

Embedded Systems Design and Modeling



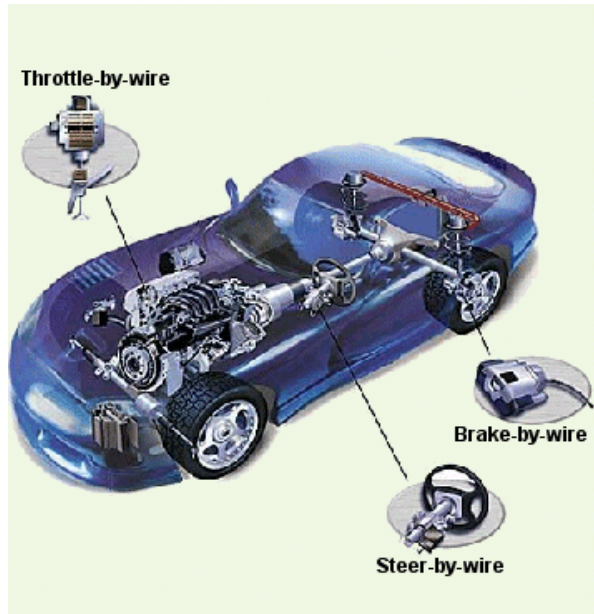
Chapter 16 Quantitative Analysis

Outline

Outline of the Lecture

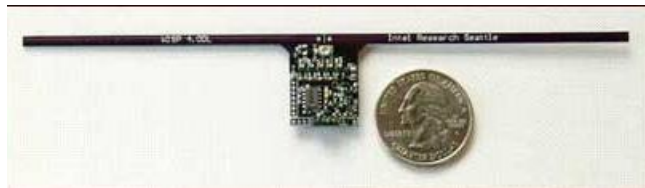
- Programs as Graphs
- Challenges of Execution Time Analysis
- Current Approaches; Measuring Execution Time
- Limitations and Future Directions

Quantitative Analysis / Verification



**Does the brake 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,
biomedical apps, etc.**

CPS Properties

- ❑ Cyber-physical systems properties:
 1. Qualitative:
 - ❑ Correctness
 - ❑ Reachability
 - ❑ Liveness
 - ❑ ...
 2. Quantitative (measurable):
 - ❑ Time to start the response (time to react)
 - ❑ Time to finish the response (execution time)
 - ❑ Resource usage (power, energy, memory, ...)
 - ❑ ...
 - ❑ For each measure, there is a constraint
- Embedded Systems Design and Modeling

Importance of Time

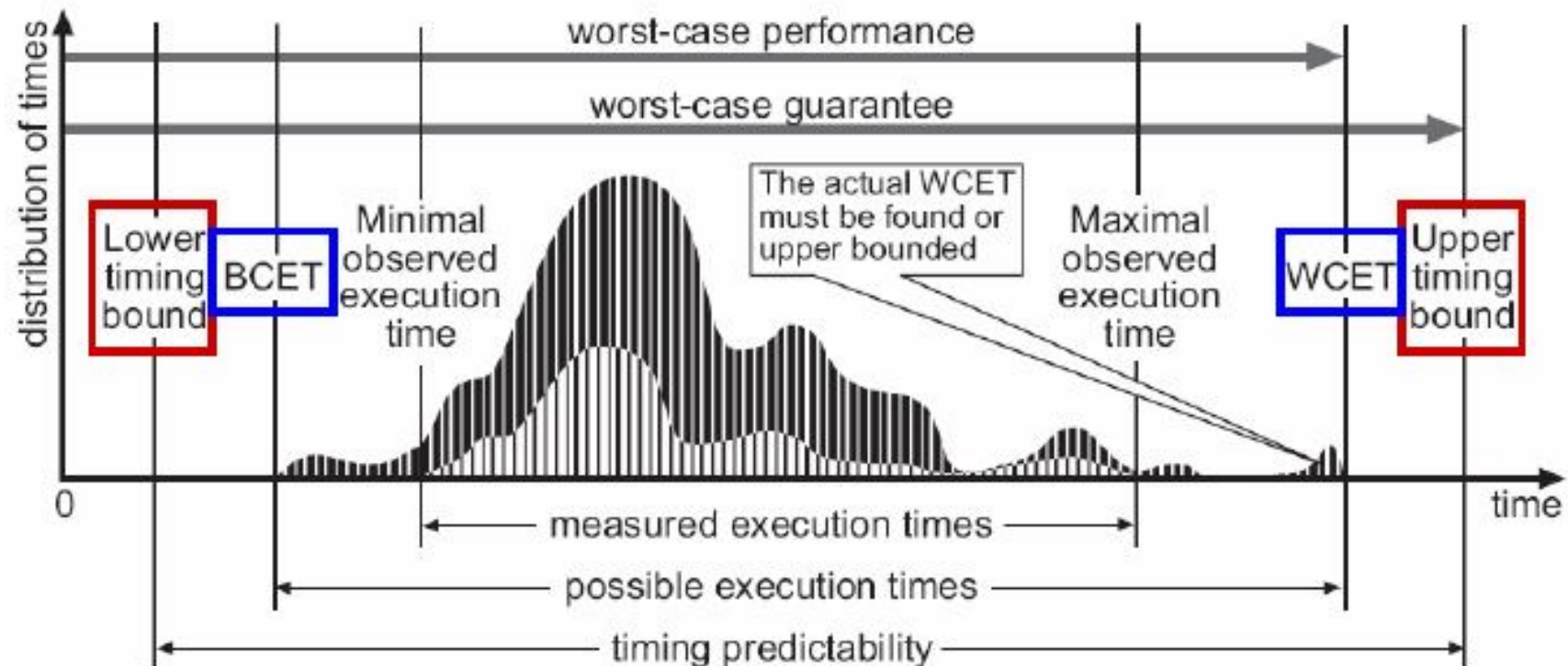
- Time is central to cyber-physical systems
- Several timing analysis problems:
 - Worst-case execution time (WCET) estimation
 - Estimating distribution of execution times
 - Threshold property: can you produce a test case that causes a program to violate its deadline?
 - Software-in-the-loop simulation: predict execution time of particular program path
- These are all various forms of the same basic problem.

WCET Definition

- 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)
- We are often concerned by WCET

WCET Taxonomy

Worst-Case Execution Time (WCET) & BCET



Estimation \neq Measurement

- Often what we need is:
 - An upper bound on the WCET or
 - A lower bound on the BCET
- Tight bound: when the computed bound equals the actual WCET or BCET
- Loose bound: if there is a considerable gap between the actual value and the computed bound
- Computing loose bounds may be much easier than tight bounds.

Problems of Interest

- Extreme case analysis:
 - WCET or BCET estimation or measurement
- Threshold analysis:
 - Instead of the actual WCET or BCET, we look for crossing a given threshold
- Average-Case analysis:
 - Instead of the actual WCET or BCET, we are interested in typical (average) amounts

Issues With WCET

- Given:

1. the code for a software task
2. 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)
- Real-time requirement evaluation

- Can WCET always be found (or estimated)?

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

Turning a Program Into A Graph

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 }
```

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

Defining Basic Blocks

- Basic block:
 - A sequence of consecutive program statements
 - The flow of control enters only at the beginning of this sequence and leaves only at the end
 - Without halting or the possibility of branching except at the end
- Examples in the next slide

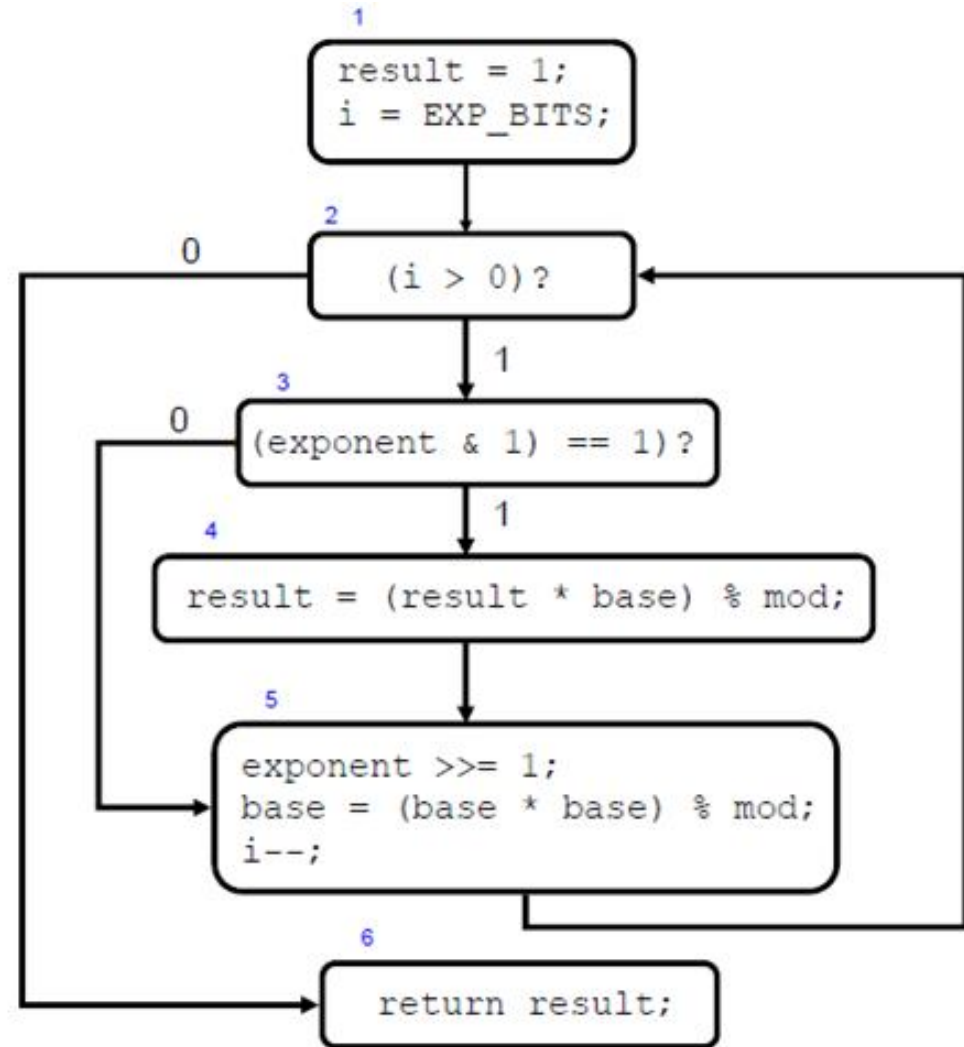
Basic Block Examples

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 }
```

Separating Data From Control

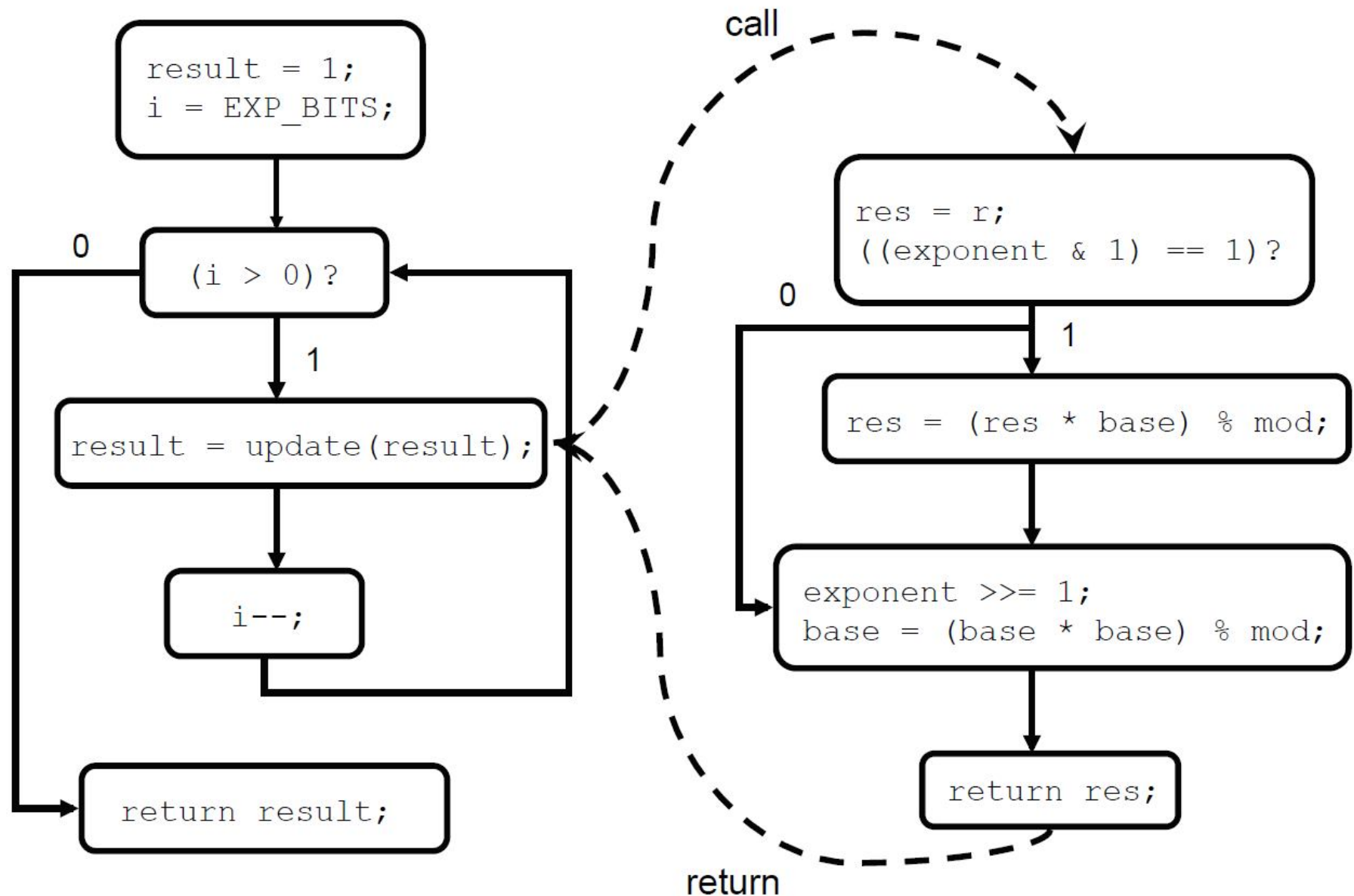
- Control flow graph (CFG):
 - A directed graph
 - Nodes: basic blocks
 - Edges: flow control
 - Possible to have special edges to transfer to and return from function calls



Effect of Function Calls

```
1  #define EXP_BITS 32
2  typedef unsigned int UI;
3  UI exponent, base, mod;
4
5  UI update(UI r) {
6      UI res = r;
7      if ((exponent & 1) == 1) {
8          res = (res * base) % mod;
9      }
10     exponent >>= 1;
11     base = (base * base) % mod;
12     return res;
13 }
14
15 UI modexp_call() {
16     UI result = 1; int i;
17     i = EXP_BITS;
18     while(i > 0) {
19         result = update(result);
20         i--;
21     }
22     return result;
23 }
```


CFG with Function Call



Factors Determining WCET

- ❑ 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

Unclear Loop Bound

- How many times this loop iterates?

```
1  typedef unsigned int UI;
2
3  UI modexp2(UI base, UI exponent, UI mod) {
4      UI result = 1;
5
6      while (exponent != 0) {
7          if ((exponent & 1) == 1) {
8              result = (result * base) % mod;
9          }
10         exponent >>= 1;
11         base = (base * base) % mod;
```

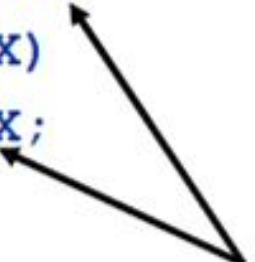
Exponential Path Space

▣ Nested loop: 2^{10000} paths!

```
4  int Ptotal, Pcnt, Ntotal, Ncnt;
5  ...
6  void count() {
7      int Outer, Inner;
8      for (Outer = 0; Outer < MAXSIZE; Outer++) {
9          for (Inner = 0; Inner < MAXSIZE; Inner++) {
10             if (Array[Outer][Inner] >= 0) {
11                 Ptotal += Array[Outer][Inner];
12                 Pcnt++;
13             } else {
14                 Ntotal += Array[Outer][Inner];
15                 Ncnt++;
16             }
17         }
18     }
19 }
```

Path Feasibility

```
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

Memory Hierarchy (1)

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:

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**?

One initial miss,
followed by all hits

Memory Hierarchy (2)

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:

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

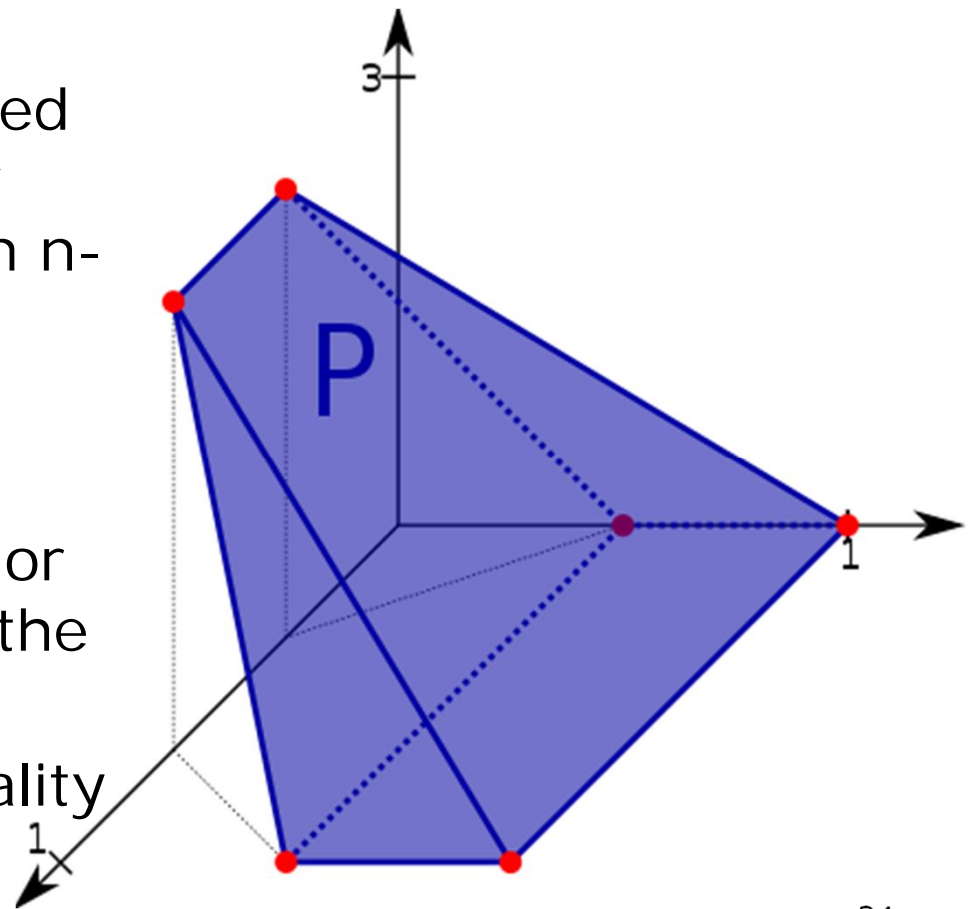
What happens
when **n=8**?

Every access to the
cache will be a miss

Linear Programming

□ Linear programming:

- The process of optimizing a linear function under linear constraints
- Each constraint is represented by an equation or inequality which is a line/plane/... in an n-dimensional space
- All constraints need to be satisfied simultaneously
- The solution (if any) lies on or within the object formed in the n-dimensional space
- In 2D, each equation/inequality is a line, hence “linear”



Integer Linear Programming

- If all solutions are integers, then it is called integer linear programming (ILP)
 - If solutions are either 1 or 0, called 0-1-ILP, ZOLP, or binary linear programming
 - If some solutions are 1 or 0, called mixed ILP (MILP)
 - In general: intractable (NP-hard)
 - But many solvers exist
 - Biggest problem: not scalable
- If at least one inequality is nonlinear, then
 - Use a nonlinear solver
 - Convert to linear and use a linear solver

WCET by ILP

Common Current Approach (high-level)

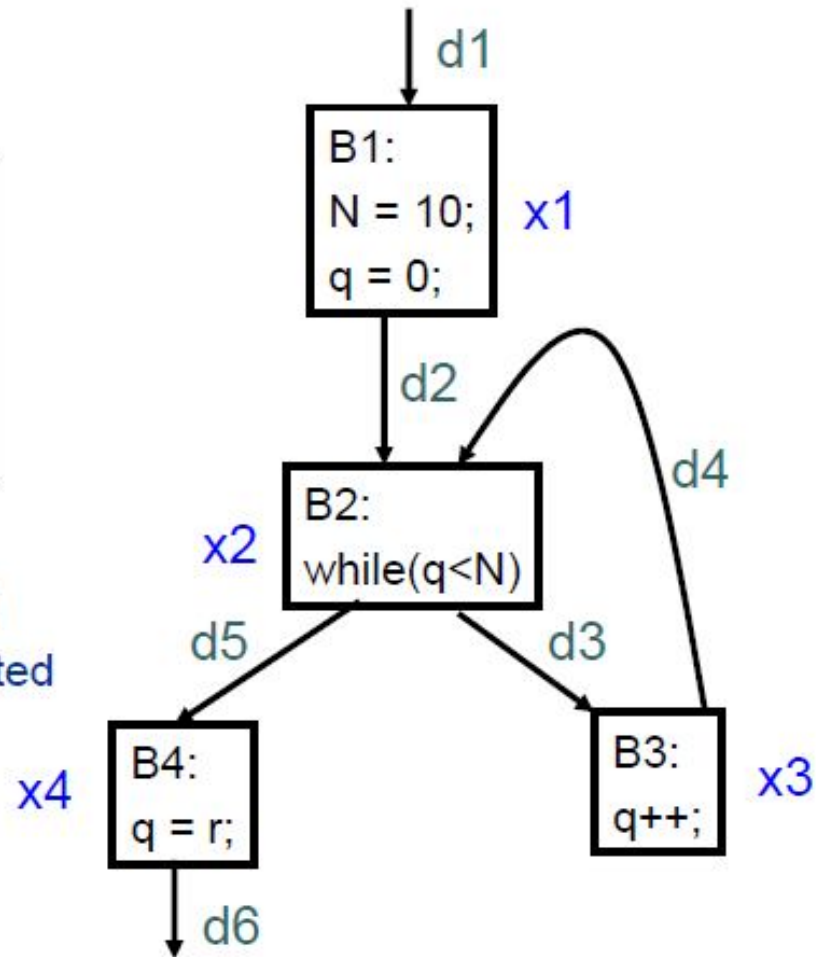
1. Manually construct processor behavior model
2. Use model to find “worst-case” starting processor states for each basic block → measure execution times of the blocks from these states
3. Use these times as upper bounds on the time of each basic block
4. Formulate an integer linear program to find the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

ILP Formulation

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

ILP Formulation (Cont'd)

Example, Revisited

$x_i \rightarrow$ # 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

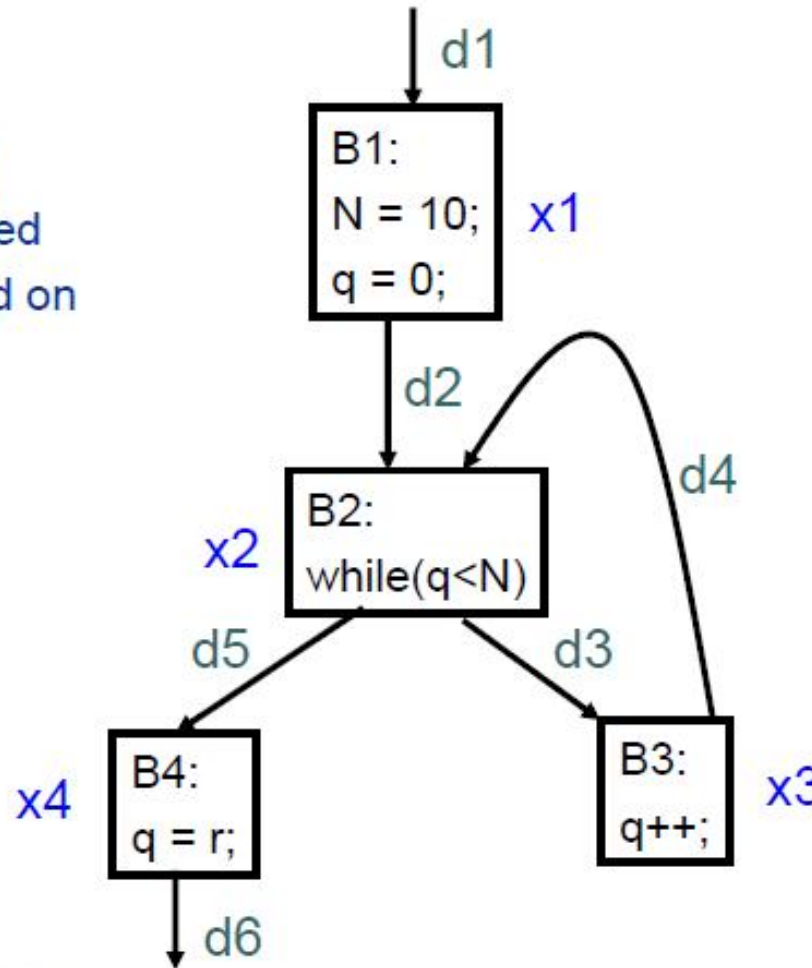
$$x_1 = d_1 = d_2$$

$$d_1 = 1$$

$$x_2 = d_2 + d_4 = d_3 + d_5$$

$$x_3 = d_3 = d_4 = 10$$

$$x_4 = d_5 = d_6$$



Results:

$$x_1 = d_1 = d_2 = 1$$

$$x_3 = d_3 = d_4 = 10$$

$$x_2 = 11$$

$$x_4 = d_5 = d_6 = 1$$

Note that logical flow constraints, cache behavior, or other constraints may be added to equations without increasing complexity!

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”

Focusing on Measurement

- ❑ How to measure run-time?
- ❑ Several techniques, with varying accuracy:
 1. Instrument code to sample CPU cycle counter
 - ❑ relatively easy to do, read processor documentation for assembly instruction
 2. Use cycle-accurate simulator for processor
 - ❑ useful when hardware is not available/ready
 3. 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

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);
}
```


Cycle Counter (Cont'd)

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?

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

Further Thoughts

WCET Open Problems

- ❑ Accurate WCET estimation/measurement requires detailed understanding of the architecture
- ❑ Analysis methods are brittle: small changes in the code and/or architecture can require complete redone
- ❑ Need to deal with concurrency, interrupt, preemption, scheduling, etc.
- ❑ Need more reliable techniques for WCET measurement

Other Quantitative Measures (1)

- Memory bound analysis:
 - Limited memory in embedded systems
 - Need to use memory efficiently
- Analysis methods:
 - Stack size: compute upper bound on stack
 - Generates a call graph to predict stack usage
 - Heap analysis: predict heap usage to avoid software crash or performance degradation
 - Harder than stack bound b/c heap usage depends on input data
 - Also depends on implementation

Other Quantitative Measures (2)

- Power and energy analysis:
 - Limited energy budget in embedded systems
 - Need to manage energy consumption efficiently
 - Energy usage depends on:
 - execution time
 - switching activity
 - both depend on the input data
 - Focus on average power consumption instead
 - Estimation by instruction profiling
 - Software benchmarking

Homework Assignments

- Chapter 16: your choice
- Due date: any time before final exam!