



Introduction to Embedded Systems

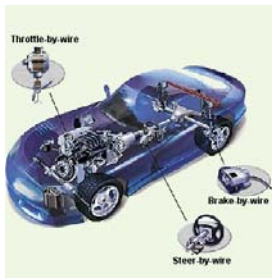
Sanjit A. Seshia

UC Berkeley
EECS 149/249A
Fall 2015

© 2008-2015: E. A. Lee, A. L. Sangiovanni-Vincentelli, S. A. Seshia. All rights reserved.

Chapter 16: Quantitative (Execution Time) Analysis

Quantitative Analysis / Verification



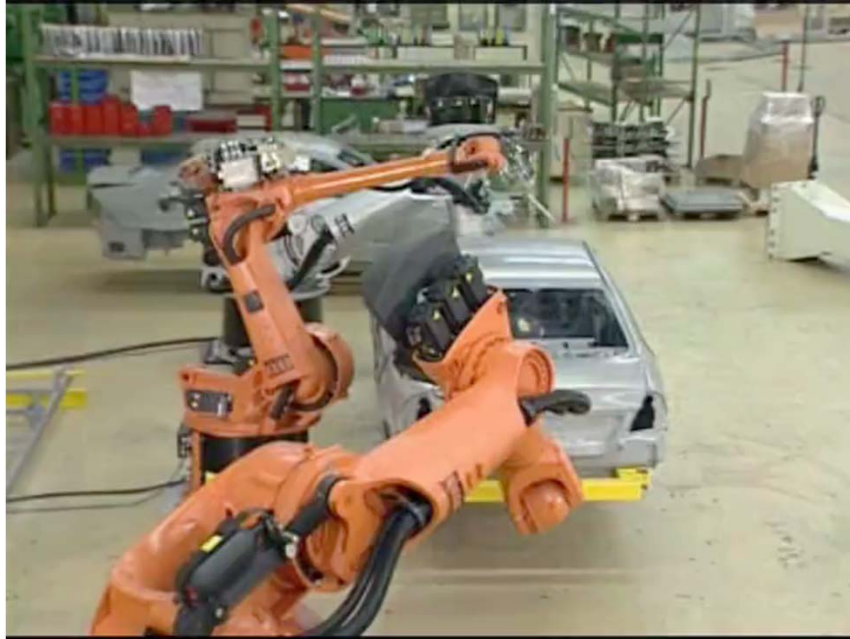
Does the brake-by-wire software always actuate the brakes within 1 ms?
Safety-critical embedded systems

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.

Quantitative Analysis, UC Berkeley: 2



Courtesy of Kuka Robotics Corp.

Quantitative Analysis, UC Berkeley: 3

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

Various forms of the same basic problem.

Quantitative Analysis, UC Berkeley: 4

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)

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

What's the WCET? BCET?

Quantitative Analysis, UC Berkeley: 5

Worst-Case Execution Time (WCET) & BCET

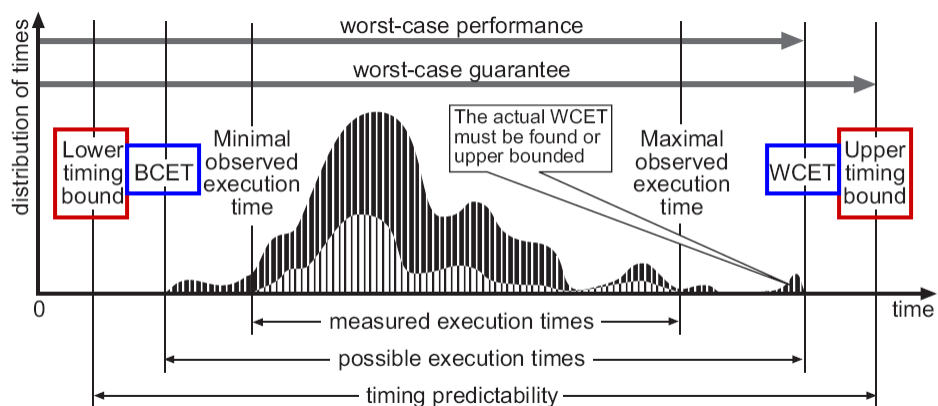


Figure from R.Wilhelm et al., ACM Trans. Embed. Comput. Sys, 2007.

Quantitative Analysis, UC Berkeley: 6

References

Material in this lecture is drawn from the following sources:

- **Chapter 16 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, 2012.

Quantitative Analysis, UC Berkeley: 7

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 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*.

Quantitative Analysis, UC Berkeley: 8

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

Quantitative Analysis, UC Berkeley: 9

Outline of the Lecture

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

Optional reading assignment: extra slides on measuring execution time using CPU cycle counter

EECS 249, UC Berkeley: 10

Example Program: Modular Exponentiation

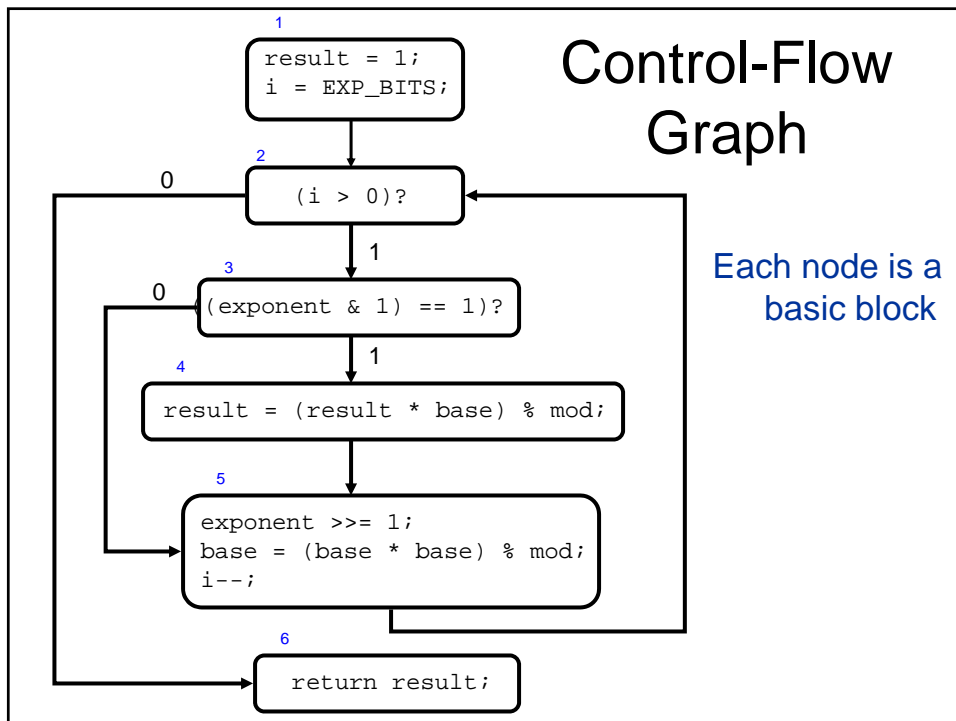
```

1 #define EXP_BITS 32
2
3 typedef unsigned int UI;
4
5 UI modexp(UI base, UI exponent, UI mod) {
6     int i;
7     UI result = 1;
8
9     i = EXP_BITS;
10    while(i > 0) {
11        if ((exponent & 1) == 1) {
12            result = (result * base) % mod;
13        }
14        exponent >>= 1;
15        base = (base * base) % mod;
16        i--;
17    }
18    return result;
19 }

```

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

EECS 249, UC Berkeley: 11



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

Quantitative Analysis, UC Berkeley: 13

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;
}
```

How many paths?

2^{10000}

Example cnt.c from WCET benchmarks, Mälardalen Univ.

Quantitative Analysis, UC Berkeley: 14

Program Path Analysis: Determining Loop Bounds

```
1 #define EXP_BITS 32
2
3 typedef unsigned int UI;
4
5 UI modexp(UI base, UI exponent, UI mod) {
6     int i;
7     UI result = 1;
8
9     i = EXP_BITS;
10    while(i > 0) {
11        if ((exponent & 1) == 1) {
12            result = (result * base) % mod;
13        }
14        exponent >>= 1;
15        base = (base * base) % mod;
16        i--;
17    }
18    return result;
19 }
```

How many times around
the while loop?

32

Quantitative Analysis, UC Berkeley: 15

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

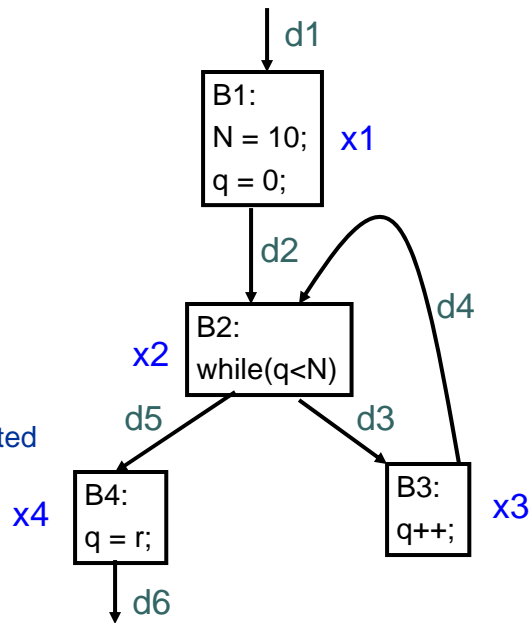
Quantitative Analysis, UC Berkeley: 16

Example

```

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

$x_i \rightarrow$ # times B_i is executed
 $d_j \rightarrow$ # times edge is executed



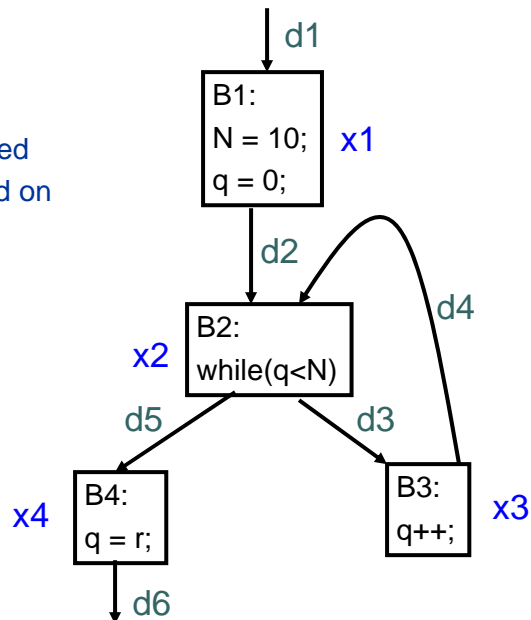
Example due to Y.T. Li and S. Malik

Quantitative Analysis, UC Berkeley: 17

Example

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

Want to
 maximize $\sum_i C_i x_i$
 subject to constraints
 $x_1 = d_1 = d_2$
 $d_1 = 1$
 $x_2 = d_2 + d_4 = d_3 + d_5$
 $x_3 = d_3 = d_4 = 10$
 $x_4 = d_5 = d_6$



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

Quantitative Analysis, UC Berkeley: 18

Integer Linear Programming

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

Want to
maximize $\sum_i C_i x_i$
subject to constraints
 $x_1 = d_1 = d_2$
 $d_1 = 1$
 $x_2 = d_2 + d_4 = d_3 + d_5$
 $x_3 = d_3 = d_4 = 10$
 $x_4 = d_5 = d_6$

The problem to solve is:

Find integer values x_i
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.

Quantitative Analysis, UC Berkeley: 19

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

Quantitative Analysis, UC Berkeley: 20

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:

$$x_i + x_j \leq 1$$

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

Quantitative Analysis, UC Berkeley: 21

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"

Quantitative Analysis, UC Berkeley: 22

Processor Behavior Analysis: Cache Effects

```

1 float dot_product(float *x, float *y, int n) {
2     float result = 0.0;
3     int i;
4     for(i=0; i < n; i++) {
5         result += x[i] * y[i];
6     }
7     return result;
8 }

```

Suppose:

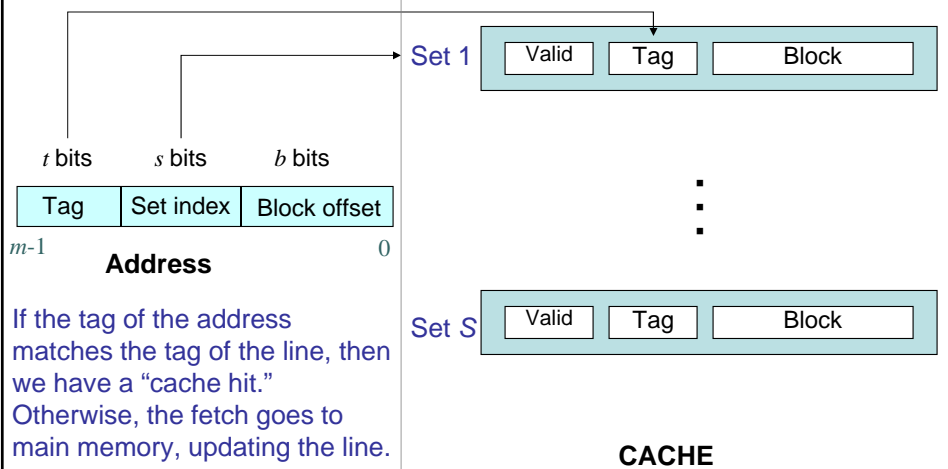
- 32-bit processor
- Direct-mapped cache holds two sets
 - 4 floats per set
 - x and y stored contiguously starting at address 0x0

What happens when $n=2$?

Quantitative Analysis, UC Berkeley: 23

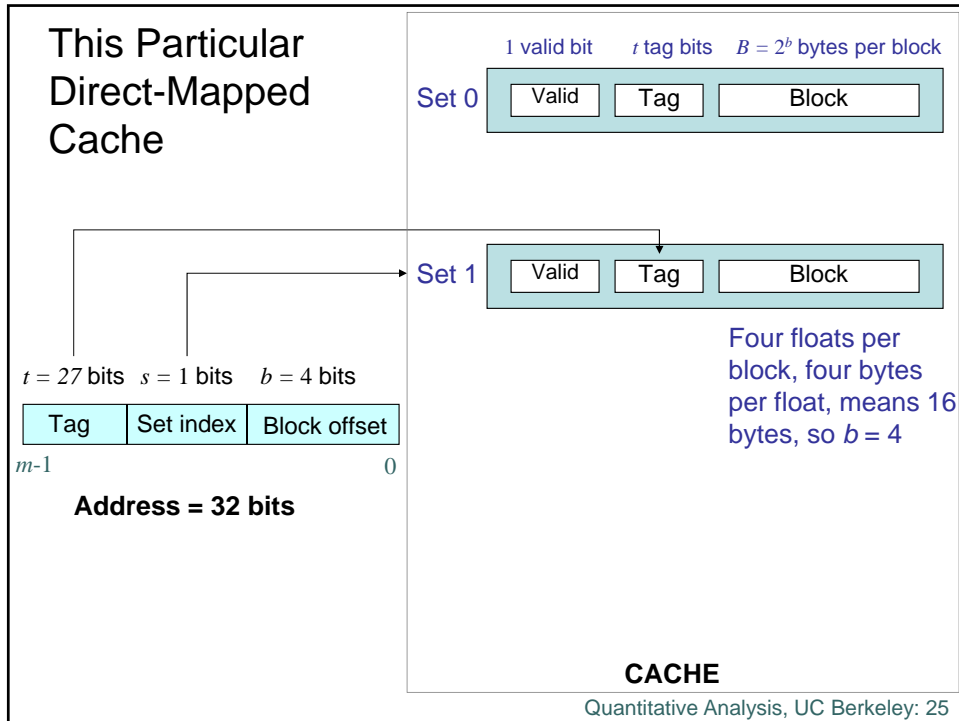
Recall Direct-Mapped Cache

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.

Quantitative Analysis, UC Berkeley: 24



Processor Behavior Analysis: Cache Effects

```

1 float dot_product(float *x, float *y, int n) {
2   float result = 0.0;
3   int i;
4   for(i=0; i < n; i++) {
5     result += x[i] * y[i];
6   }
7   return result;
8 }
  
```

What happens when $n=2$?

Suppose:

- 32-bit processor
- 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.

Quantitative Analysis, UC Berkeley: 26

Processor Behavior Analysis: Cache Effects

```

1 float dot_product(float *x, float *y, int n) {
2     float result = 0.0;
3     int i;
4     for(i=0; i < n; i++) {
5         result += x[i] * y[i];
6     }
7     return result;
8 }

```

What happens when $n=8$?

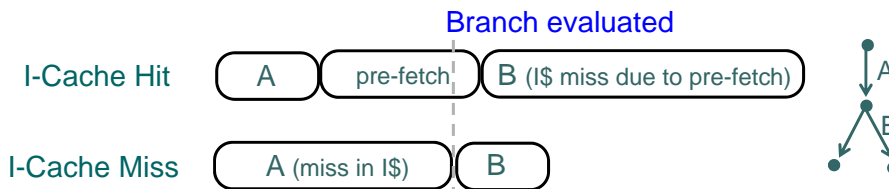
$x[0]$ will miss, pulling $x[0-3]$ into the set 0. Then $y[0]$ will miss, pulling $y[0-3]$ into the same set, evicting $x[0-3]$. Every access will be a miss!

Suppose:

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

Quantitative Analysis, UC Berkeley: 27

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.]

Quantitative Analysis, UC Berkeley: 28

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
- ...

Quantitative Analysis, UC Berkeley: 29

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

Quantitative Analysis, UC Berkeley: 30

Some WCET Estimation Tools

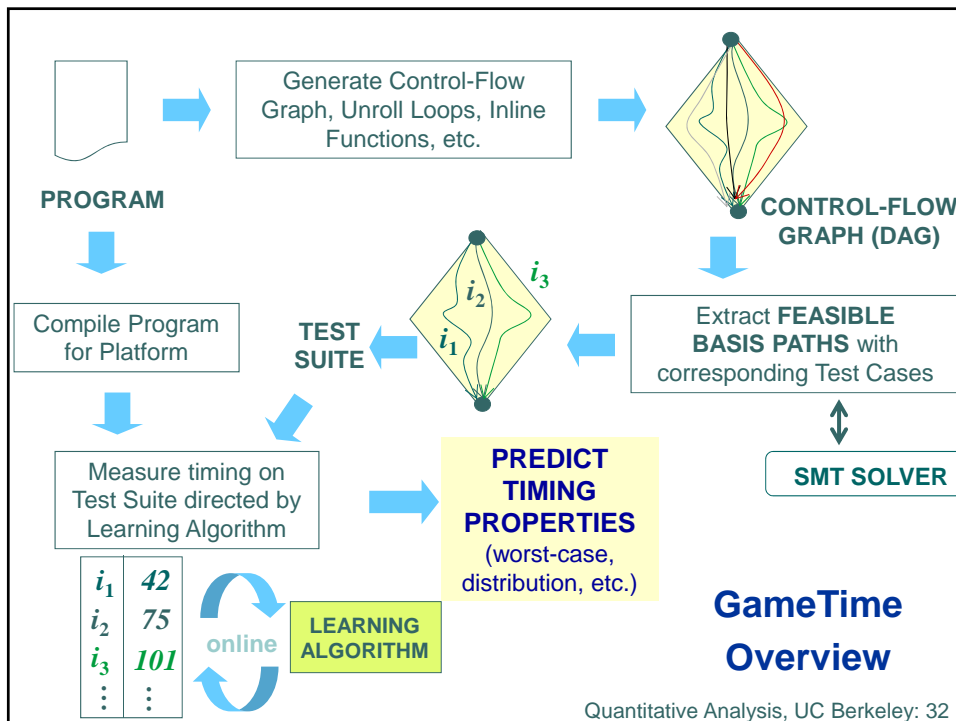
Commercial Tools: aiT, RapiTime, ...

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!

See sidebar in Ch 16 for more information.

Quantitative Analysis, UC Berkeley: 31



Optional Reading Material

Quantitative Analysis, UC Berkeley: 33

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

Quantitative Analysis, UC Berkeley: 34

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]

Quantitative Analysis, UC Berkeley: 35

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]

Quantitative Analysis, UC Berkeley: 36

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]

Quantitative Analysis, UC Berkeley: 37

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]

Quantitative Analysis, UC Berkeley: 38

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]

Quantitative Analysis, UC Berkeley: 39

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();
    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);
```

Quantitative Analysis, UC Berkeley: 40

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]

Quantitative Analysis, UC Berkeley: 41

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

Quantitative Analysis, UC Berkeley: 42