ECE382M.20: System-on-a-Chip (SoC) Design

Lecture 4 – SoC Performance Analysis

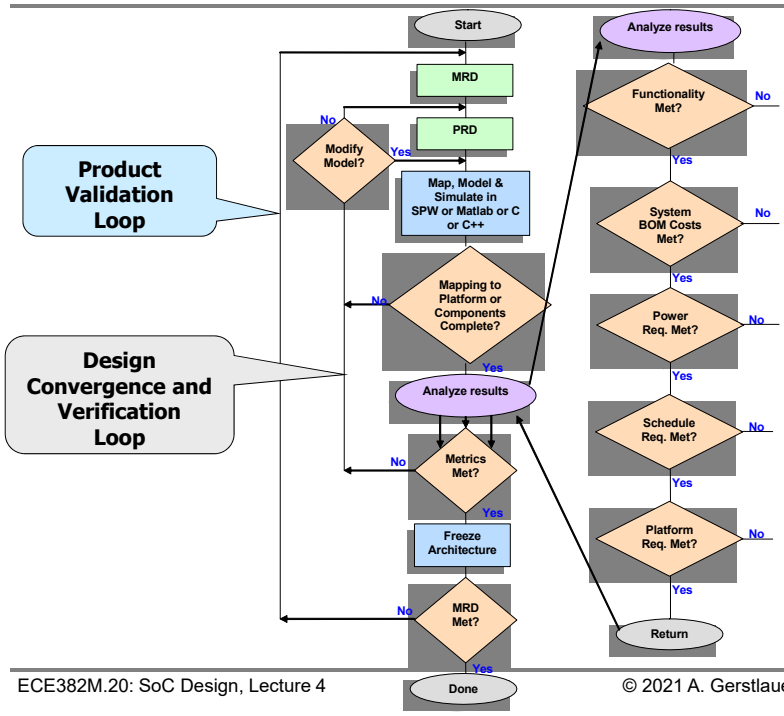
Sources:
Prof. Jacob Abraham, UT Austin
Prof. Lothar Thiele, ETH Zurich
Prof. Reinhard Wilhelm, Saarland Univ.

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

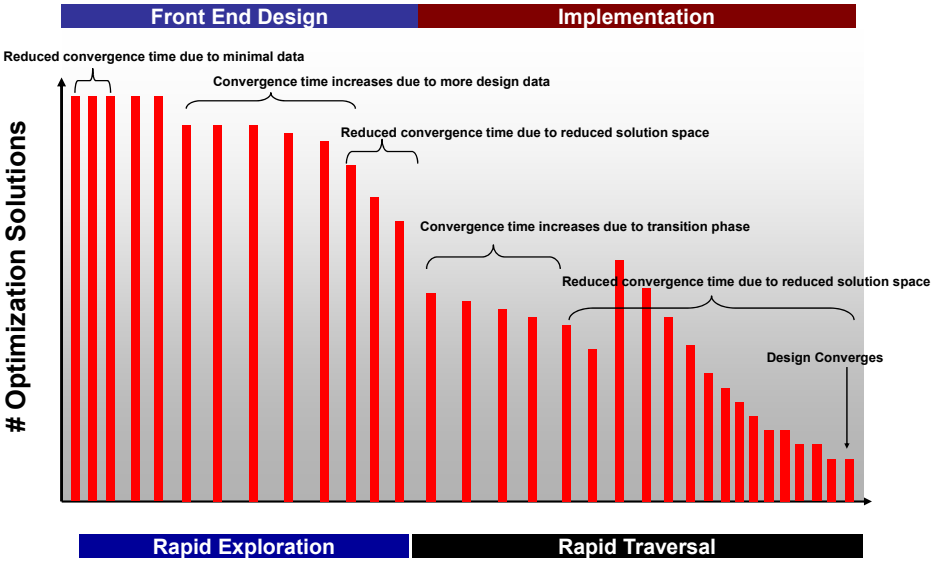
Lecture 4: Outline

- **SoC design flow**
  - Iterative design convergence
- **Requirement & performance analysis**
  - Algorithm- and application-level analysis
  - System-level prototyping
  - Component-level estimation

SoC Design Flow



Design Convergence

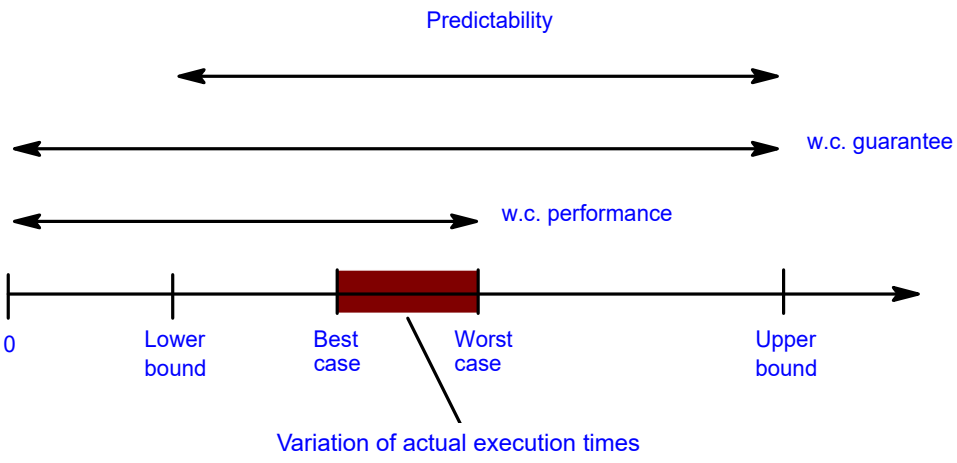


Performance of a System

- **Depends on many factors**
  - System design (algorithms and data structures)
  - Implementation (code)
  - Execution platform architecture
  - The workload to which it is subjected
  - The metric used in the evaluation

➢ **Interactions between these factors**

Performance Analysis



➢ **Simulation (dynamic) vs. analysis (static)**
  - Tightness of bounds, under- vs. over-estimation

Evaluating a Design

- **Algorithm and application level**
  - Analysis of complexity
  - Identify bottlenecks, evaluate tradeoffs
- **System level**
  - Virtual platform prototyping
  - Physical prototyping
- **Component level**
  - Software partition
    - Profiling, measurement
    - Simulation, tracing
    - Power/performance/… estimation
  - Hardware partition
    - Modeling & estimation

Algorithm-Level Analysis

- **Theoretical complexity analysis**
  - *O*() notation ("order-of")
  - Example: Sorting
    - "Bubble" sort
    - Merge sort
  - Example: Fourier transform
    - Discrete Fourier transform
    - Fast Fourier transform
- **Analysis of code**
  - Example: BCH* encoding
    - Code in C
    - Find the number of XOR and AND operations performed in the loop as a function of *k*
    - Assume *length* is 1024, and in any bit position, 0 and 1 are equally likely

** A BCH code is a multilevel, cyclic, error-correcting, variable-length digital code used to correct multiple random error patterns. BCH codes may also be used with multilevel phase-shift keying whenever the number of levels is a prime number or a power of a prime number.*

BCH Code

```
encode_bch()
{
register int     i, j;
register int     feedback;
for (i = 0; i < length - k; i++)
     bb[i] = 0;
for (i = k - 1; i >= 0; i--) {
     feedback = data[i] ^ bb[length - k - 1];
     if (feedback != 0) {
          for (j = length - k - 1; j > 0; j--)
               if (g[j] != 0)
                    bb[j] = bb[j - 1] ^ feedback;
               else
                    bb[j] = bb[j - 1];
          bb[0] = g[0] && feedback;
     } else {
          for (j = length - k - 1; j > 0; j--)
               bb[j] = bb[j - 1];
          bb[0] = 0;
     }
}
}
```


Application-Level Profiling

- **Execute code on physical or simulated machine**
  - (Cross-)Compile down to binary
  - Run on real processor or instruction-set simulator
    - Benchmarks and input data test vectors
- **Instrument code and collect metrics at runtime**
  - Include effect of processor instruction set and architecture
    - For the given runtime platform (not necessarily the intended target)
  - Many profiling tools for data gathering and analysis
    - *gprof*, etc.
  - Various interfaces, levels of automation, and approaches to information presentation
  - A lot of work in the high performance computing community

➢ Effect of instrumentation on measured results?

Instrumentation Techniques

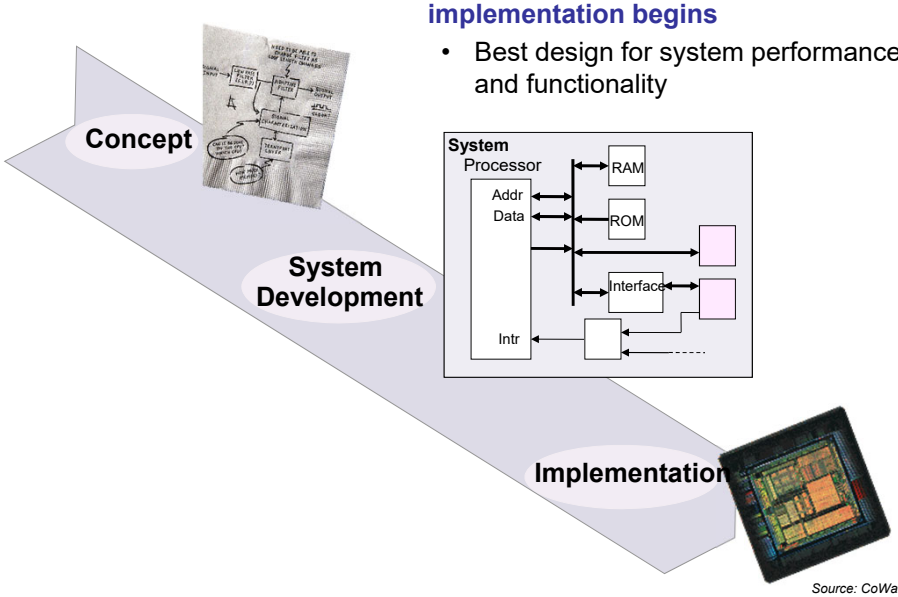
- **Program instrumentation techniques**
  - Manual: Programmer inserted directives
  - Automatic: No direct user involvement
    - Sampling [gprof]
    - Binary rewriting [PIN]
    - Dynamic Instrumentation
- **Hardware instrumentation techniques**
  - Timers, memory system performance, processor usage, etc.
  - Available mostly through special registers or memory mapped location
    - Example: Pentium Pro MSRs (model (machine)-specific registers)
      - » Counters for clock cycles, memory read/write, L1 cache misses, pipeline flushes, etc.
  - Hardware assisted trace generation
- **Operating system instrumentation techniques**
  - Behavior of virtual memory, file system, file cache, etc.
  - Access via APIs
- **Network instrumentation techniques**
  - Passive, e.g. RMON protocol packet header fields for monitoring
  - Active, e.g. ping, NWS in grid style computing.

Darknet Profiling

- **PC-based reference code**
  - Designed to run in a Linux environment, but in this course, we are going to make it run on an embedded processor (ARM A53)
  - ➢ Profile in both physical (board) and simulated ARM env.
    - Measure performance and identify bottlenecks
- **Board profiling (Lab 1)**
  - `gprof` measures where program spends its time and which functions call other functions while it was executing
    - Flat profile: total amount of time program spends executing each function
    - Call graph: how much time was spent in each function and its children
- **Virtual platform profiling (Lab 2)**
  - Run `gprof` in virtual platform prototype
    - Flat and call graph profiles
    - Timing accuracy of virtual platform? (QEMU timing model!)

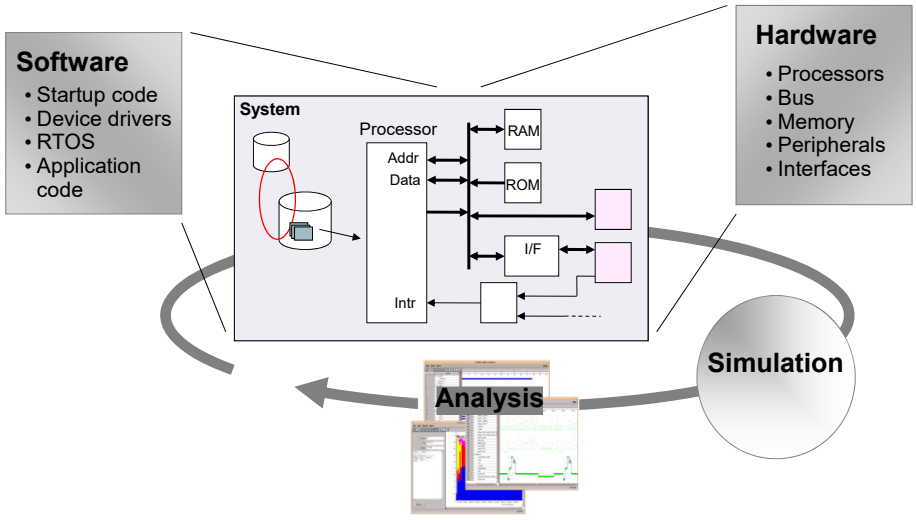
System-Level Development

- **Get the architecture right before implementation begins**
  - Best design for system performance and functionality



Source: CoWare, Inc.

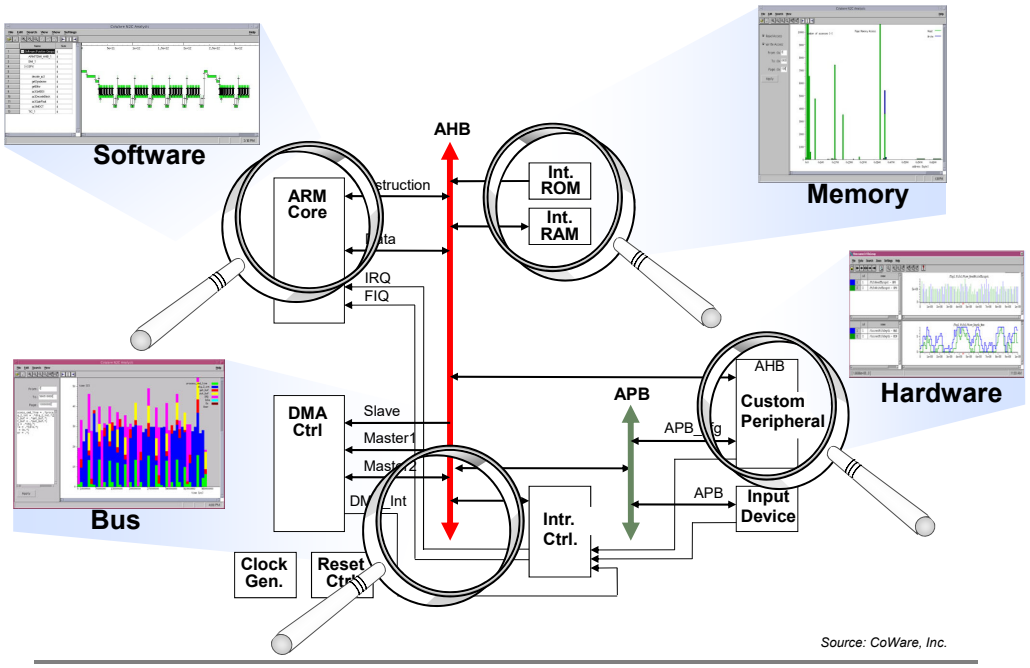
Virtual Platforms



- **Virtual prototyping**
  - Component-level simulation in transaction-level model (TLM)

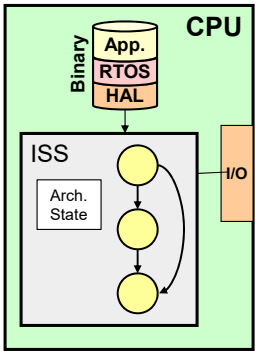
Source: CoWare, Inc.

Architecture Analysis



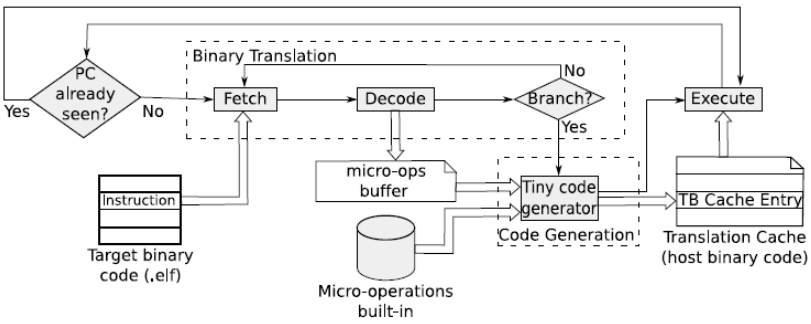
Component-Level Simulation

- **Cycle-accurate**
  - Describe micro-architecture/RTL in C
- **Interpreted instruction-set simulation**
  - Cycle-accurate/-approximate
  - Functionality/behavior only
- **Compiled ISS**
  - Binary translation
  - Offline vs. just-in-time (self-modifying code)
  - Functional only (no/rough timing, e.g. CPI=1)
- **Source-level/host-compiled simulation**
  - C model describing functionality/behavior
  - Back-annotate with timing and other metrics



Quick EMUlator (QEMU)

- **Open-source, binary-translating ISS [Bellard'05]**
  - Emulates a variety of architectures (x86, ARM, PowerPC)
  - Stand-alone or full-system model (peripherals to boot OS)



Source: M. Gilgor, N. Fournel, F. Pétrot, "Using Binary Translation in Event Driven Simulation for Fast and Flexible MPSoC Simulation," CODES+ISSS'09

Component-Level Estimation

- **Static analysis and prediction of performance**
  - Need for hard guarantees in embedded space
    - Reliability, safety, etc.
- ➢ **Worst-Case Execution Time (WCET) estimation**
  - ➢ Tightness of bounds?
- **Analysis of modern processors is very difficult**
  - Dynamic effects
    - Pipeline introduces dependencies on input sequence (instructions)
    - Cache effects
    - Branch prediction
  - Hazards lead to timing accidents & penalties
    - Structural (resource being used by another)
    - Data (dependence for data calculations)
    - Control (calculating next address – branches, interrupts)

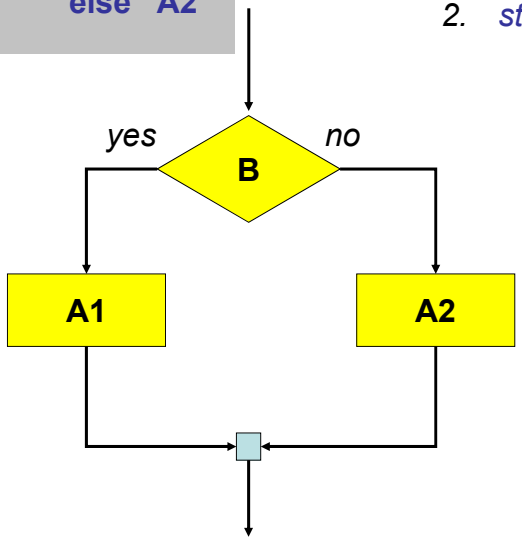
Overall WCET Approach

- **Compute upper bounds along the program structure**
  - Programs are hierarchically structured
  - Compute the upper bound for a construct from the upper bounds of its constituents
  - Basic blocks of code at the leaves of the hierarchy
- ***Micro-architecture analysis***
  - Abstract interpretation of code in basic blocks
  - Determines WCET for each basic block (in contexts)
  - Exclude as many timing accidents as possible
- ***Worst-case path determination***
  - Control flow graph (CFG) of basic blocks
  - Map to an integer linear program
  - Determines upper bound and associated path

Conditional Statements

A ≡ if B
then A1
else A2

Constituents of A:
1. *condition* B
2. *statements* A1 *and* A2



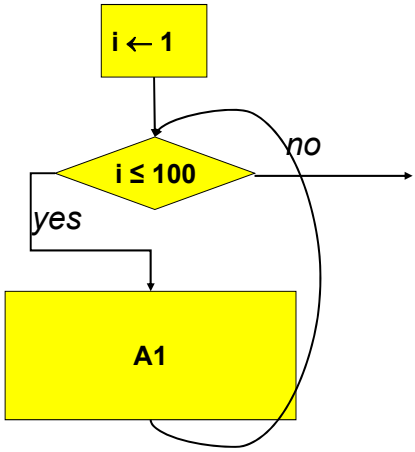
wcet(A) =

wcet(B) +

max(wcet(A1),
wcet(A2))

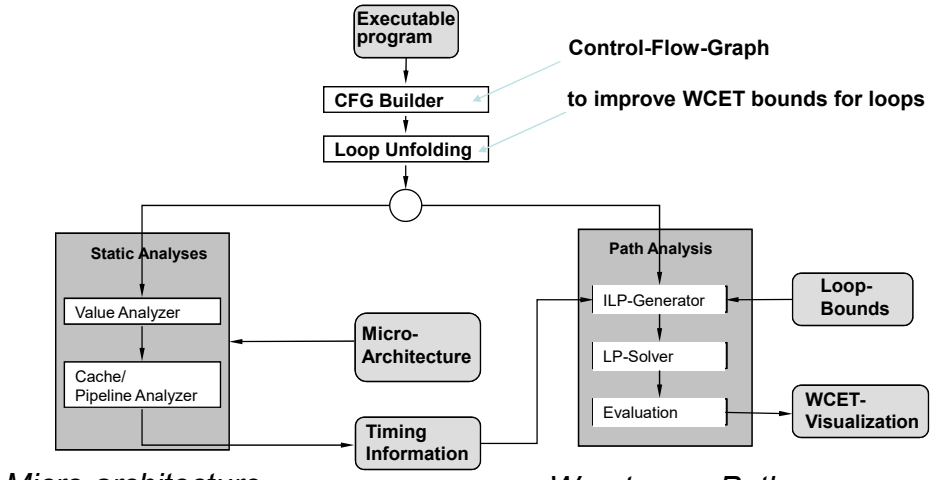
Loops

A ≡ for i ← 1 to 100 do A1



wcet(A) =
wcet(i ← 1) +
100 × (wcet(i ≤ 100) +
wcet(A1)) +
wcet(i ≤ 100)

aiT Tool



Micro-architecture Analysis

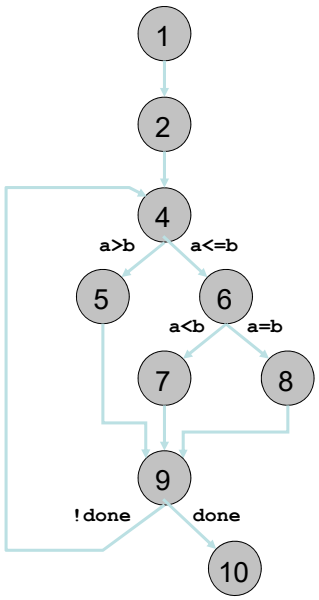
Worst-case Path Determination

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.

Control Flow Graph (CFG)

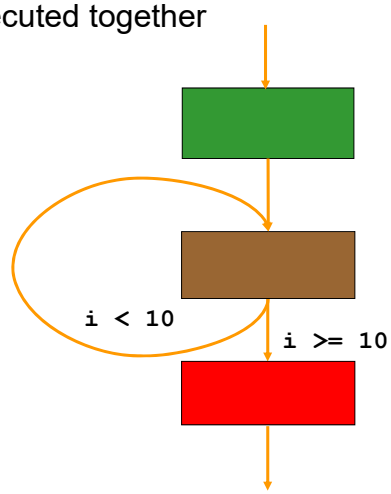
```
   what_is_this {
1     read (a,b);
2     done = FALSE;
3     repeat {
4       if (a>b)
5          a = a-b;
6       elseif (b>a)
7          b = b-a;
8       else done = TRUE;
9     } until done;
10    write (a);
   }
```



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
```



Calculation of the WCET

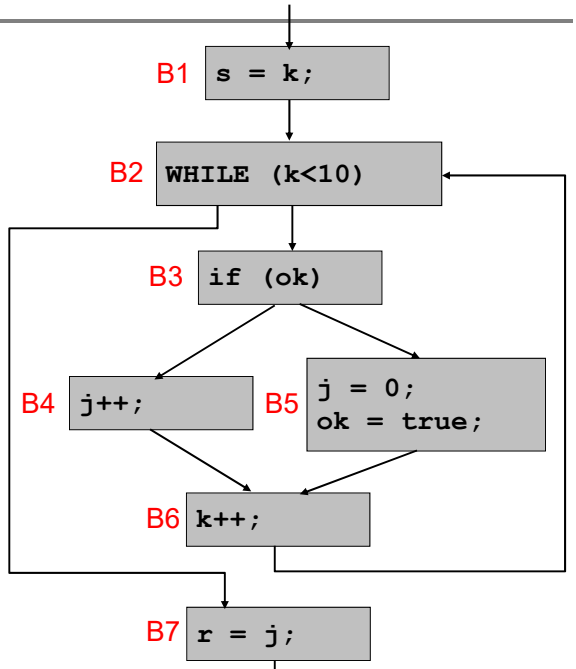
- ***Definition:*** **A program consists of N basic blocks, where each basic block B_i has a worst-case execution time c_i and is executed for exactly x_i times. Then, the WCET is given by**

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

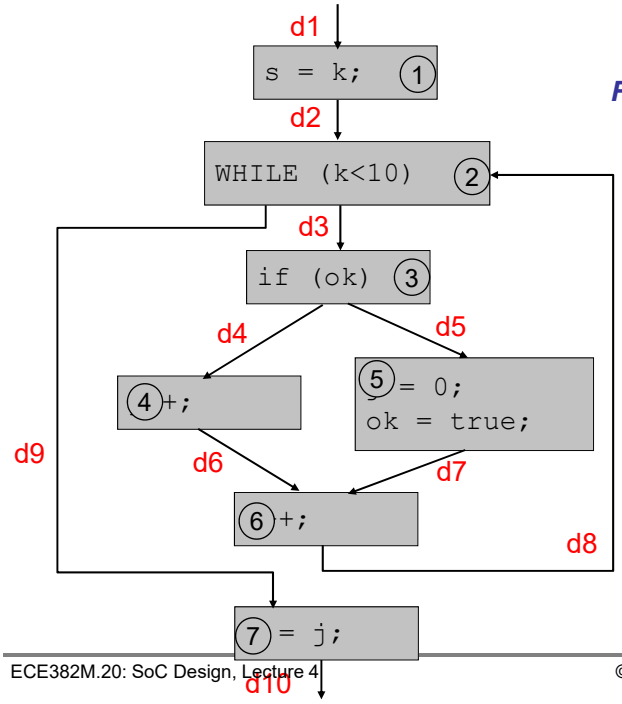
  - The c_i values are determined using static analysis.
  - How to determine x_i?
    - Structural constraints given by the program structure
    - Additional constraints provided by the programmer (bounds for loop counters, etc.; based on knowledge of the program context)

Example

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



Structural Constraints



Flow equations:

$d1 = d2 = x_1$

$d2 + d8 = d3 + d9 = x_2$

$d3 = d4 + d5 = x_3$

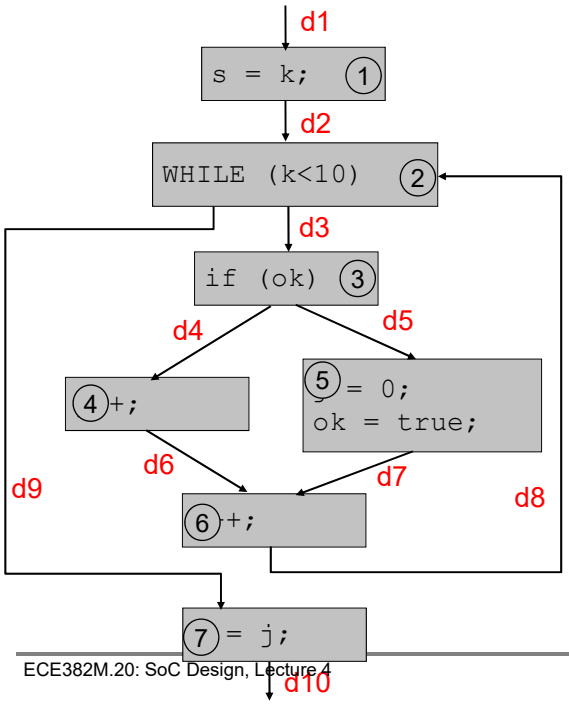
$d4 = d6 = x_4$

$d5 = d7 = x_5$

$d6 + d7 + d8 = x_6$

$d9 = d10 = x_7$

Additional Constraints



Loop is executed for at most 10 times:

$x_3 <= 10 \cdot x_1$

B5 is executed for at most one time:

$x_5 <= 1 \cdot x_1$

Integer Linear Program (ILP)

- **ILP with structural and additional constraints**

program is executed once

$$WCET = \max\left\{ \sum_{i=1}^{N} c_i \cdot x_i \;\middle|\; d_1 = 1 \wedge \right.$$

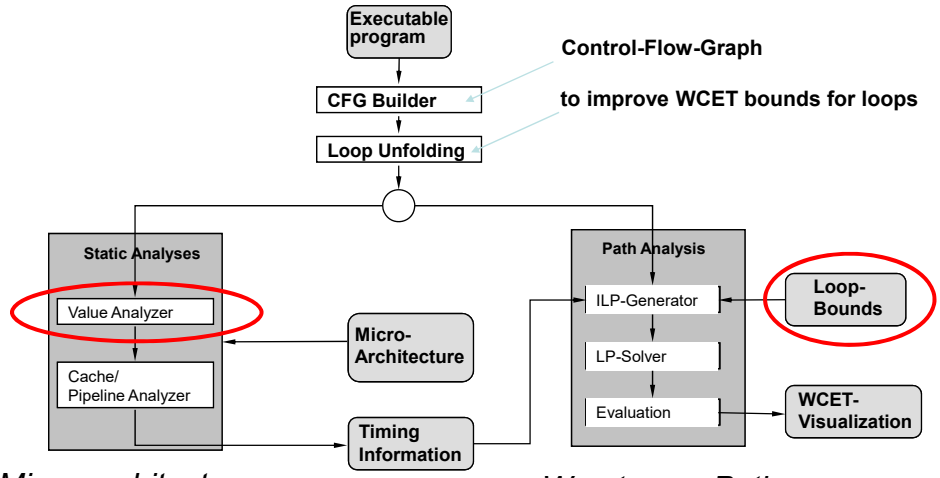
$$\sum_{j \in in(B_i)} d_j = \sum_{k \in out(B_i)} d_k = x_i, i = 1...N \wedge$$

structural constraints

$$\left. ... \right\}$$

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

Overview

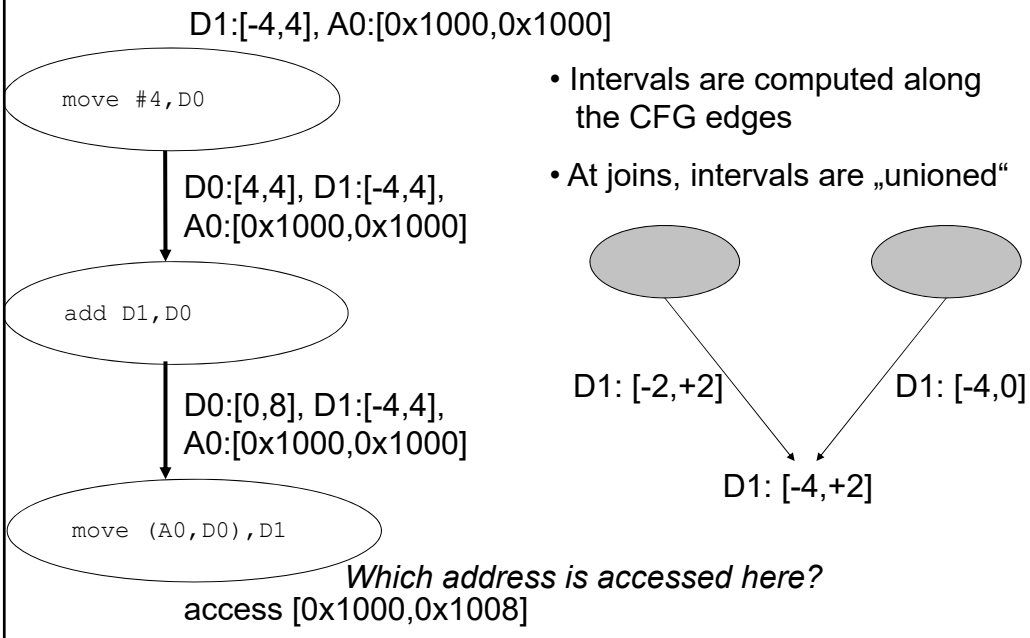


Value Analysis

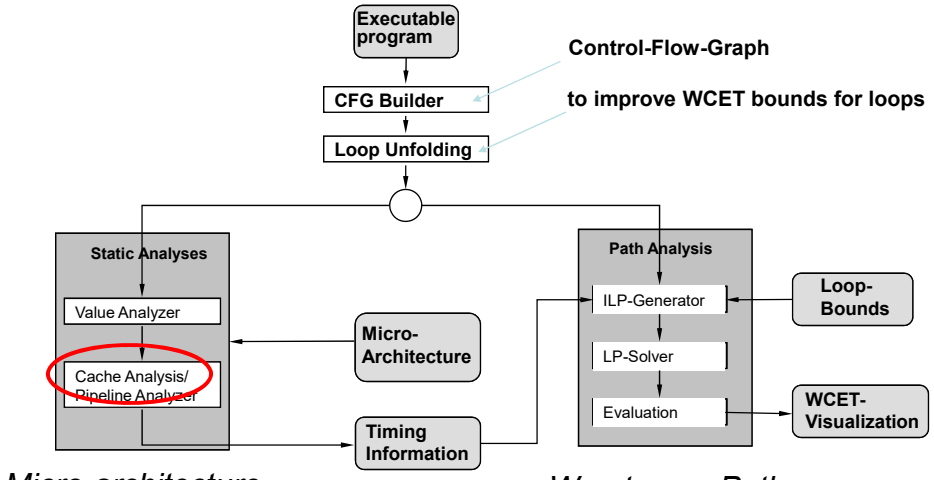
- ***Motivation***:
  - Provide access information to data-cache/pipeline analysis
  - Detect infeasible paths
  - Derive loop bounds
- ***Method*: calculate intervals at all program points, i.e. lower and upper bounds for the set of possible values occurring in the machine program (addresses, register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

Value Analysis



Overview



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.**

Static Cache Analysis

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

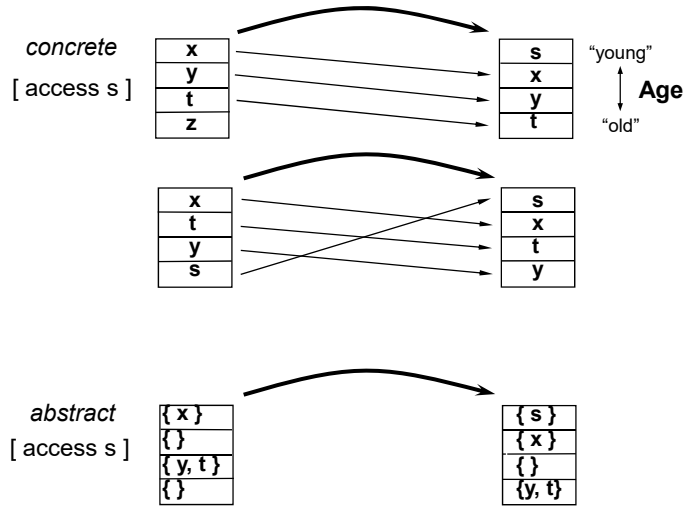
➢ ***Must Analysis***

- 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.

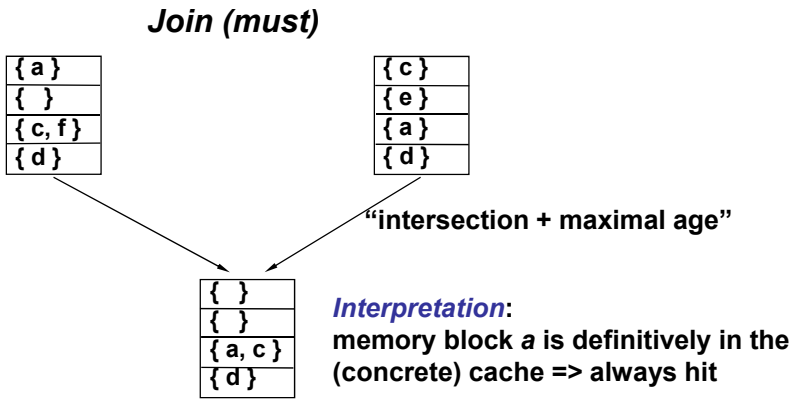
➢ ***May Analysis***

- 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.

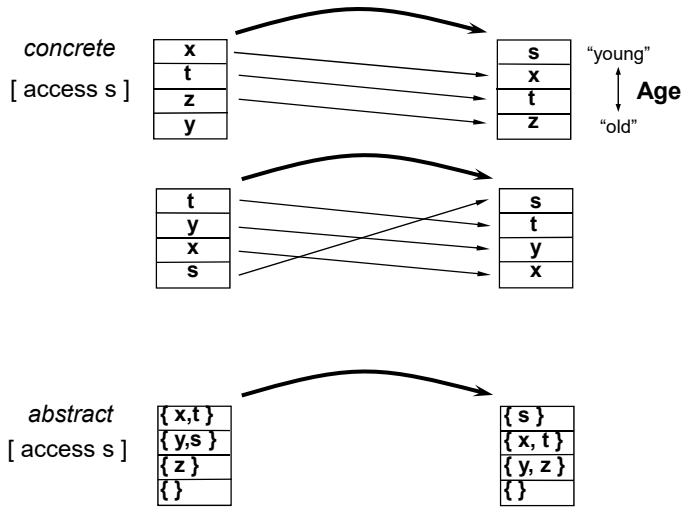
Cache with LRU: Must Analysis



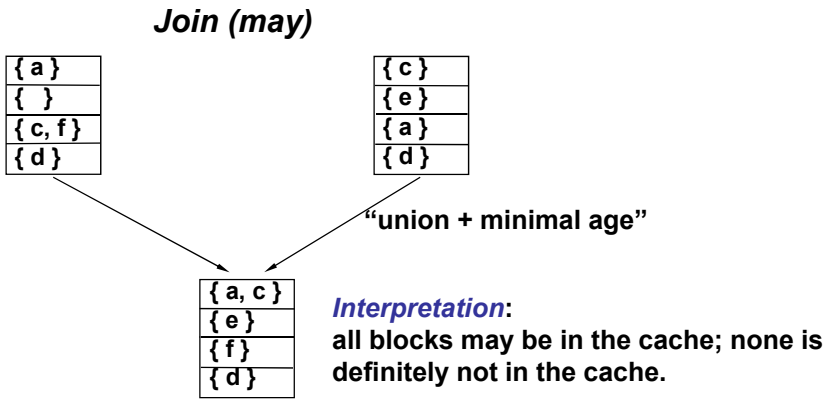
Must Analysis: Join



Cache with LRU: May Analysis

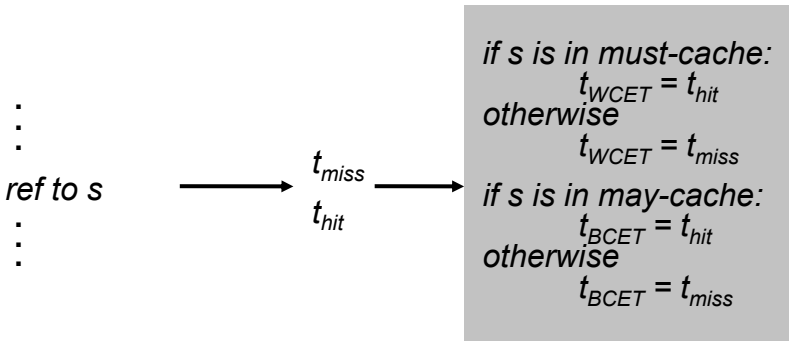


May Analysis: Join



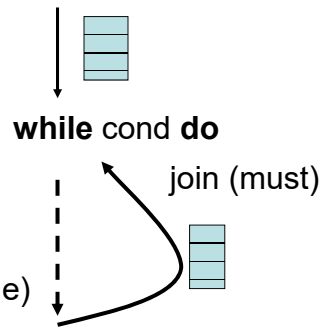
Cache Analysis for WCET

- **Information about cache contents sharpens timings**

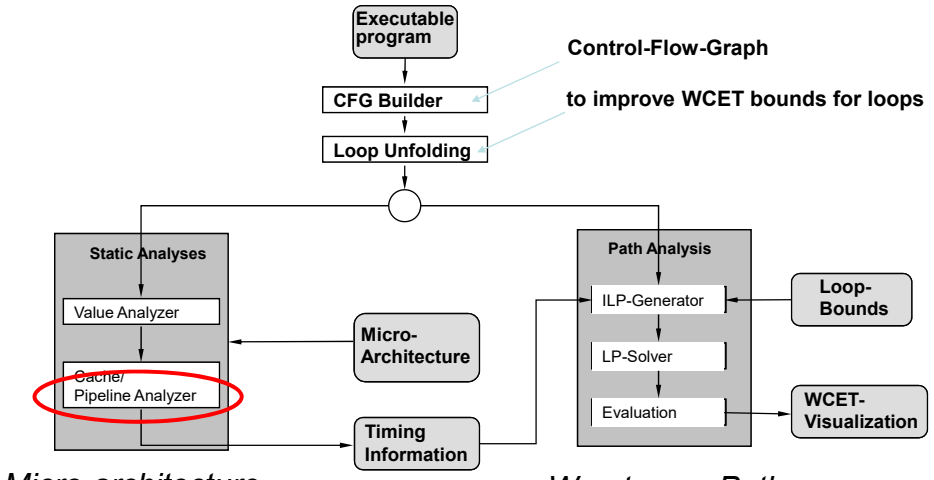


Cache Analysis Contexts

- **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)



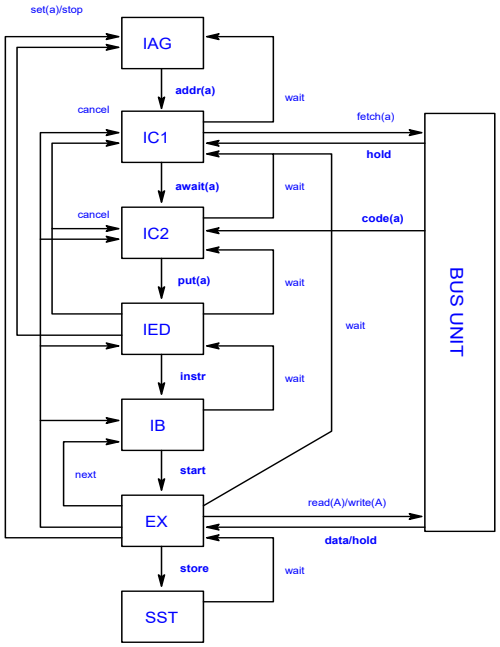
Overview



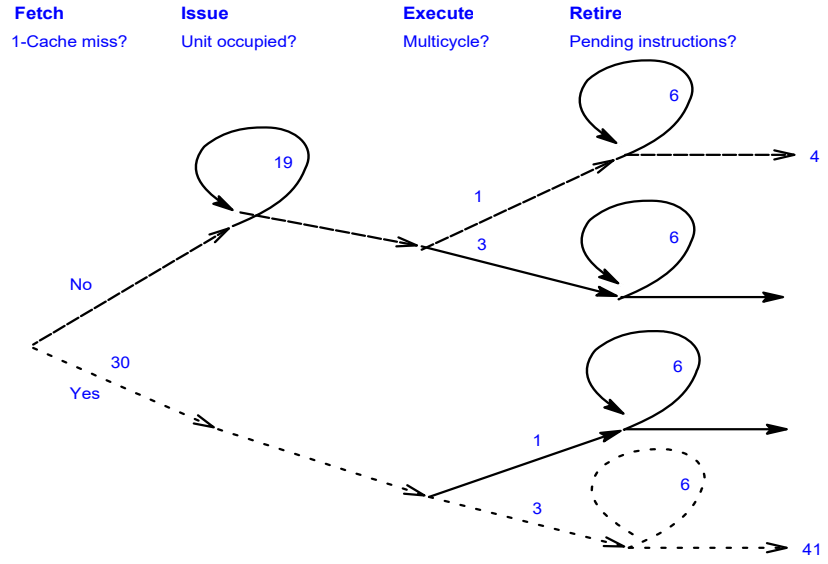
Micro-architecture Analysis

Worst-case Path Determination

Motorola ColdFire 5307 Pipeline



Execution of Multiply Instruction



Static Analysis of Hazards

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

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

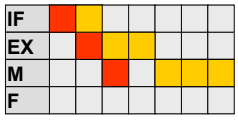
Hit

***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**

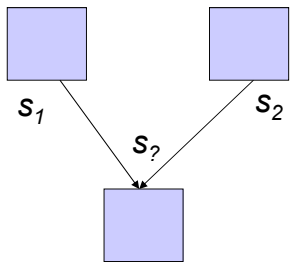


Abstract Pipeline Execution

- **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

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



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

Lecture 4: Summary

- ✓ **SoC design flow**
  - ✓ Iterative design convergence
- **Requirement & performance analysis**
  - ✓ Algorithm- and application-level analysis
  - ✓ System-level prototyping
  - ✓ Component-level simulation or estimation