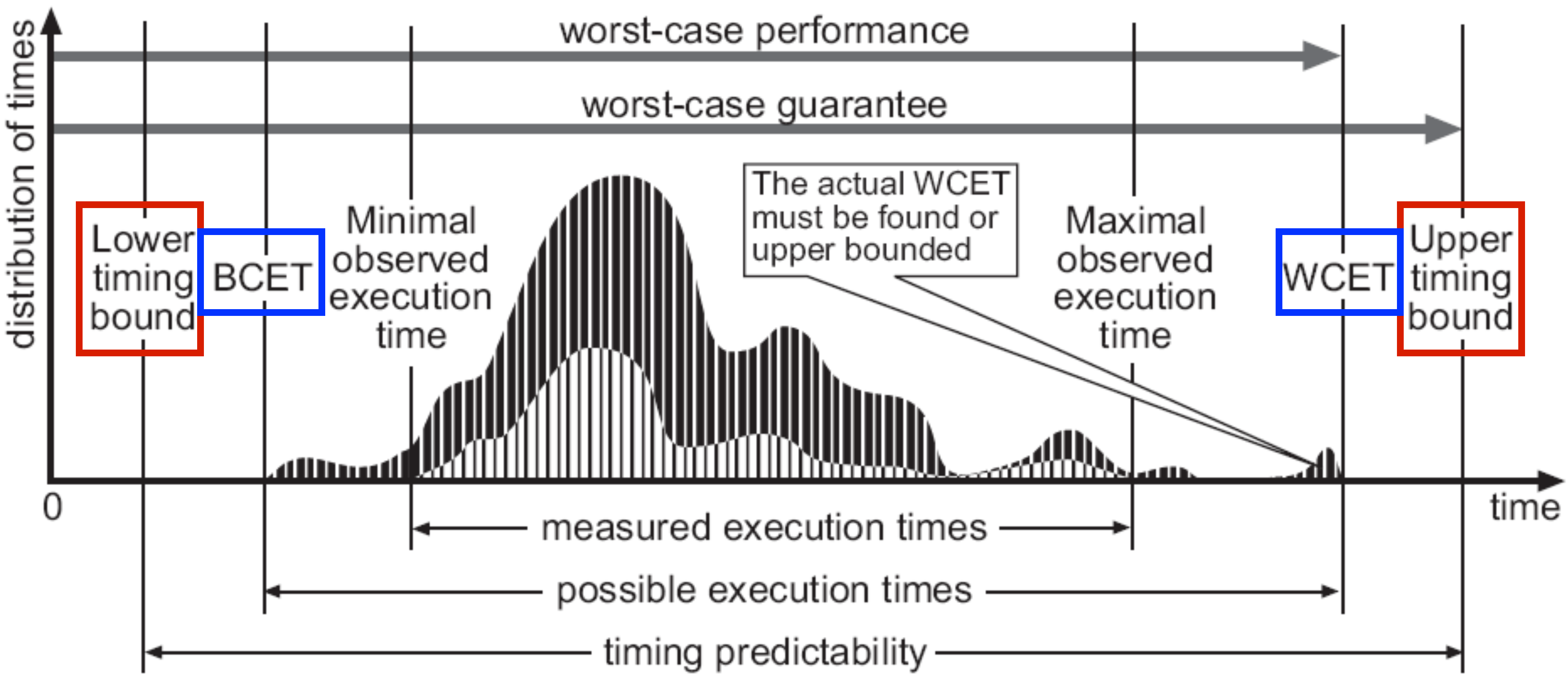


Figure from R.Wilhelm et al., ACM Trans. Embed. Comput. Sys, 2007.

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

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

- Programs as Graphs
- Challenges of Execution Time Analysis
- Current Approaches; Measuring Execution Time
- Limitations and Future Directions

Example Program: Modular Exponentiation

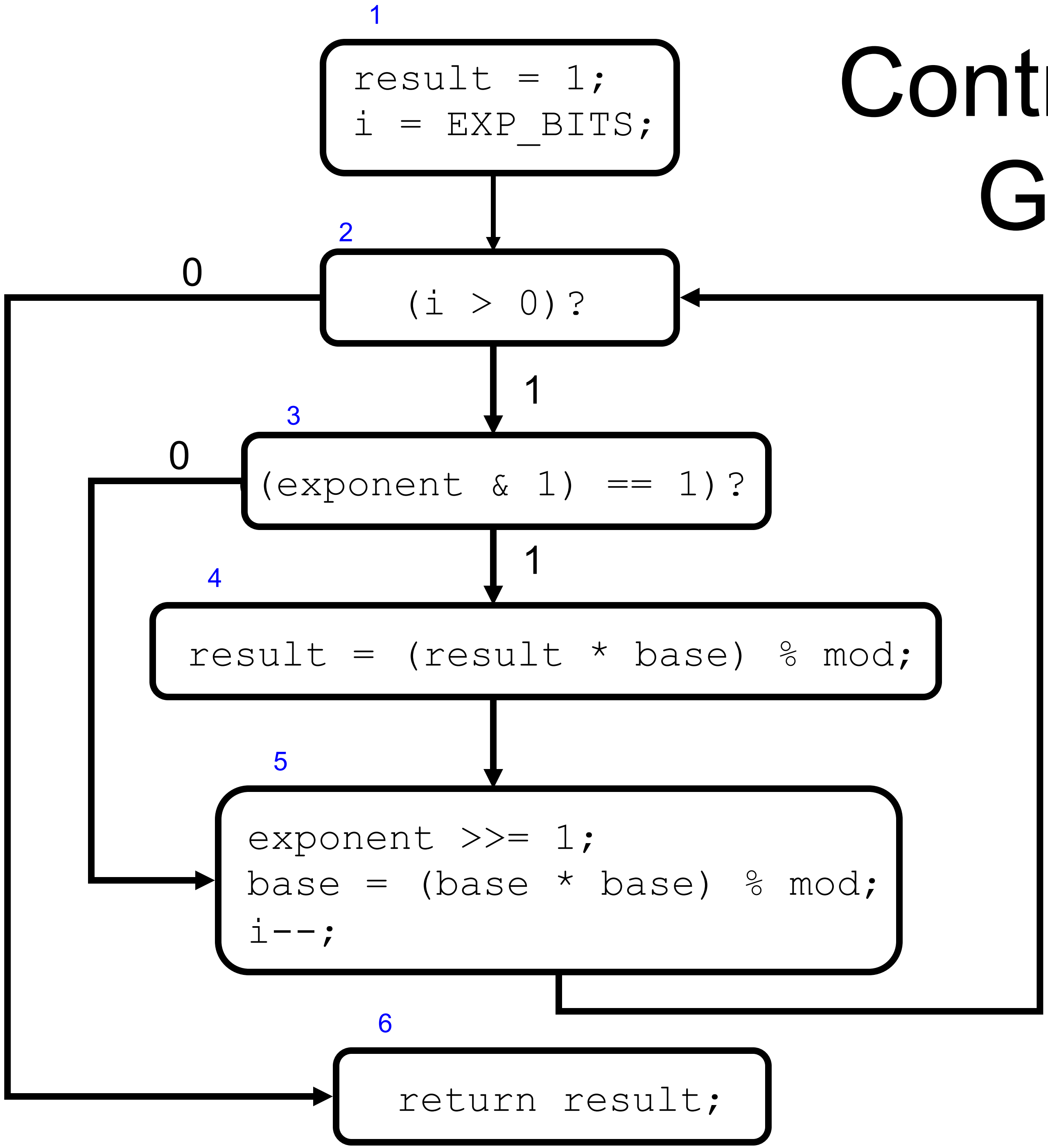
```
#define EXP_BITS 32

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;
}
```

Control-Flow Graph



Each node is a basic block

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: 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;
}
```

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)

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

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**?

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

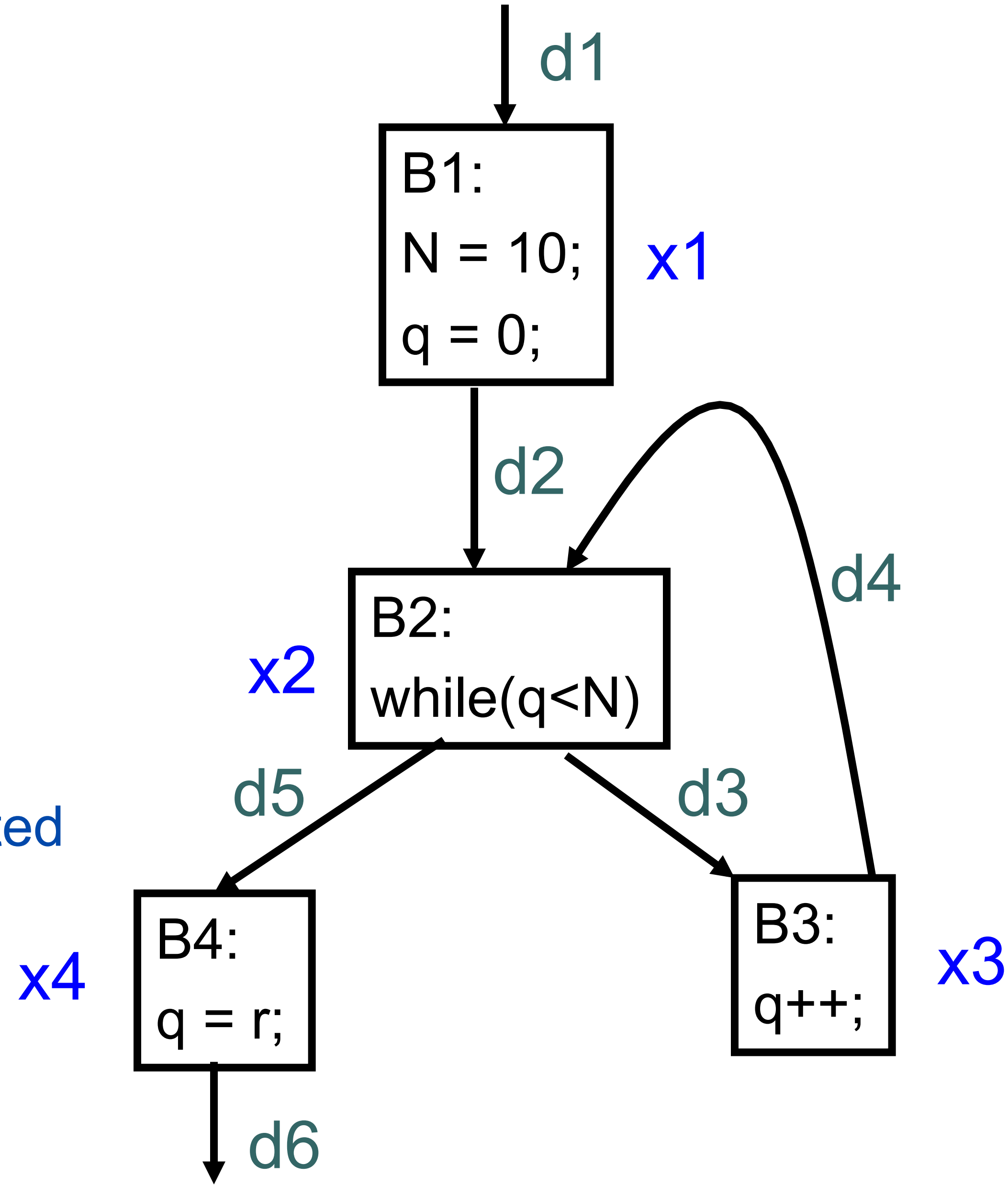
What happens when **n=8**?

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, Revisited

xi → # times Bi is executed

dj → # times edge is executed

C_i → measured upper bound on time taken by Bi

Want to

maximize $\sum_i C_i xi$

subject to constraints

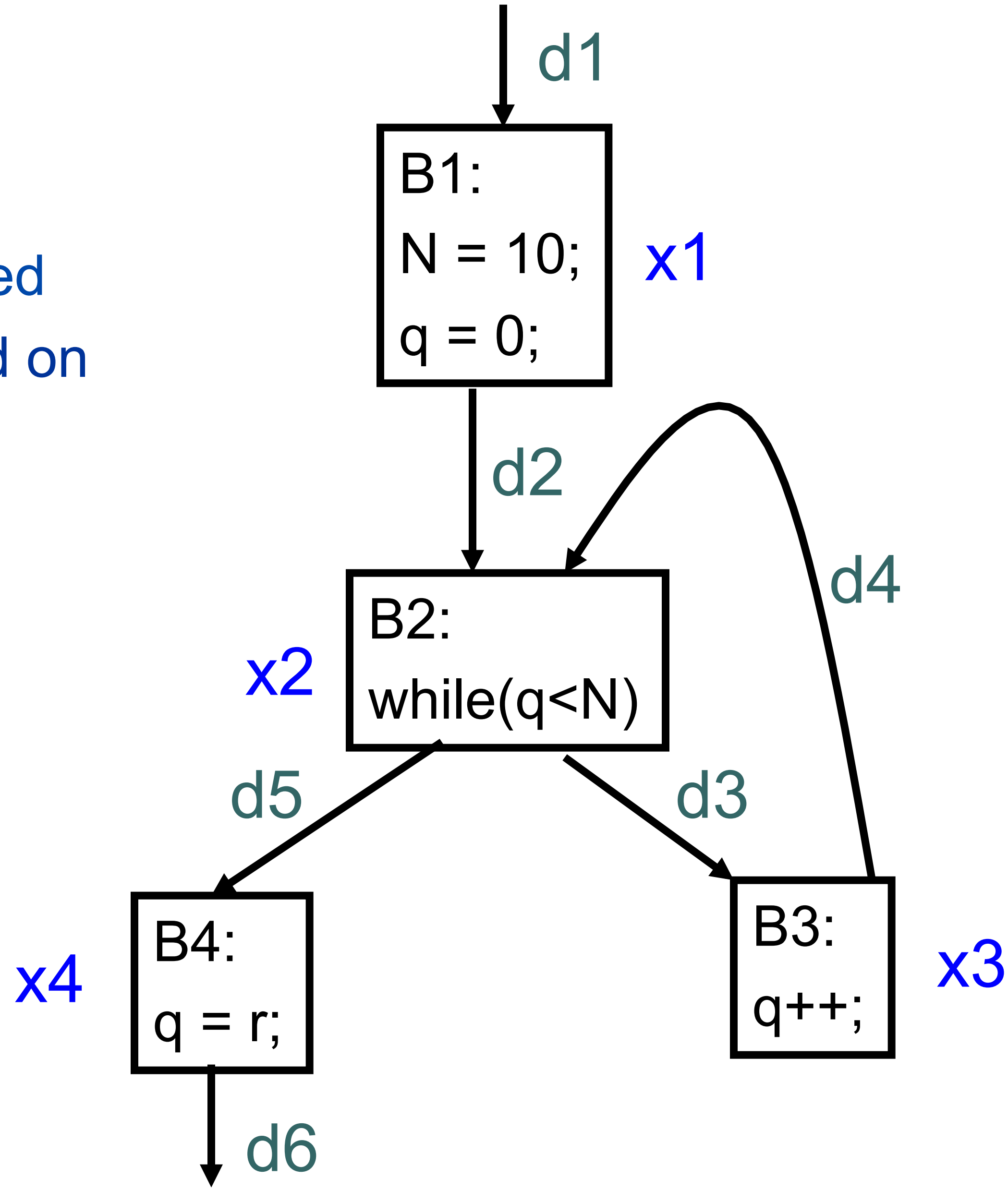
x1 = d1 = d2

d1 = 1

x2 = d2+d4 = d3+d5

x3 = d3 = d4 = 10

x4 = d5 = d6



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

Timing Analysis and Compositionality

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

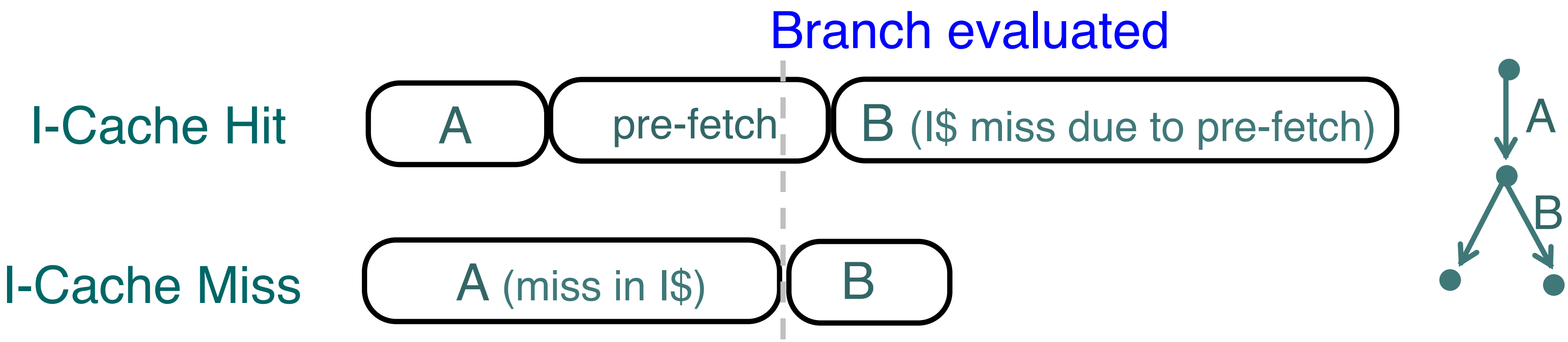
Is WCET(T) = WCET(A) + WCET(B) ?

NOT ALWAYS!

WCETs cannot simply be composed ☹

→ Due to dependencies "through environment"

Timing Anomalies



<u>Scenario 1:</u> 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.

<u>Scenario 2:</u> 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
- …

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]

Measurement Pitfalls

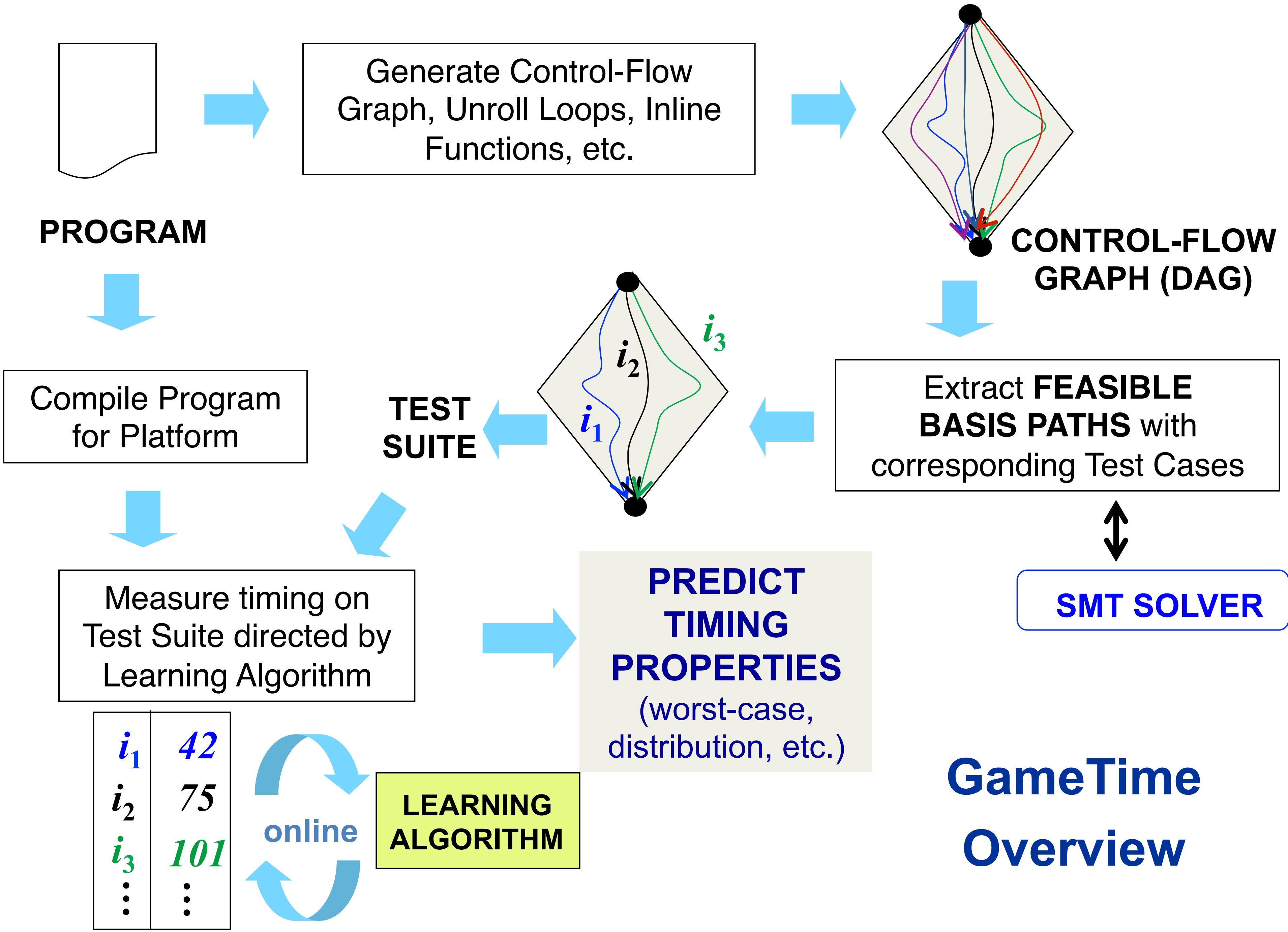
- Instrumentation incurs small overhead
  - measure long enough code sequence to compensate
- Cache effects can skew measurements
  - “warm up” the cache before making measurement
- Multi-tasking effects: counter keeps going even when the task of interest is inactive
  - take multiple measurements and 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

Some WCET Estimation Tools

Commercial Tools: aiT, RapiTime, …

University/Research Tools: **GameTime**, Chronos, …

See sidebar in Ch 15 for more information.



PROGRAM
Generate Control-Flow Graph, Unroll Loops, Inline Functions, etc.
CONTROL-FLOW GRAPH (DAG)
Extract FEASIBLE BASIS PATHS with corresponding Test Cases
SMT SOLVER
TEST SUITE
i_1
i_2
i_3
Compile Program for Platform
Measure timing on Test Suite directed by Learning Algorithm
i_1 42
i_2 75
i_3 101
online
LEARNING ALGORITHM
PREDICT TIMING PROPERTIES (worst-case, distribution, etc.)
GameTime Overview

Open Problems

- Architectures are getting much more complex.
  - Can we create processor models without the agonizing pain?
  - Can we change the architecture to make timing analysis easier? [See PRET machine project]
- Analysis methods are "Brittle" – small changes to code and/or architecture can require completely re-doing the WCET computation
  - 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

Dealing with Overhead & Cache Effects

- Always execute function once to “warm up” cache
- Keep doubling number of times execute P() until reach

```
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]