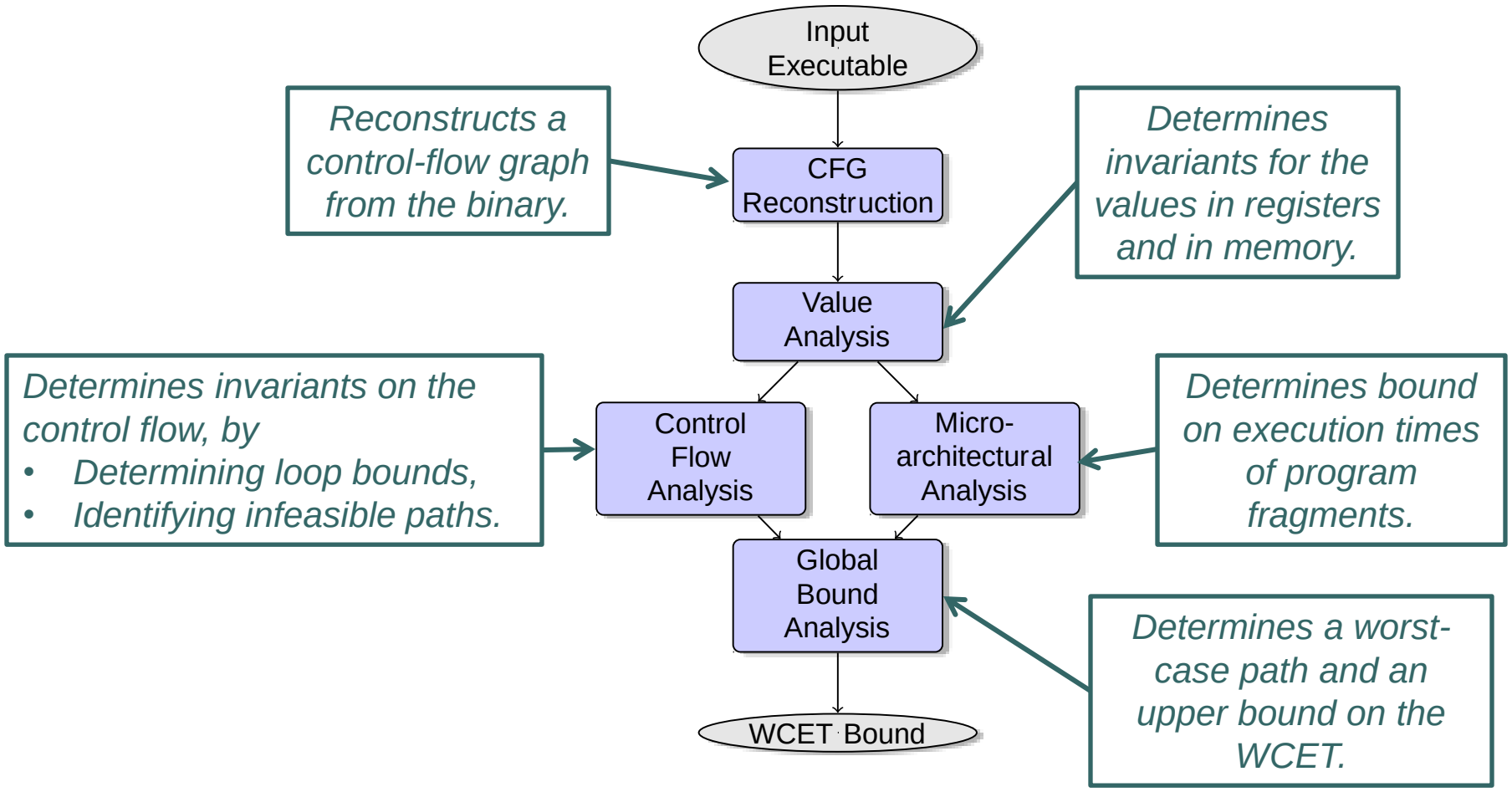
Verification of Real-Time Systems

Microarchitectural Analysis

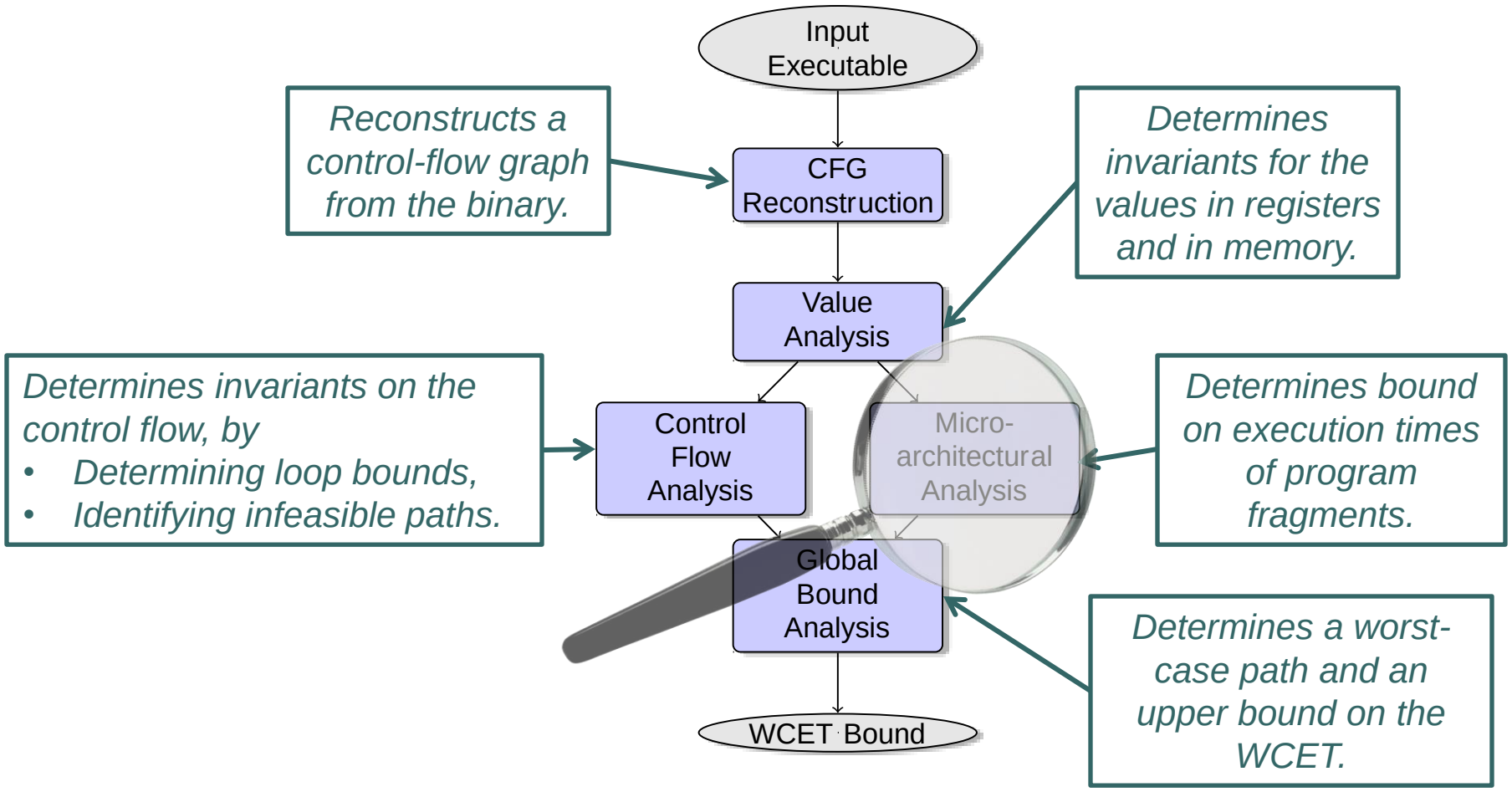
Jan Reineke

Advanced Lecture, Summer 2015

Structure of WCET Analyzers



Structure of WCET Analyzers



Microarchitectural Analysis

Ideal 1970s world: one instruction = one cycle

Real world:

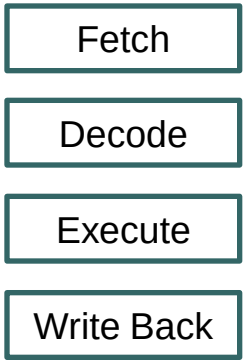
- Pipelining
- Branch prediction + speculative execution
- Caches
- DRAM

➔ Execution time of individual instruction highly variable and dependent on state of microarchitecture

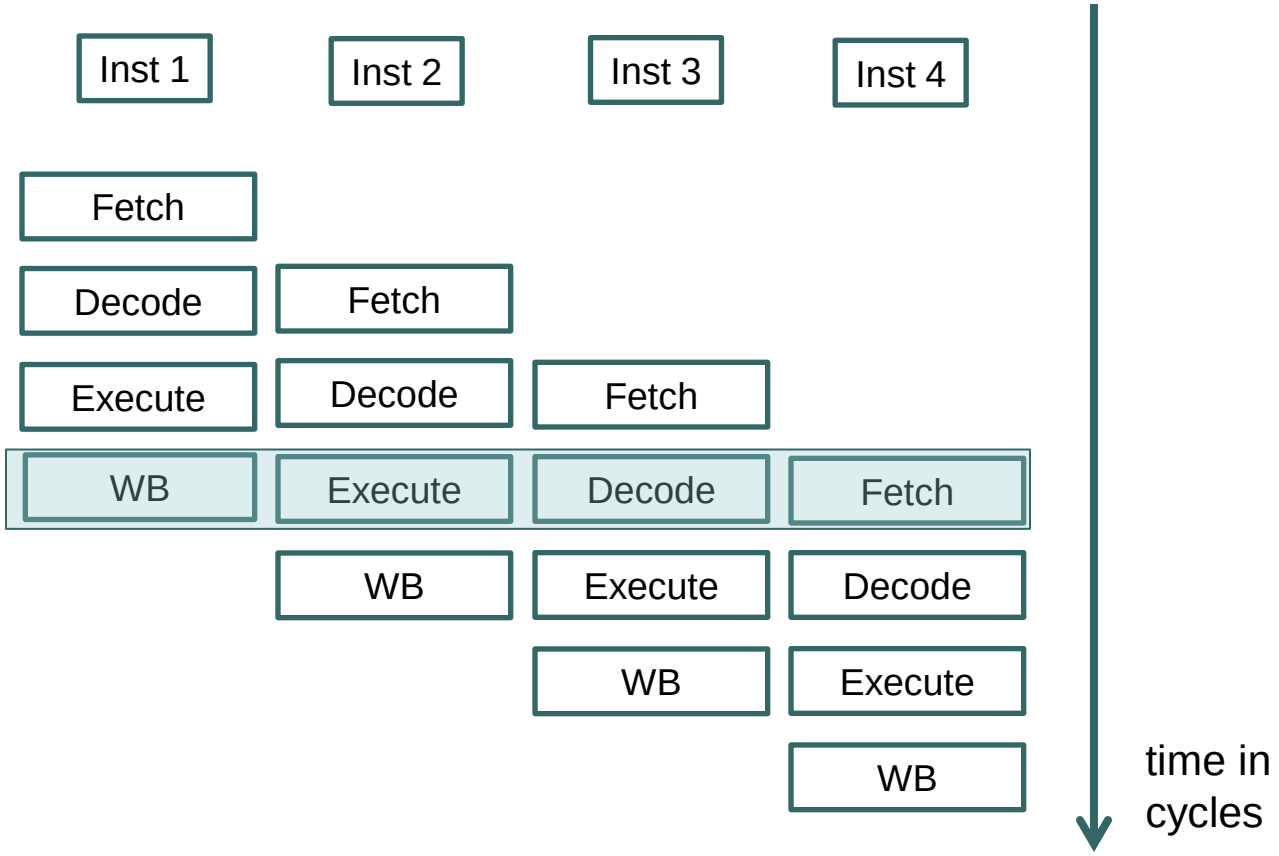
➔ Need to determine in which states the microarchitecture may be at a point in the program

Hardware Features: Pipelining

- Instruction execution is split into several stages
- Several instructions can be executed in parallel
- Some pipelines can start more than one instruction per cycle: VLIW, Superscalar
- Some processors can execute instructions out-of-order
- Practical Problems: Hazards and cache misses



Hardware Features: Pipelining



Ideal Case: One Instruction per Cycle

Pipeline Hazards

Pipeline Hazards:

- Data Hazards: Operands not yet available (Data Dependences)
- Resource Hazards: Consecutive instructions use same resource
- Control Hazards: Conditional branch
- Instruction-Cache Hazards: Instruction fetch causes cache miss

Assuming worst case everywhere is not an option!

Static exclusion of hazards

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

lwz r4, 20(r1)

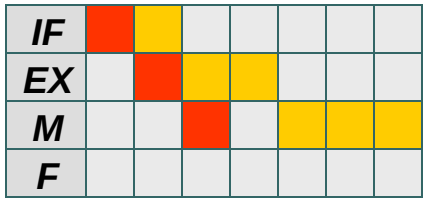
Hit

Dependence analysis: elimination of data hazards

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

Operand ready

Resource reservation tables: elimination of resource hazards



Microarchitectural Analysis as Abstract Interpretation

- Ingredients of an Abstract Interpretation
  - Concrete Semantics that captures property of interest
  - Abstract Semantics + Relation to Concrete Semantics
- Thus, wanted:
  - Concrete Semantics that captures execution time of basic blocks
  - Abstraction of this Concrete Semantics

View of Processor as a State Machine

- Processor (pipeline, cache, memory, inputs) viewed as a *big* state machine, performing transitions every clock cycle
- Starting in an initial state s_0, transitions are performed, until a final state is reached, producing a trace $(s_0, \ldots, s_n)$ of states:
  - Final state s_n: program terminated
  - # transitions = n = execution time of program
- Can split execution into subsequences corresponding to basic blocks
  - Execution time of a basic block **b**

    = length of subtrace executing **b**

A Concrete Pipeline Executing a Basic Block

function *exec* (*b* : **basic block**, *s* : **concrete pipeline state**)

t : **trace**

Interprets instruction stream of *b* starting in state *s* producing trace *t.*

Successor basic block is interpreted starting in initial state *last(t).*

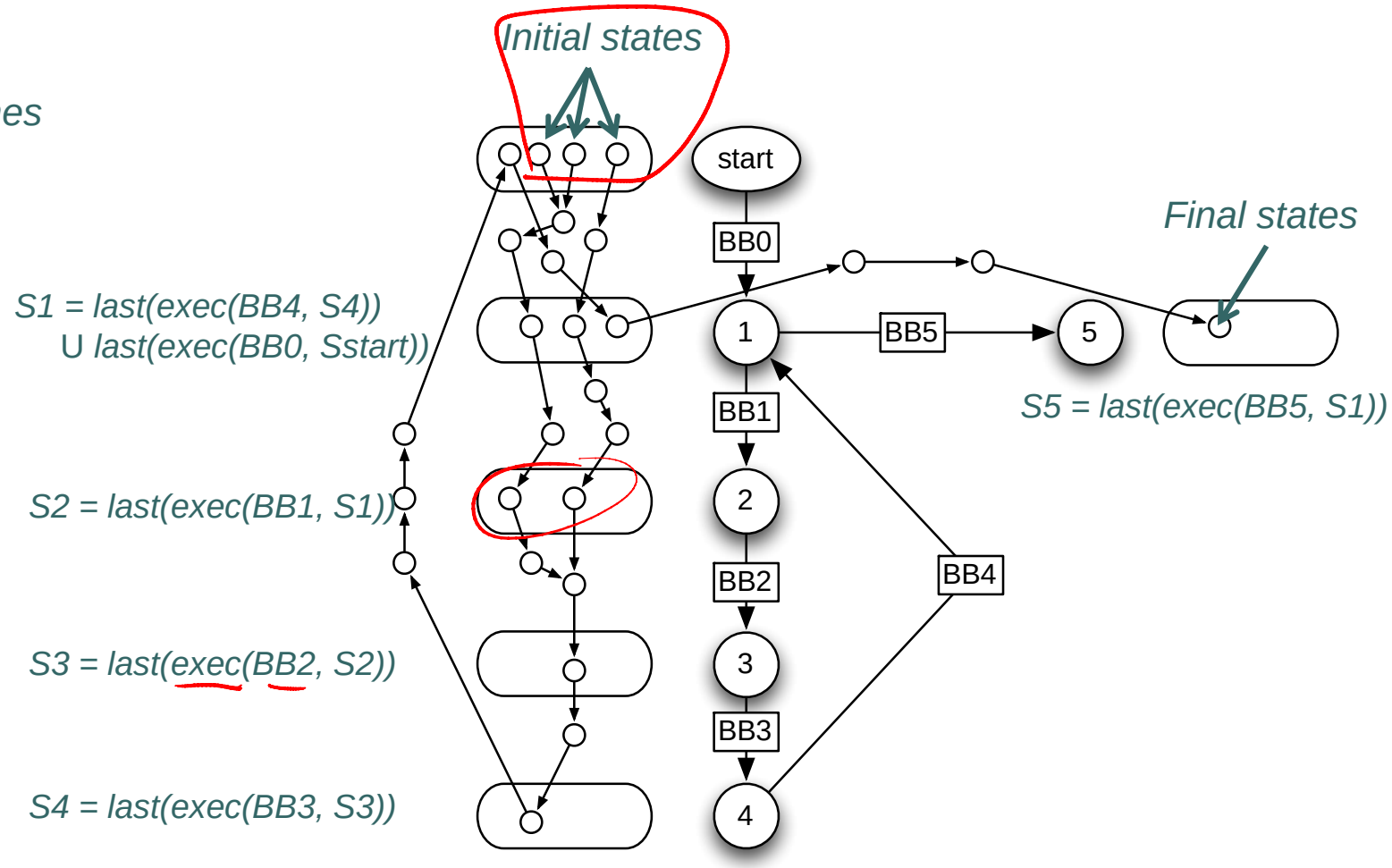
length(t) gives number of cycles for basic block *b.*

As in previous cases, we can lift *exec* from single pipeline states to sets of pipeline states to arrive at a Collecting Trace Semantics.

Illustration: Collecting Trace Semantics

Basic Block Execution Times (in cycles):

BB0: 2 or 3
BB1: 2 or 3
BB2: 2 or 3
BB3: 2
BB4: 4
BB5: 3



Sets of reachable states and traces can again be defined as least fixed point of set of equations.

An Abstract Pipeline Executing a Basic Block

function exec (*b* : **basic block**, *s* : **abstract pipeline state**) ***t*** : **trace**

Interprets instruction stream of *b* starting in state *s* producing abstract trace ***t.***

*length(**t**)* gives number of cycles.

What is different?
Abstraction introduces Nondeterminism!

- In the concrete pipeline model, one state resulted in one new state after a one-cycle transition
- Now, in the abstract model, one state can have several successor states:
  - In general: need to explore all successor states, cache miss not necessarily worse than cache hit → Timing Anomalies

An Abstract Pipeline Executing a Basic Block

function analyze (*b* : **basic block**, *S* : **analysis state**) *T* : **trace**

Analysis states = $PS \rightarrow CS_{\perp}$

PS = set of abstract pipeline states

CS = lattice of abstract cache states

Can be interpreted as
- *set of abstract pipeline states (= those that do not map to bottom)*
- *one abstract cache state for each pipeline state in this set*

Interprets instruction stream of *b* starting in state ***S*** producing abstract trace ***T*** of analysis states.

*max(length(**T**))* - upper bound for execution time

*last(**T**)* - initial analysis state for successor block

*Why maintain **sets** of abstract pipeline states?*

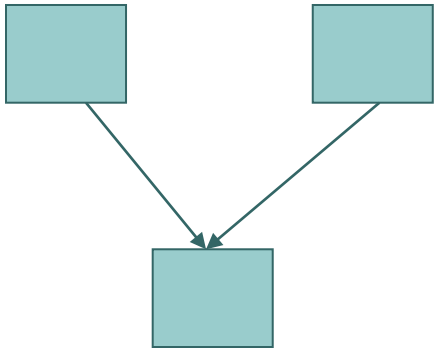
Domain of Analysis States

Analysis states = $PS \to CS_\perp$

Join/Order of analysis states:

$$A \sqcup B = \lambda p \in PS.A(p) \sqcup_{CS} B(p)$$

"Union of sets of abstract pipeline states"
+ "Join of corresponding abstract cache states"



Concretization:

$$\gamma(AS) := \bigcup_{ps \in PS} \{\langle p, c\rangle \mid p \in \gamma_{PS}(ps) \wedge c \in \gamma_{CS}(AS(ps))\}$$

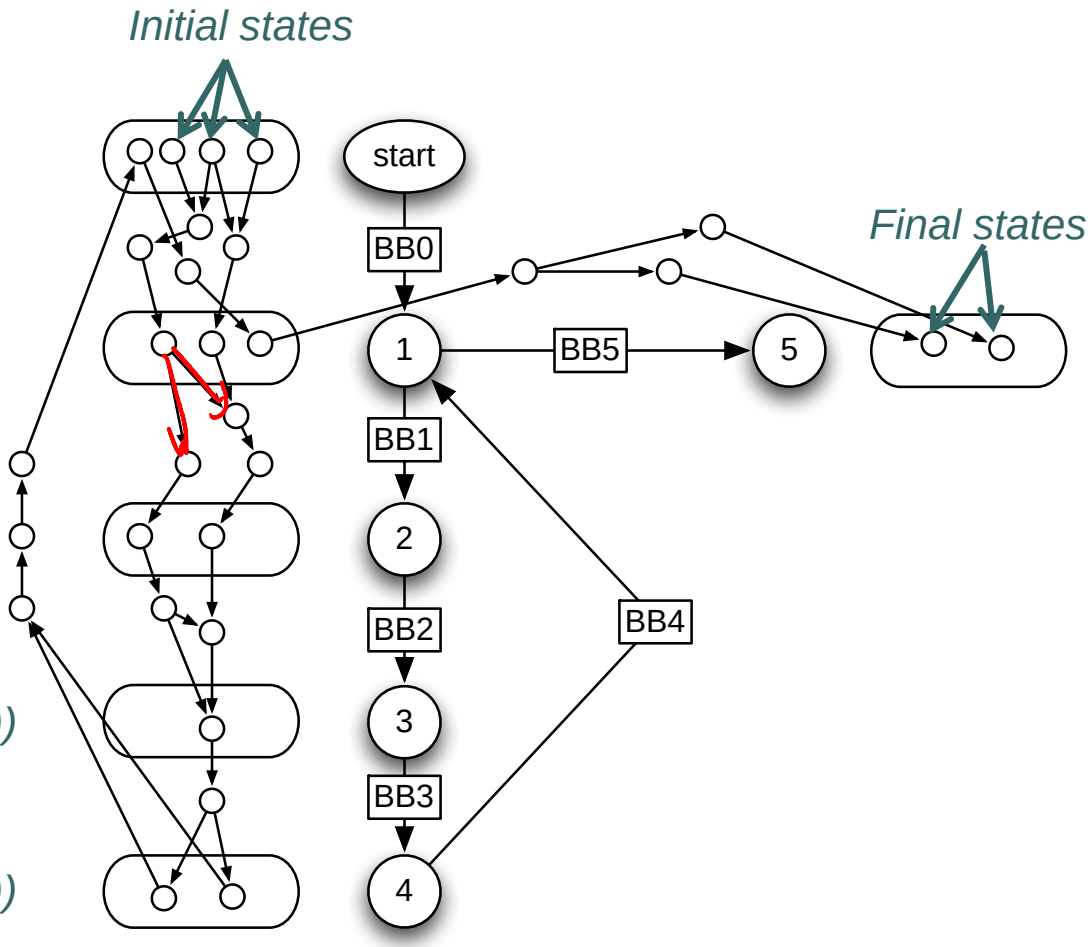
Concretization of abstract pipeline state

Concretization of corresponding abstract cache state

Illustration: Abstract Collecting Trace Semantics

Basic Block Execution Times (in cycles):

- *BB0: 2 or 3*
- *BB1: 2 or 3*
- *BB2: 2 or 3*
- *BB3: 2*
- *BB4: 4*
- *BB5: 3*



S3 = last(analyze(BB2, S2))

S4 = last(analyze(BB3, S3))

Sets of reachable states and traces can again be defined as least fixed point of set of equations.

Conclusions

- Execution time of basic blocks is property of a trace semantics
- Microarchitectural analysis integrates analyses of pipeline and cache behavior
- So far, no “good” abstraction for pipeline states → analysis maintains sets of (almost concrete) abstract pipeline states
- Analysis needs to consider all cases due to timing anomalies (more about these later)