EE382N.23: Embedded System Design and Modeling

Lecture 11 – Estimation and Evaluation

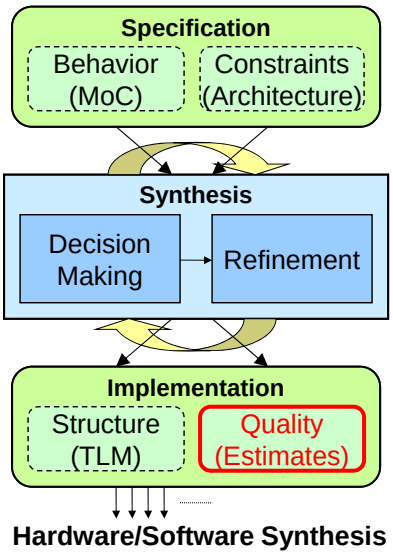
Andreas Gerstlauer
Electrical and Computer Engineering
University of Texas at Austin
gerstl@ece.utexas.edu

Lecture 11: Outline

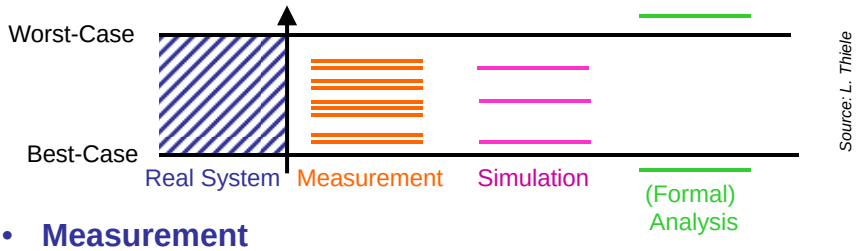
- **Evaluation and estimation**
  - Methods
- **Analysis**
  - Component- and system-level estimation
- **Simulation**
  - Simulation methods
- **Hybrid approaches**
  - Semi-analytical methods
  - Machine learning-based methods

System-Level Synthesis

- **X-Chart**



Evaluation and Estimation Methods



- **Measurement**
  - Fast (real time), exhaustive?
  - Requires physical implementation
- **Simulation**
  - Speed vs. accuracy tradeoffs
  - Quality of testbench, corner cases?
- **Analysis**
  - Worst-case/best-case assumptions
  - Tightness of upper/lower bounds? Dynamic effects?

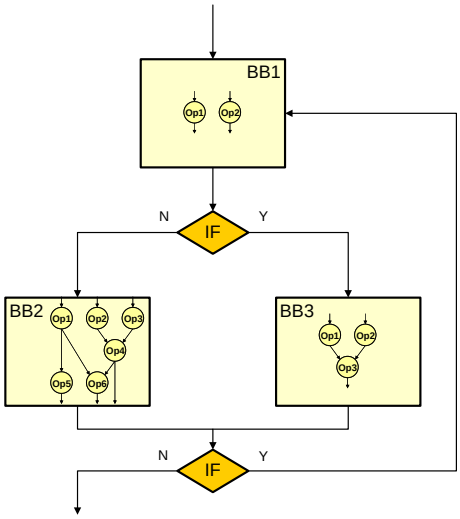
Analysis Methods

- **Static analysis**
  - Symbolic, mathematical models for avg/best/worst case
  - ➢ Worst-case execution time analysis (WCET) of single task
  - ➢ Real-time scheduling of single processor
  - ➢ Analytical performance models of computer systems
- **Probabilistic analysis**
  - Statistical models, distributions for “average” case
  - ➢ Queuing theory for computer systems
- **Deterministic dynamic analysis**
  - Min-plus/max-plus algebra, upper/lower bounds over time
  - ➢ Network calculus, real-time calculus
  - ➢ Modular Performance Analysis (MPA) of parallel systems

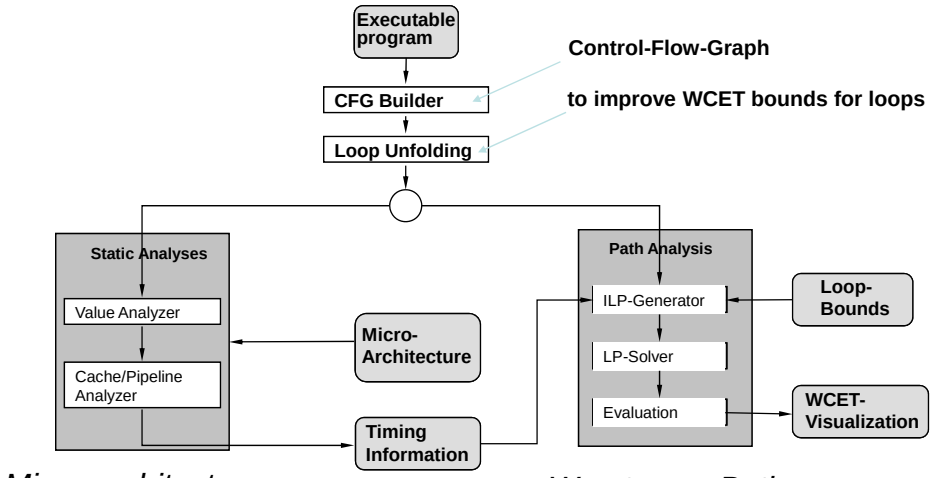
Task-Level Analysis

- **Worst-case execution time (WCET)**
  - Micro-architecture analysis
    - Compute bounds for each basic execution block
    - Symbolically simulate statements on processor model (pipeline)
    - ➢ Conservative assumptions for dynamic effects (caches, predictors)
  - Path analysis
    - Enumerate possible paths and take maximum of block sequence
    - ➢ Possible paths often highly dynamic (loop bounds, false paths)
  - ➢ Basis for back-annotation or static system analysis
    - ➢ Combine static code analysis with dynamic system simulation
    - ➢ Static or dynamic model of inter-process cross-dependencies

Control/Data Flow Graph (CDFG)



aiT Tool



Source: L. Thiele

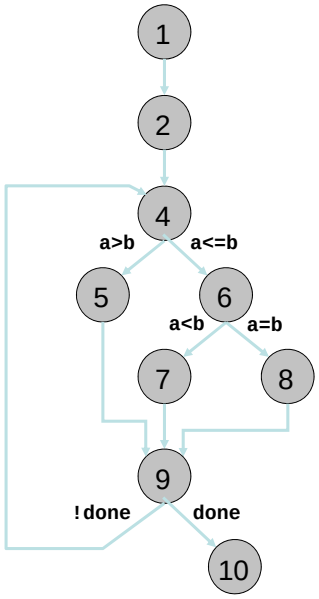
Program Path Analysis

- ***Program Path Analysis***
  - Which sequence of instructions is executed in the worst-case (longest runtime)?
  - ***Problem***: the number of possible program paths grows exponentially with the program length
- ***Model***
  - We know the upper bounds (number of cycles) for each basic block from static analysis
  - Number of loop iterations must be bounded
- ***Concept***
  - Transform structure of CFG into a set of (integer) linear equations.
  - Solution of the Integer Linear Program (ILP) yields bound on the WCET.

Source: L. Thiele

Control Flow Graph (CFG)

```
   what_is_this {
1     read (a,b);
2     done = FALSE;
3     repeat {
4       if (a>b)
5          a = a-b;
6       elseif (b>a)
7          b = b-a;
8       else done = TRUE;
9     } until done;
10    write (a);
   }
```

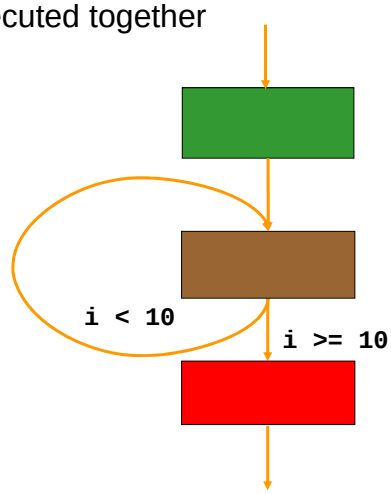


Source: L. Thiele

Control Flow Graph (CFG)

- **The nodes are the basic blocks**
  - Single point of entry & exit
  - Instructions in block always executed together

```
      i := 0
      t2 := 0
   L  t2 := t2 + i
      i := i + 1
      if i < 10 goto L
      x := t2
```



Source: L. Thiele

Calculation of the WCET

- ***Definition:*** **A program consists of N basic blocks, where each basic block B_i has a worst-case execution time c_i and is executed for exactly x_i times. Then, the WCET is given by**

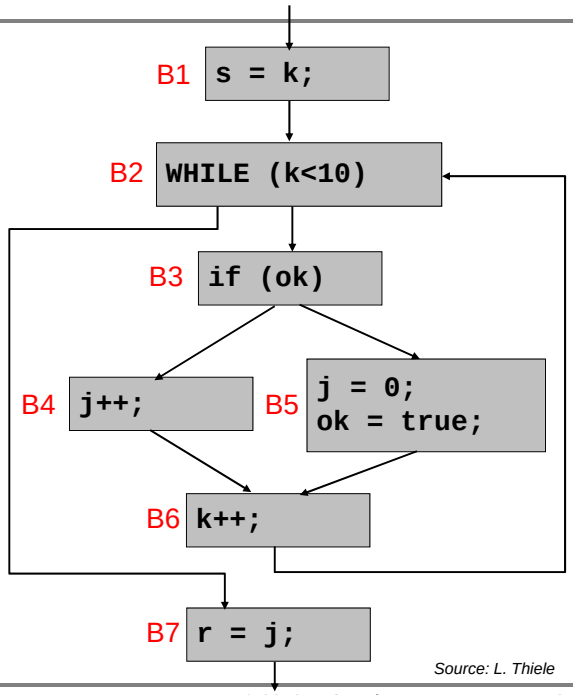
$$WCET = \sum_{i=1}^{N} c_i \cdot x_i$$

  - The c_i values are determined using static analysis.
  - How to determine x_i?
    - Structural constraints given by the program structure
    - Additional constraints provided by the programmer (bounds for loop counters, etc.; based on knowledge of the program context)

Source: L. Thiele

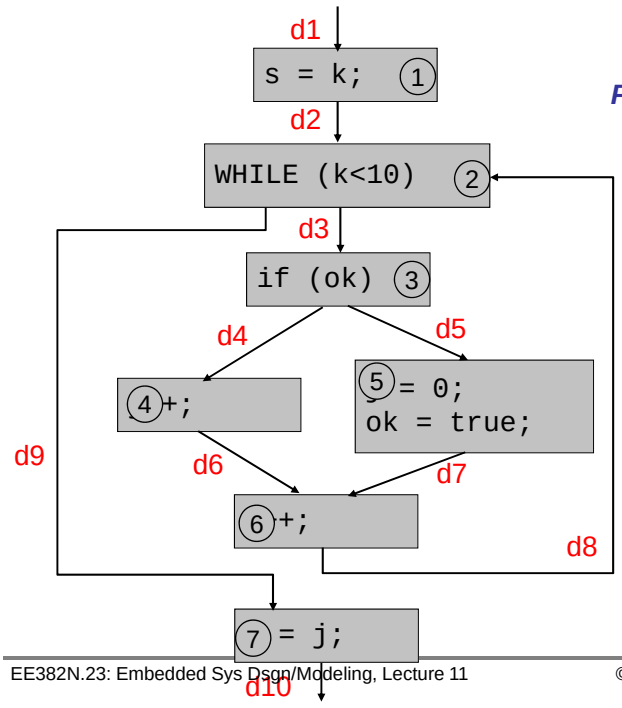
Example

```
/* k >= 0 */
s = k;
WHILE (k < 10) {
   IF (ok)
      j++;
   ELSE {
     j = 0;
     ok = true;
   }
   k ++;
}
r = j;
```



Source: L. Thiele

Structural Constraints



Flow equations:

$d1 = d2 = x_1$

$d2 + d8 = d3 + d9 = x_2$

$d3 = d4 + d5 = x_3$

$d4 = d6 = x_4$

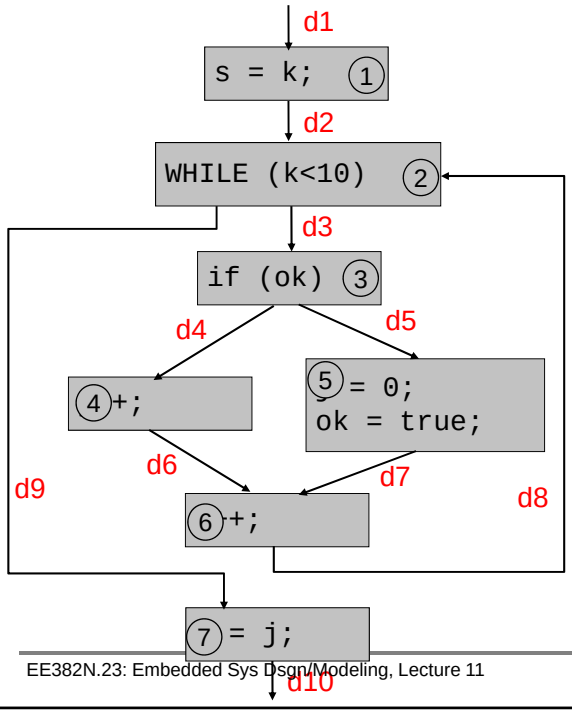
$d5 = d7 = x_5$

$d6 + d7 = d8 = x_6$

$d9 = d10 = x_7$

Source: L. Thiele

Additional Constraints



Loop is executed for at most 10 times:

$x_3 <= 10 \cdot x_1$

B5 is executed for at most one time:

$x_5 <= 1 \cdot x_1$

Source: L. Thiele

Integer Linear Program (ILP)

- **ILP with structural and additional constraints**

program is executed once

$$WCET = \max\{\sum_{i=1}^{N} c_i \cdot x_i \mid d_1 = 1 \wedge$$

$$\sum_{j \in in(B_i)} d_j = \sum_{k \in out(B_i)} d_k = x_i, i = 1..N \wedge$$

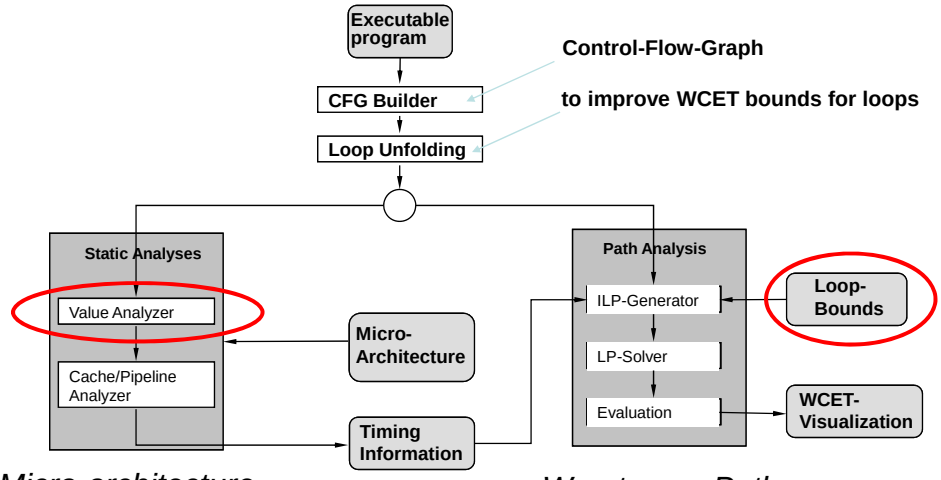
$$...\}$$

structural constraints

- **Apply standard ILP solver**
  - NP-complete, i.e. exponential complexity!

Source: L. Thiele

Overview



Micro-architecture Analysis

Worst-case Path Determination

Source: L. Thiele

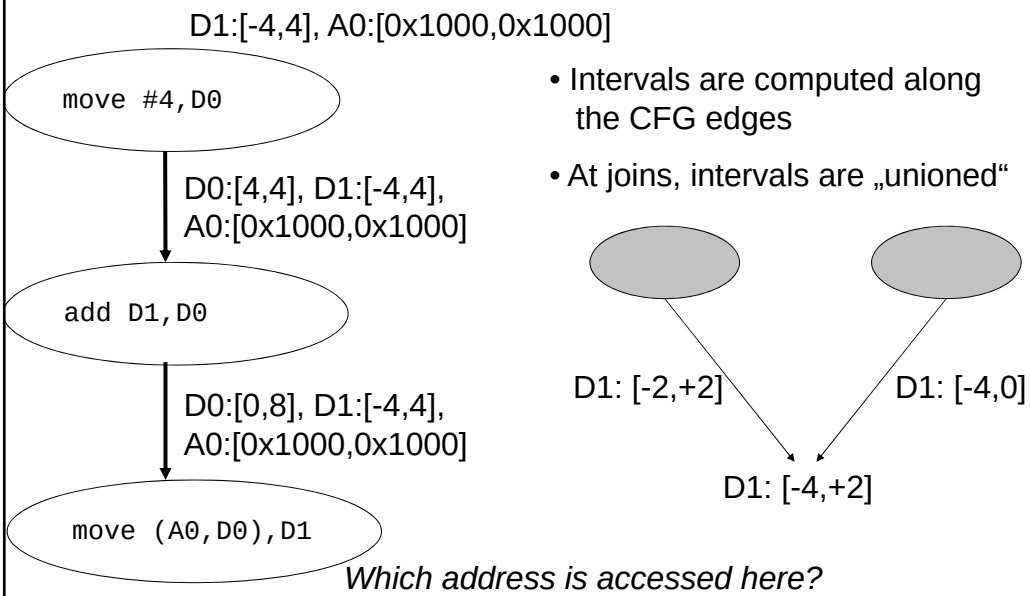
Value Analysis

- ***Motivation*:**
  - Provide access information to data-cache/pipeline analysis
  - Detect infeasible paths
  - Derive loop bounds
- ***Method*: calculate intervals at all program points, i.e. lower and upper bounds for the set of possible values occurring in the machine program (addresses, register contents, local and global variables)**

➢ **Abstract interpretation (AI)**
  - *Semantics-based method* for static program analysis
  - Perform the program's computations using value descriptions or *abstract values* in place of the concrete values, start with a description of all possible inputs
  - Supports *correctness* proofs

Source: L. Thiele

Value Analysis



- Intervals are computed along the CFG edges
- At joins, intervals are „unioned“

Which address is accessed here?

Source: L. Thiele

Caches

- **Caches are used, because**
  - Speed gap between CPU and main memory
  - ➢ Fast but expensive on-chip memory in between
- **Every memory access goes through the cache**
  - Block *m* containing *a* is in the cache (hit):
    request for *a* is served in the next cycle.
  - Block *m* is not in the cache (miss):
    m is transferred from main memory to the cache,
    m may replace some block in the cache,
    request for *a* is served ASAP while transfer still continues.
- **Several *replacement strategies*: LRU, PLRU, FIFO,...
  determine which line to replace.**

Source: L. Thiele

Static Cache Analysis

- **Abstraction**
  - Sets of memory blocks in single cache lines
    - From values to locations, ignoring arithmetic

➢ ***Must Analysis***

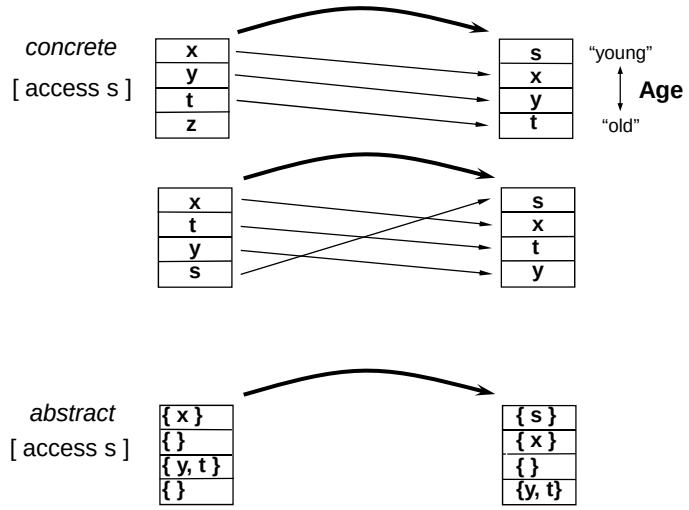
- For each program point (and calling context), find out which blocks are in the cache
- Determines safe information about cache hits. Each predicted cache hit reduces WCET.

➢ ***May Analysis***

- For each program point (and calling context), find out which blocks may be in the cache. Complement says what is not in the cache
- Determines safe information about cache misses. Each predicted cache miss increases BCET.

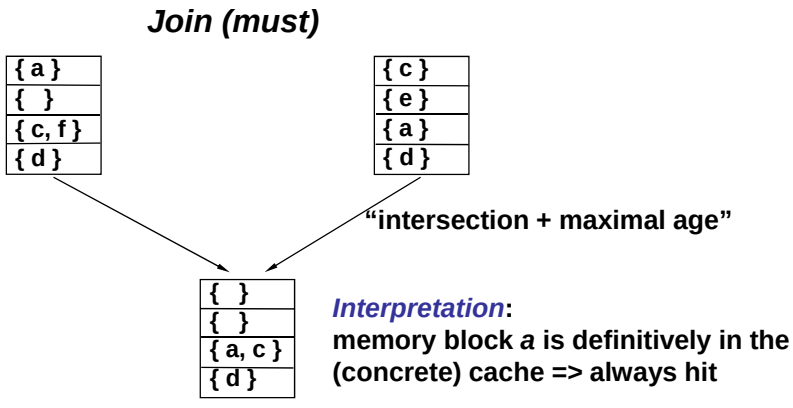
Source: L. Thiele

Cache with LRU: Must Analysis



Source: L. Thiele

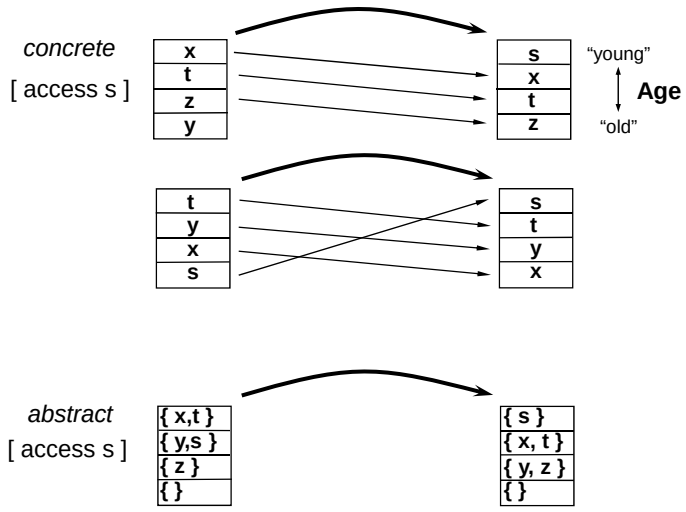
Must Analysis: Join



***Interpretation*:**
memory block *a* is definitively in the (concrete) cache => always hit

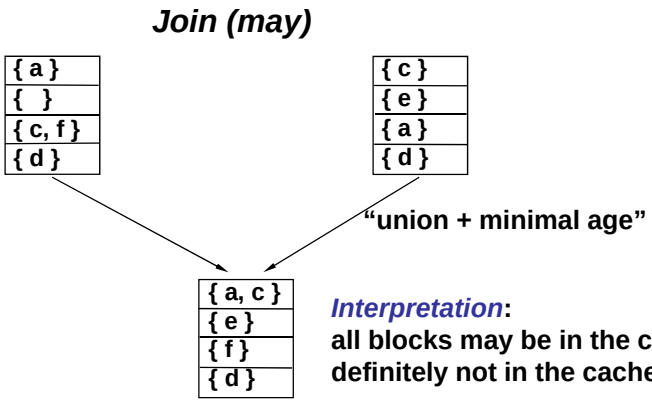
Source: L. Thiele

Cache with LRU: May Analysis



Source: L. Thiele

May Analysis: Join

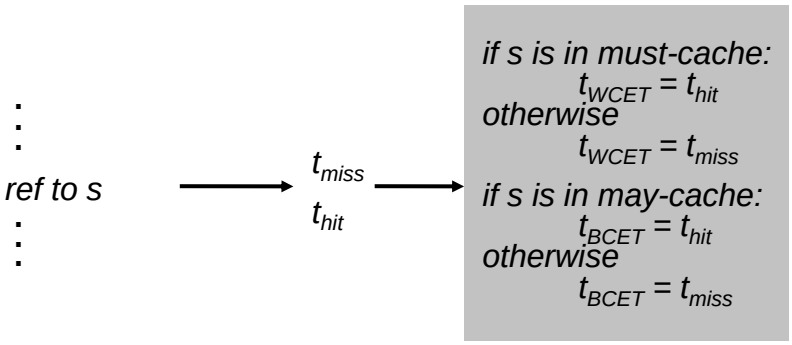


***Interpretation*:**
all blocks may be in the cache; none is definitely not in the cache.

Source: L. Thiele

Cache Analysis for WCET

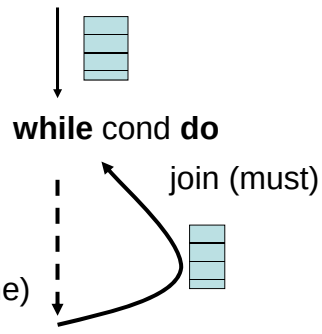
- **Information about cache contents sharpens timings**



Source: L. Thiele

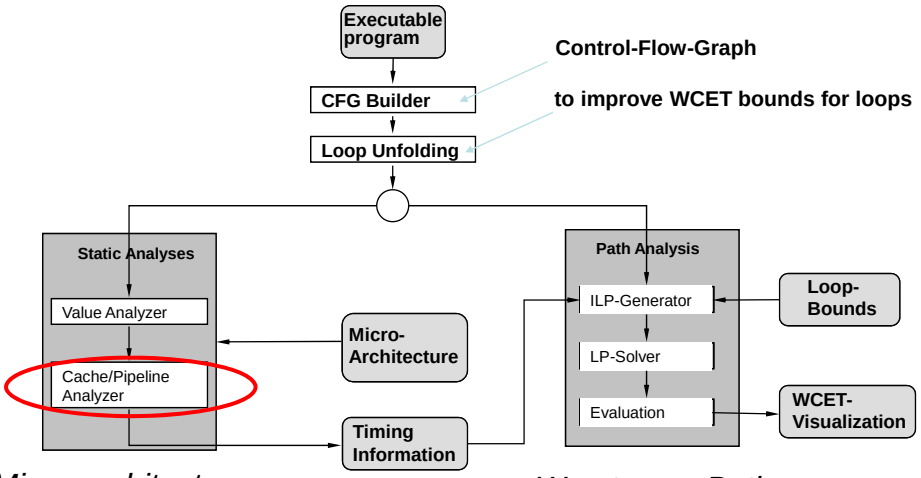
Cache Analysis Contexts

- **Cache contents depends on the context**
  - Calls and loops
- **First iteration loads the cache**
  - Intersection looses most of the information
- **Distinguish as many contexts as useful**
  - 1 unrolling for caches
  - 1 unrolling for branch prediction (pipeline)



Source: L. Thiele

Overview



Source: L. Thiele

Static Analysis of Hazards

***Cache analysis*: prediction of cache hits on instruction or operand fetch or store**

```
lwz r4, 20(r1)
```

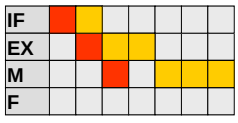
Hit

***Dependency analysis*: analysis of data/control hazards**

```
add r4, r5,r6
lwz  r7, 10(r1)
add r8, r4, r4
```

Operand ready

***Resource reservation tables*: analysis of resource hazards**



Source: L. Thiele

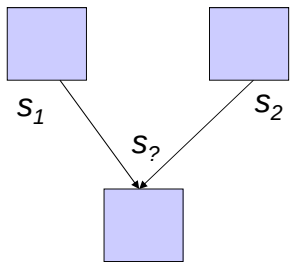
Abstract Pipeline Execution

- **exec (*b* : basic block, *s* : abstract pipeline state) *t*: trace**
  - Interprets instruction stream of *b* (annotated with cache information) starting in state *s* producing trace *t*
  - *length(t)* gives number of cycles
- **What is abstracted?**
  - Abstract states may lack information, e.g. about cache contents
  - Assume local worst cases is safe (in the case of no timing anomalies)
  - Traces may be longer (but never shorter) than in reality

Source: L. Thiele

Context

- ***Starting state* for basic block? In particular, if there are several predecessor blocks?**
- **Solutions**
  - Sets of states
  - Combine by assuming that local worst case is safe



Source: L. Thiele

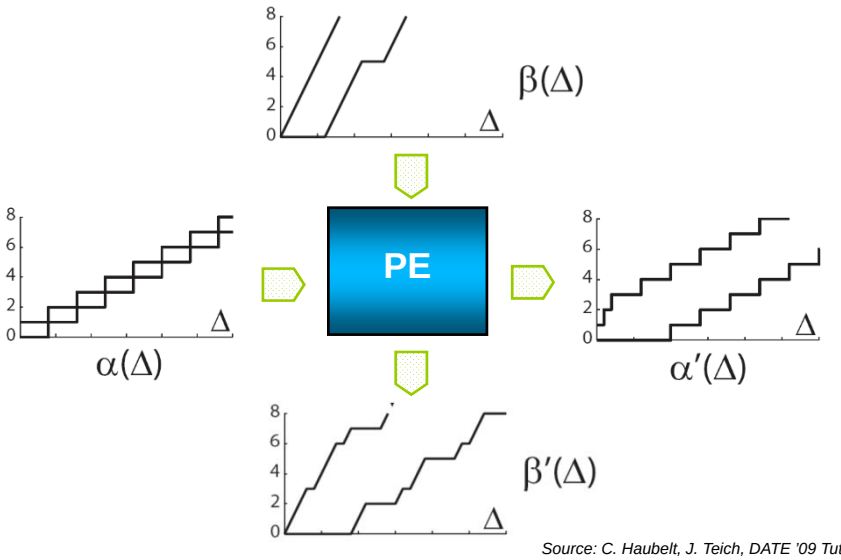
Summary of WCET Analysis Steps

- ***Value analysis***
- ***Cache analysis* using statically computed effective addresses and loop bounds**
- ***Pipeline analysis***
  - Assume cache hits where predicted,
  - Assume cache misses where predicted or not excluded.
  - Only the "worst" result states of an instruction need to be considered as input states for successor instructions!
- ***Path analysis***
  - Compute final WCET estimate

Source: L. Thiele

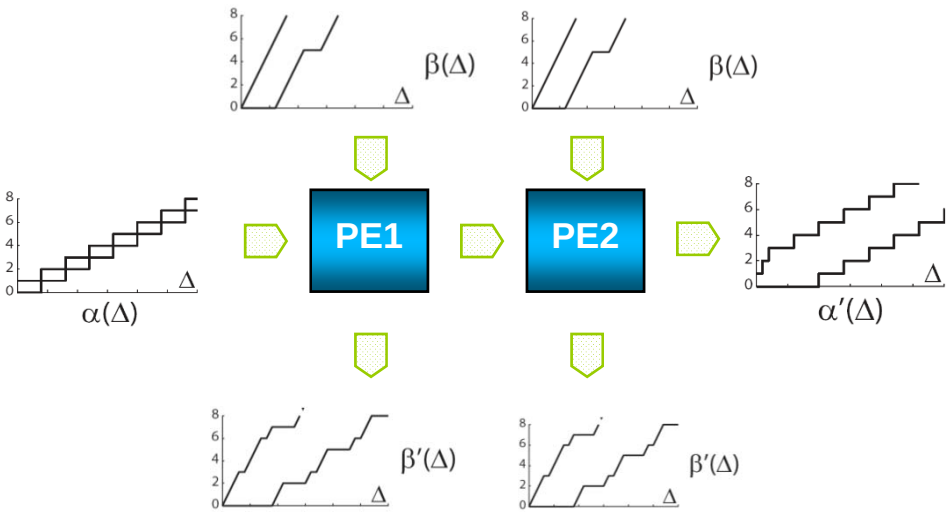
System-Level Analysis

- **Modular Performance Analysis (MPA)**
  - Real-time calculus (RTC)



Source: C. Haubelt, J. Teich, DATE '09 Tutorial

MPSoC Analysis with MPA



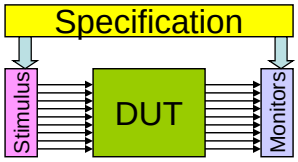
Source: C. Haubelt, J. Teich, DATE '09 Tutorial

Lecture 11: Outline

- Evaluation and estimation
  - Methods
- Analysis
  - Component- and system-level estimation
- **Simulation**
  - Simulation methods
- **Hybrid approaches**
  - Semi-analytical methods
  - Machine learning-based methods

Simulation

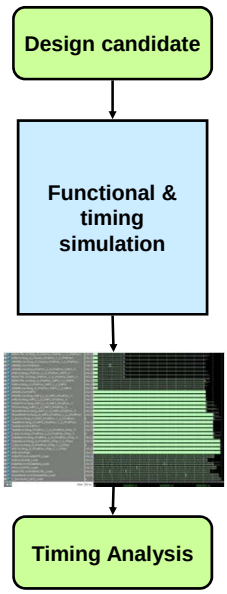
- **Create stimuli and simulate model**



- **Inputs**
  - Specification
    - Used to create interesting stimuli and monitors (golden output)
  - Model of DUT
    - Typically written in HDL or C or both
- **Output**
  - Failed test vectors (validation)
  - Quality metrics (evaluation)
- **Speed vs. accuracy**
  - Fundamental tradeoff

Simulation Methods

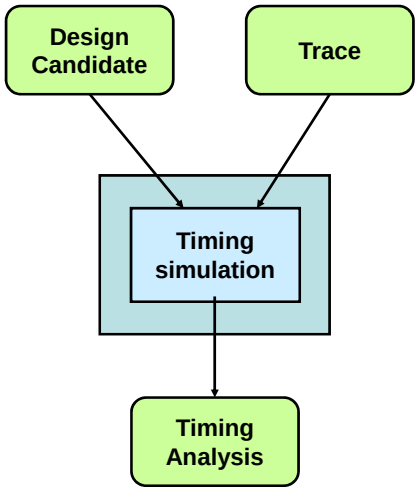
- **Component models**
  - Functional model
  - Timing, energy, … models
- **Co-simulation**
  - System description language & model
  - Generates a trace
- **Simulation trace analysis**
  - Check functional results
  - Extract metrics from trace
  - Latency, throughput, etc.



Source: C. Haubelt, J. Teich

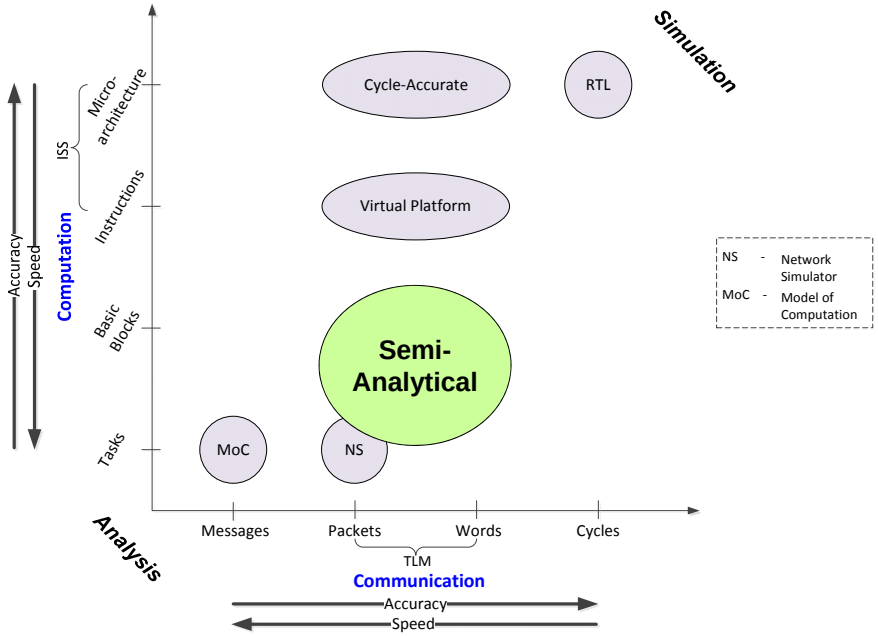
Trace-Driven Simulation

- **Drive simulation via pre-existing, static traces**
  - Traces for system block behavior
  - Traces obtained from fast functional-only simulation
- **Examples**
  - Trace-driven simulation
  - Arrival curve extraction from traces
  - Trace generation from arrival curves

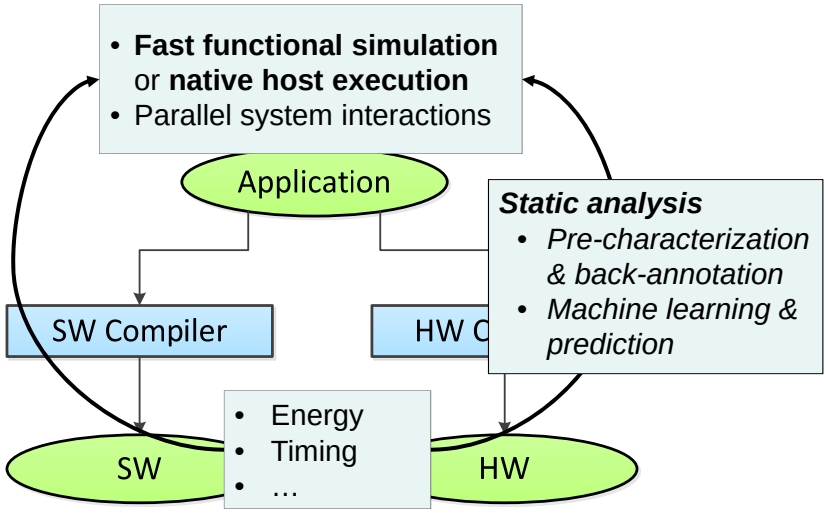


Source: C. Haubelt, J. Teich, DATE '09 Tutorial

System-Level Modeling Space

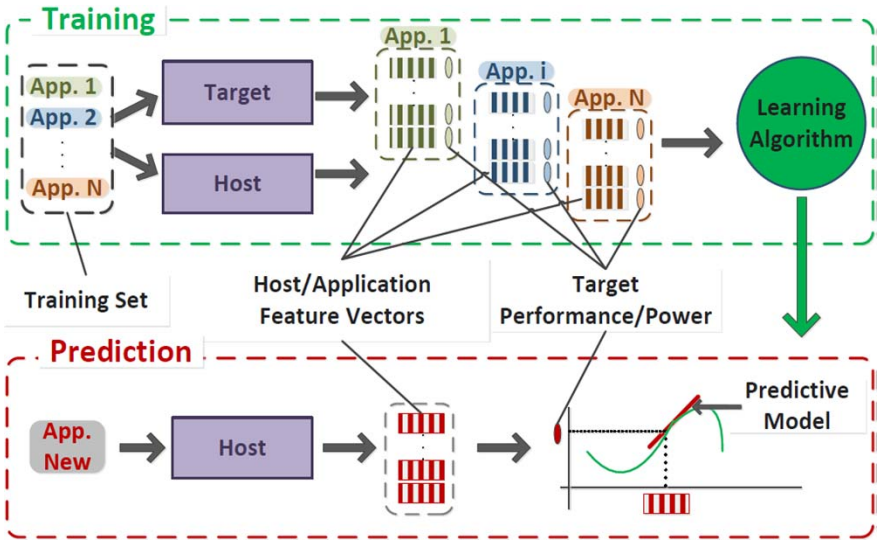


Semi-Analytical Modeling



Learning-Based Methods

- **Learning-based prediction**



Source: X. Zheng, L. John, A. Gerstlauer. "LACross: Learning-Based Analytical Cross-Platform Performance and Power Prediction," IJPP, 2017.

Software Models

- **Predict on target CPU while running on host CPU**
  - Using hardware counters on host as features
  - Predict target performance and power
  - At program phase level

➢ **Instrumentation-based**
  - Compiler-based instrumentation at basic block granularity
  - Collect features and train/call model every *N* basic blocks

➢ **Sampling-based**
  - Source-oblivious at binary level using timer interrupts
  - Sample alignment during training

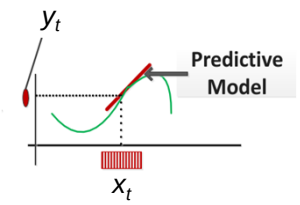
Learning Formulation

- **Given training set (x_i, y_i)**
  - $x_i \in \mathbb{R}^d$: d-dimensional counter feature vector from host
  - $y_i \in \mathbb{R}$: reference performance/power on target
- **Want to find function $F(x_i) \approx y_i$**
  - Fundamentally non-linear

➢ **Locally linear approximation $F_t(x_t)$ at input x_t**

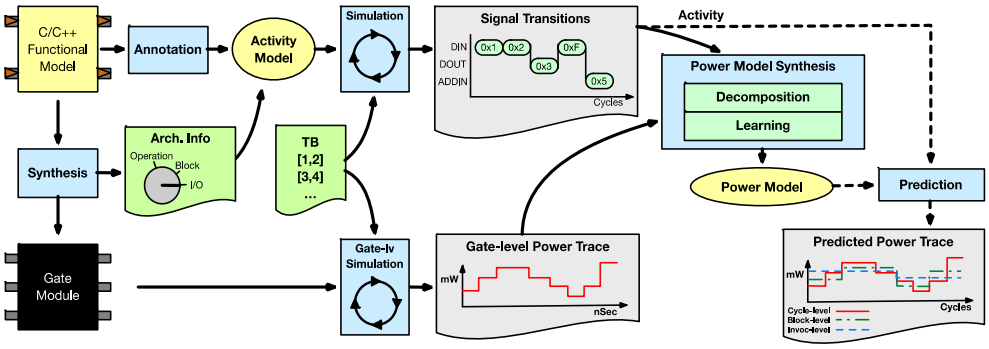
$$F_t(x_t) = \theta_t^T x_t$$

  - Around neighborhood of x_t
  - LASSO regression to solve for θ_t



Learning-Based Hardware Models

- **Learning-based IP power modeling**
  - White / grey / black box models
  - Operation / block / I/O activity from functional simulation
  - Predict gate-level trace at cycle / block / invocation granul.

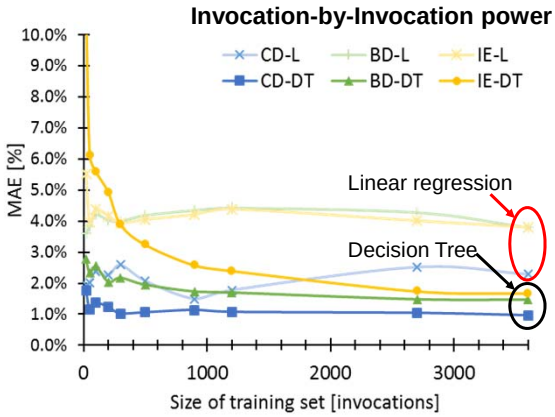


➢ **Data-dependent, Fast**

Source: D. Lee, A. Gerstlauer, "Learning-Based, Fine-Grain Power Modeling of System-Level Hardware IPs," ACM TODAES, 23(3), 2018..

Learning Formulation

- **Dedicated, domain-specific learning formulations**
  - Structural model decomposition & feature selection
- **Advanced, non-linear regression models**
  - Traditional, not deep learning w/ small training size



- L : Linear regression
- DT: Decision Tree regression

- CD : Cycle decomposed model
- BD : Block decomposed model
- IE : Invocation ensemble model

Lecture 11: Summary

- **Static analysis**
  - Worst/best/average case bounds
  - Execution time analysis of single task
  - Real-time calculus for concurrent systems
- **Simulation**
  - Detailed system simulation
  - Trace-based simulation
- **Hybrid approaches**
  - Semi-analytical modeling
  - Machine learning-based prediction