Spring 2009 - Real-Time Systems

http://www.neu-rtes.org/courses/spring2009/

Chapter 2
Worst-Case Execution Time Analysis

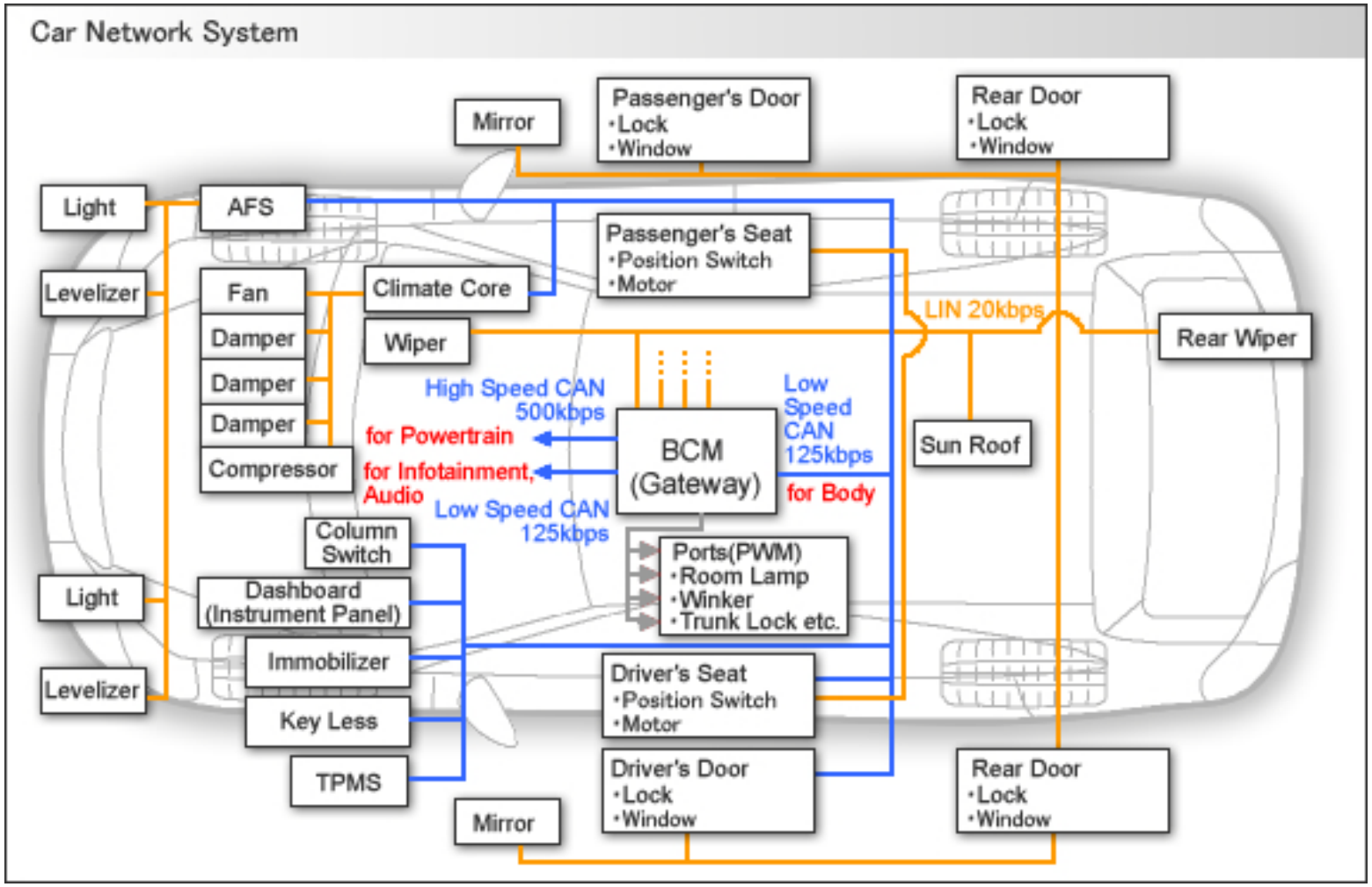
Real-Time Embedded Systems Laboratory
Northeastern University

Objectives

- In this chapter, you are supposed to learn:
  - What is WCET, and why WCET
  - How to obtain the WCET of a program
  - Static analysis methods and measurement-based methods
  - Practices on WCET analysis of RTOS
  - New challenges and future trends on WCET analysis

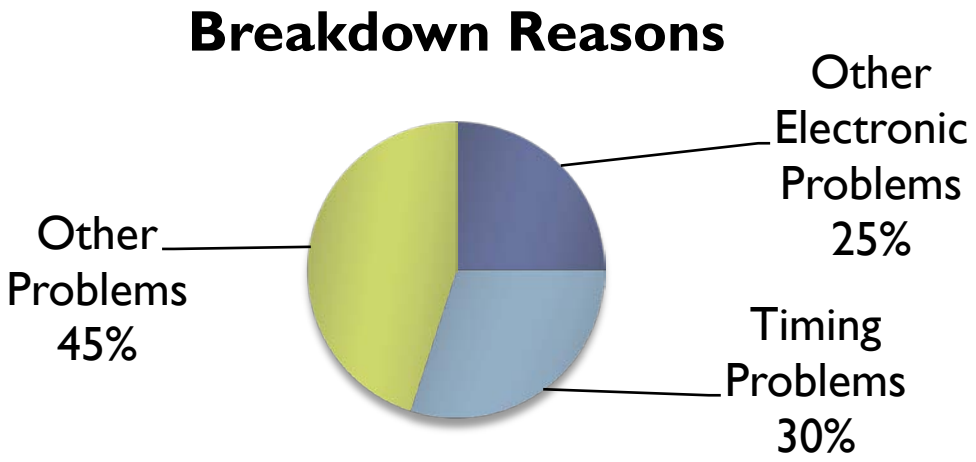
Contents

An Example of Distributed RTS



The Need for Timing Validation

- An Example in Car Industry
  - Today, a new car typically contains 80 ECUs
  - The car electronic systems are provided by multiple OEMs
  - The challenge of integration
  - Increasingly complex processors are used
- Related reports show that



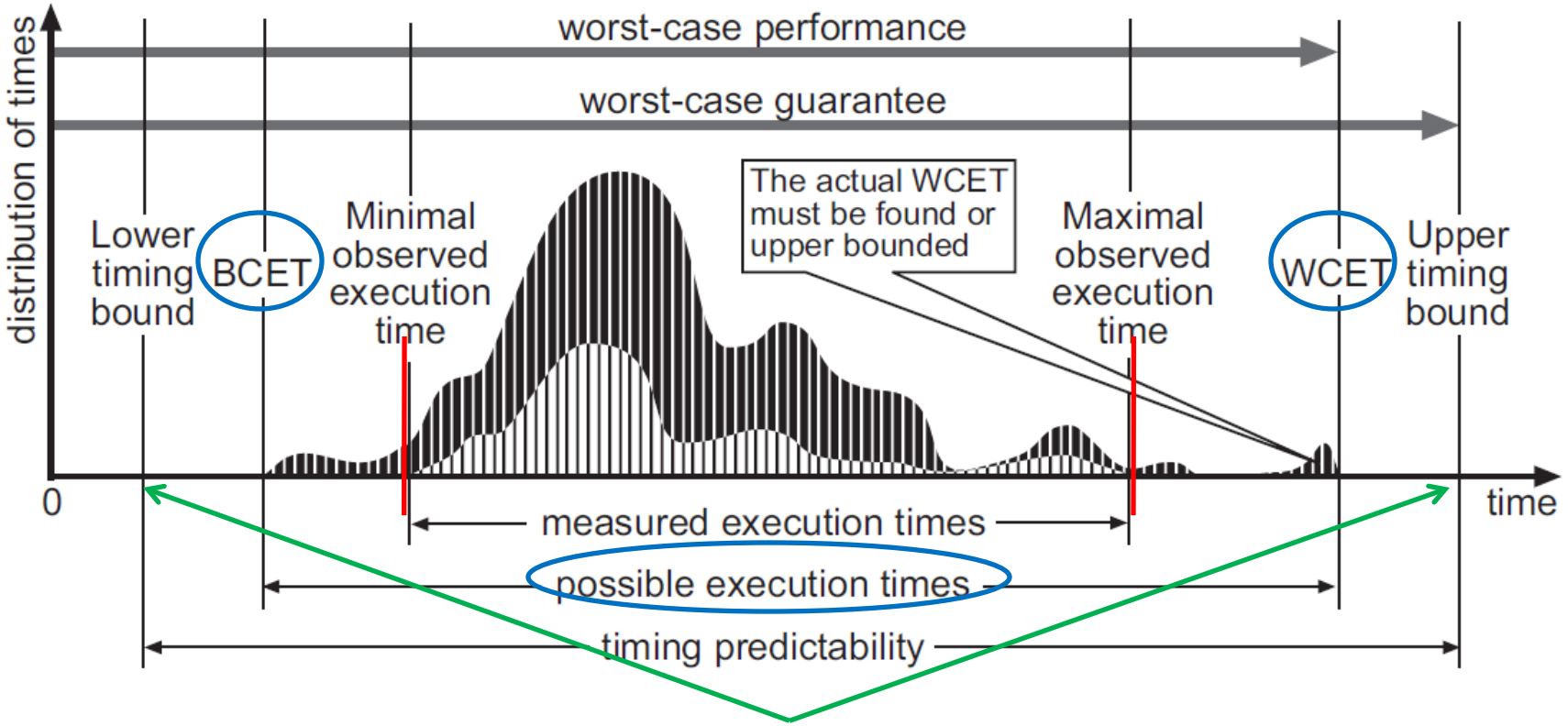
A Simplest Form of Exe. Time Variation

```
Void signal_processing (){
        curr_signal = read_signal();
        if (curr_signal < threshold){
                signal_transformation();    // some +-*/ ops.
        }
        else{
                error_handling_routine();
                // complex error handling operations
        }
}
```

In this signal processing task, the real operations performed depends on the inputted signals. Different signals lead to different operations, then different execution time.

Almost all real-life programs have variable execution time.

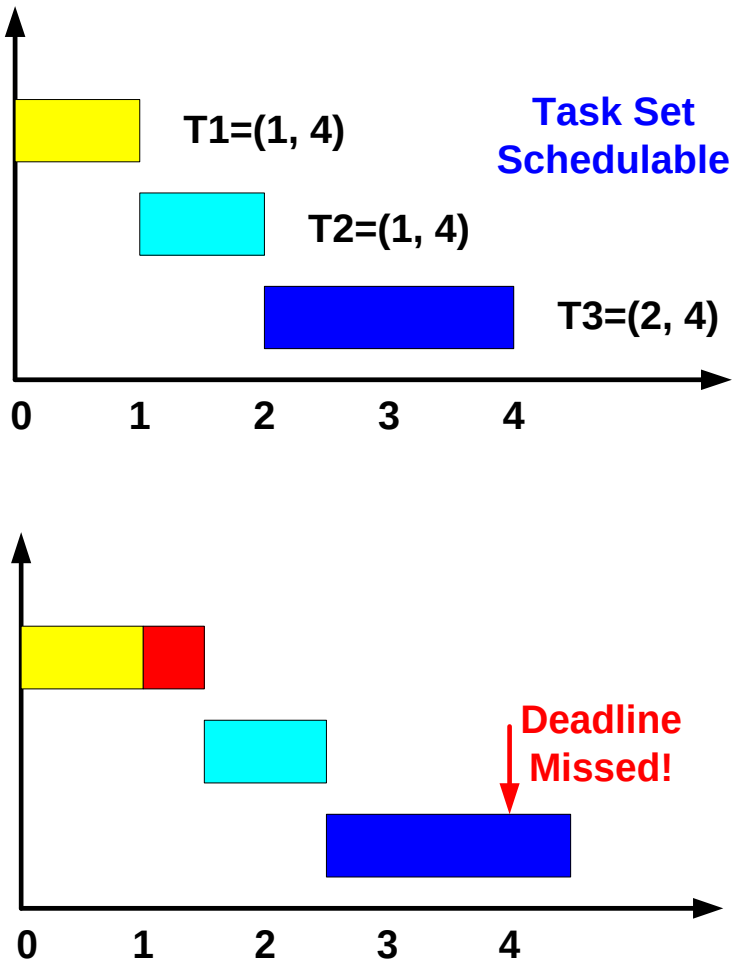
What is WCET?



Worst-case execution time of a task is NOT response time of a task, the latter contains not only execution time, but also the durations of preemptions and blockings.

Why WCET Analysis?

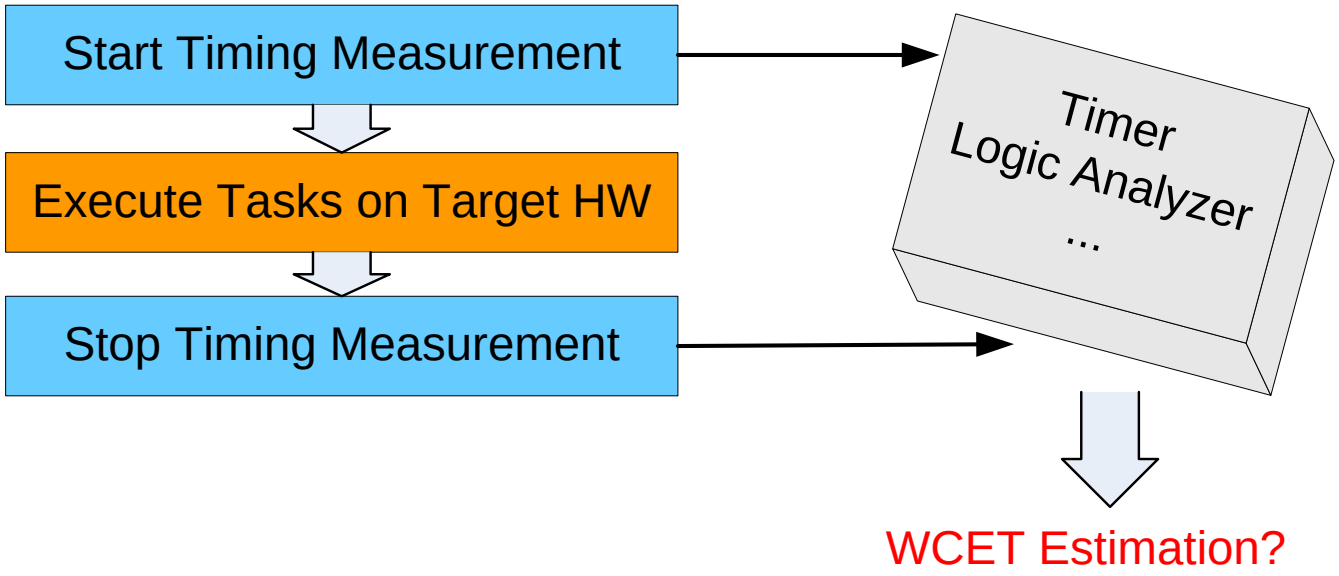
- Hard real-time systems must satisfy stringent timing constraints; whether the constraints are satisfied or not should be analyzed at design time
- Real-time schedulability test requires WCET of each task, and an incorrect result leads to timing failure
- On the right is an example of the result led by incorrectly estimated WCET



WCET Analysis Quality

- Safety:
  - The estimated upper bound should always enclose the actual WCET
- Tightness:
  - The estimated upper bound should be as close as possible to the actual WCET
- Complexity:
  - There is a trade-off between accuracy and analysis complexity
  - Analyzers should balance it according to practical requirements
- The trade-off between analysis complexity and the quality of results

Why Not Just Measure WCET?



Why Not Just Measure WCET?

- Why NOT?
  - It is intractable to cover all execution traces of a program (Think of a program with 10,000 loop iterations and an if-then-else as the loop body, $2^{10,000}$ traces)
  - Hard to guarantee worst-case data input
  - hard to simulate worst-case processor state
  - Need real hardware
- BUT
  - Measurement-based methods are easy to implement
  - Can get a rough estimation of the execution time
  - Compliment with other analysis techniques to make the results trustworthy

Static Analysis Techniques

- How it works?
  - Given a program executable and the hardware the program is running, use mathematical methods to calculate the safe upper bound without any simulation
- Pros
  - Math theorems guarantee safety
  - So mandatory in safe-critical hard real-time systems
- Cons
  - Need to build complex mathematical models
  - Long analysis time for complex programs

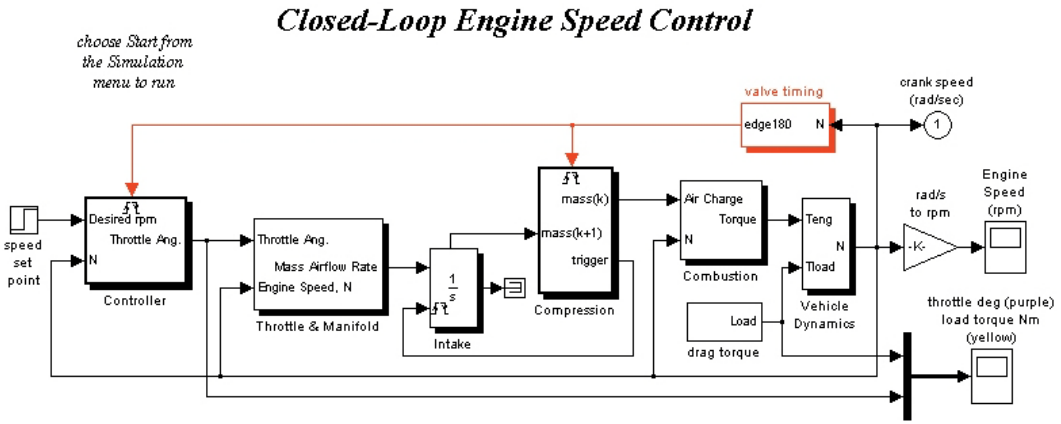
The Ingredients of WCET Analysis

- Flow Facts
  - Flow facts give us information on the control flow of the programs, such as infeasible paths and loop counts, etc.
  - Automatic flow facts extraction and manual annotation
  - How to annotate flow facts in the program

The Ingredients of WCET Analysis

- The Representation Levels of Programs

Matlab/Simulink – Component-based Design



C/C++/Java – High-Level Language

```
Int filtez(int *bpl, int *dlt)
{
    int i;
    long int zl;
    zl = (long)(*bpl++)*(*dlt++);
    for (i = 1; i < 6; i++)
        zl+=(long)(*bpl++)*(*dlt++);
    return ((int) (zl >> 14));    /* x2 here */
}
```

Assembly or Machine Code

```
STMFD   SP!, {LR}
STMFD   SP!, {LR}
STMFD   SP!, {R0-R12}        ; Push registers
MRS     R4,  CPSR            ; Push current CPSR
TST     LR, #1               ; called from Thumb mode?
ORRNE   R4,  R4, #0x20       ; If yes, Set the T-bit
STMFD   SP!, {R4}
```

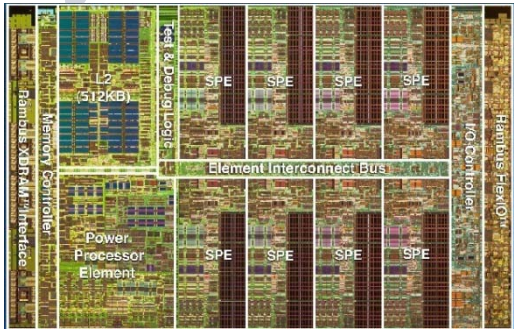
The Ingredients of WCET Analysis

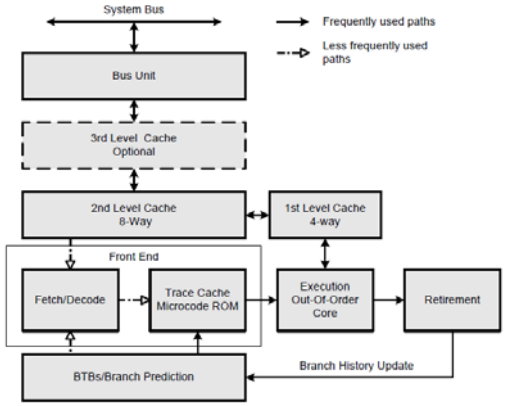
- The Target Hardware

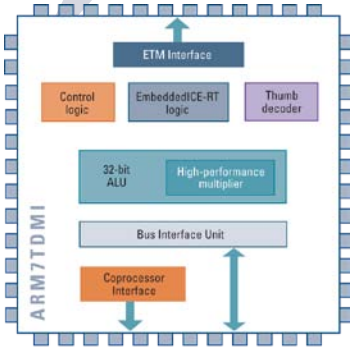
Multi-core Processors

Complex Processors with Pipeline and Cache

Simple Single-Core Processor







Hardware is becoming more and more complex, hard to analyze!

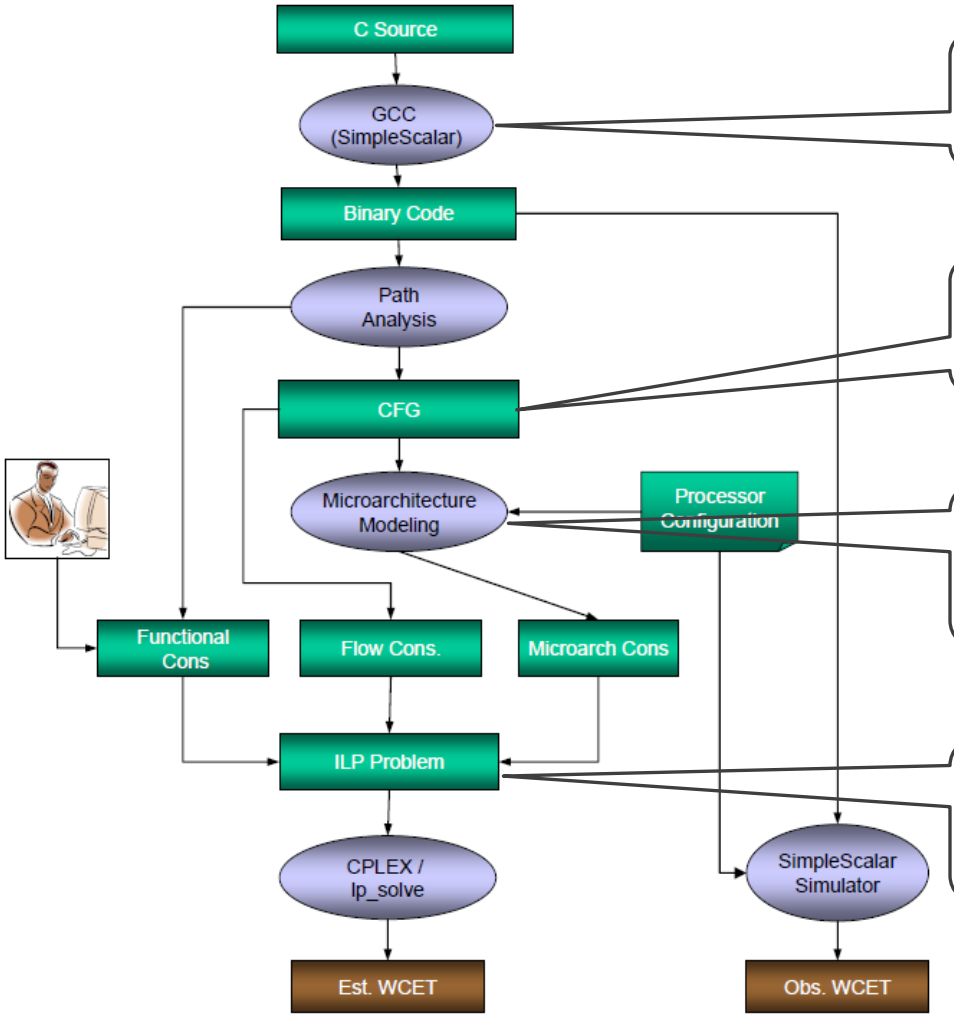
Remarks on the Ingredients

- The Representation Levels of Programs
  - Precise timing analysis has to be done after all program transformations
  - Generally, it is much easier to extract or annotate flow facts in a higher representation level
  - The flow facts should be mapped from higher level to lower level correctly, probably this mapping is done in parallel to code transformation

- Hardware in real-time systems are becoming more and more complex with features to improve average-case performance (throughput), but less predictable, e.g. timing anomaly

Contents

A Generic Workflow of Static Analysis



1 Compile source code into the binary of target hardware

2 Reconstruct the Control-Flow Graph from the binary

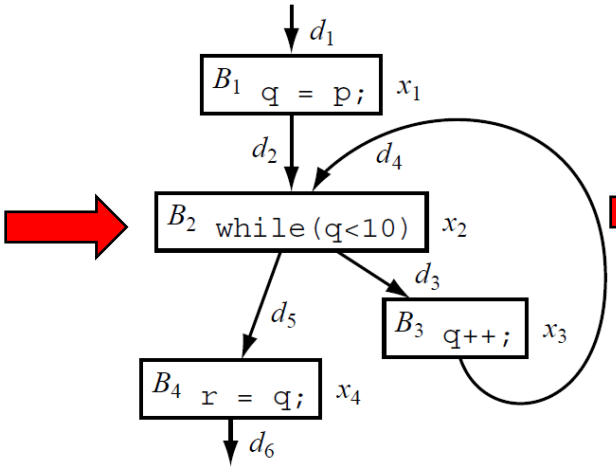
3 Model the HW architecture, calculate the execution time of each basic block in the CFG

4 Calculate the WCET using some DSE tools, e.g. ILP solvers, constraint solvers, model checkers

Figure from Chronos@NUS

An Example of the Workflow

```
/* p >= 0 */
q = p;
while(q<10)
  q++;
r = q;
```



Do micro-arch modeling to get the execution time of each BB

$$wcet = MAX \sum_{t_{i\_j} \in T_B} (cost(t_{i\_j}) * x_{i\_j}) \ \ s.t. \ \ flowfacts \quad (1)$$

$$\forall\, BBi,\ b_i = \sum_{dst(t_{k\_i})=BBi} x_{k\_i} = \sum_{src(t_{i\_j})=BBi} x_{i\_j} \quad (2)$$

Estimated WCET value

$$\forall\, Loop_i,\ \ b_{Tail_i} \leq lpb_i \cdot \sum_{BBj \in BL_i} b_j$$

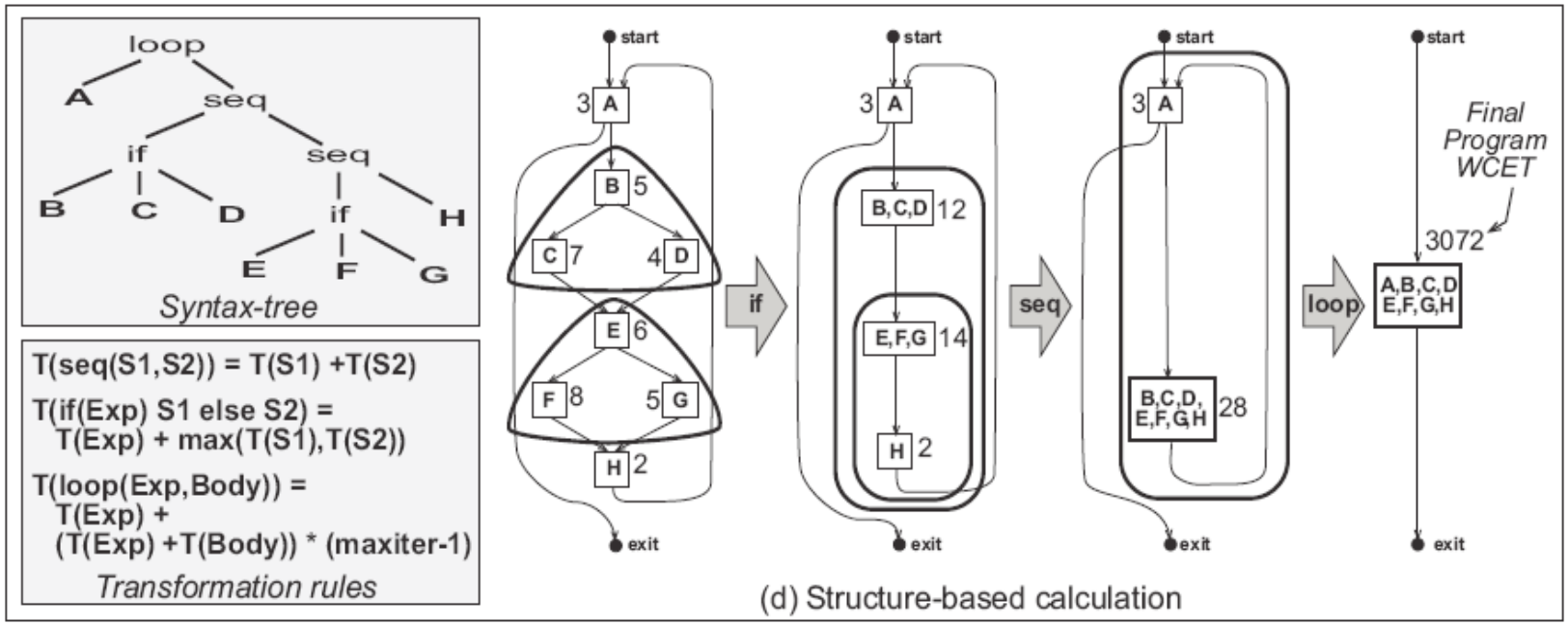
What is Path Analysis?

- Path Analysis
  - To identify the execution trace that leads to the longest execution time
  - To identify infeasible paths of the program
  - Path analysis is a "Design Space Exploration" problem

- Popular Techniques
  - Tree-based methods (Timing Schema)
  - Path-based methods
  - Implicit Path Enumeration Technique (IPET)

Timing Schema

- Represent the program in a syntax tree
- Calculate the WCET of a program by folding the tree



Timing Schema

- Some General Assumptions
  - No recursion
  - Explicit function calls
  - No "goto"s
  - Bounded loop with single entry and single exit
- The Rules

$construct$	$MAXT$		
primitive	$maxt(primitive)$	$=$	$\tau(primitive)$
sequence	$maxt(sequence)$	$=$	$\sum_i maxt(construct_i)$
alternative	$maxt(alternative)$	$=$	$maxt(condition)+$ $\max(maxt(construct_1), maxt(construct_2))$
$\text{loop}_{\text{number}}$	$maxt(loop_{head})$	$=$	$maxt(init) + maxt(condition)+$ $count * (maxt(body) + maxt(condition))+$ $maxt(overrun\_statement)$
	$maxt(loop_{tail})$	$=$	$count * (maxt(body) + maxt(condition))+$ $maxt(overrun\_statement)$
$\text{loop}_{\text{time}}$	$maxt(loop_{time})$	$=$	$time + maxt(timeout\_statement)$
subroutine	$maxt(subroutine)$	$=$	$\tau(organization) + maxt(body)$

An Example

```
int calc_center(image, x_center, y_center)                                  /*  44 */
char    image[MAX_ROWS][MAX_COLS];
int     *x_center, *y_center;
{
    int pixel_count, x_coord, y_coord, x_sum, y_sum;                        /*  48 */

    pixel_count = x_sum = y_sum = 0;

    FOR(y_coord = 0; y_coord < MAX_ROWS; y_coord++) SCOPE MAX_COUNT(MAX_ROWS)   /* (42+26;76;32) */
    {                                                 /* loop3 */
        FOR(x_coord = 0; x_coord < MAX_COLS; x_coord++) MAX_COUNT(MAX_COLS)     /* (42+26;76;32) */
        {                                             /* loop4 */
            if (image[x_coord][y_coord])              /* alt2  */           /* 146 */
            {
                int weight;

                MAX_COUNT(MAX_AREA);                          /* marker */    /* 40 */
                weight = calc_weight(image, x_coord, y_coord);              /* 310 + */
                x_sum += x_coord * weight;
                y_sum += y_coord * weight;
                pixel_count += weight;
            }
        }
    }

    if (pixel_count)                                  /* alt3  */           /*  22 */
    {
        *x_center = x_sum / pixel_count;                                    /* 424 */
        *y_center = y_sum / pixel_count;
    }
    else
        *x_center = *y_center = 0;                                          /*  56 */

    return 1;                                                               /*  14 */
}
```

An Example (2)

$$
\begin{aligned}
maxt(calc\_center) &= 44 + 48 + maxt(loop_3) + maxt(alt_3) + 14 = \\
&= \underline{551\ 475\ 096} \\
maxt(loop_3) &= 42 + 76 + 200 * ((maxt(loop_4) + 32) + 76) + 26 = \\
&= 551\ 474\ 544 \\
maxt(loop_4) &= 42 + 76 + 640 * ((maxt(alt_2) + 32) + 76) + 26 = \\
&= 2\ 757\ 264 \\
maxt(alt_2) &= 146 + \max(310 + maxt(calc\_weight), 0) = 4\ 200 \\
maxt(alt_3) &= 22 + \max(424, 56) = 446
\end{aligned}
$$

$$
\begin{aligned}
maxt(calc\_weight) &= 44 + 72 + maxt(loop_1) + 122 = 3\ 744 \\
maxt(loop_1) &= 54 + 84 + 3 * ((maxt(loop_2) + 32) + 84) + 26 = 3\ 506 \\
maxt(loop_2) &= 54 + 84 + 3 * ((maxt(alt_1) + 32) + 84) + 26 = 998 \\
maxt(alt_1) &= 146 + \max(16, 0) = 162
\end{aligned}
$$

The Workflow of Timing Schema

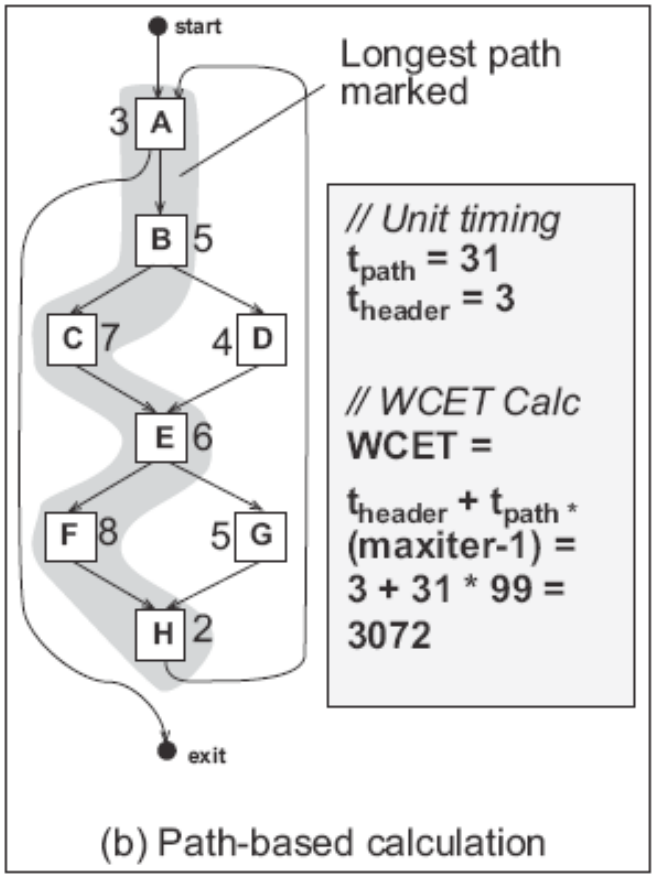
- Decomposition
  - Decompose a statement into its primitive components (atomic blocks)
- Code Prediction
  - Predict the implementation (compiled instructions) of each atomic block
- Execution Time of the Atomic Blocks
  - Calculate the execution times of the atomic blocks according to the execution times of the instructions
- Execution Time of the Statements
  - Calculate the execution times of the statements according to the execution times of the atomic blocks

An Evaluation of Timing Schema

- Pros
  - Simple method with cheap computation effort
  - Scale very well with program size

- Cons
  - Cannot deal with generic flexible program structures
  - Limited ability on specifying flow facts
  - Suffers compiler optimization

Path-Based Methods



- The upper bound is determined by: first calculating the bounds of all paths, and then searching the path with longest execution time
- Possible paths are represented explicitly

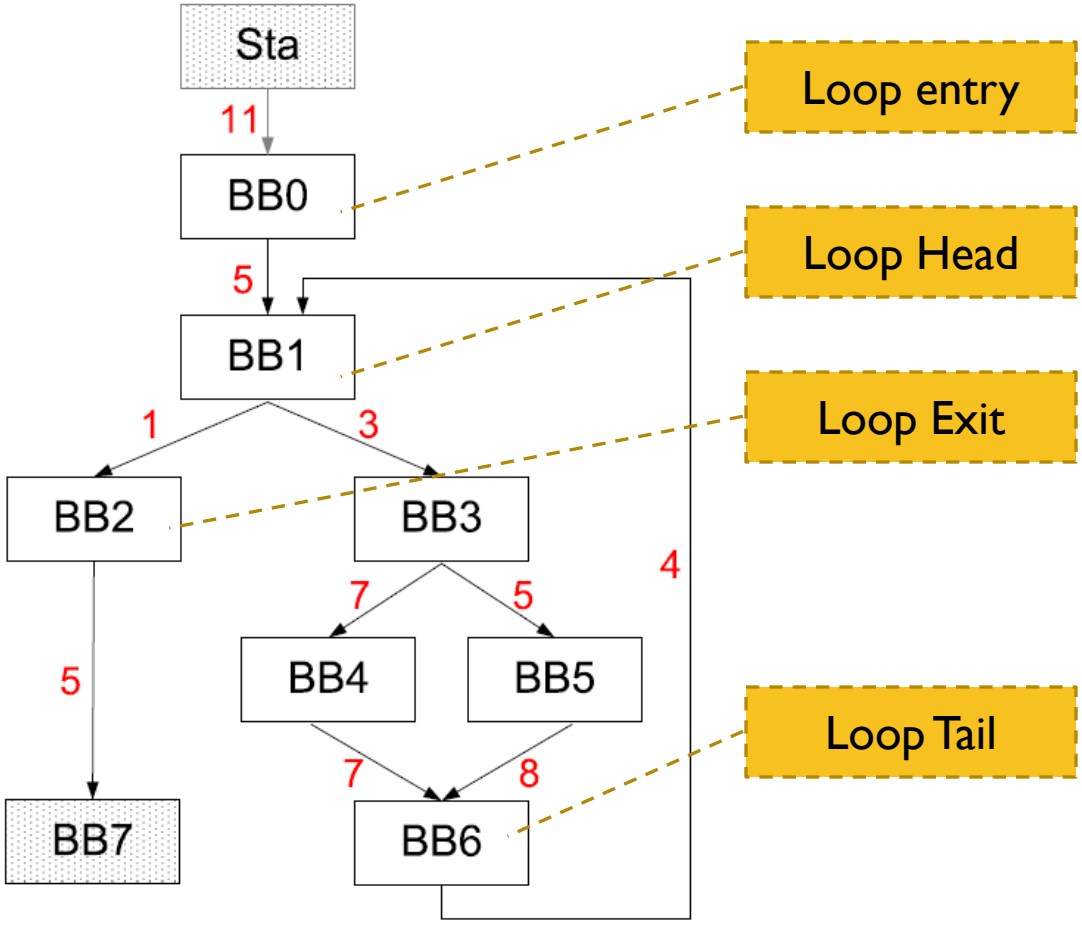
Model Checking

- Model Checking of WCET is Path Based
  - The state space is all the possible program paths
  - The model checkers deal with paths explicitly

- Basic Idea
  - Construct the CFG of a program as input
  - Transform the CFG into the MC model
  - Search the path with the longest execution time

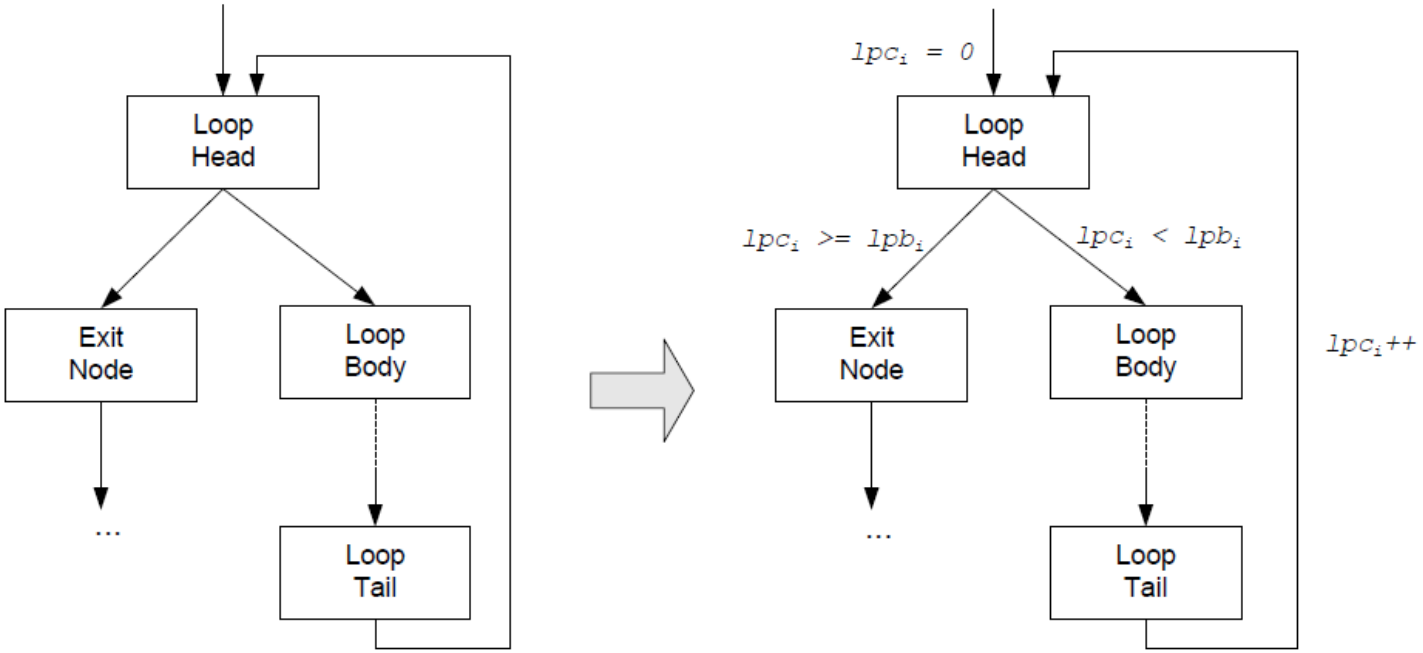
CFG Reconstruction – An Example

```
void main()
{
   int b;
   int i = 0, j = 0;
   while (i < 10){
      if (b)
         j++;
      else
         j--;
      i++;
   }
}
```



(a) A motivating example (b) The CFG

CFG → Model Checking Model



The model checker runs an FSM, where each box represents a state in the FSM, and the arcs represent the transitions. Labels on arcs specify the transition conditions.

The Optimization Procedure

- We can ask the model checker "is it YES or NO that 'for all execution paths starting from the initial state, globally WCET is not greater than N".
- Additional procedures are needed to find the actual value of N

Algorithm 1 Finding the WCET using binary search

input: The model M of a model checker, initial value of N
output: The optimal value found

```
set the upper and lower bound of binary search
while (lower bound < upper bound - 1)
   middle = (lower bound + upper bound) / 2;
   check the property [ ] ø(middle)
   if ( [ ] ø(middle) is satisfied)
      upper bound = middle;
   else
      lower bound = middle;
end while
return upper bound
```

For example,
If the actual WCET is 100, then
TRUE, for N= 100
FALSE, for N= 99

Evaluation of the Path-Based Methods

- Pros
  - Allows simple integration of HW modeling in the analysis (expressiveness)
  - Guaranteed exact results

- Cons
  - Scalability problems (exponential state space)
  - If you use model checkers, some unknown performance bottlenecks may occur

Implicit Path Enumeration Technique

- Can obtain exact answer without exhaustive search of all the paths
- Hint: the objective is to determine the worst-case execution time, not the worst-case execution path
- Idea: finding the worst-case execution time → finding the worst-case execution count of each basic block

Implicit Path Enumeration Technique

- Solutions
  - The problem of finding the worst-case execution counts can be formulated as an Integer Linear Programming (ILP) problem or a constraint programming problem
  - The more constraints, the more accurate results

$$t_{max} = \max\left(\sum_i c_i x_i\right)$$

execution time of basic block B_i (constant)

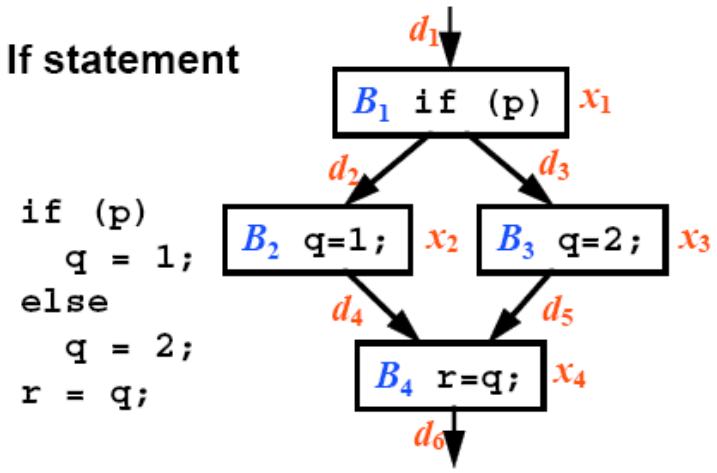
execution count of basic block B_i (variable)

subject to a set of constraints:

$$\mathbf{Ax} \leq \mathbf{B}$$

Implicit Path Enumeration Technique

- Constraints – Restrictions on x-variables
  - Structural constraints: extracted directly from the CFG



At each node:

$$Basic\ block\ count = \sum inputs = \sum outputs$$

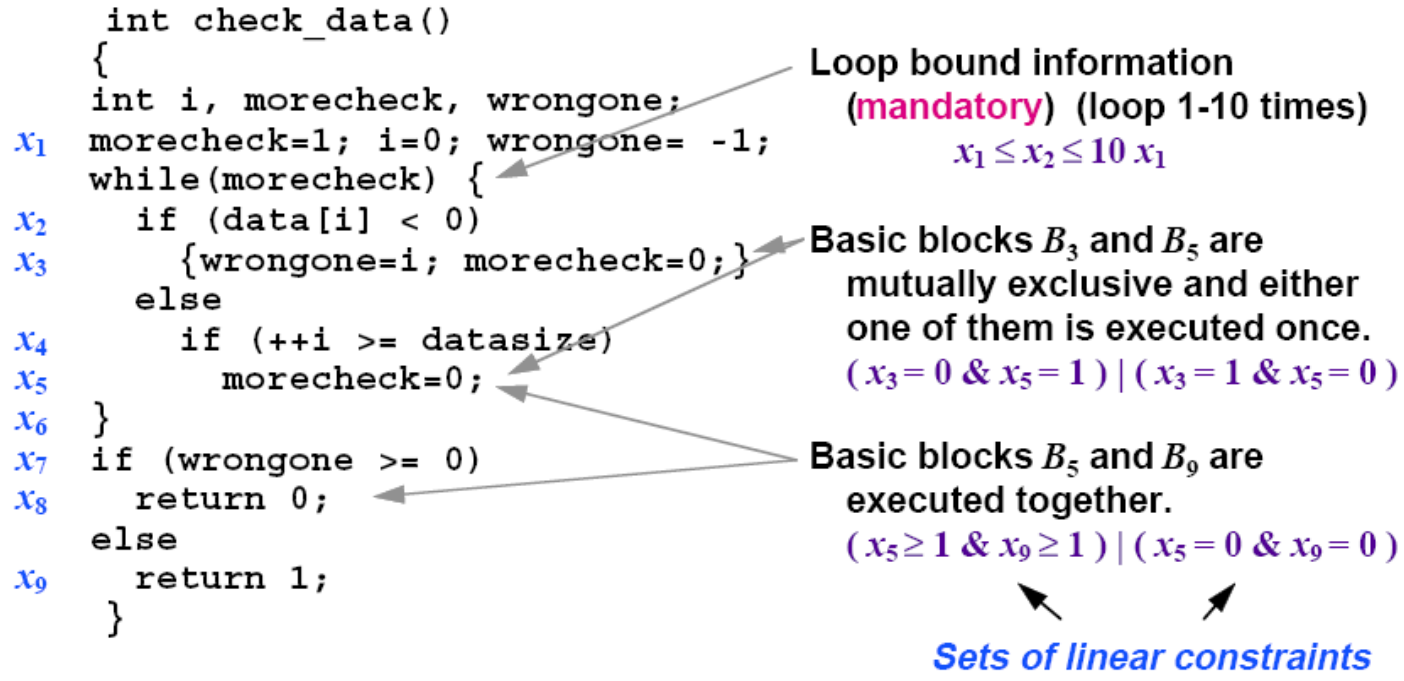
Structural Constraints:

$$x_1 = d_1 = d_2 + d_3$$
$$x_2 = d_2 = d_4$$
$$x_3 = d_3 = d_5$$
$$x_4 = d_4 + d_5 = d_6$$

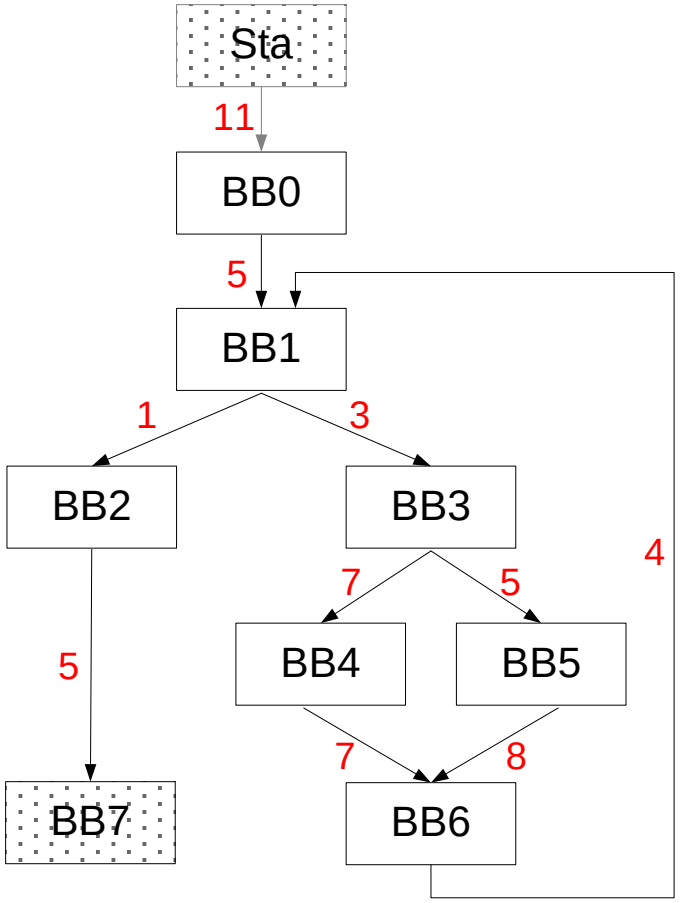
Linear constraints

Implicit Path Enumeration Technique

- Constraints – Restrictions on x-variables
  - Functional constraints: telling how the program works, e.g. how many times a loop iterates



An Example of ILP Formulation




```
Maximize

11 dSta_0 + 5 d0_1 + 1 d1_2 + 3 d1_3
+ 5 d2_7 + 7 d3_4 + 5 d3_5 + 7 d4_6
+ 8 d5_6 + 4 d6_1

Subject to

\ === tcfg constraints ===
dSta_0 = 1
b0 - d0_1 = 0
b0 - dSta_0 = 0
b1 - d1_2 - d1_3 = 0
b1 - d0_1 - d6_1 = 0
b2 - d2_7 = 0
b2 - d1_2 = 0
b3 - d3_4 - d3_5 = 0
b3 - d1_3 = 0
b4 - d4_6 = 0
b4 - d3_4 = 0
b5 - d5_6 = 0
b5 - d3_5 = 0
b6 - d6_1 = 0
b6 - d4_6 - d5_6 = 0
b7 - d2_7 = 0

b0 = 1
b7 = 1
b6 - 10 b0 <=  0

// Definition of integers is omitted
```

An Evaluation of IPET

- Pros
  - Allows to consider complex flow facts
  - Generation of constraints is simple and direct
  - Efficient tools

- Cons
  - Solving ILP is generally NP-hard (luckily, the WCET problem can be reduced to network flow problem, which requires less solving time)
  - Still difficult to encode the flow facts that specify execution ordering

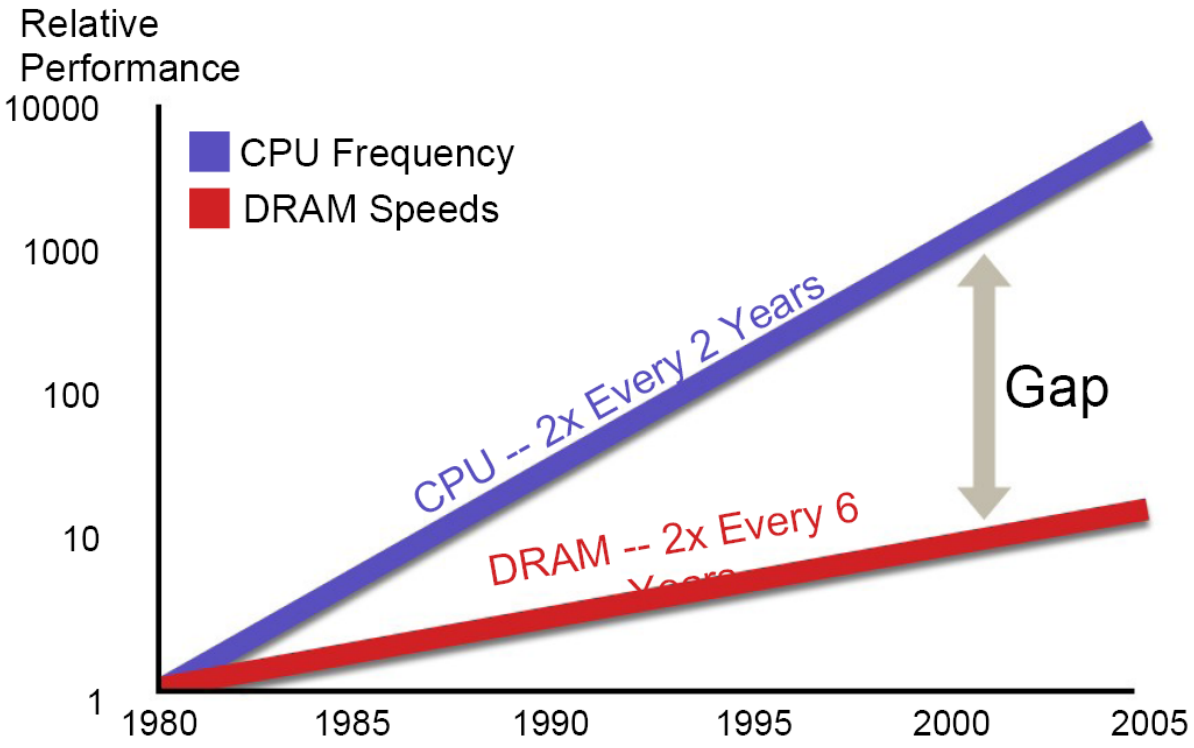
Contents

Micro-Architecture Analysis

- Why Micro-Architecture Analysis?
  - The execution time depends not only on the program itself, but also on the hardware where the program executes
  - Modern processors have lots of complex features that can result in unpredictable execution time variation, which is very hard to analyze
  - Timing Anomaly
- What Are Included in Micro-Architecture Analysis?
  - Cache analysis
  - Pipeline analysis (multiple issue, out-of-order pipelines)
  - Branch prediction and speculative execution
  - ……

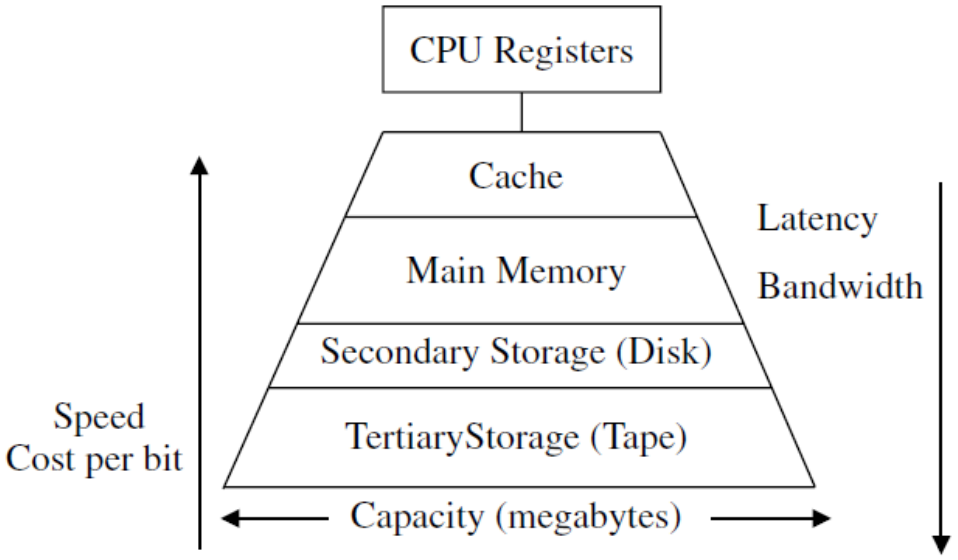
Cache in a Nutshell

- Why Cache?
  - The "memory wall"



Cache in a Nutshell

- Why Cache?
  - Cost-speed trade-off
  - Program temporal/spatial locality
  - Memory hierarchy

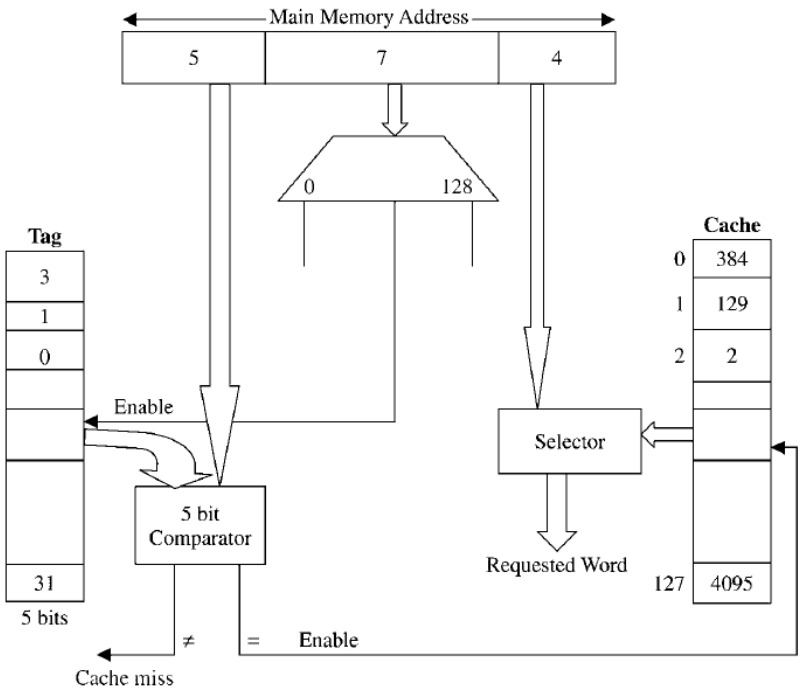


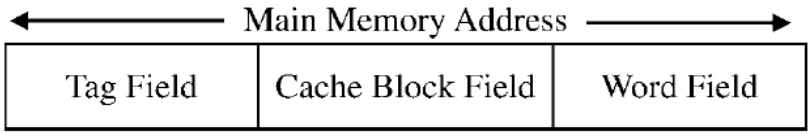
Cache in a Nutshell

- Types of Caches
  - L1 Instruction Cache (32KB)
  - L1 Data Cache (32KB)
  - L2/L3 Unified Cache (512KB ~ 6MB)
  - Shared cache in multicores
- Associativity
  - Cache are organized in terms of "cache lines"
  - Associativity specifies how the cache lines are organized and how to map a memory block into the cache
  - Direct-mapped
  - Full-associative
  - Set-associative

Cache in a Nutshell

- Direct-mapped Cache





i = x % n;

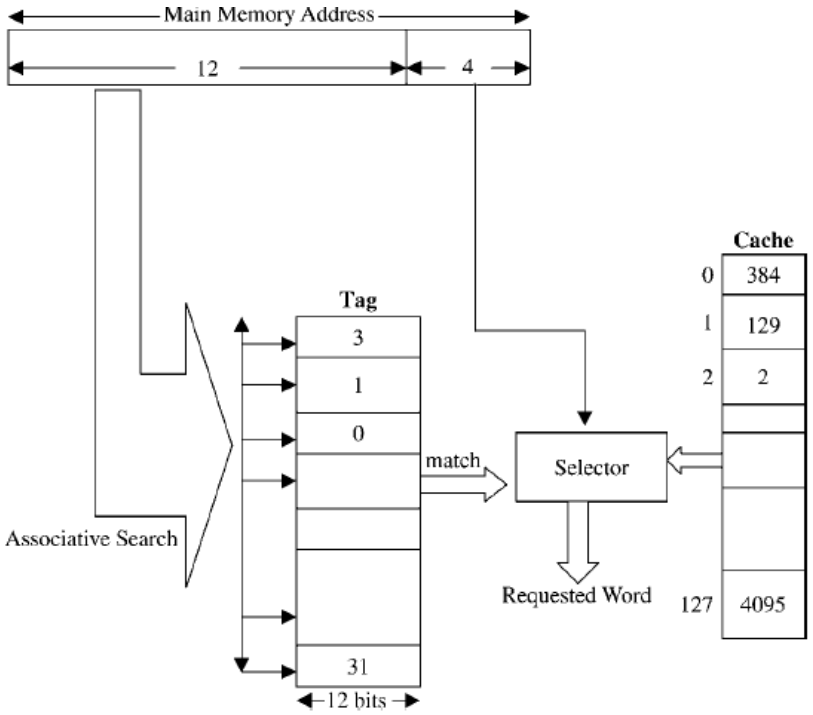
Easy to implement

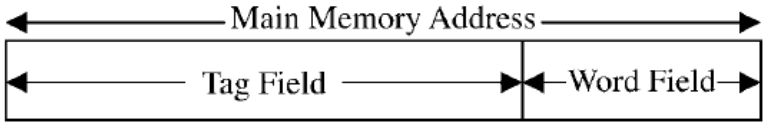
Fast scan

But high miss ratio!

Cache in a Nutshell

- Full-associative Cache





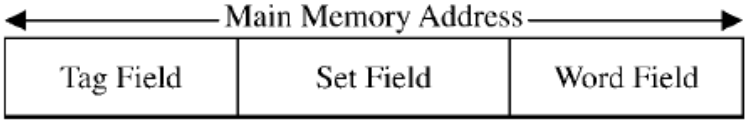
A memory block can be mapped to any cache line if not occupied

Efficient use of the cache

But notorious scan and replacement overhead!

Cache in a Nutshell

- Set-associative Cache

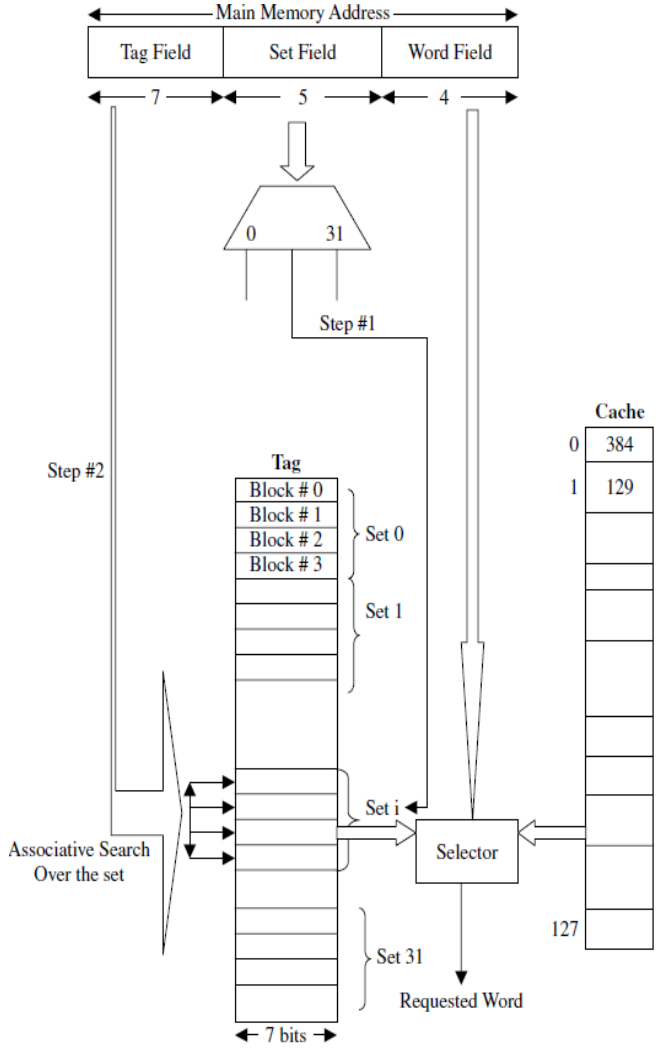


i = (x % #sets) + A (0≤A≤set size)

A clever trade-off between direct-mapped caches and full-associative caches

Much less overhead than FA, but still harder to analyze than DM

Good news to GP-architecture guys, but not so good to Real-Time guys



Cache in a Nutshell

- Replacement Policy
  - If cache miss occurs, kick out which cache line?
  - Round-robin, LRU, pseudo-LRU
  - Different cache replace policies have different predictability
- Write Policy
  - Write-through: whenever there is a write to the cache content, the data is immediately written to the corresponding main memory address, regardless of hits or misses
  - Write-back: only write dirty cache data to main memory when the cache block is replaced, requires special bits in cache to tag dirty data

Cache Analysis in WCET Analysis

- Without cache analysis
  - In each BB, all memory accesses take fixed cycles, no variation
  - The execution time of a BB is not affected by the execution history
  - When there is cache, all the situations are different

- Analysis of different types of caches
  - I-cache with different replacement policy
  - I-cache or D-cache?
  - Single-level or multi-level?
  - Dedicated cache or shared cache?

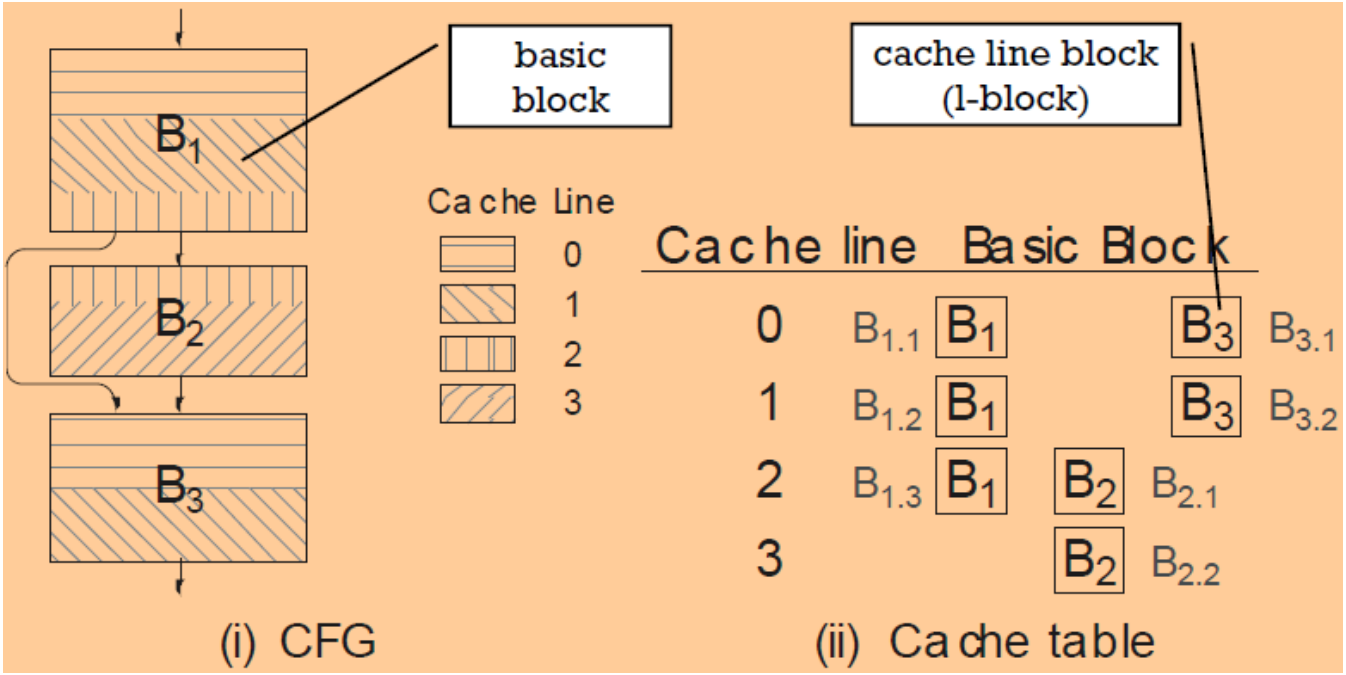
Cache Analysis in the IPET Framework

- Idea
  - Model new constraints related to cache behavior into the original ILP problem
  - No fundamental changes to the structure of the ILP problem

- How to?
  - For each instruction, determine
    - Cache hit execution counts, time
    - Cache miss execution counts, time
    - → go into the basic blocks

Line Blocks

- The objective cache analysis is to determine how many misses and hits in each BB → analyze conflicting memory blocks



Modified ILP Formulation

Let:

$x_{i.j}^{hit}$ – cache hit count of l-block $B_{i.j}$

$x_{i.j}^{miss}$ – cache miss count of l-block $B_{i.j}$

$c_{i.j}^{hit}$ – exec. time of l-block $B_{i.j}$ given that it is a cache hit

$c_{i.j}^{miss}$ – exec. time of l-block $B_{i.j}$ given that it is a cache miss

n_i – number of l-blocks of basic block B_i

Maximize:

$$\sum_{i}^{N}\sum_{j}^{n_i}\left(c_{i.j}^{hit}x_{i.j}^{hit}+c_{i.j}^{miss}x_{i.j}^{miss}\right)$$

subject to:

$$x_i = x_{i.j}^{hit} + x_{i.j}^{miss} \qquad j = 1,2,\ldots,n_i$$

structural constraints

functionality constraints

cache constraints

New Cache Constraints

- only **one** l-block Bk.l maps to the same cache line (first access is miss):

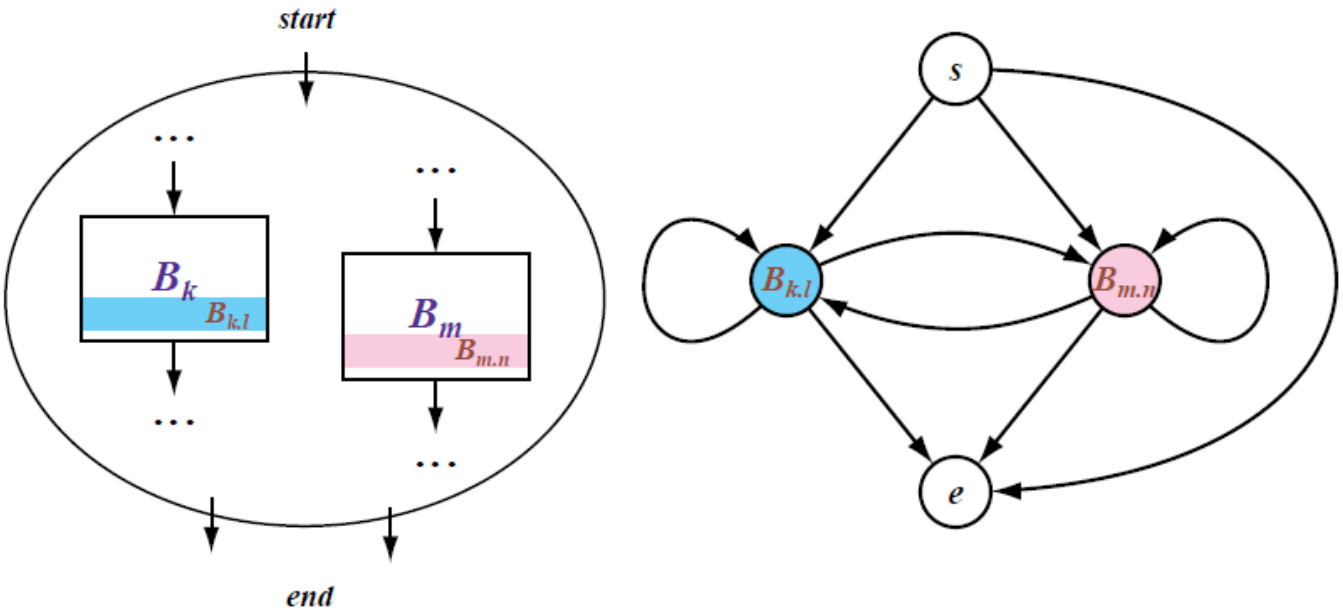
$$x_{k.l}^{miss} \leq 1$$

- only two or more **non-conflicting** l-blocks map to the same cache line (first access is miss):

$$x_{k.l}^{miss} + x_{m.n}^{miss} \leq 1$$

- two ore more **conflicting** l-blocks → use **CCG**

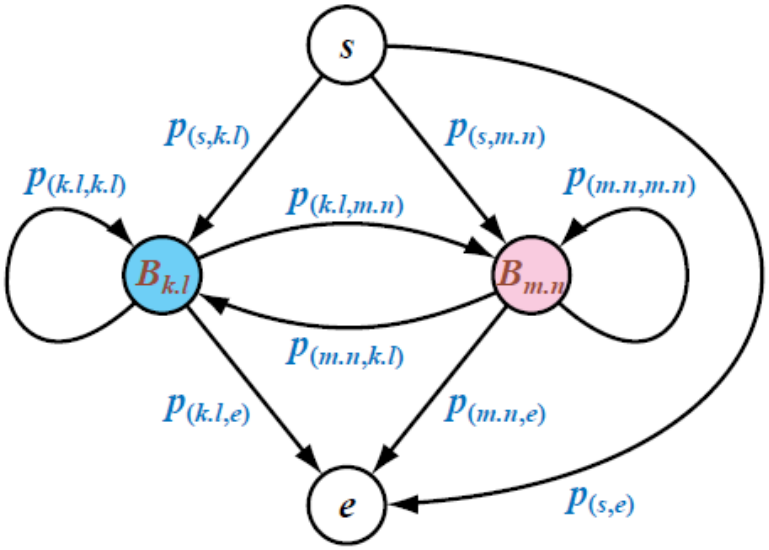
Cache Conflict Graph (CCG)



Control Flow Graph (CFG)

Cache Conflict Graph (CCG)

Generating Constraints from CCG



Flow at node $B_{k.l}$:

$$x_k = p_{(s,k.l)} + p_{(m.n,k.l)} + p_{(k.l,k.l)}$$
$$= p_{(k.l,e)} + p_{(k.l,m.n)} + p_{(k.l,k.l)}$$

Cache hit count for *l-block* $B_{k.l}$:

$$p_{(k.l,k.l)} \le x_{k.l}^{hit} \le p_{(s,k.l)} + p_{(k.l,k.l)}$$

Starting Condition:

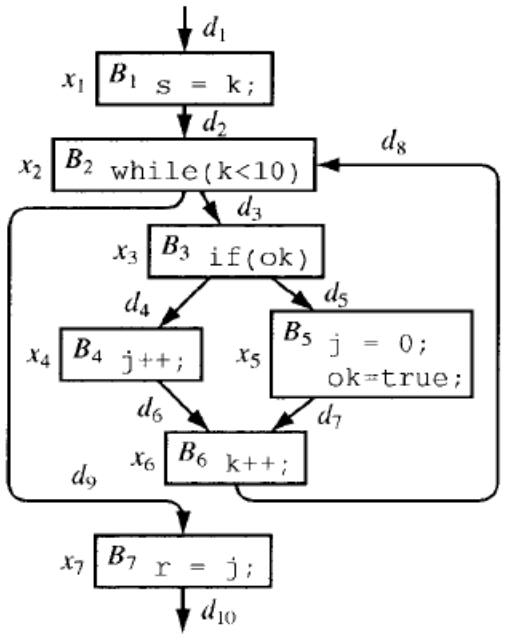
$$p_{(s,k.l)} + p_{(s,m.n)} + p_{(s,e)} = 1$$

Tightening the Constraints

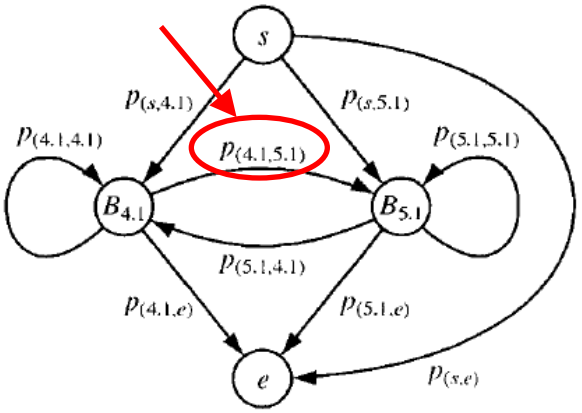
- Assumptions for the Example
  - Each BB is mapped to a single cache line
  - BB1 conflicts with BB6, BB4 conflicts with BB5

```
/* k >= 0 */
s = k;
while (k < 10) {
  if (ok)
    j++;
  else {
    j = 0;
    ok = true;
  }
  k++;
}
r = j;
```

(i) Source code



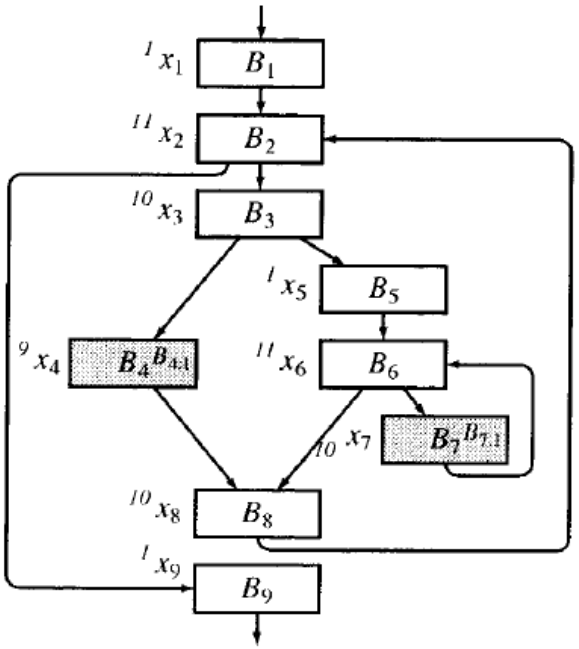
(ii) Control flow graph (CFG)



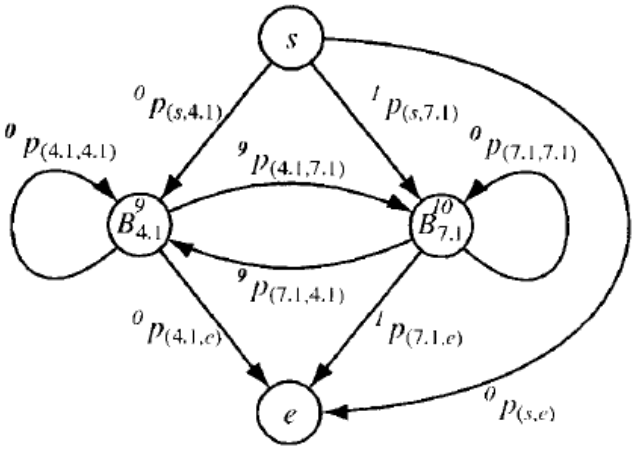
(i) *l-block* $B_{4.1}$ conflicts with *l-block* $B_{5.1}$

$p_{(4.1,5.1)} = 0$

Tightening the Constraints



(i) CFG



(ii) CCG

x3 = 10 ·x1

x7 = 10·x5

x4 = 9·x1

We already know: $0 \leq p_{(i,j,\,u.v)} \leq \min(x_i, x_u).$

But this needs to be tightened: $p_{(s,\,7.1)} + p_{(4.1,7.1)} \leq x_5.$

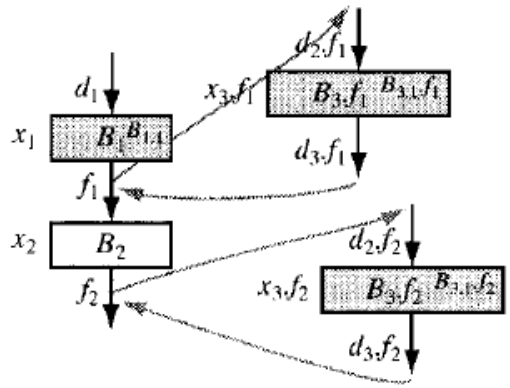
Inter-Procedure Calls

- d1 = 1, x1 = d1 = f1, x2 = f1 = f2, d2.f1 = f1
- x3.f1 = d2.f1 = d3.f1, d2.f2 = f2
- x3.f2 = d2.f2 = d3.f2, x3 = x3.f1 + x3.f2
- $X^{hit}3.1 = p(3.1.f1, 3.1.f2)$

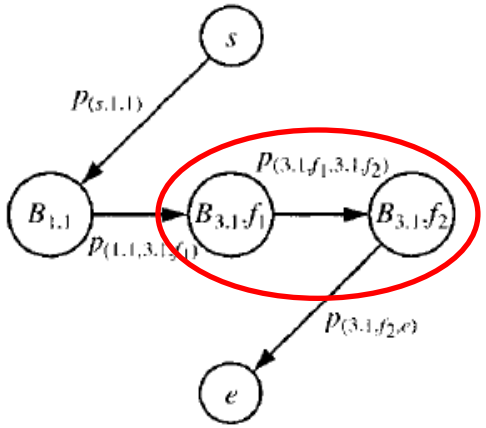
```
     void main()
     {
B1     inc(&i);
B2     inc(&j);
     }


     void inc(int *pi)
     [
B3     *pi++;
     }
```

(i) Code fragment



(ii) CFG with 2 instances of function `inc`



(iii) CCG

Direct-Mapped → Set-Associative

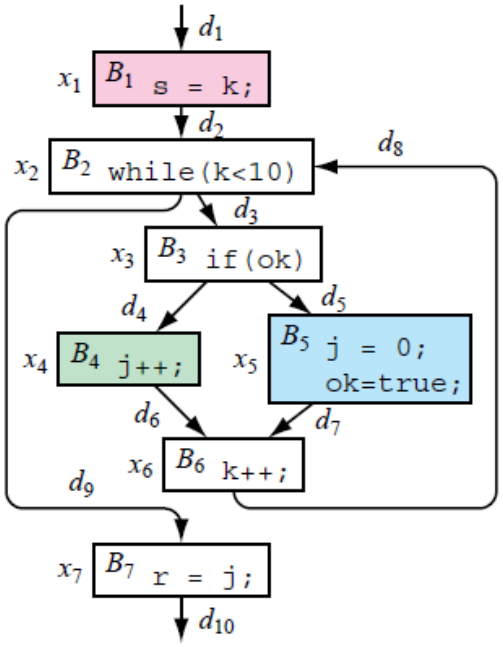
- What's the Difference?
  - Since conflicting domains are set-associative sets, there are more potential conflicts to be analyzed
  - Cache replacement policy affects analysis
- What to do?
  - We need to maintain cache states

[X,X], [X,$B_{i,j}$], [X,$B_{k,l}$], [X,$B_{m,n}$], [$B_{i,j}$,$B_{k,l}$], [$B_{i,j}$,$B_{m,n}$], [$B_{k,l}$,$B_{i,j}$], [$B_{k,l}$,$B_{m,n}$], [$B_{m,n}$,$B_{i,j}$] and [$B_{m,n}$,$B_{k,l}$].

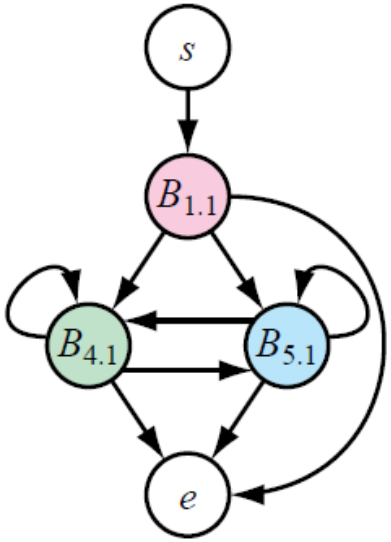
$$\sum_{i=0}^{n} \frac{m!}{(m-i)!}$$

  - CCG → CSTG (a more concrete form of CCG)
  - Cost function is unchanged, but cache constraints are different now

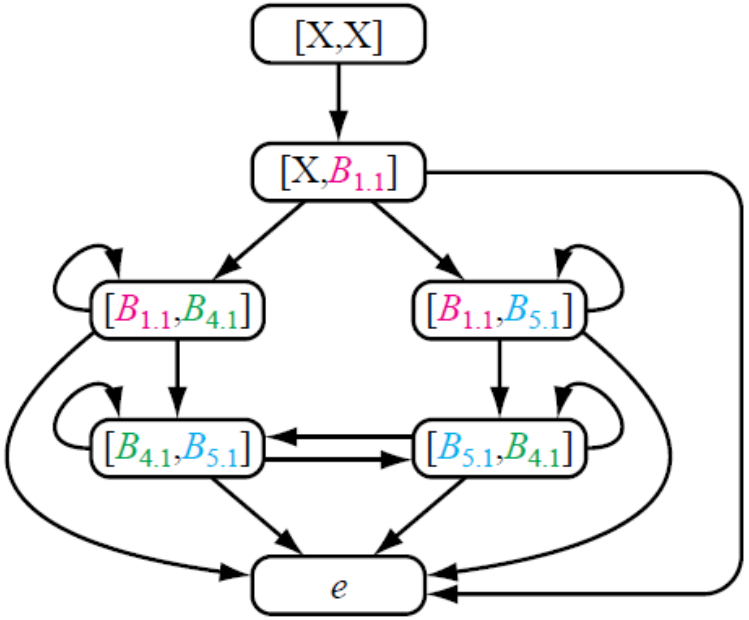
Cache State Transition Graph



CFG

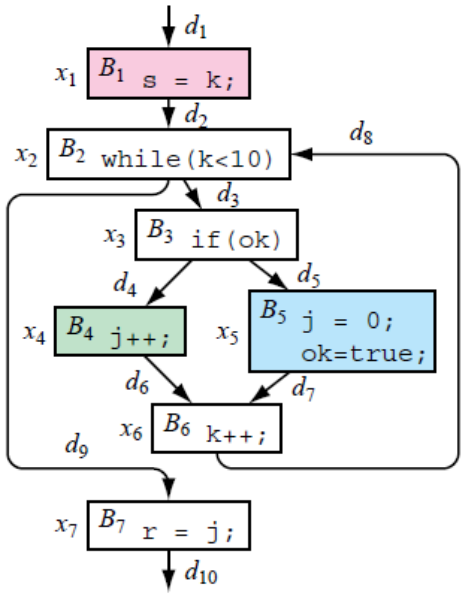


CCG

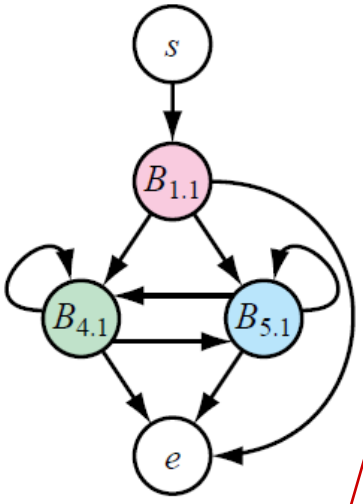


Cache state transition graph

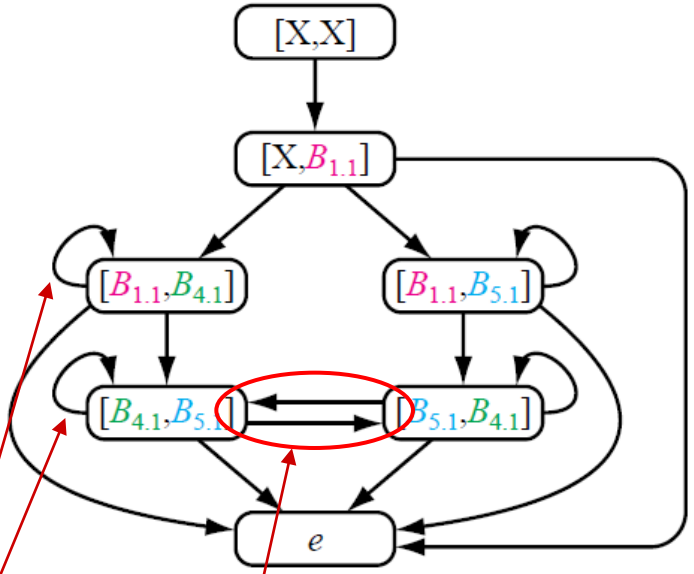
New Cache Constraints



CFG



CCG



Cache state transition graph

1. The execution count of $B_{m,n}$ = the sum of inflow with $B_{m,n}$ in the right most line entry
2. For each node, sum of inflow = sum of outflow
3. Starting condition
4. Cache hit lower bound:

$$x_{y,z}^{hit} \geq \sum_{u,v} P([u,v,y,z],\ [u,v,y,z]) + \sum_{u,v} P([y,z,u,v],\ [u,v,y,z])$$

Data Cache Analysis

- Two sub-problems
  - Determine load/store addresses
  - Model worst case data cache hit/miss counts
- Difficulties
  - L/S addresses may be ambiguous or may change, usually dynamic data structures are banned for static analysis
  - Data flow analysis is required
- Solutions
  - Extend cost functions to include data cache miss penalties
  - Use linear constraints to solve address ambiguity problems

Two-Level Analysis

- Data flow analysis
  - To determine the absolute data addresses of LD/ST instructions
  - Very difficult, but algorithms already established
- Data cache conflict analysis
  - Given the results of data flow analysis, construct a data cache conflict graph, and use ILP techniques to bound the data cache hit and miss counts
- Cinderella works on the second sub-problem

Modified Cost Functions

$$x_i = m_{addr}^{hit} + m_{addr}^{miss}.$$

$$\text{Exec. time} = \sum_i \sum_j (c_{i.j}^{i\_hit} x_{i.j}^{hit} + c_{i.j}^{i\_miss} x_{i.j}^{miss}) + \sum_{addr} (c_{addr}^{d\_hit} m_{addr}^{hit} + c_{addr}^{d\_miss} m_{addr}^{miss})$$

Data Cache Conflict Graph

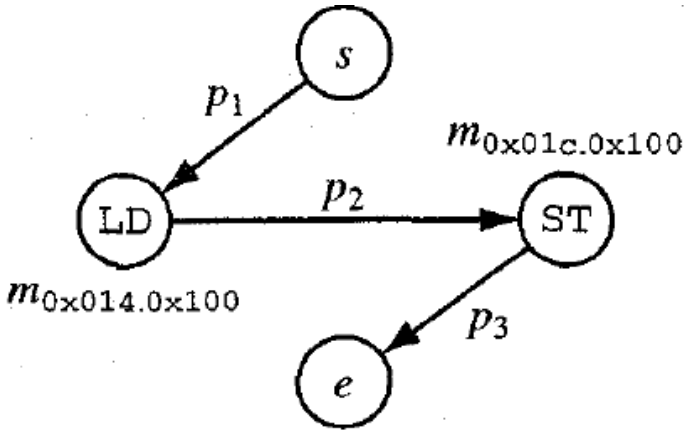
- Idea
  - By data flow analysis, we can identify a set of possible data addresses accessed by LD/ST instr.
  - Different LD/ST instructions that access the addresses in the same data cache set may leads to data cache miss
  - Similar to I-cache analysis, use data cache conflict graph to capture the control flow of LD/ST instructions to analyze potential data hits and misses

Data Cache Conflict Graph

```
        void f() {
          int i, array[10];

x1        i = 0;
          while (i < 10) {
x2          ++array[i];
            ++i;
          }
x3      }
```

```
x1   0x000   lda      0x0, r2          // i = 0
     0x004   lda      0xa, r3
     0x008   lda      0x100, r4
     0x010   cmpibge  r2, r3, 0x028
x2   0x014   ld       (r4)[r2*4], r5   // load array[i]
     0x018   addi     r5, 1, r5        // ++array[i]
     0x01c   st       r5, (r4)[r2*4]   // store array[i]
     0x020   addi     r2, 1, r2        // ++i
     0x024   cmpibl   r2, r3, 0x014
x3   0x028   ret
     .data
     0x100   // array elements
```



Assume data cache is direct-mapped, and each cache line has 4 bytes

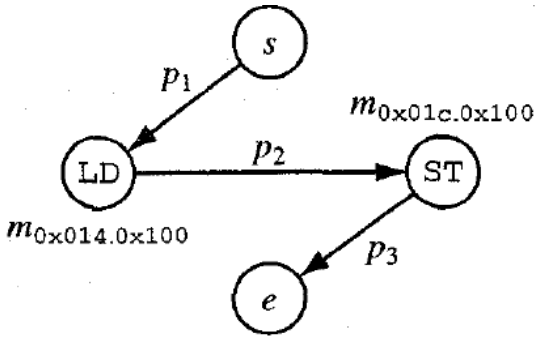
Data address rage [0x100, 0x124] span 10 data cache lines

Take the set at 0x100 for example, see the graph on the left

New Constraints

- In D-CCG, sum of inflow = sum of outflow

$$m_{0x014.0x100} = p_1 = p_2$$
$$m_{0x01c.0x100} = p_2 = p_3$$
$$p_1 = 1$$



- The bounds on the execution counts of each LD/ST instruction instance

$$\sum_{addr_j} m_{0x014.addr_j} = x_2$$
$$\sum_{addr_j} m_{0x01c.addr_j} = x_2$$

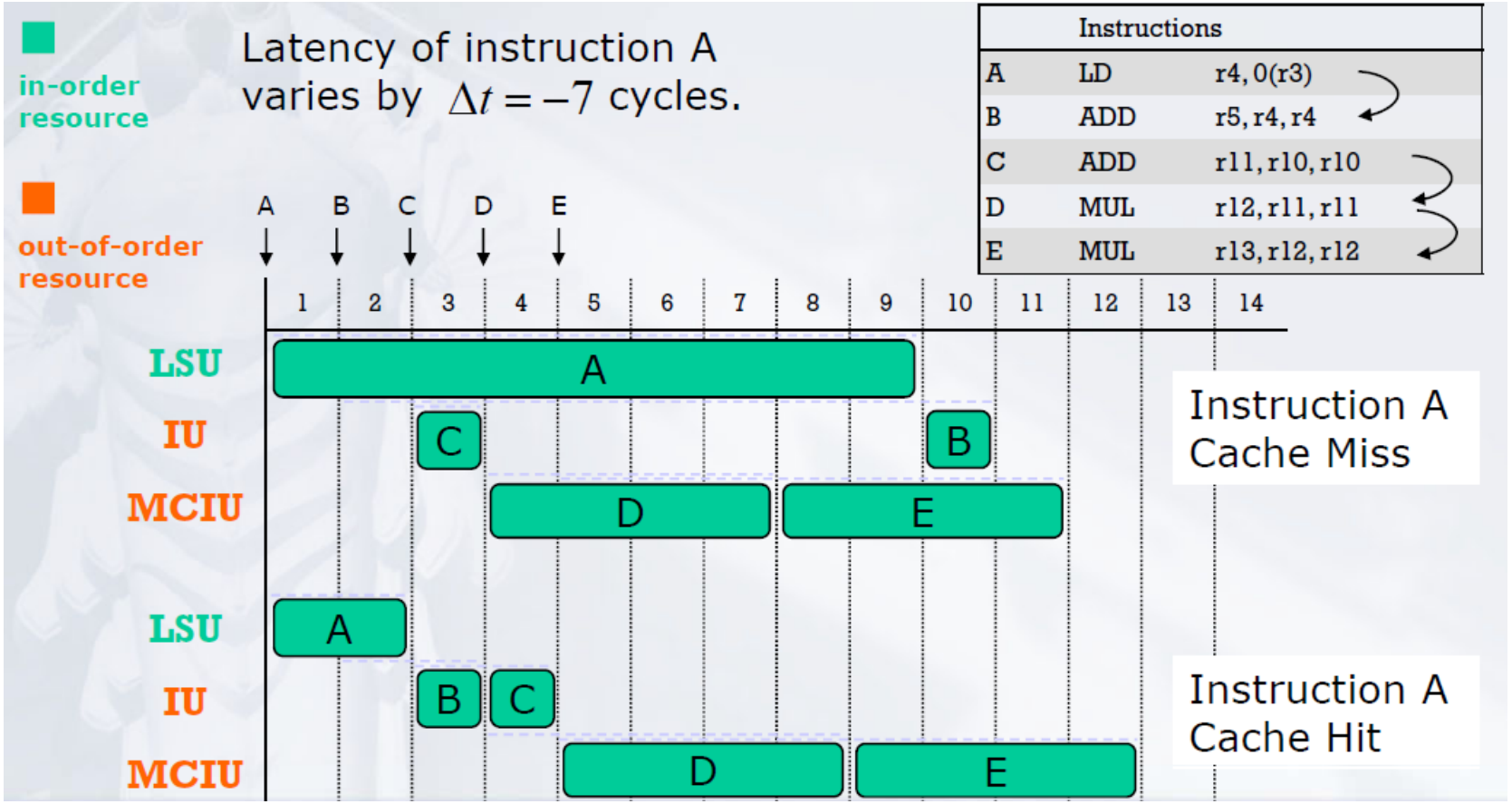
- Hit and miss relation
  - LD-incurred cache miss is similar to instruction cache
  - ST-incurred cache miss depends on write policies: write through or write back, with/without write allocate

An Evaluation of the Above Analysis

- Pros
  - An elegant way to integrate hardware modeling into WCET calculation
- Cons
  - The number of ILP constraints grows greatly, because the CCG is a fine-grained representation of cache states
  - So the time to solve the ILP problem may be very long, not feasible for real-life programs
- Solutions
  - Try some other methods that can do cache analysis in a more coarse-grained way by sacrificing some precision

Timing Anomaly

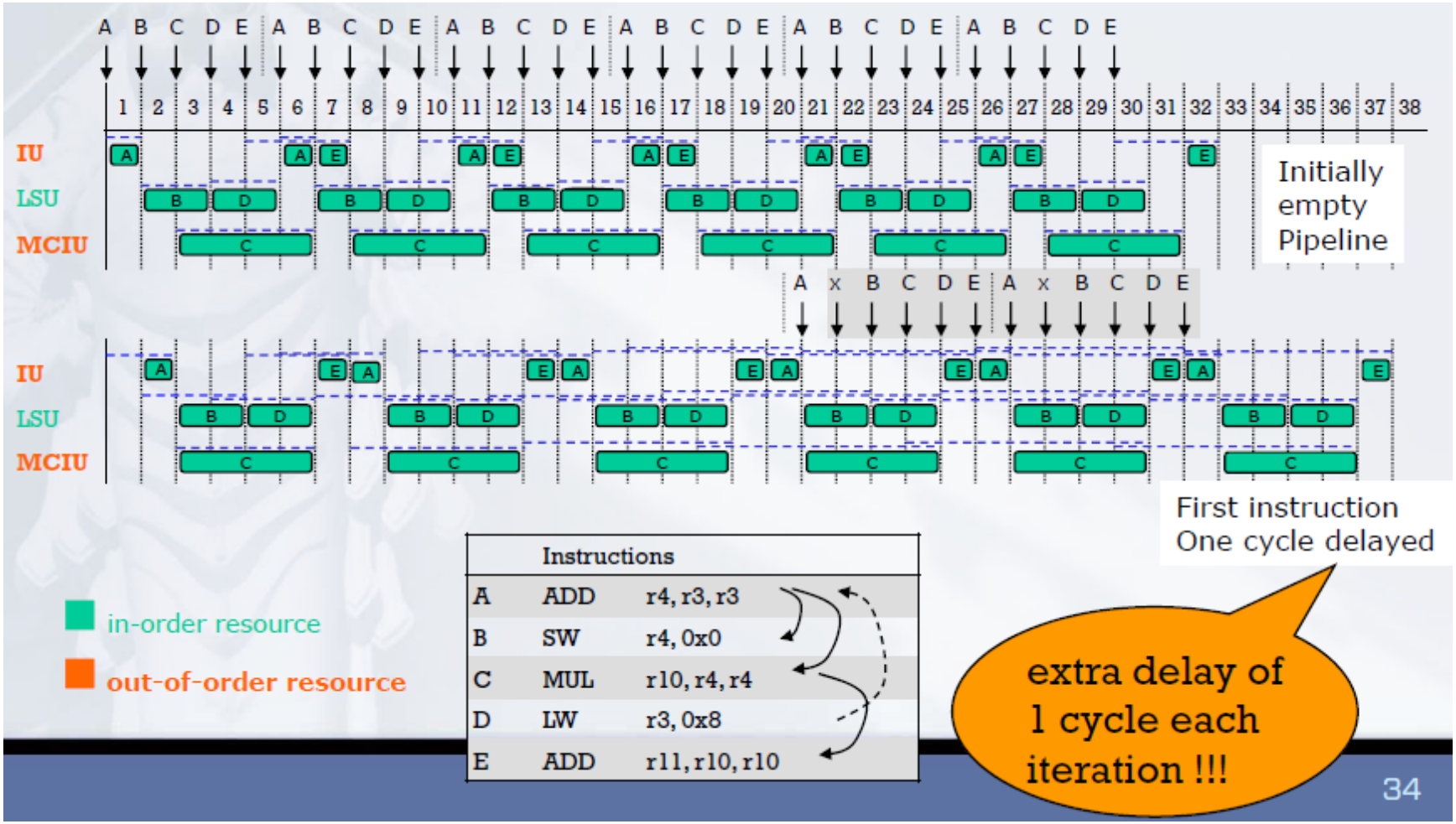
- Counterintuitive Behaviors



Timing Anomaly

- A Formal Definition
  - Δt – Latency variations of several instructions S' (the whole instruction sequence is S)
  - ΔC – execution time change of the whole instruction sequence
- As long as one of the following conditions hold, we say that a timing anomaly occurs
  - $\Delta t > 0 \rightarrow \Delta C < 0$
  - $\Delta t < 0 \rightarrow \Delta C > 0$
  - $\Delta t > 0 \rightarrow \Delta C > \Delta t$
  - $\Delta t < 0 \rightarrow \Delta C < \Delta t$

Domino Effect



Possible Solutions

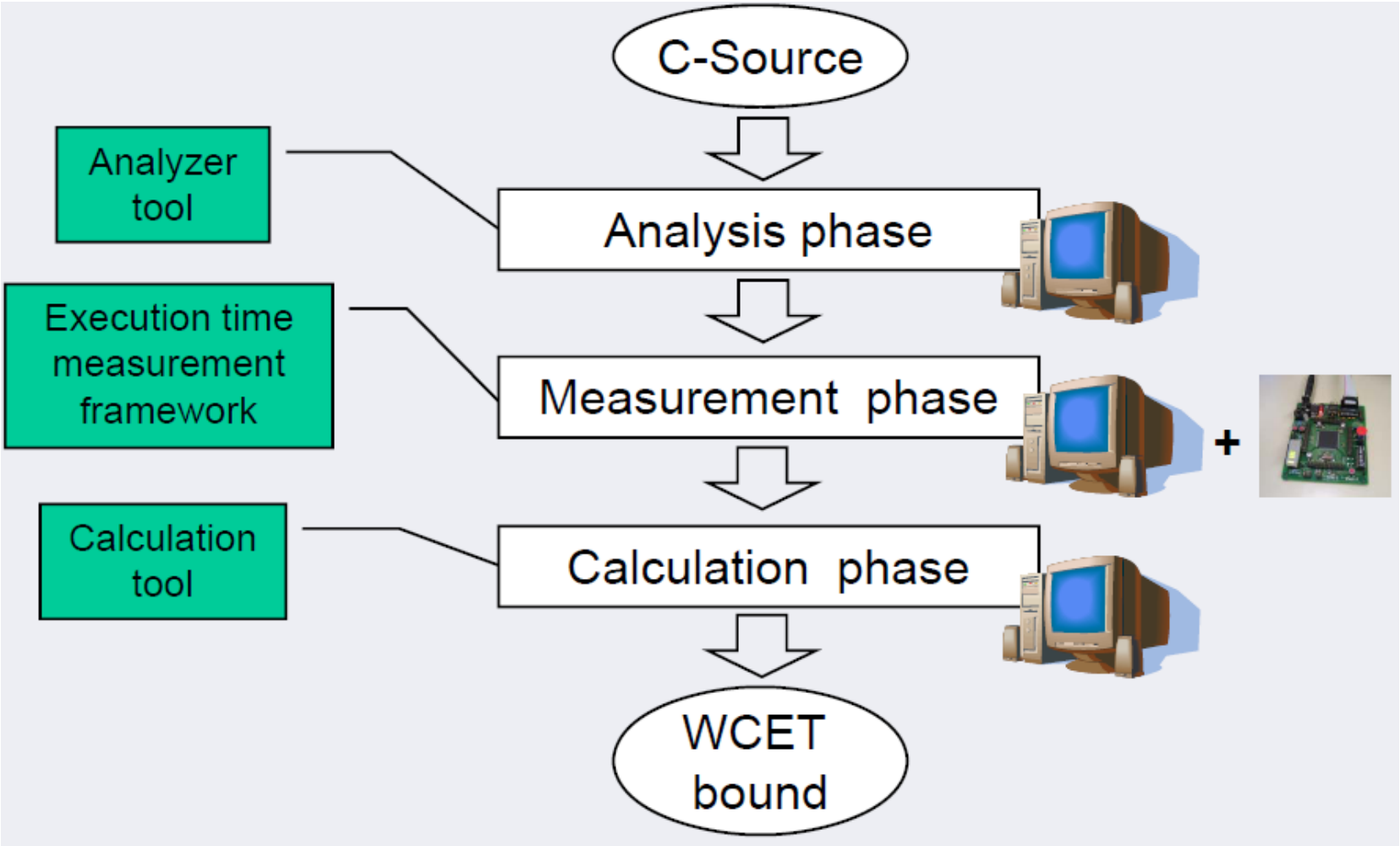
- Occurrence of timing anomalies depends on both hardware features and code structure
- How to eliminate timing anomalies?
  - De-active caches
  - Use synchronization points
  - Choose more predictable hardware platform
  - Code reordering

Contents

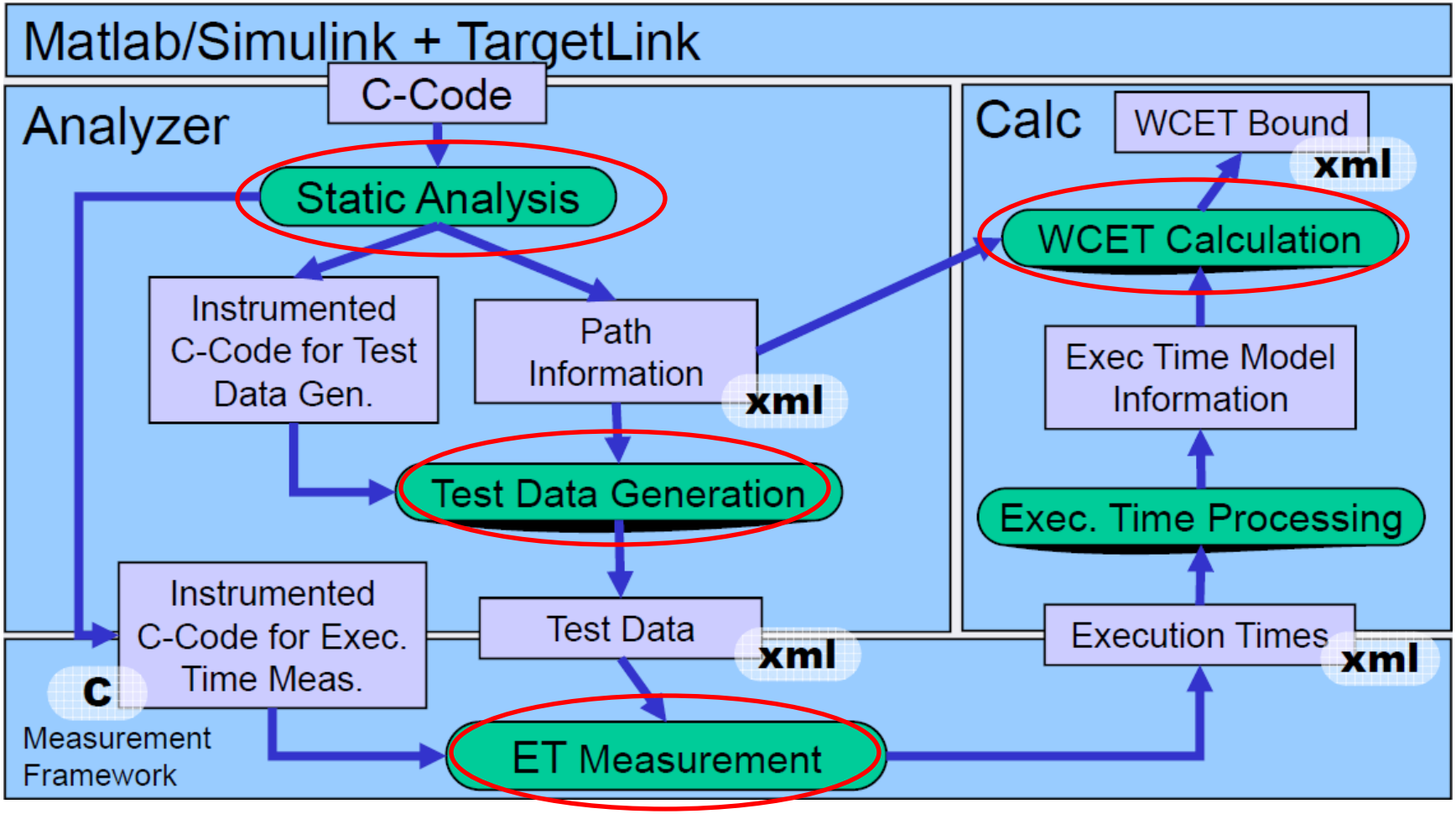
A Review of Problems of Static Analysis

- Problems of Static Analysis
  - Computation efforts exerted to cover all possible situations → possible scalability problems
  - Hard to conduct micro-architecture models
  - Micro-arch analysis of complex hardware may encounter scalability problems
- So Measurement-Based Methods
  - What can we benefit from it?
  - How to do measurement-based analysis?
  - What are the technical issues?

Measurement-Based Methods – The Big Picture



Tool Architecture



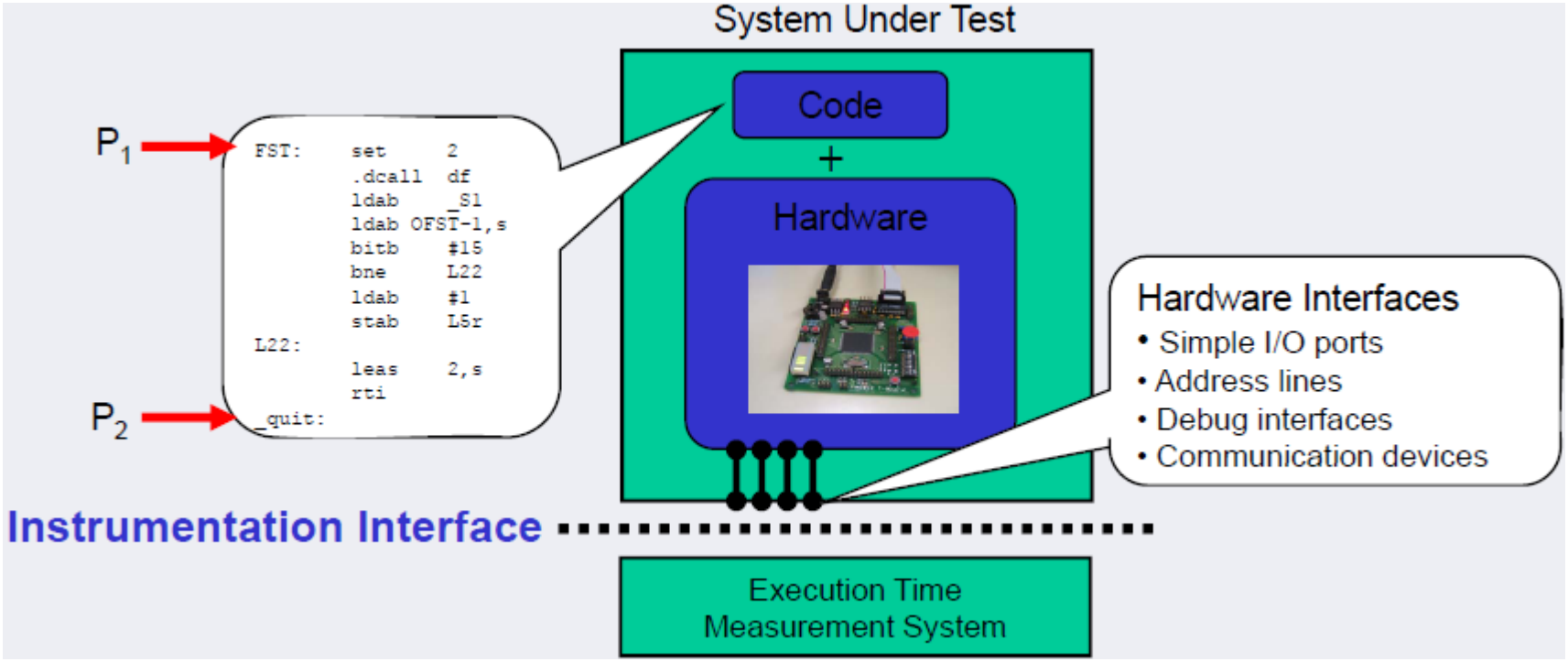
Issues in Measurement-Based Methods

- How to measure?
  - Measurement tools: HW, SW
  - End-to-end, or just measure code segments?
- How to cover more execution traces?
  - Due to worst-case input
  - Due to worst-case hardware states
  - Path/Trace coverage
- What do the results reveal?
  - Single WCET value, or a ET distribution?
  - This issue equals "what's the use of measurement-based methods?"

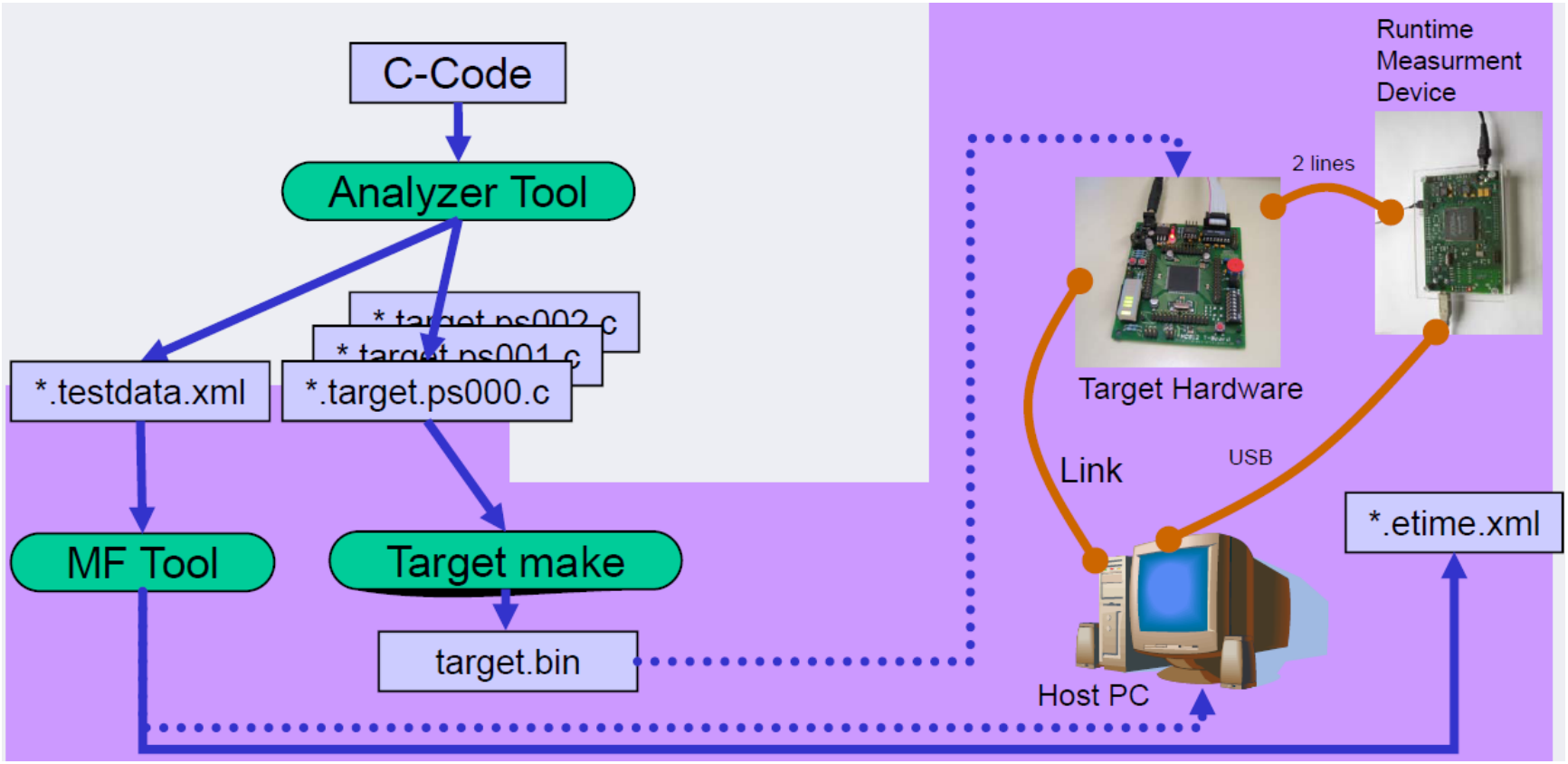
How to Measure?

- End-to-end or measuring code segments?
  - End-to-end is easy, but inaccurate, intractable
  - Measurement of code segments + Calculation
- How to Measure?
  - Software instrumentation
    - Put time recording in the analyzed codes
    - Accuracy?
  - Hardware instrumentation
    - Logic Analyzers, oscilloscopes, …

Hardware Instrumentation



Execution Time Measurement Framework



Instrumentation Methods

- Requirements
  - Instrumentations (IPs) may not alter program flow or execution time in an unknown or unpredictable way. IPs have to be persistent if changing either.
  - Execution always starts with the same (known) state (cache, pipeline, branch prediction, ...)
- Design Decisions
  - Control flow manipulation? Input data generation?
  - Number of measurement runs?
  - Resource consumption?
  - Required devices?
  - Installation effort?

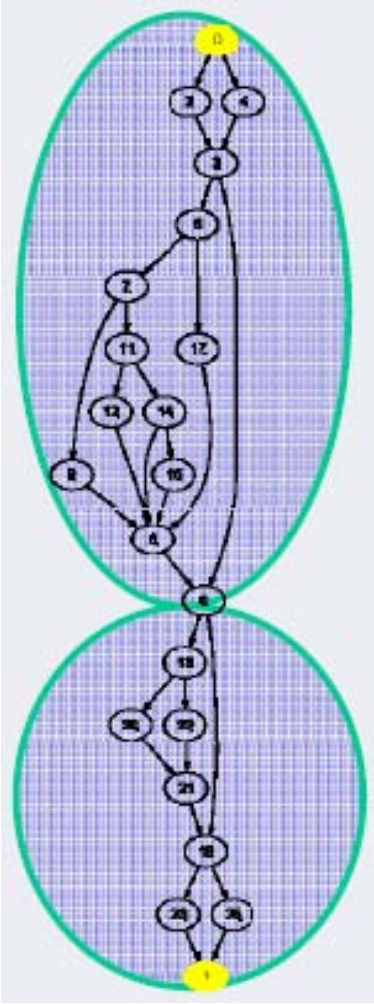
The Steps of Measurement-Based Analysis

1. Static analysis: reconstruct CFG from the code
2. Program partitioning
3. Test data generation
4. Execution time instrumentation
5. WCET calculation

- This is only one exemplary workflow, other measurement-based methods may have different workflow

Program Partitioning

- What is a program segment?
  - Roughly a sub-graph of the CFG
- Why program partitioning?
  - Reduce problem state space → reduce analysis efforts
  - Precision is sacrificed
- Partitioning granularity
  - Fewer segments → less instrumentation efforts but higher analysis computation overhead
- "Good" partitioning
  - Balance "the # of program segments" and "the average # of paths per segment"

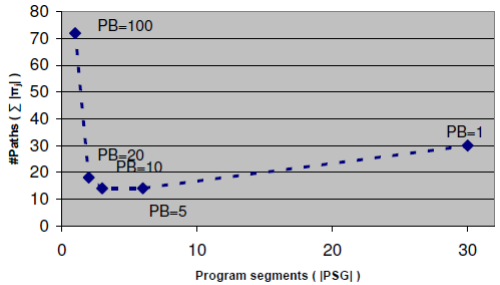


Program Partitioning

- An Example of Program Partitioning

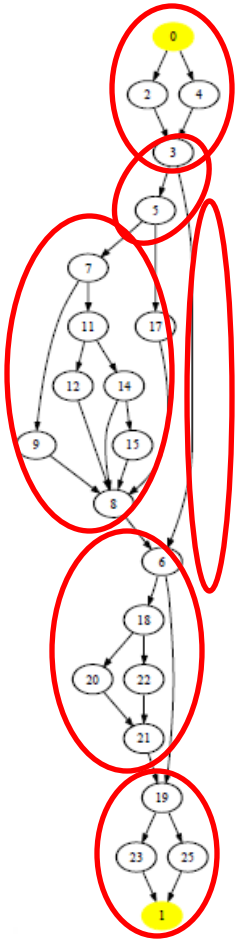
Path Bound	|PSG|	#Paths ($\sum \|\pi_j\|$)
1	30	30
5	6	14
10	3	14
20	2	18
100	1	72

(a) Partitioning Results



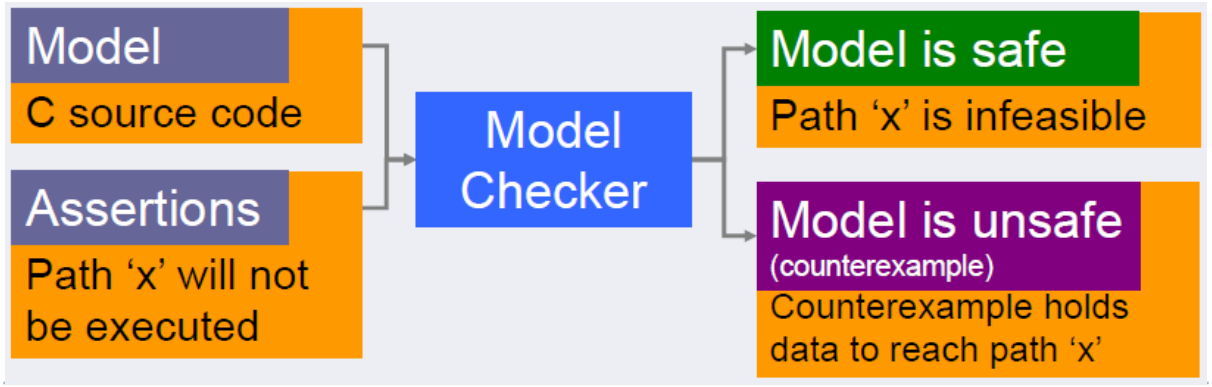
(b) Dependency between $|PSG|$ and $\sum |\Pi_j|$

```
1   int x;
2
3   int main_nice_partitioning(
4           int y, int i, int a, int b)
5   {
6     if (x == 1) {
7       x++; // BB 2
8     } else {
9       x--; // BB 4
10    }
11    // BB 3
12    if (b == 1) {
13      // BB 5
14      if (a == 1) {
15        // BB 7
16        if (x == 3) {
17          x++; // BB 9
18        } else {
19          // BB 11
20          if (x == 2) {
21            x++; // BB 12
22          } else {
23            // BB 14
24            if (x == 4) {
25              x++; // BB 15
26            }
27          }
28        }
29      } else {
30        x++; // BB 17
31      }
32      x++; // BB 8
33    }
34    // BB 6
35    if (b == 2) {
36      // BB 18
37      if (a == 1) {
38        x++; // BB 20
39      } else {
40        x--; // BB 22
41      }
42      x++; // BB 21
43    }
44    // BB 19
45    if (y == 1) {
46      x++; // BB 23
47    } else {
48      x--; // BB 25
49    }
50  }
```

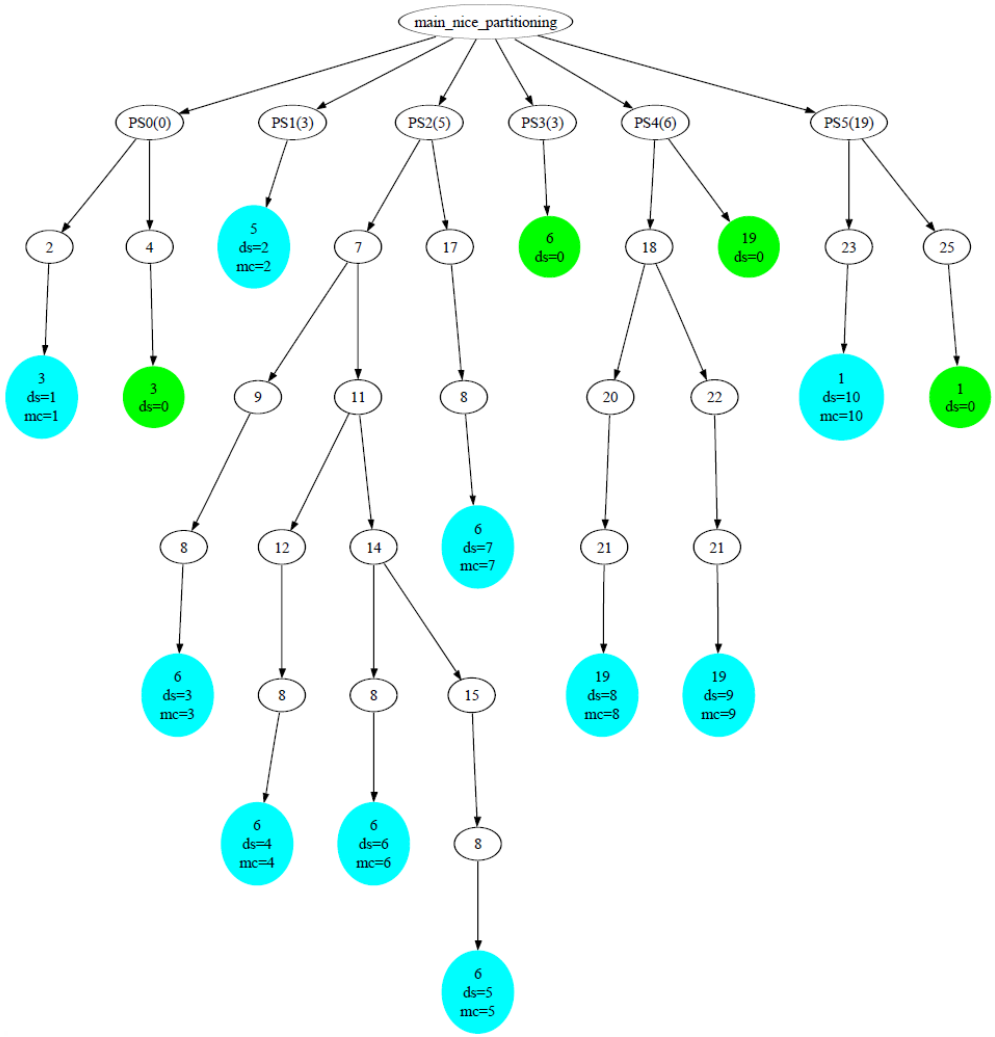


Test Data Generation

- What is the so-called "test data"?
  - Roughly, the values of a set of variables that leads to one of the paths of a program segment
- What is the use of "test data"?
  - Put code instrumentations at the segment boundaries, and set the test data to some specific values, which can leads the program to the desired path
- How to obtain "test data"? – model checking



Test Data Generation



Test Data Generation

- Execution Time Measurement
  - Use software instrumentation to guide the program
  - Use hardware instrumentation to measure execution time
- Enforcing Predictable Hardware States
  - Challenge: on complex hardware where the instruction timing depends on the execution history
  - Code instrumentations can be used to enforce an a-priori known state at the beginning of a program segment, thus avoiding the need for considering the execution history
- WCET Calculation
  - Use ILP, Model Checking, or any optimization tools to do longest path search

Probabilistic WCET Analysis

- What is probabilistic WCET analysis?
  - It gives you a distribution of the execution time of a program, instead of single WCET value
- Why probabilistic WCET analysis?
  - To determine the probability distribution of the execution times of tasks, then used to do probabilistic schedulability analysis in soft real-time systems
  - Helping to detect the "WCET hotspot", used for WCET reduction
  - Helping to analyze the execution behaviors of a program

Probabilistic WCET Analysis

- Solution: Probabilistic Timing Schema
  - Timing Schema
    - W(A) = exec time A
    - W(A;B) = W(A)+W(B)
    - W(if E then A else B) = W(E) + max(W(A),W(B))
  - Probabilistic Timing Schema
    - Sequential execution: Z = X + Y
    - Distribution functions: F(x) = P[X ≤ x], G(y) = P[Y ≤ y]
    - To compute H(z) = P[X + Y ≤ z]
    - If X and Y are independent $H(z) = \int_x F(x)G(z-x)dx$
    - If joint distribution between X and Y is given as J(x, y) $H(z) = \int_{x+y=z} j(x,y)$
    - If the joint distribution is unknown $H(z) = \int_{x+y=z} \frac{\partial^2 \min(F(x), G(y))}{\partial x \partial y}$

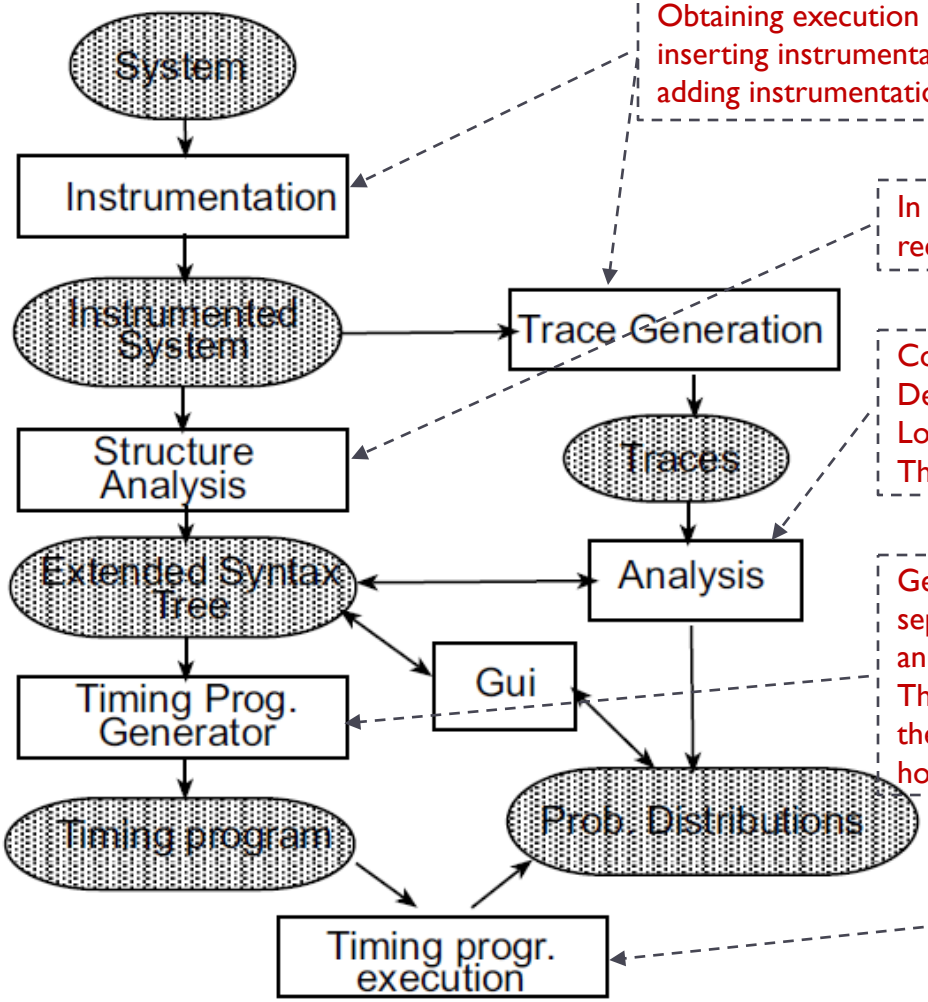
Probabilistic WCET Analysis

- Probabilistic Timing Schema
  - Conditional execution: Z = max(X,Y)
  - Z = E + max(X,Y), max(X,Y) has the distribution H(z)

$$H(z) = \int_{\max(x,y)=z} \frac{\partial^2 \min((F(x), G(y))}{\partial x \partial y}$$

  - Iteration: can be analyzed as a combination of sequence execution and conditional execution, loop bounds should be known
- Determining Probability Distributions
  - To determine the actual distribution of the execution times of individual units (basic blocks)
  - Run the units under a large number of test scenarios

The pWCET Analysis Tool



Obtaining execution traces. This is done by manually or automatically inserting instrumentation calls into the source code, or by automatically adding instrumentation codes into the compiled assembly code

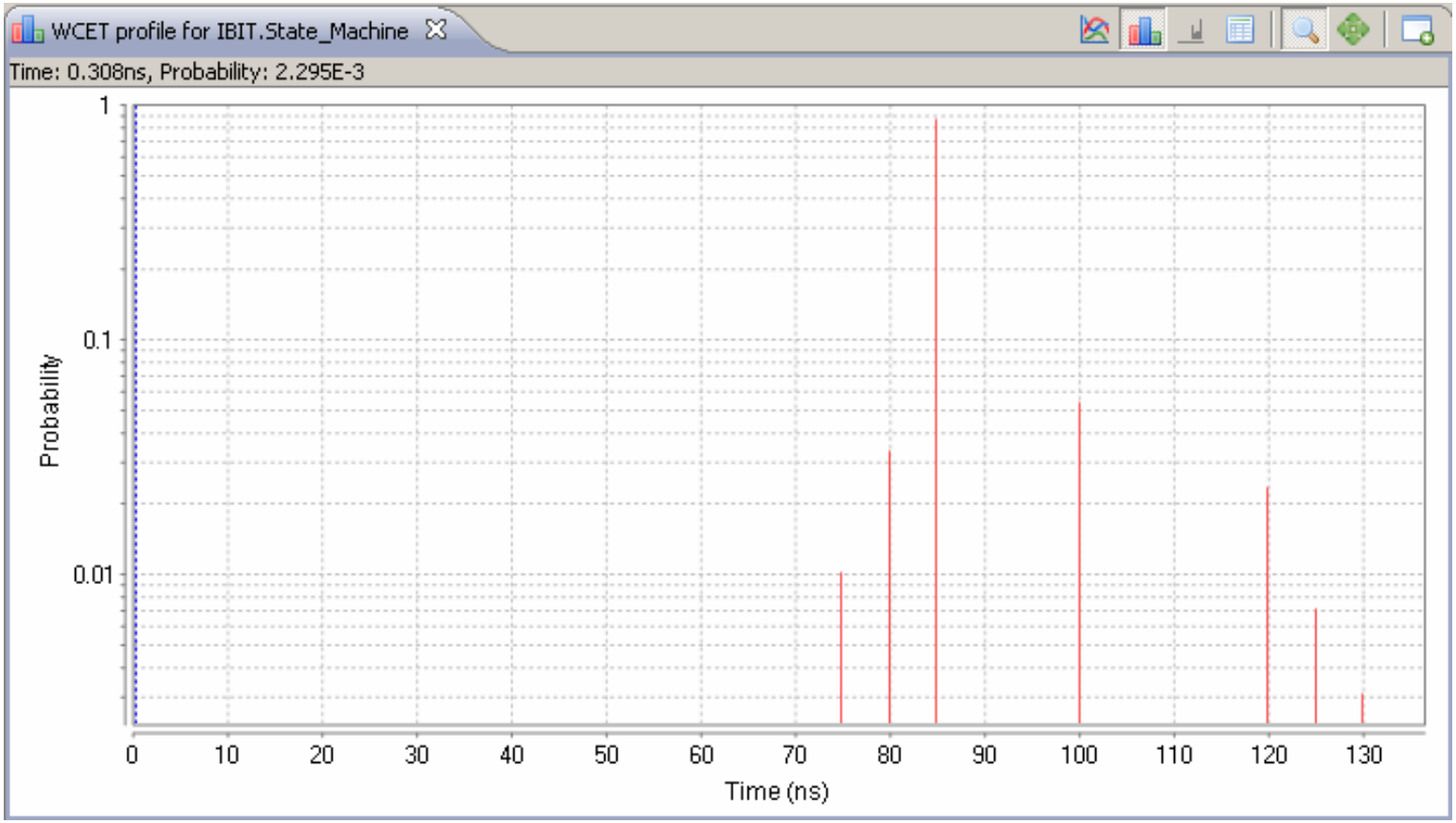
In this step, the CFG of the assembly code is reconstructed, and then converted into a syntax tree

Compute the distribution functions of each node from the traces;
Determine the joint distribution function of pairs of nodes;
Loop identification, loop iteration extracted;
This step is VERY computation expensive!!

Generate a program for WCET calculation, this is based on separating the timing analysis into a program generation part and an execution part.
The generator traverses the tree in reversed order and applies the timing schema rules, and the results is a set of commands on how to compute the timing program for the given tree.

Run the generated program with the program to be analyzed, and calculate the probabilistic distribution of the execution times of the program.

RapiTime Exemplary Results Report



A Survey of WCET 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. Vienna M. Vienna H.	- Genetic Algorithms Model Checking	static program analysis segment measurements segment measurements	IPET n.a. 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

A Survey of WCET Tools

- Support of Architectural Features

Tool	Caches	Pipeline	Periphery
aiT	I/D, direct/set associative, LRU, PLRU, pseudo round robin	in-order/out-of-order	PCI bus
Bound-T	-	in-order	-
RapiTime	n.a.	n.a.	n.a.
SymTA/P	I/D, direct/set-associative, LRU	n.a.	n.a.
Heptane	I-cache, direct, set associative, LRU, locked caches	in-order	-
Vienna S. Vienna M. Vienna H.	jump-cache n.a. n.a.	simple in-order n.a. n.a.	- n.a. n.a.
SWEET	I-cache, direct/set associative, LRU	in-order	-
Florida	I/D, direct/set associative	in-order	-
Chalmers	split first-level set-associative, unified second-level cache	multi-issue superscalar	-
Chronos	I-cache, direct, LRU	in-order/out-of-order, dyn. branch prediction	-

Contents

Introduction of This Research Topic

- Real-Life Real-Time Systems are Composed of
  - RTOS
  - Applications
- Timing Correctness of a Real-Time System is guaranteed by
  - Schedulability analysis in the high level
  - WCET analysis in the low level
- Applying WCET tools for application programs to RTOS
  - Poor results are reported (up to 86% pessimism)
  - Hard to handle some RTOS specific programs
- Additional analysis techniques are required!

WCET Analysis of RTEMS

- Research Group
  - Antoine Colin & Isabelle Puaut @ IRISA
- Experiment Setup
  - WCET tool: Heptane (tree-based)
  - RTOS: RTEMS
  - Manual revision to codes
  - 12 out of 85 system calls, span across 91 files, 14,532 LOC

WCET Analysis of RTEMS

- Problem 1: unstructured control flow
  - Such as goto statements, multiple loop exits, …
  - Because Heptane is a tree-based WCET analysis tool
  - Consequences: (1) rewriting the codes; (2) only a small subset of RTEMS system calls are analyzed
- Problem 2: Dynamic function calls implemented through function pointers
  - Real called functions are determined at runtime
  - Solutions: replace them with static ones

WCET Analysis of RTEMS

- Problem 3: Hard to determine loop bounds since the loop bounds are related to dynamic runtime behaviors
  - Task queue, message queue manipulation
  - Solution: Manually bound loops by an investigation of RTOS codes
- Problem 4: Blocking system calls
- Problem 5: Context switch overhead

- Putting them all together, an average of 86% pessimism in the estimated results is reported

- Challenges to WCET Analysis – Side Effects
  - It is apparent that the state space can be reduced via composable or hierarchical design/analysis
  - Side effects are defined as task interactions that cannot be traced back to task interface. For example, the shared cache may enable task A to influence the execution time of task B by displacing B's data in the shared region.
  - Side effects are a big problem to composable timing analysis

Predictable Architecture Design @ TuWein

- Side Effects in Simple Hardware Architectures
  - Variable program execution time due to
    - Unpredictable data input
    - Instructions with variable execution cycles dependent on operands
- In Complex Hardware Architectures
  - Different task instances may have different execution time
  - Scheduling without preemption: task instances from different tasks may execute alternatively, creating complex hardware states which are hard to predict
  - Scheduling with preemption: HW states change at preemption points, hard to predict when preemption will happen
  - Modern complex pipelines → flush not practical

Predictable Architecture Design @ TuWein

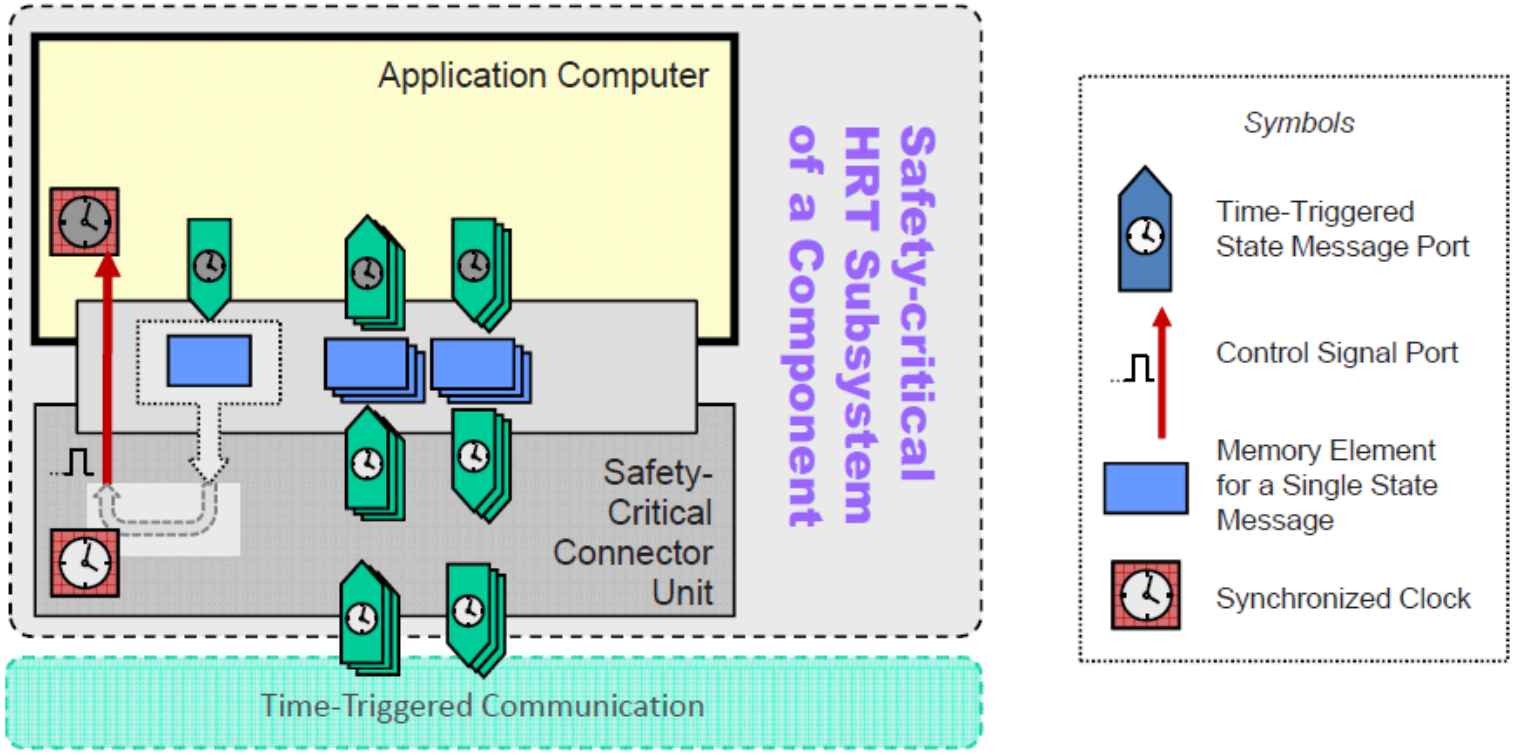
- Side Effects in Multicore Processors
  - Shared cache: if two tasks on two different cores share the same cache lines, it is hard to bound the effects of mutual replacement of cache contents
  - Other shared resources have similar problems
  - Simultaneous Multi-Threading (SMT): also called hyper-threading by Intel, multiple tasks on the same core share the function units at instruction level, hard to analyze the execution time of each task with good precision

Predictable Architecture Design @ TuWein

- Solutions
  - The basic philosophy of Puschner's solutions is to try every possibility to AVOID unwanted interactions
    - (1) The use of single-path code in all tasks
    - (2) The execution of a single task/thread per core
    - (3) The use of simple in-order pipelines
    - (4) Statically scheduled access to shared memory in CMPs
  - The solutions require redesign in both hardware and software (at both system level and application level)

Predictable Architecture Design @ TuWein

- An RTOS for a Time-Predictable Computing Node



Predictable Architecture Design @ TuWein

- Requirements on Hardware Architectures
  - The execution times of instructions are independent of the operand values
  - The CPU support a conditional move instruction having invariable execution times
  - Direct-mapped or set-associative caches with LRU
  - Memory access times are invariable for all data items
  - The CPU has a programmable instruction counter that can generate an interrupt when a given number of instructions has been completed

Predictable Architecture Design @ TuWein

- The SW Architecture – Task Model
  - Simple Task Model
    - I/O operations will never block a task
    - No statements for explicit I/O or synchronization within a task
    - All inputs are ready at task startup
    - Outputs are ready in the output variables when the task completes
  - Single-path Tasks
    - Transformation techniques

- Single-Path Transformation

```
if cond                          tmp1 := expr1;
then result := expr1;            tmp2 := expr2;
else result := expr2;            test cond;
                                 movt result, tmp1;
                                 movf result, tmp2;
```

if *cond*
then $(v_1,...,v_n) := F1(v_1',...,v_m')$
else $(v_1,...,v_n) := F2(v_1',...,v_m')$

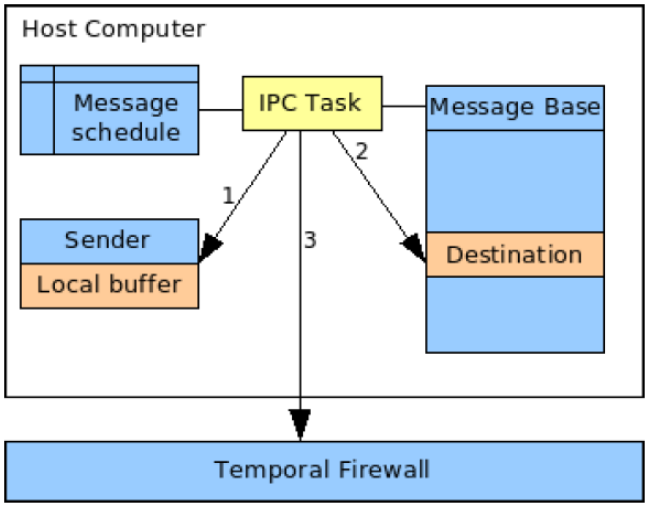
$(h_1,...,h_n) := F1(v_1',...,v_m')$
$(h_1',...,h_n') := F2(v_1',...,v_m')$
cond: $(v_1,...,v_n) := (h_1,...,h_n)$
not *cond*: $(v_1,...,v_n) := (h_1',...,h_n')$

-- conditions so far: *cond-old*
while *cond-new* **do** max *expr* times
  stmts;

$finished_x$:= **false**;
for i_x := 1 **to** *expr* **do**
begin
  if not *cond-new*
  then $finished_x$:= **true**;
  if *cond-old* **and not** $finished_x$
  then *stmts*;
end

Predictable Architecture Design @ TuWein

- The SW Architecture – RTOS
  - There must be no jitter in the execution times of the RTOS routines
  - Kernel designed using the single-path techniques
  - Communications: messages are scheduled at fixed time off-line



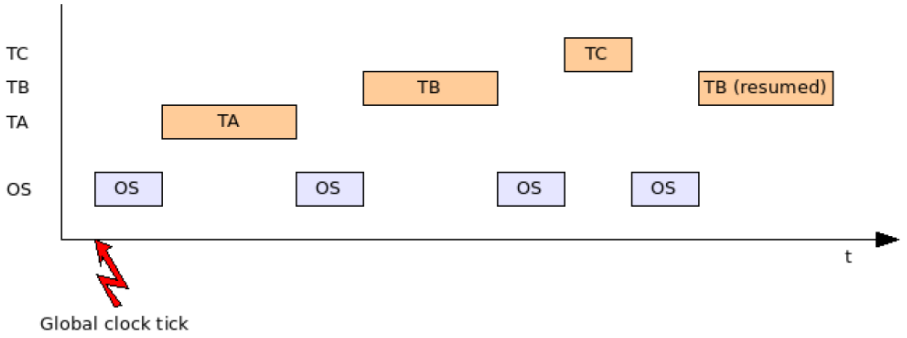
1 Local buffer accessed by tasks
2 Global buffer managed by IPC
3 Inter-node communication
4 Message schedule defined off-line

Predictable Architecture Design @ TuWein

- The SW Architecture – RTOS
  - Scheduler
    - Time-triggered
    - Schedule is determined off-line
    - Scheduler invoked at each global clock tick
    - Mode-switch is implemented by schedule switch, also determined off-line
    - Tasks are divided into "initialization phase" and "real-time phase", the former is non-real-time, the latter is managed by the RTOS

Predictable Architecture Design @ TuWein

- An Example



Task	Time of activation	Execution time	Termination
Kernel	0	1341	1341
Scheduler	1341	5634	6975
Kernel	6975	1626(+43)	8644
T_A	8644	611	9255
Kernel	9255	1626(+43)	10924
IPC	10924	628	11552
Kernel	11552	1626(+43)	13221
T_B	13221	1000	14221
Kernel	14221	1626(+43)	15890
T_C	15980	37	15927
Kernel	15927	1626(+43)	17596
$T_B(resumed)$	17596	1771	19369

Predictable Architecture Design @ TuWein

- Evaluations
  - Puschner has posed insights on design for predictability
  - Single-path technique is too costly and rigid
  - Requiring both specialized hardware and software (RTOS) may be impractical
  - In all, the ultimate predictability is achieved at the cost of system flexibility

Combined Schedulability & WCET Analysis

- Schneider studied combined schedulability & WCET analysis in his Ph.D. thesis, issues discussed in his work include
  - The quality of WCET analysis of RTOS can be improved by considering both the applications and the RTOS
  - In real-life multitasking real-time systems, tasks are executed in an interleaving manner (interruptions), but this is not considered in traditional WCET analysis, under such a circumstance, both scheduling and WCET must be re-think

Combined Schedulability & WCET Analysis

- Why Combined Schedulability & WCET Analysis?
  - Traditional schedulability and WCET analysis are performed in a hierarchical manner where the WCETs of the tasks are calculated first, then the results are fed to schedulability analysis
  - It is implied that even a task is interrupted, the WCET of all its segments equals the WCET of the task without interruptions
  - In multi-tasking systems running on complex hardware, the assumptions for hierarchical analysis is invalidated

Combined Schedulability & WCET Analysis

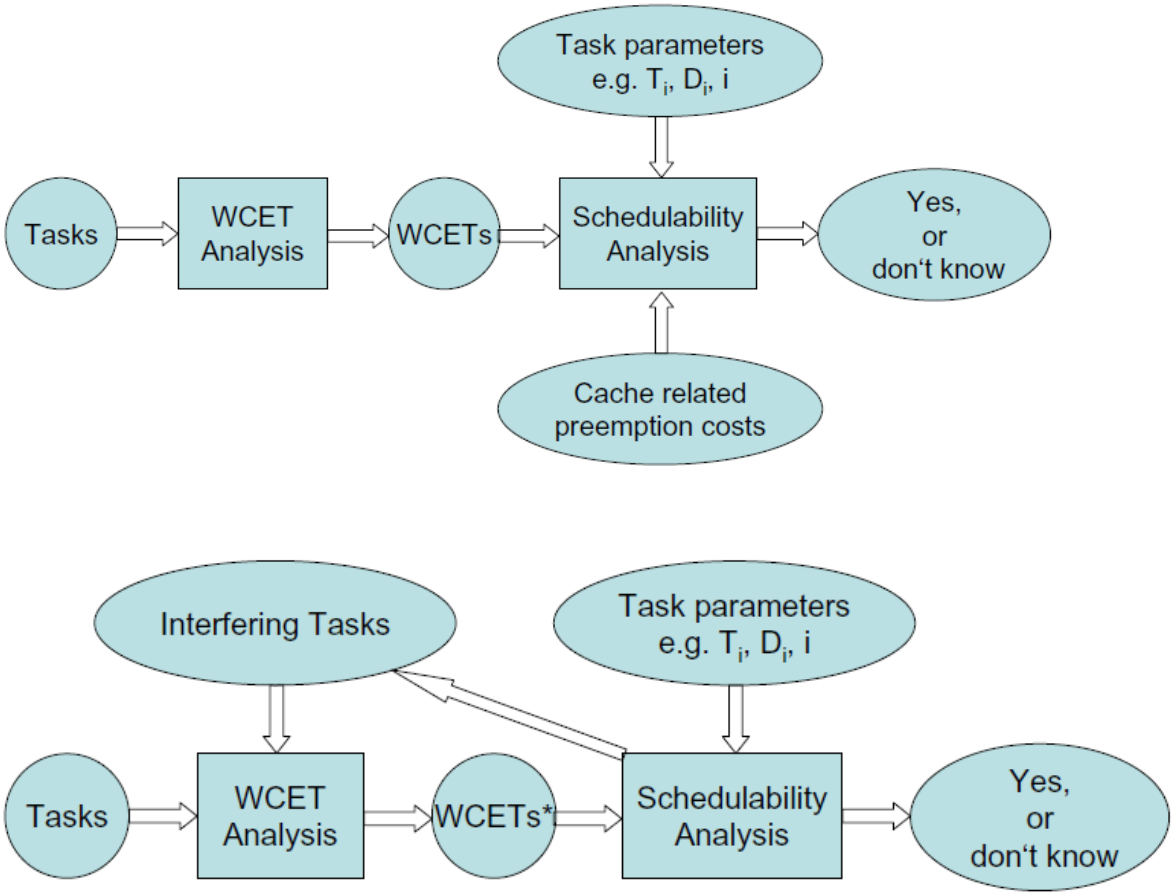
- Why the assumption is invalidated?
  - As we have discussed in previous slides, the WCET of a program highly depends on the processor states in presence of complex hardware
  - If a program is interrupted during execution, when it resumes, the hardware state is not identical to that at the interruption point, the influences are complex:
    - Some needed cache contents are swapped out, so the WCET in presence of interruption is larger than that without interruption
    - If timing anomaly occurs, the displacement of cache contents may leads to a smaller WCET

Combined Schedulability & WCET Analysis

- How to deal with these problems?
  - Consider the scheduling behavior within the WCET analysis process, and capture the state change at the interruption points
  - Re-calculate the WCET by considering the state change
  - Re-do schedulability analysis with new WCET values

Combined Schedulability & WCET Analysis

- The Old and New Analysis Framework



A Summary of Research Practices in WCET Analysis of RTOS

	Colin	Schneider	Sandell	Petters	Puschner
RTOS Analyzed	RTEMS	OSE	OSE	L4	
Analysis Tool	HEPTANE	aiT	aiT/SWEET	Petters' Tool	
Average Overestimation	86%	n/a	n/a	n/a	
Problems Due to Program Features					
Irreducible Program Structure	P2			P2	
Indexed Jumping				S	
Problems Due to Lack of Application Information					
Hard to Bound Loops Due to Runtime Properties	P2,3		P2,3	N	
Dynamic Function Calls	P4			P4	
Blocking System Calls	N				
Lack of Knowledge on System Call Contexts		P3			
Lack of Knowledge on RTOS Running Mode			P2		
Problems Due to Task Switching and Inter-Task Interference					
Timing Effects Due to Task Switching		P1			P4
Timing Anomalies Due to Preemption		P1			P4
Inaccurate Execution Time of Context Switches	N	S	N		P4
Inter-Task Interference Due to Resource Sharing on Multicores				N	N

* "S": the problem is properly solved
* "N": the problem is circumvented in related research
* "P": the problem is partially solved, but needs further development. Possible problems may be low scalability of the analysis (P1), too much user intervention required (P2), low quality of the results (P3), or the adopted techniques are too restrictive (P4)

A Summarization of Problems

- Problem 1: Irreducible program structures
  - Solution: choose a proper WCET tool
- Problem 2: Lack of application information greatly affects analyzability and the precision of the results
  - Bounding loops
  - Dynamic function calls and blocking system calls
  - System call context and RTOS working mode
  - Solution: extract helpful information from applications
- Problem 3: multi-tasking
  - Solution: develop analysis techniques that can safely bound the effects of task switching

Challenges on WCET Analysis of RTOS

- Does Single WCET Value Suffice?
  - The running of RTOS is mode-based, so a single WCET value regardless of execution mode is not sensible
  - Related techniques, such as parametric ILP should be developed
- Considering Both Applications and RTOS
  - Application information may be very useful to RTOS analysis, e.g. bounding loops
  - What kinds of application information should be communicated to the analyzer?
  - How can these information be communicated to the analyzer?

Challenges on WCET Analysis of RTOS

- Combined Schedulability and WCET Analysis
  - There is a mutual communication between schedulability analysis and WCET analysis
  - Control of the state space explosion
- Raising the Degree of Automation
  - Almost all related research practices reported low degree of automation in the analysis
  - WCET tool designers must always keep the issue of "automation" in mind when designing tools
  - The degree of automation is the largest factor that affects the usability of a WCET tool

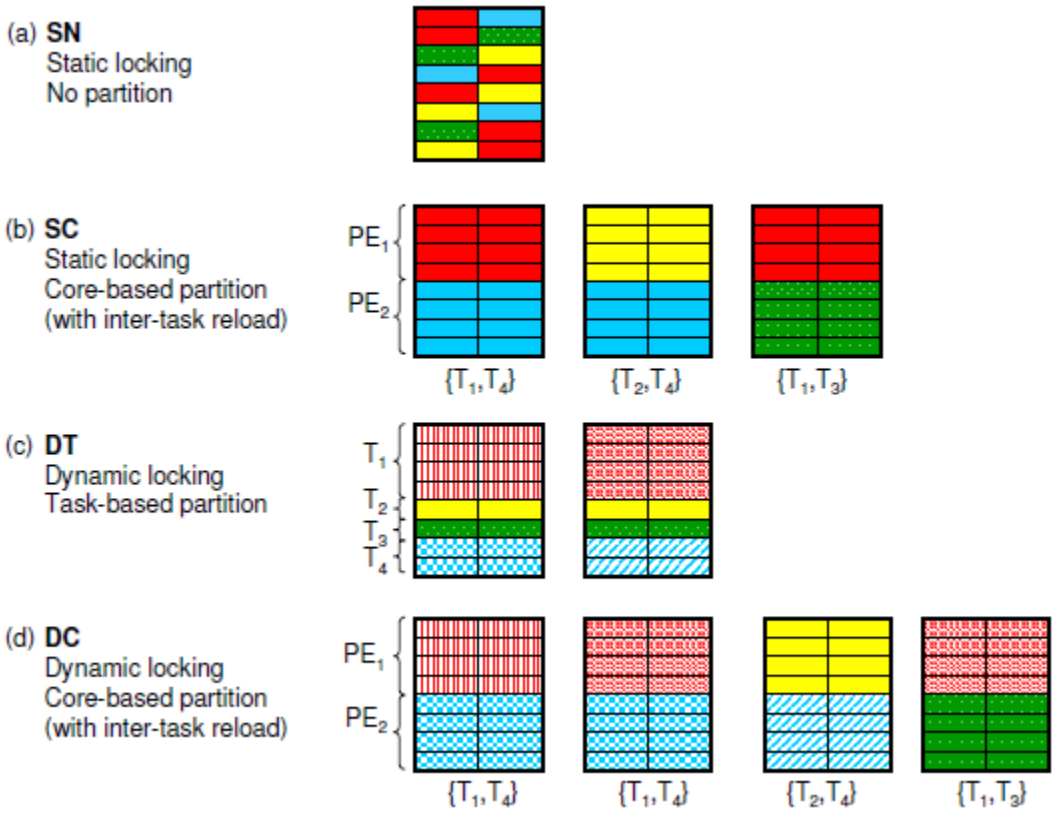
Challenges on WCET Analysis of RTOS

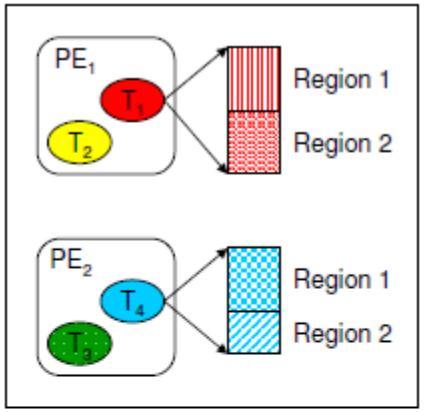
- Managing Analysis Complexity in the Multicore Era
  - Problem: fine-grained access to shared resources (L2 cache, on-chip bus, …), and for most existing architectures, we have very limited ability to control the behavior of these shared resources
  - Solution: Performance isolation techniques (cache partitioning), since such techniques can "create" an isolated environment for each core, and at the same time still maintains the flexibility that shared resources provide with

Challenges on WCET Analysis of RTOS

- To Design or to Analyze?
  - Analyze
    - No need to change existing hardware or system; analysis must be done if you're to analyze fabricated systems
    - But lots of hardware features or management policies are not designed for real-time, these features make the analysis very hard
    - To guarantee predictability on unpredictable hardware, a lot of pessimism is introduced into the results → system over design
  - Design
    - To design hardware or software with the consideration of real-time from scratch can yield very predictable systems
    - Predictability is achieved by sacrificing flexibility
    - New hardware requires re-design of the system, from hardware, to programming tools, to OS and applications
  - A Graceful Balance!

Cache Partitioning and Locking





	Static locking	Dynamic locking
No partition	**SN**	--
Task-based partition	ST	**DT**
Core-based partition	**SC**	**DC**

(e) Comparison Table

Partitioning is used to avoid inter-task interference regardless of single- or multi-core.
Locking is used to enforce predictability in terms of cache hits/misses

Contents

Trends in Hardware

- More software control
  - Software-controlled cache locking
  - Scratchpad memory
  - More predictable caches or pipelines
- Multi-core processors
  - + multiple simple cores
  - - Shared cache → inter-task interference
  - - Share whatever, on-chip buses or networks
- Execution Behavior
  - Traditionally, researchers assume single task execute on single core, but this is not necessarily the whole story
  - A big gap between WCET and ACET

Trends in Software

- Levels of Abstraction
  - Traditionally C code or assembly code
  - A trend towards higher-level abstraction, e.g. OO languages, model-based design
  - More dynamic control structure, hard to reconstruct CFG
  - more dynamic data structure, memory access
  - Java VM, JIT compilation

- Component-based design
  - FSM synthesize highly unstructured code
  - Parameterized execution time/WCET

Trends in Analysis Techniques

- WCET-aware Compilation
  - Try to tackle the analysis complexity problem in compilers
  - Develop compilers that can generate predictable codes
- Raise Automation Level
  - Automatic extraction of flow facts, less user intervention
  - Flow facts mapping across different representation levels
- Parametric WCET Analysis
  - Obtain a function for WCET results, instead of a single WCET value
- Integrate WCET analysis with power-aware techniques
- Integrate WCET analysis with scheduling analysis

Contents

Recommended Readings

- **Books**
- Flemming Nielson, et al, *Principles of Program Analysis*, Springer, 2004.
- John L. Hennessy and David A. Patterson, Computer Architecture, A Quantitative Approach, 4th edition, Elsevier, 2006.
- Mostafa Abd-El-Barr and Hesham El-Rewini, *Fundamentals of Computer Organization and Architecture*, John Wiley & Sons, 2005.
- **Related Course Pages**
- http://ti.tuwien.ac.at/rts/teaching/courses/wcet-ss08
- **Referenced Papers**
- Surveys and Overview Papers
- R. Wilhelm, et al. *The worst-case execution-time problem—overview of methods and survey of tools*. Trans. on Embedded Computing Sys., 7(3):1–53, 2008.
- Mingsong Lv, Nan Guan, Yi Zhang, Qingxu Deng, Ge Yu, *Static Timing Analysis of Real-Time Opearting Systems – Survey of Research and New Challenges*, NEU-RTES lab report, 2008.
- Bjorn Lisper. *Trends in Timing Analysis*. 2006.
- C. Ferdinand and R. Heckmann. *Worst-case execution time - a tool provider's perspective*. In ISORC 2008.
- Raimund Kirner, Peter Puschner. *Classification of WCET Analysis Techniques*. In ISORC 2005.
- Rainmund Kirner, Peter Pushner. *Obstacles in Worst-Case Execution Time Analysis*. In ISORC 2008.

Recommended Readings

- J. Gustafsson. *Usability aspects of wcet analysis*. In ISORC 2008.
- J. Gustafsson and A. Ermedahl. *Experiences from applying wcet analysis in industrial settings*. In ISORC 2007.
- Jan Gustafsson, et al. *ALL-TIMES – a European Project on Integrating Timing Technology*. 2008.
- Static Analysis
- Xianfeng Li, et al. Chronos: *A Timing Analyzer for Embedded Software*. 2006.
- Yau-Tsun Steven Li, et al. Cinderella: *A Retargetable Environment for Performance Analysis of Real-Time Software*. In EuroPar 1997.
- Yau-Tsun Steven Li, et al. *Performance Analysis of Embedded Software Using Implicit Path Enumeration*. In DAC 1995.
- Yau-Tsun Steven Li, et al. *Performance Estimation of Embedded Software with Instruction Cache Modeling*. 1999.
- Yau-Tsun Steven Li, et al. *Cache Modeling for Real-Time Software: Beyond Directed-Mapped Instruction Caches*. 1996.
- Mingsong Lv, et al. *Performance Comparison of Techniques on Static Path Analysis of WCET*. In EUC 2008.
- T. Lundqvist and P. Stenstrom. *Timing Anomalies in Dynamically Scheduled Microprocessors*. In RTSS 1999.
- J. Reineke, et al. *A Definition and Classification of Timing Anomalies*. In WCET 2006.
- Measurement-Based Analysis
- Ingomar Wenzel, et al. *Measurement-Based Timing Analysis*. 2008.

Recommended Readings

- Ingomar Wenzel. *Measurement-Based Timing Analysis of Superscalar Processors*. Ph.D. thesis, 2006.
- Guillem Bernat, et al. *pWCET: a Tool for Prebabilistic Worst-Case Execution Time Analysis of Real-Time Systems*. 2003.
- Guillem Bernat, et al. *WCET Analysis of Probabilistic Hard Real-Time Systems*. In RTSS 2002.
- WCET Analysis of RTOS
- M. Carlsson, J. Engblom, A. Ermedahl, J. Lindblad, and B. Lisper. *Worst-case execution time analysis of disable interrupt regions in a commercial real-time operating system*. 2002.
- D. Sandell, A. Ermedahl, J. Gustafsson, and B. Lisper. *Static timing analysis of real-time operating system code*. In 1st International Symposium on Leveraging Applications of Formal Methods, 2004.
- A. Colin and I. Puaut. *Worst-case execution time analysis of the rtems real-time operating system*. 13th Euromicro Conference on Real-Time Systems, 2001.
- G. Khyo, P. Puschner, and M. Delvai. *An operating system for a time-predictable computing node*. The 6th IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, pages 150–161, 2008.
- P. Puschner and M. Schoeberl. *On composable system timing, task timing, and wcet analysis*. In WCET 2008.
- P. Puschner. *Transforming execution-time boundable code into temporally predictable code*. In 17th World Computer Congress - Stream on Distributed and Parallel Embedded Systems, 2002.
- J. Schneider. *Combined schedulability and wcet analysis for real-time operating systems*. Ph.D. thesis of Saarland University, Germany, 2002.

Recommended Readings

- J. Schneider. *Why you can't analyze rtoss without considering applications and vice versa*. 2nd International Workshop on Worst-Case Execution Time Analysis, 2002.
- M. Singal and S. M. Petters. *Issues in analysing l4 for its wcet*. Proceedings of the 1st International Workshop on Microkernels for Embedded Systems, 2007.
- Vivy Suhendra, Tulika Mitra. Exploring Locking & Partitioning for Predictable Shared Caches on Multi-cores. In DAC 2008.
- Tools & Projects
- ALL-Times: http://www.mrtc.mdh.se/projects/all-times/
- aiT: www.ait.com
- Bound-T: www.tidorum.fi/bound-t/
- RapiTime: www.rapitasystems.com
- SymTA/P:
- Heptane: http://www.irisa.fr/aces/work/heptane-demo/heptane.html
- Vienna: http://www.wcet.at/
- SWEET: http://www.mrtc.mdh.se/projects/wcet/
- OTAWA: http://www.otawa.fr/
- Chalmers: http://www.ce.chalmers.se/research/group/hpcag/project/wcet.html
- Chronos: http://www.comp.nus.edu.sg/~rpembed/chronos/

Visit Our Website ☺

- The Website of Real-Time Embedded Systems Laboratory, Northeastern University
  - http://www.neu-rtes.org
  - http://www.neu-rtes.org/courses/spring2009/
- You can find
  - General information on the projects conducted in our lab
  - Research and publications
  - Research information and contacts of the members
  - Some useful research links
- Write me emails if you have questions in WCET or RTS
  - mingsong@research.neu.edu.cn