Run-Time Guarantees for Real-Time Systems

Reinhard Wilhelm
Saarbrücken

AbsInt
Angewandte Informatik GmbH

UNIVERSITÄT
DES
SAARLANDES

Structure of the Talks

1. Introduction,
   - problem statement,
   - tool architecture,
   - static program analysis
2. Caches
   - must, may analysis
3. Pipelines
   - Abstract pipeline models
   - Integrated analyses
4. Results, conclusions, and future work

Industrial Needs

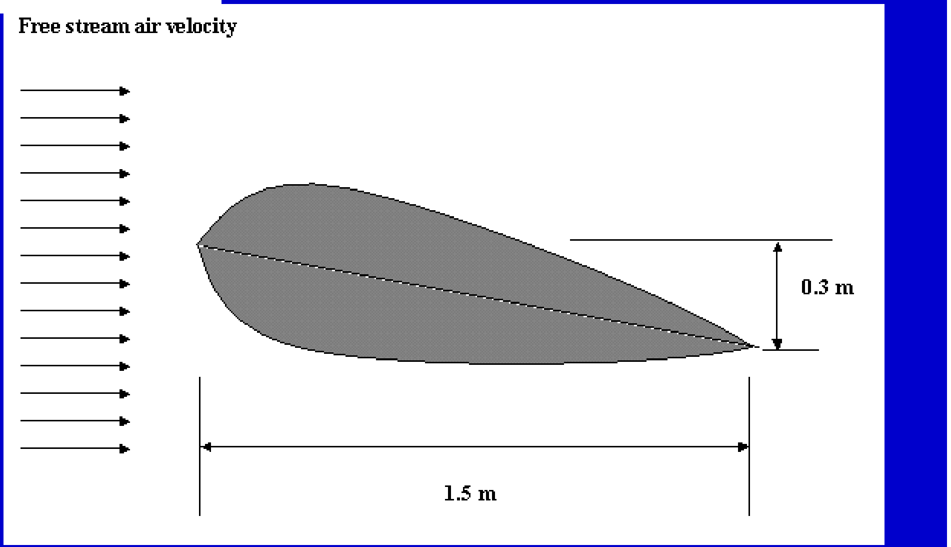
- Hard real-time systems, often in safety-critical applications abound
  - Aeronautics, automotive, train industries, manufacturing control

Sideairbag in car,

Reaction in <10 mSec

Wing vibration of airplane,

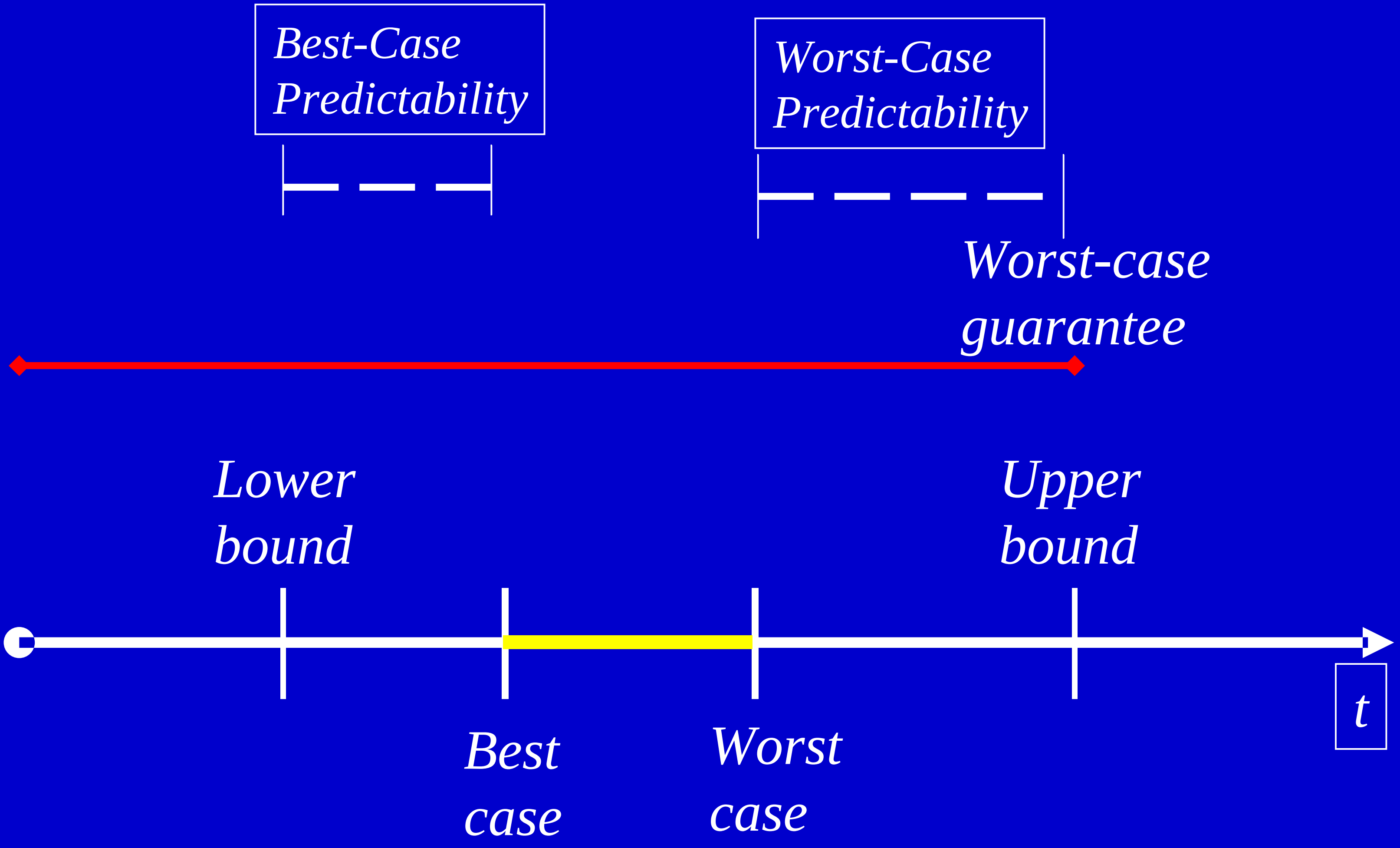
sensing every 5 mSec



Hard Real-Time Systems

- Embedded controllers are expected to finish their tasks reliably within time bounds.
- Task scheduling must be performed
- Essential: upper bound on the execution times of all tasks statically known
- Commonly called the Worst-Case Execution Time (WCET)
- Analogously, Best-Case Execution Time (BCET)

Basic Notions



Best-Case Predictability

Worst-Case Predictability

Worst-case guarantee

Lower bound

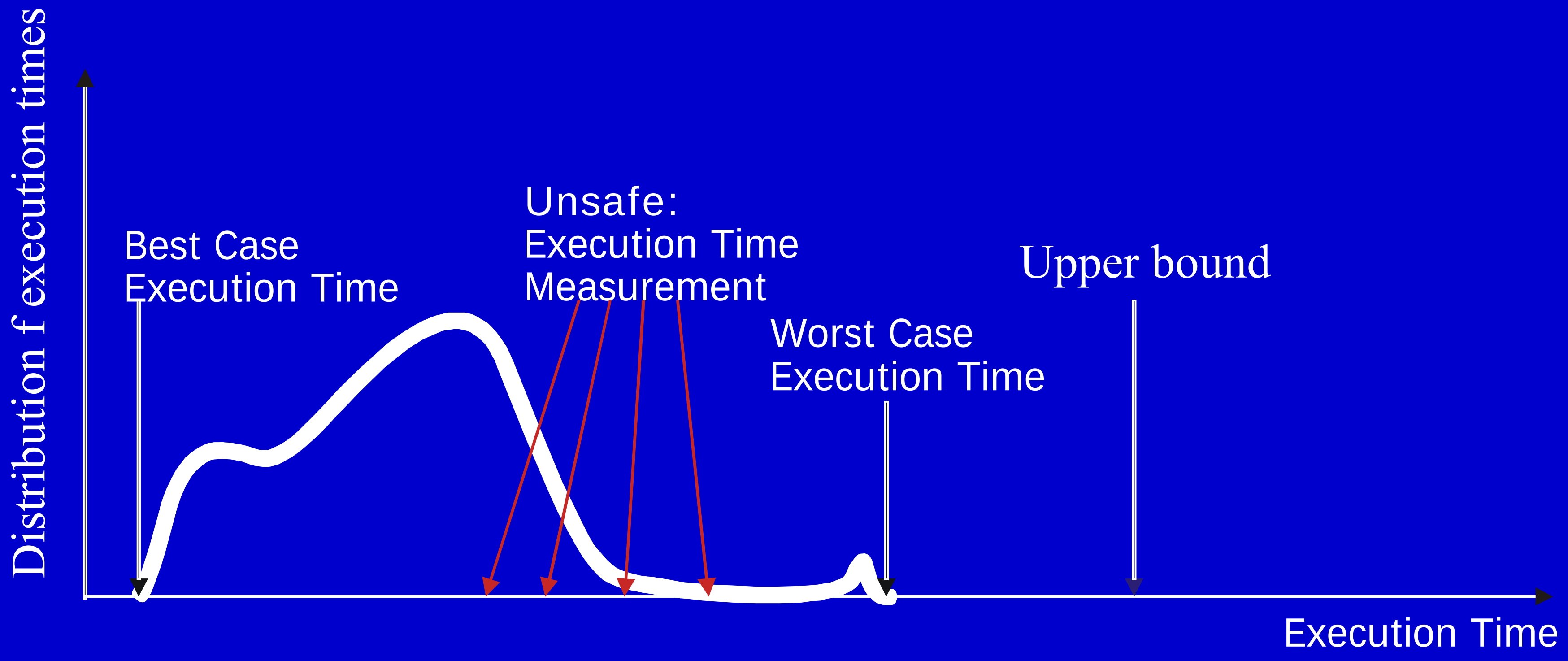
Upper bound

Best case

Worst case

t

Measurement – Industry´s “best practice”



Works if either

- *worst-case input can be determined, or*
- *exhaustive measurement is performed*

Otherwise, determine upper bound from execution times of instructions

(Most of) Industry’s Best (?) Practice

- Measurements: determine execution times directly by observing the execution or a simulation on a set of inputs.

  Does not guarantee an upper bound to all executions.
- Exhaustive execution in general not possible! Too large space of input domain x set of initial execution states.

Structure-based Approach

Compute upper bounds along the structure of the program

- Programs are hierarchically structured;
- statements are nested inside statements.
- so, compute the upper bound for a statement from the upper bounds of ist constituents

Sequence of Statements

A ≡ A1; A2;

Constituents of A*:*
A1 *and* A2

Upper bound for A
is the sum of the upper
bounds for A1 and A2

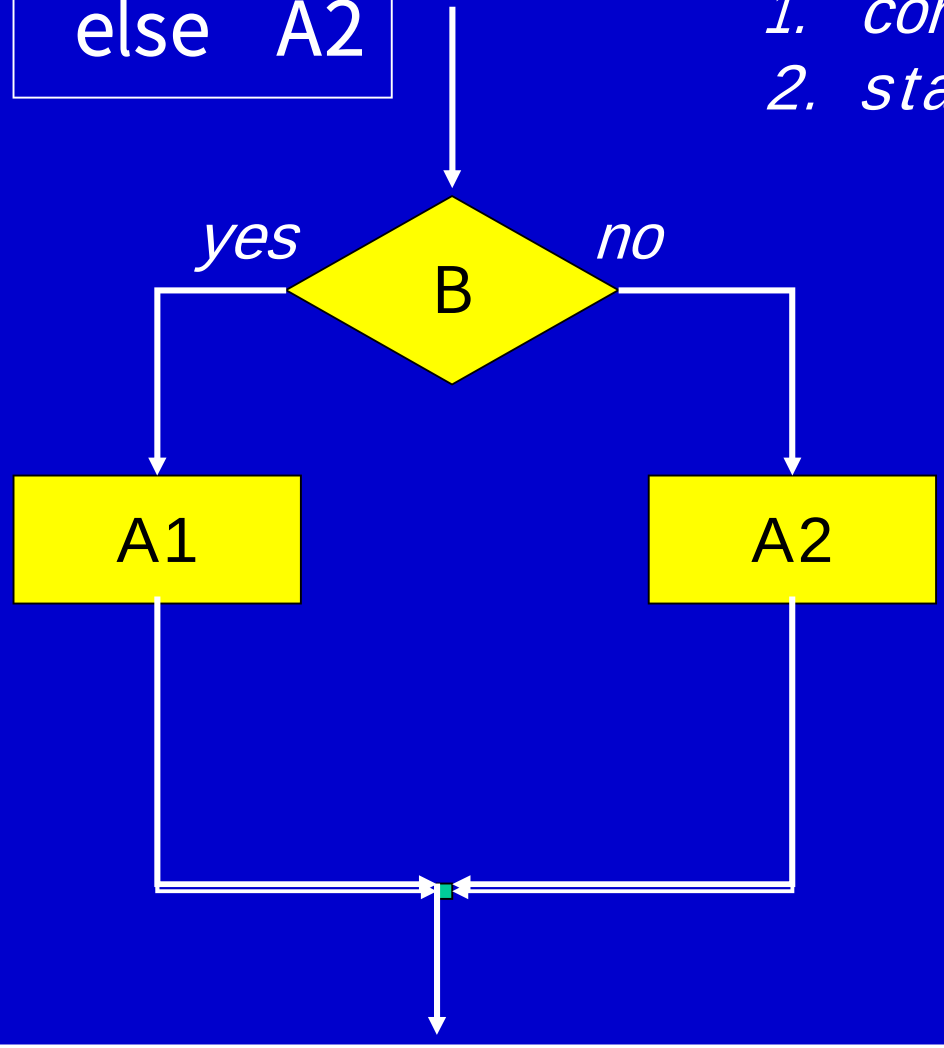
$$\mathrm{ub}(A) = \mathrm{ub}(A1) + \mathrm{ub}(A2)$$

Conditional Statement

```
A ≡ if B
      then  A1
      else  A2
```

Constituents of A:

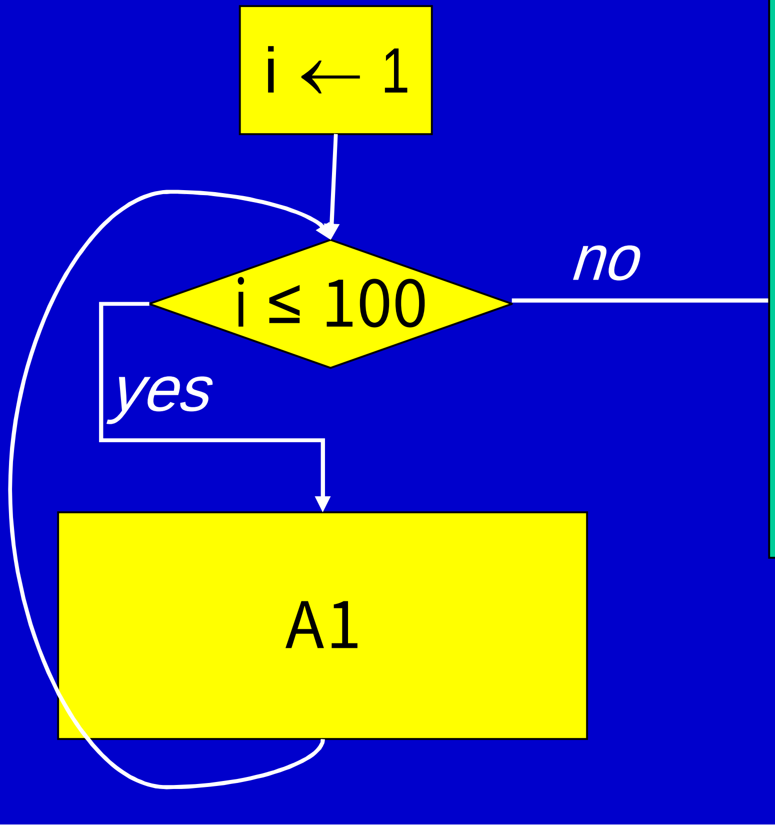
1. *condition* B
2. *statements* A1 *and* A2



$$ub(A) = ub(B) + \max(ub(A1), ub(A2))$$

Loops

$A \equiv$ for $i \leftarrow 1$ to 100 do
A1



$$
\begin{aligned}
ub(A) = \; & ub(i \leftarrow 1) + \\
& 100 \times (\, ub(i \leq 100) + \\
& \qquad ub(A1)\,) + \\
& ub(\, i \leq 100)
\end{aligned}
$$

How to start?

- Assignment

  $x \leftarrow a + b$

Assumes constant exexcution times for instructions

Add

cycles	
store m	14
move	1

Not applicable to modern processors!

$\mathrm{ub}(x \leftarrow a + b) =$

$\quad \mathrm{cycles}(\text{load a}) +$

$\quad \mathrm{cycles}(\text{load b}) +$

$\quad \mathrm{cycles}(\text{add}) +$

$\quad \mathrm{cycles}(\text{store x})$

Modern Hardware Features

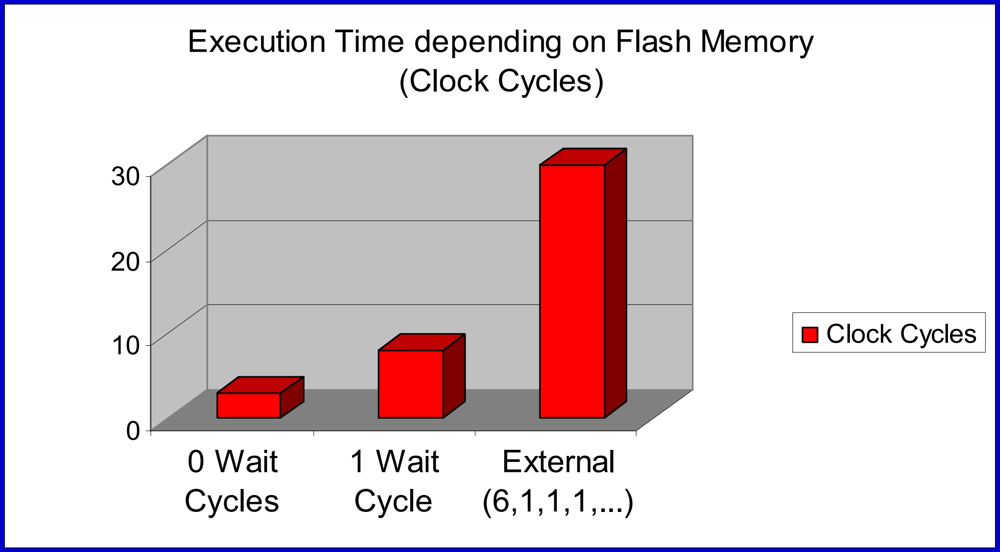
- Modern processors increase performance by using: Caches, Pipelines, Branch Prediction, Speculation
- These features make WCET computation difficult: Execution times of instructions vary widely
  - Best case - everything goes smoothely: no cache miss, operands ready, needed resources free, branch correctly predicted
  - Worst case - everything goes wrong: all loads miss the cache, resources needed are occupied, operands are not ready
  - Span may be several hundred cycles

Access Times

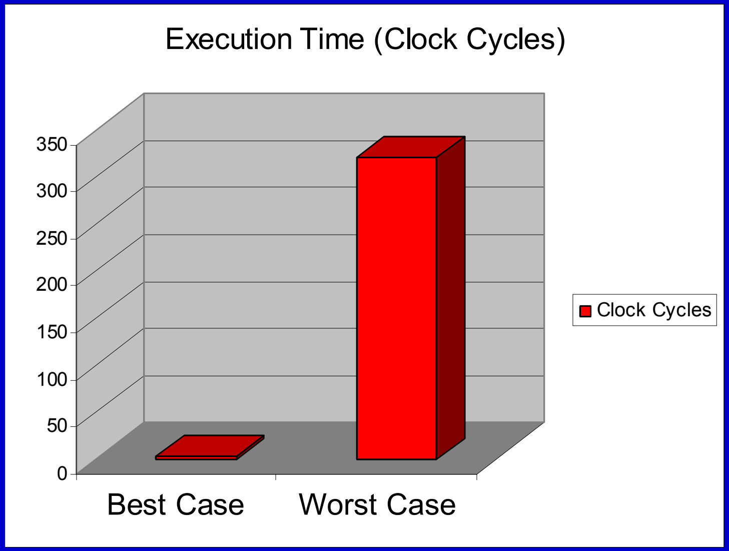
x = a + b;

```
LOAD      r2, _a
LOAD      r1, _b
ADD       r3,r2,r1
```

MPC 5xx



PPC 755



(Concrete) Instruction Execution

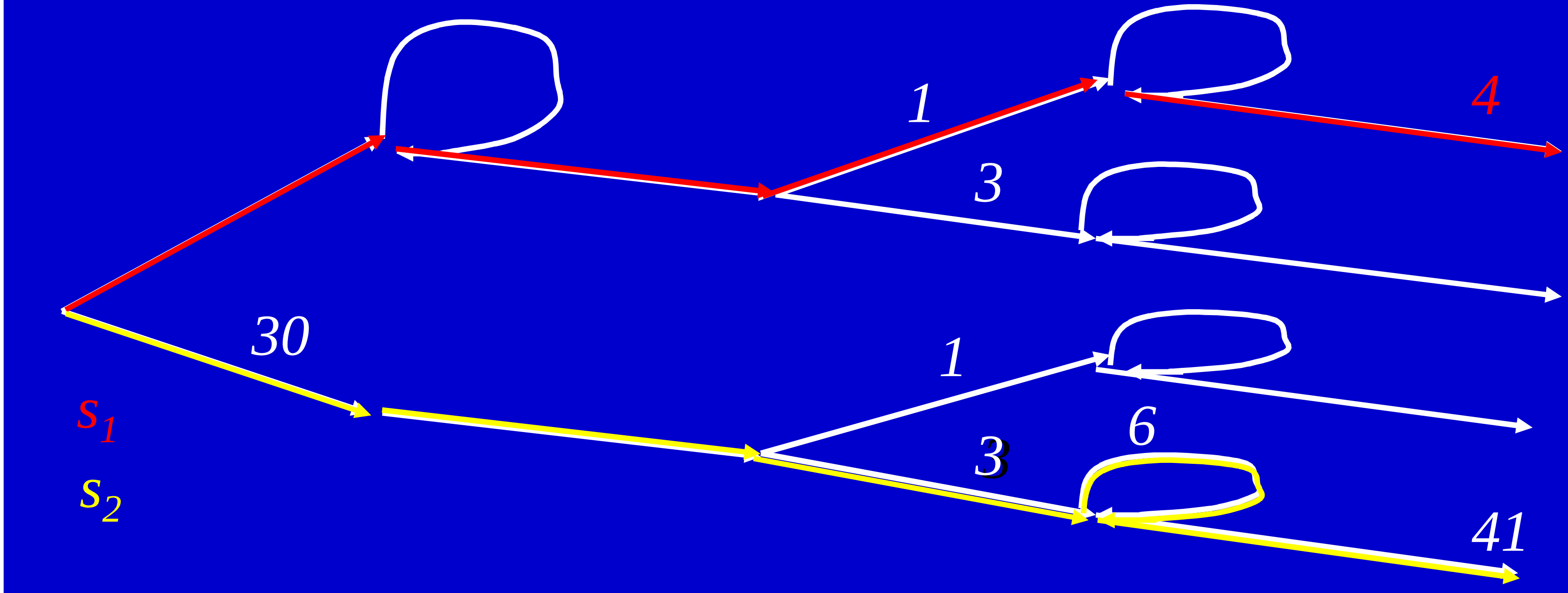
`mul`

Fetch
I-Cache miss?

Issue
Unit occupied?

Execute
Multicycle?

Retire
Pending instructions?



Timing Accidents and Penalties

Timing Accident – cause for an increase of the execution time of an instruction

Timing Penalty – the associated increase

- Types of timing accidents
  - Cache misses
  - Pipeline stalls
  - Branch mispredictions
  - Bus collisions
  - Memory refresh of DRAM
  - TLB miss

History-Sensitivity of Execution Times
- Problem and Chance -

Contribution of the execution of an instruction to a program‘s execution time

- depends on the execution state, i.e., on the execution so far,
- can be bounded if strong invariants about all execution states at this instruction are available.

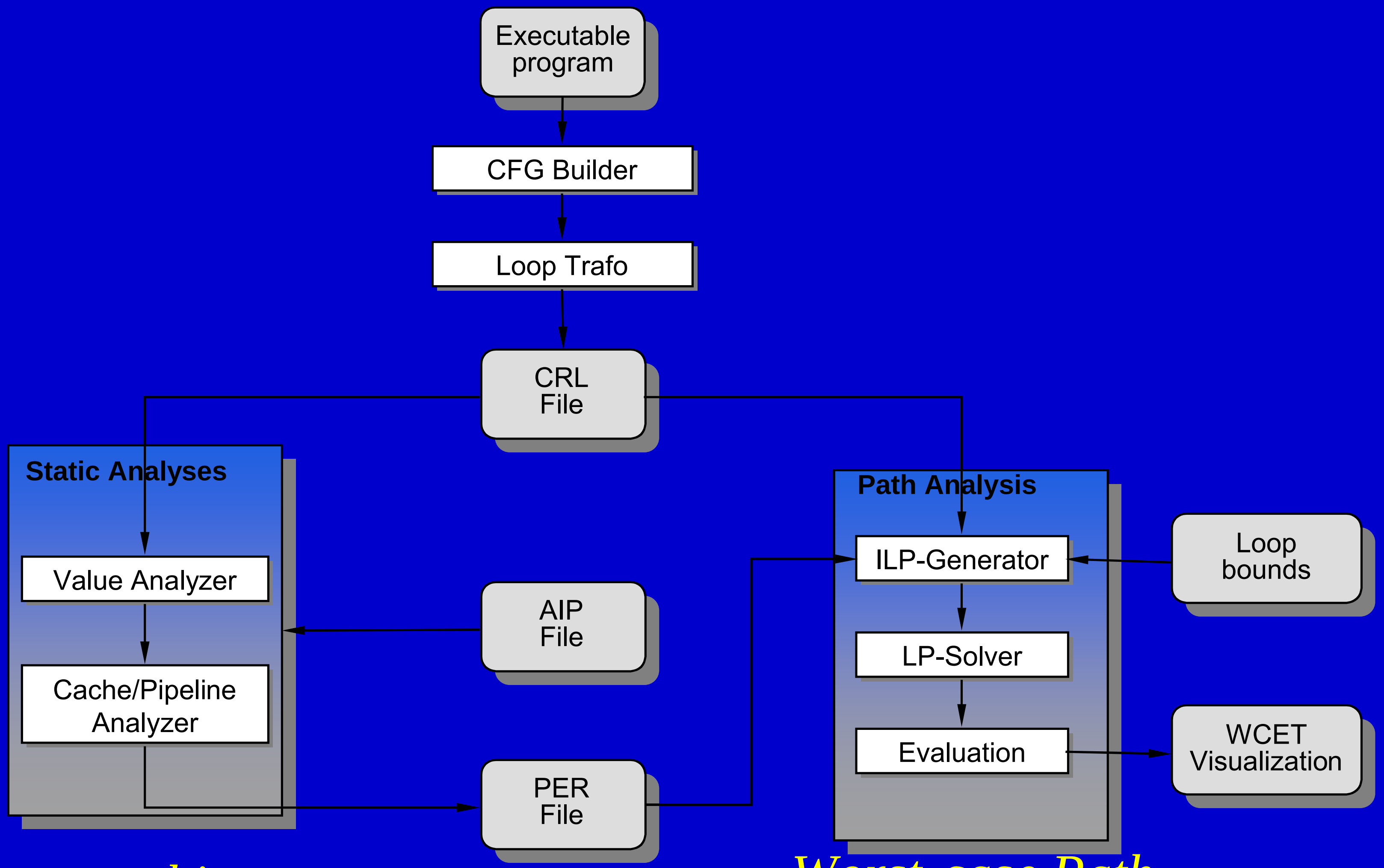
The Challenge

- Goal: Determine reliable and precise upper bounds for the program‘s execution times
- Method: Determine precise upper bounds for the program‘s instructions in their control-flow context

Overall Approach: Natural Modularization

1. Micro-architecture Analysis:
   - Uses Abstract Interpretation
   - Excludes as many Timing Accidents as possible
   - Determines WCET for basic blocks (in contexts)
2. Worst-case Path Determination
   - Maps control flow graph to an integer linear program
   - Determines upper bound and associated path

Overall Structure



Micro-architecture Analysis

Worst-case Path Determination

Murphy’s Law in Timing Analysis

- Naïve, but safe guarantee accepts Murphy’s Law: Any accident that may happen will happen
- Consequence: hardware overkill necessary to guarantee timeliness
- Example: Alfred Rosskopf, EADS Ottobrunn, measured performance of PPC with all the caches switched off (corresponds to assumption ‘all memory accesses miss the cache’)
  Result: Slowdown of a factor of 30!!!

Fighting Murphy’s Law

- Static Program Analysis allows the derivation of Invariants about all execution states at a program point
- Derive Safety Properties from these invariants: Certain timing accidents will never happen.
  Example: At program point p, instruction fetch will never cause a cache miss
- The more accidents excluded, the lower the upper bound
- (and the more accidents predicted, the higher the lower bound)

Abstract Interpretation (AI)

- semantics-based method for static program analysis
- Basic idea of AI: 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
- AI supports correctness proofs
- Tool support (Program-Analysis Generator PAG)

Abstract Interpretation – the Ingredients

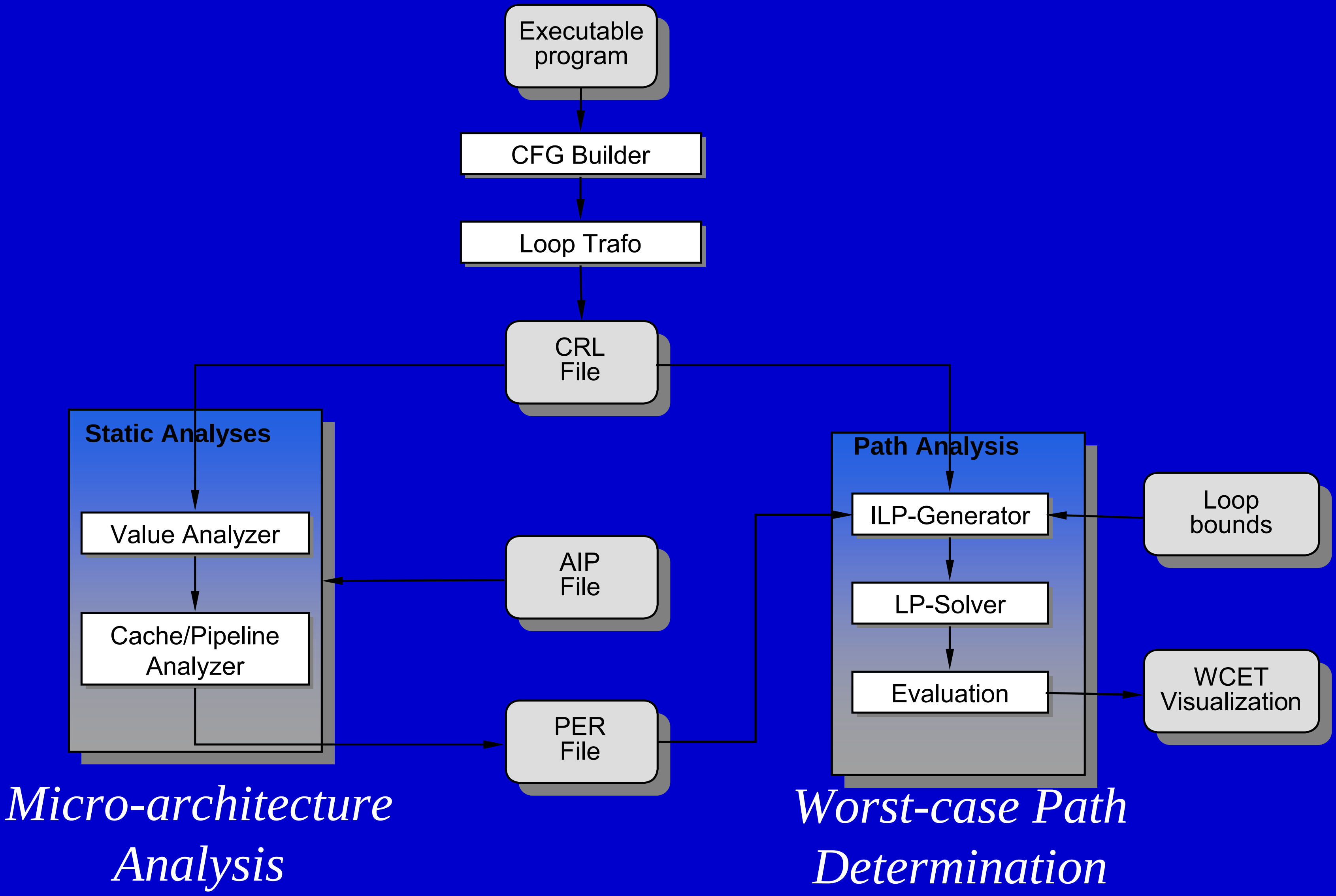
- abstract domain – complete semilattice, related to concrete domain by abstraction and concretization functions,
  e.g. $L \rightarrow \text{Intervals}$,
  where $\text{Intervals} = LB \times UB$,
  $LB = UB = \text{Int} \cup \{-\infty, \infty\}$
  instead of $L \rightarrow \text{Int}$
- abstract transfer functions for each statement type – abstract versions of their semantics
  e.g. $+^{\#}$: Intervals × Intervals → Intervals where
  $[a,b] +^{\#} [c,d] = [a+c, b+d]$ with + extended to $-\infty$, ∞
- a join function combining abstract values from different control-flow paths – lub on the lattice
  e.g. t : Interval × Interval → Interval where
  $[a,b]$ t $[c,d] = [\mathit{min}(a,c), \mathit{max}(b,d)]$ with *min*, *max* extended to $-\infty$, ∞

Abstract Interpretation – the Ingredients and one Example

- abstract domain – complete semilattice, related to concrete domain by abstraction and concretization functions,
  e.g. intervals of integers (including -∞, ∞) instead of integer values
- abstract transfer functions for each statement type – abstract versions of their semantics
  e.g. arithmetic and assignment on intervals
- a join function combining abstract values from different control-flow paths – lub on the lattice
  e.g. “union” on intervals
- Example: Interval analysis (Cousot/Halbwachs78)

Value Analysis

Overall Structure

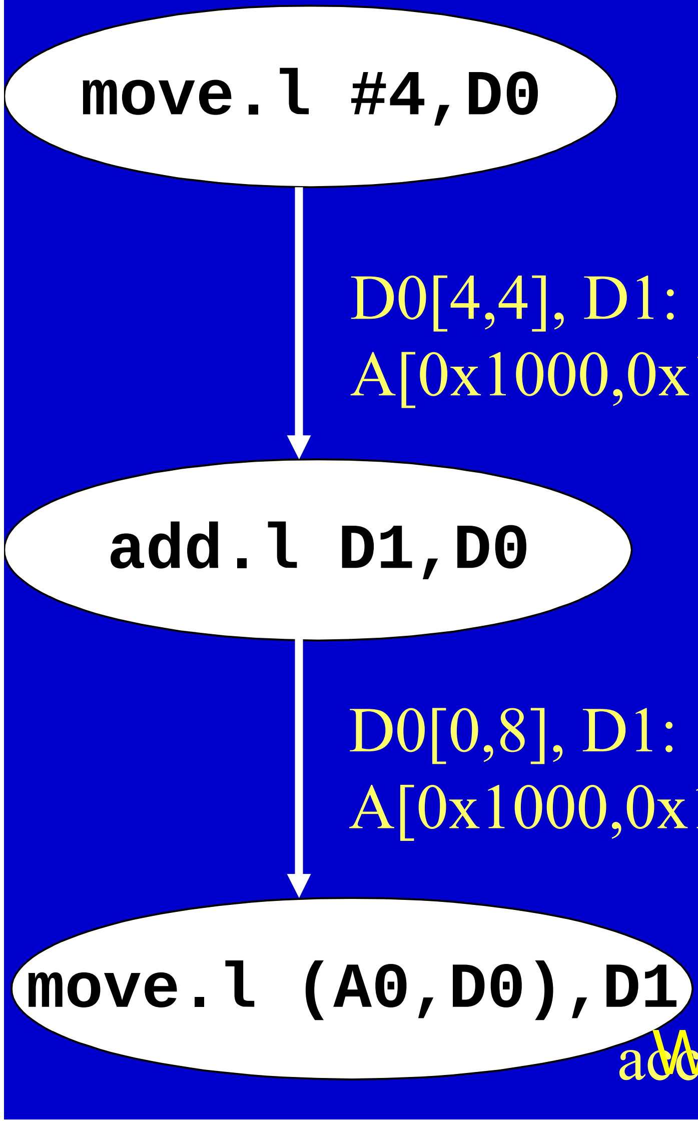


Value Analysis

- 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) (Cousot/Halbwachs78)

Value Analysis II

D1: [-4,4], A[0x1000,0x1000]



`move.l #4,D0`

D0[4,4], D1: [-4,4], A[0x1000,0x1000]

`add.l D1,D0`

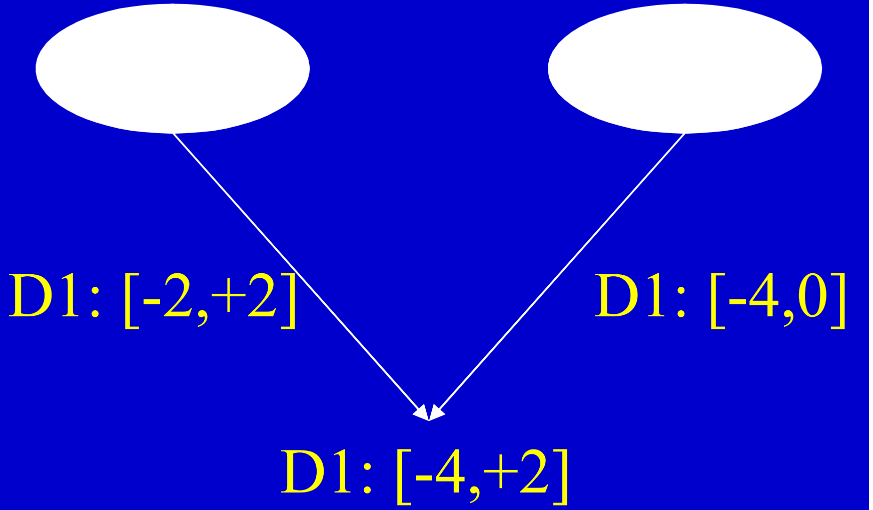
D0[0,8], D1: [-4,4], A[0x1000,0x1000]

`move.l (A0,D0),D1`

access [0x1000,0x1008]

Which address is accessed here?

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



D1: [-2,+2]

D1: [-4,0]

D1: [-4,+2]

Value Analysis (Airbus Benchmark)

Task	Unreached	Exact	Good	Unknown	Time [s]
1	8%	86%	4%	2%	47
2	8%	86%	4%	2%	17
3	7%	86%	4%	3%	22
4	13%	79%	5%	3%	16
5	6%	88%	4%	2%	36
6	9%	84%	5%	2%	16
7	9%	84%	5%	2%	26
8	10%	83%	4%	3%	14
9	6%	89%	3%	2%	34
10	10%	84%	4%	2%	17
11	7%	85%	5%	3%	22
12	10%	82%	5%	3%	14

1Ghz Athlon, Memory usage <= 20MB

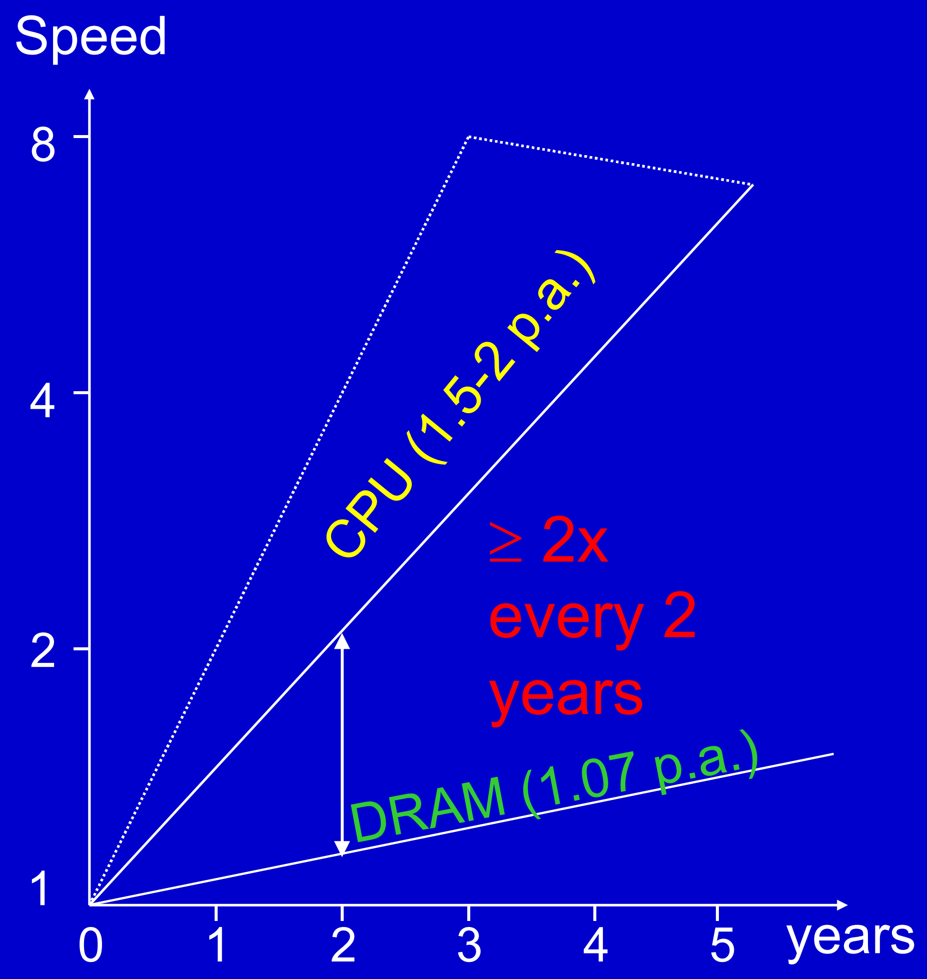
Good means less than 16 cache lines

Caches

Caches: Fast Memory on Chip

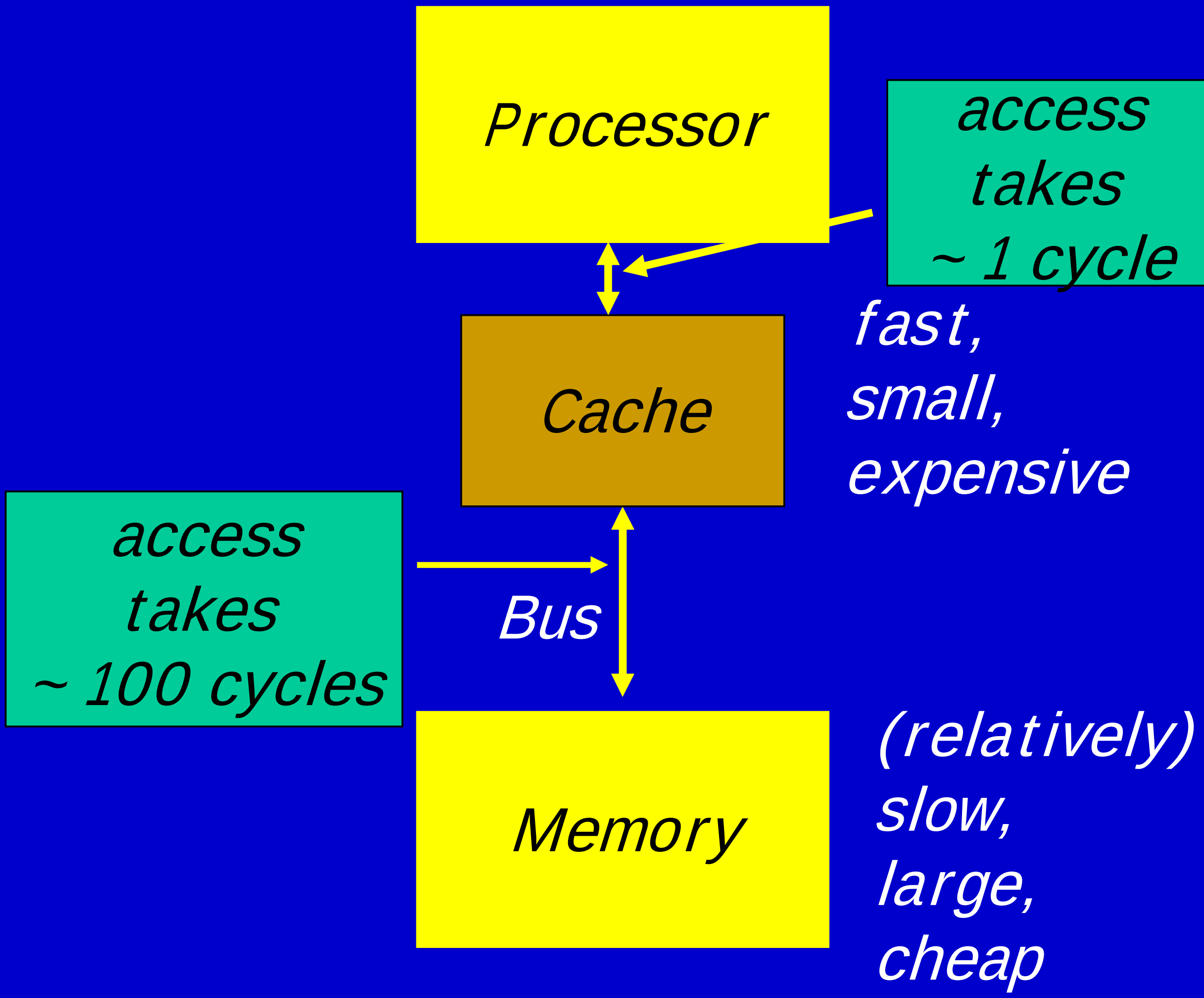
- Caches are used, because
  - Fast main memory is too expensive
  - The speed gap between CPU and memory is too large and increasing
- Caches work well in the average case:
  - Programs access data locally (many hits)
  - Programs reuse items (instructions, data)
  - Access patterns are distributed evenly across the cache

Speed gap between processor & main RAM increases



P.Marwedel

Caches



Processor

access takes ~ 1 cycle

Cache

fast, small, expensive

access takes ~ 100 cycles

Bus

Memory

(relatively) slow, large, cheap

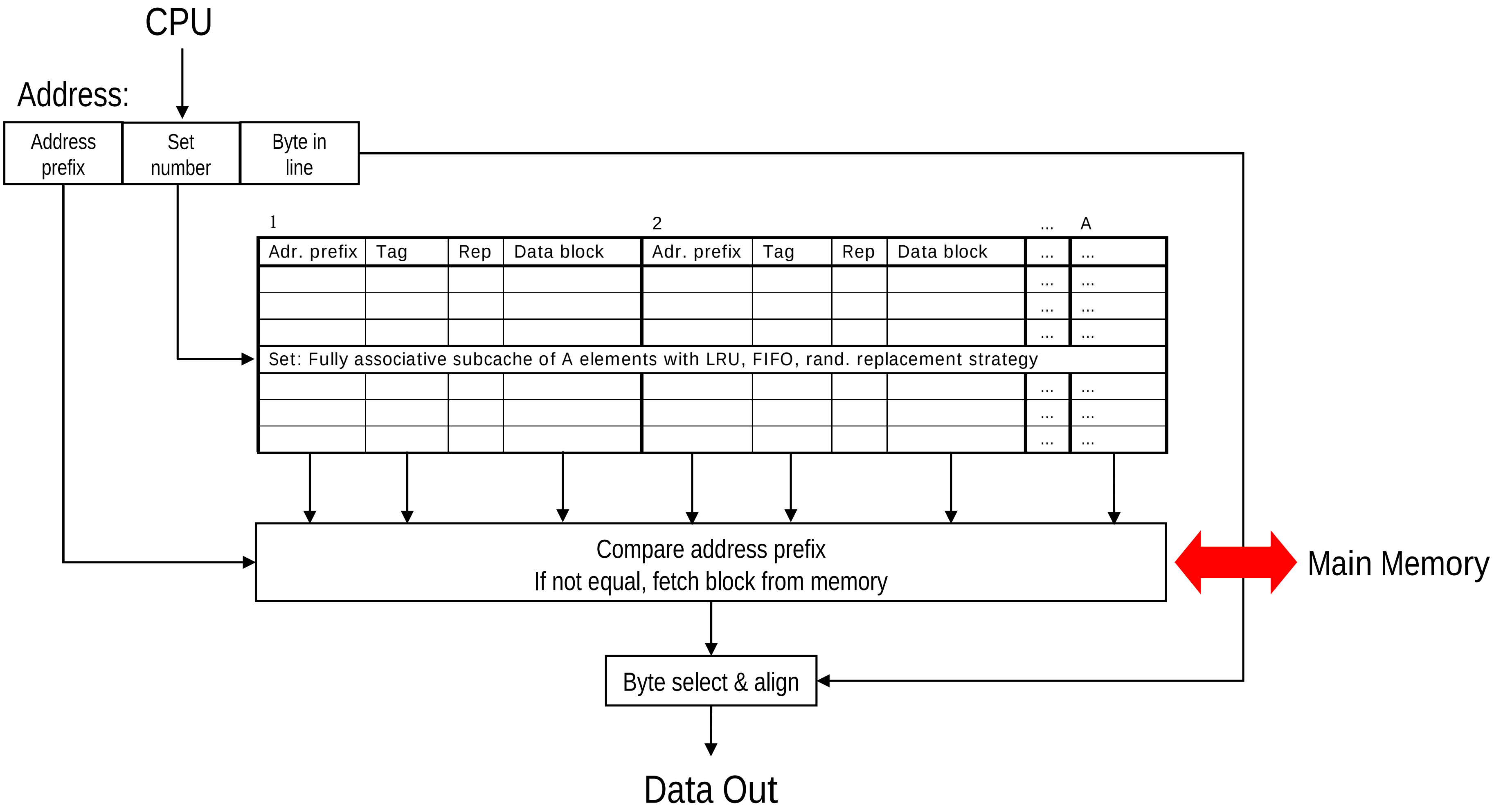
Caches: How the work

CPU wants to read/write at memory address *a*,
sends a request for *a* to the bus

Cases:

- Block *m* containing *a* in the cache (hit):
  request for *a* is served in the next cycle
- Block *m* 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

A-Way Set Associative Cache



CPU

Address:

Address prefix	Set number	Byte in line

1				2				...	A
Adr. prefix	Tag	Rep	Data block	Adr. prefix	Tag	Rep	Data block	...	...
								...	...
								...	...
								...	...
Set: Fully associative subcache of A elements with LRU, FIFO, rand. replacement strategy									
								...	...
								...	...
								...	...

Compare address prefix
If not equal, fetch block from memory

Main Memory

Byte select & align

Data Out

LRU Strategy

- Each cache set has its own replacement logic => Cache sets are independent: Everything explained in terms of one set
- LRU-Replacement Strategy:
  - Replace the block that has been Least Recently Used
  - Modeled by Ages
- Example: 4-way set associative cache

age	0	1	2	3
	m_0	m_1	m_2	m_3
Access m_4 (miss)	m_4	m_0	m_1	m_2
Access m_1 (hit)	m_1	m_4	m_0	m_2
Access m_5 (miss)	m_5	m_1	m_4	m_0

Cache Analysis

How to statically precompute cache contents:

- Must Analysis:

  For each program point (and calling context), find out which blocks are in the cache

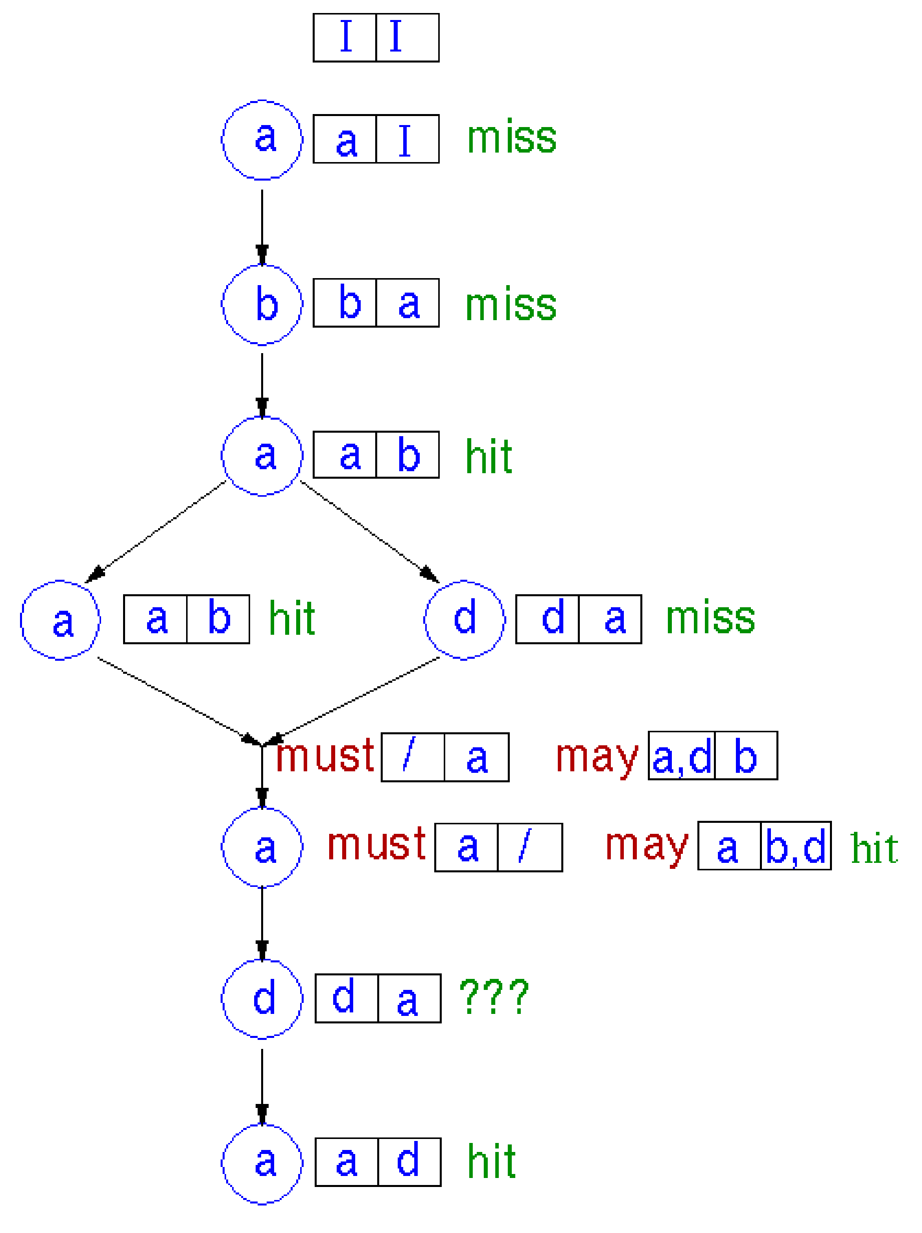
- May Analysis:

  For each program point (and calling context), find out which blocks may be in the cache

  Complement says what is not in the cache

Must-Cache and May-Cache-Information

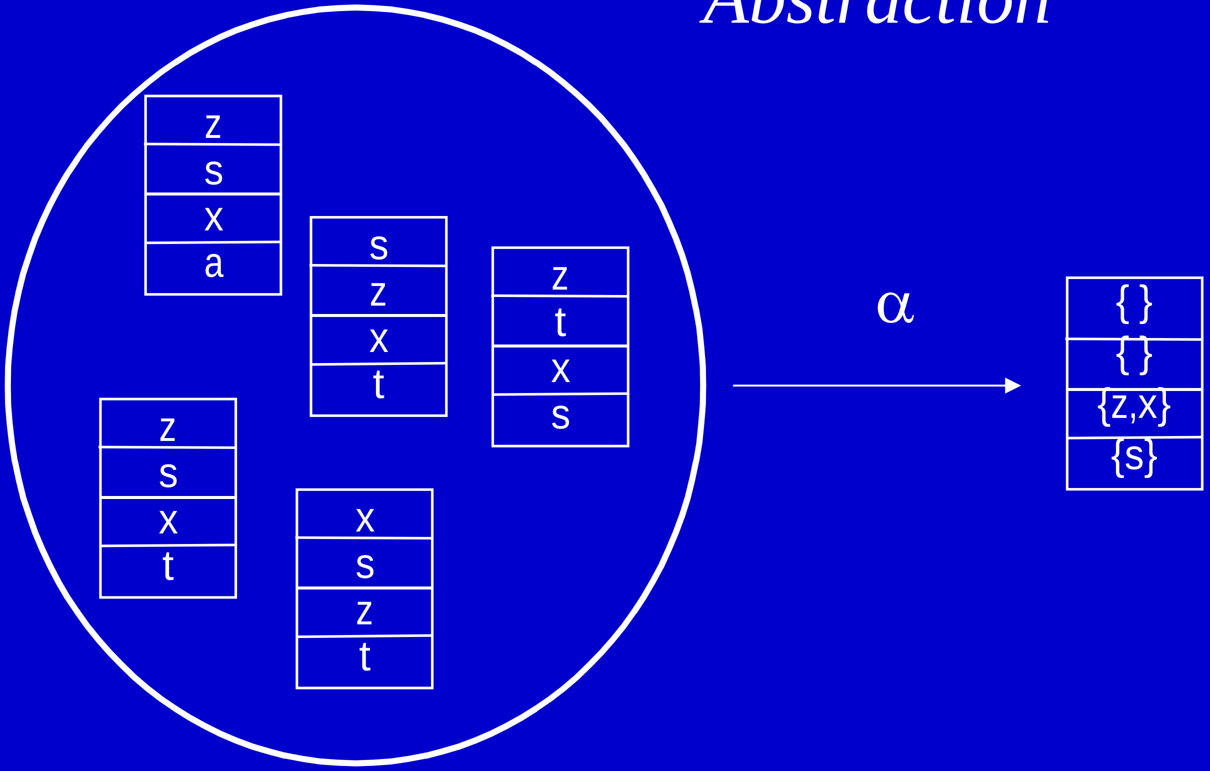
- Must Analysis determines safe information about cache hits
  Each predicted cache hit reduces WCET
- May Analysis determines safe information about cache misses
  Each predicted cache miss increases BCET



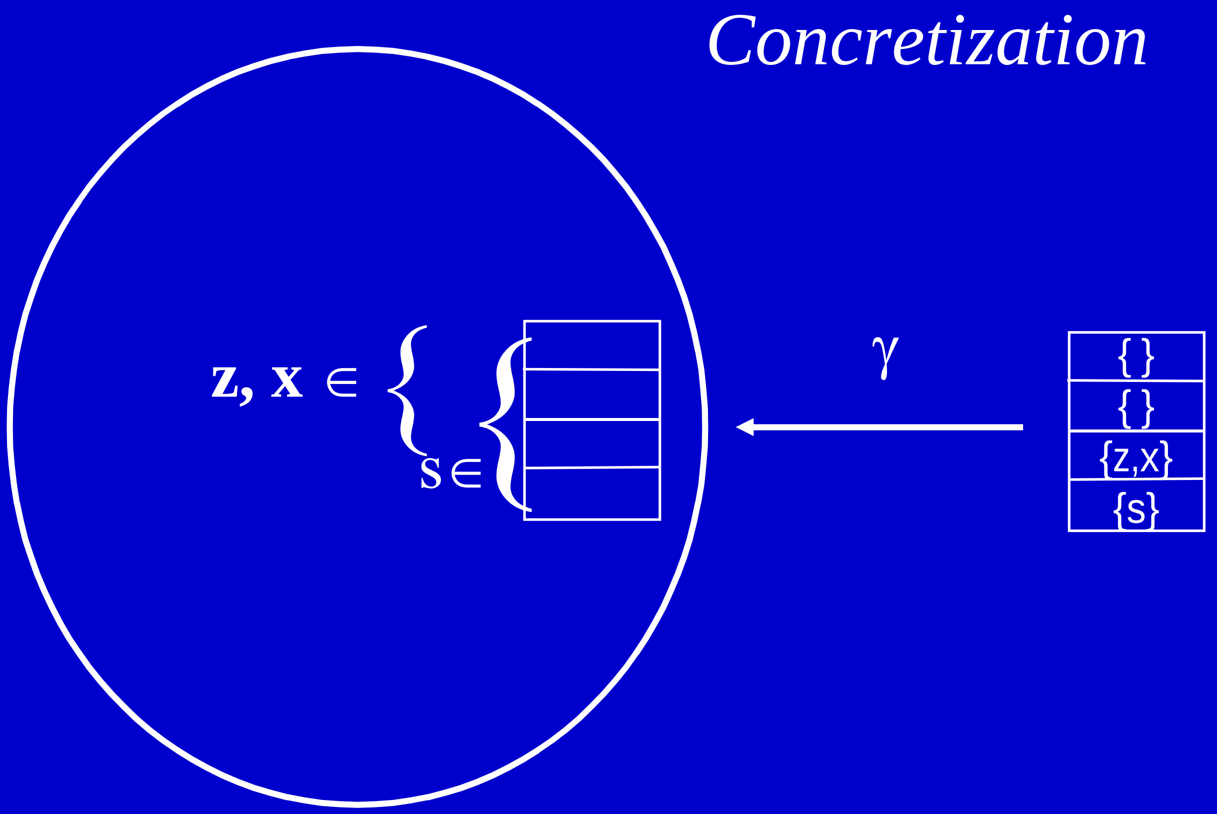
Example:
Fully Associative Cache
(2 Elements)

Abstract Domain: Must Cache

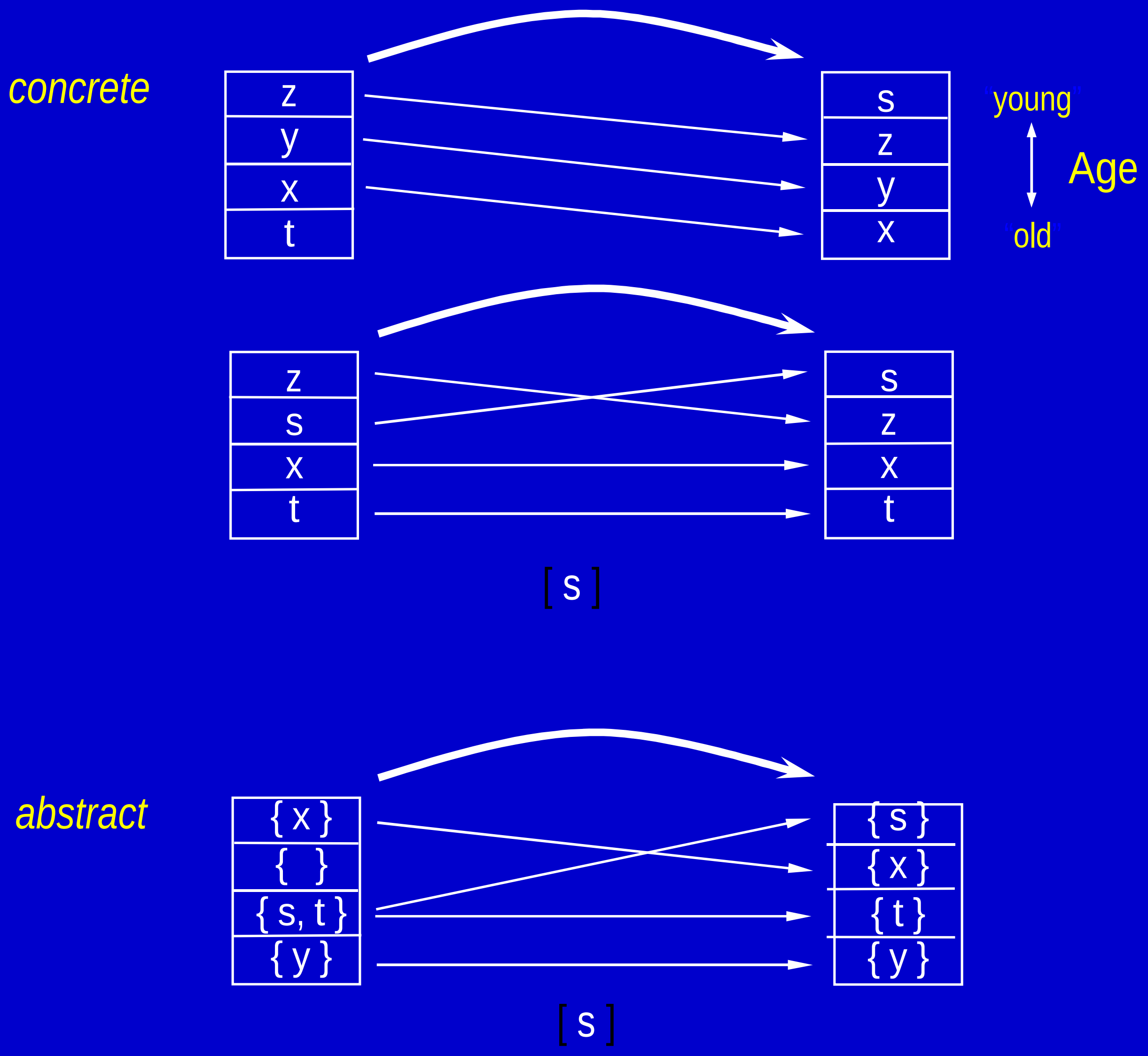
Abstraction



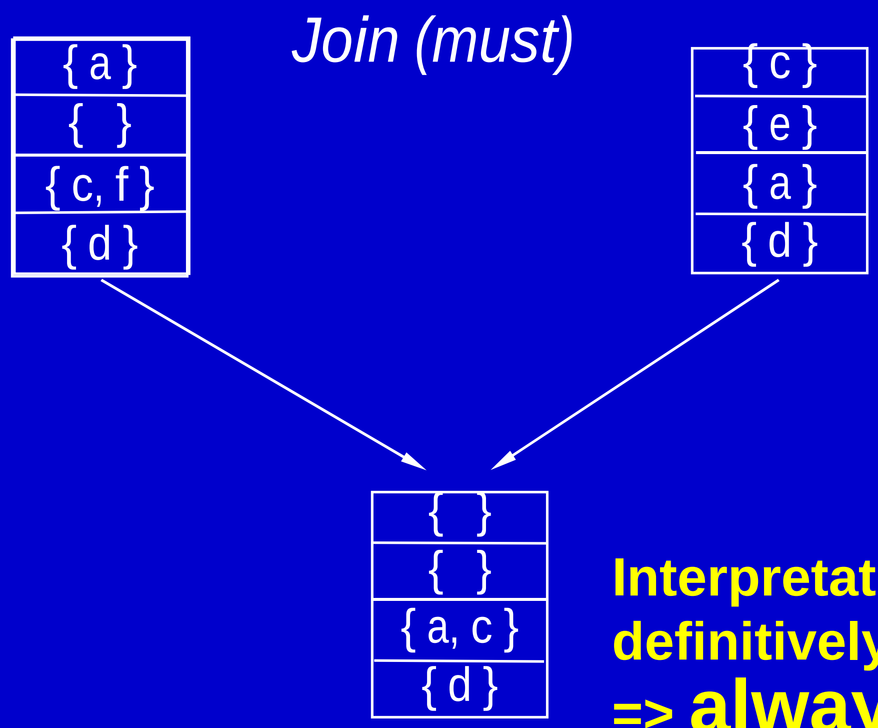
Abstract Domain: Must Cache



Cache with LRU Replacement: Transfer for must



Cache Analysis: Join (must)



Join (must)

{ a }
{ }
{ c, f }
{ d }

{ c }
{ e }
{ a }
{ d }

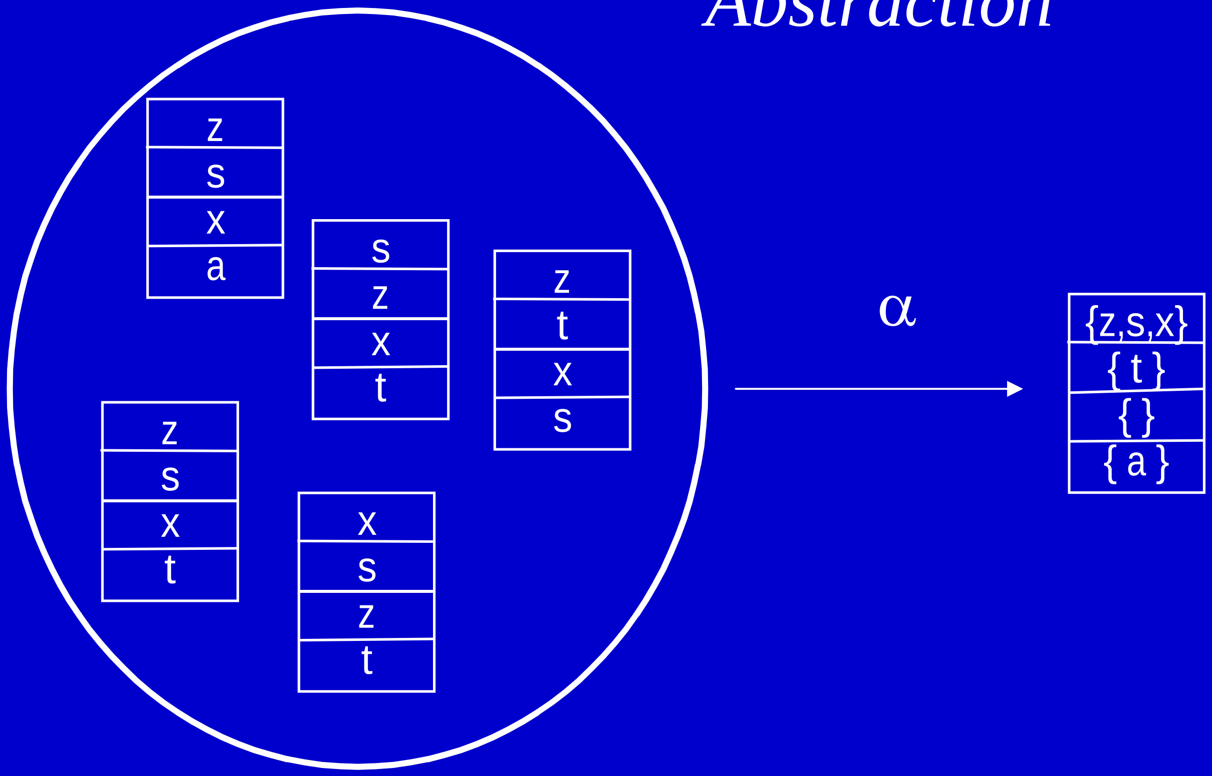
{ }
{ }
{ a, c }
{ d }

"intersection
+ maximal age"

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

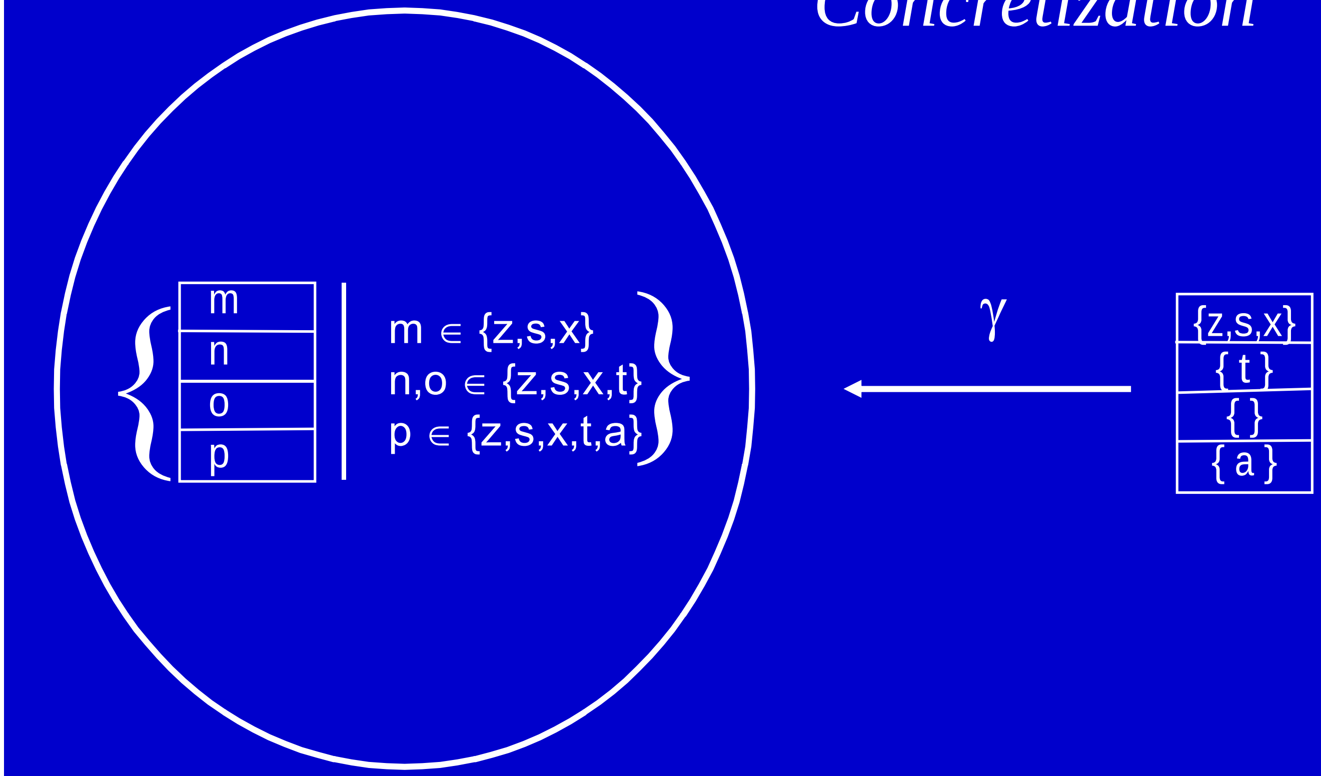
Abstract Domain: May Cache

Abstraction

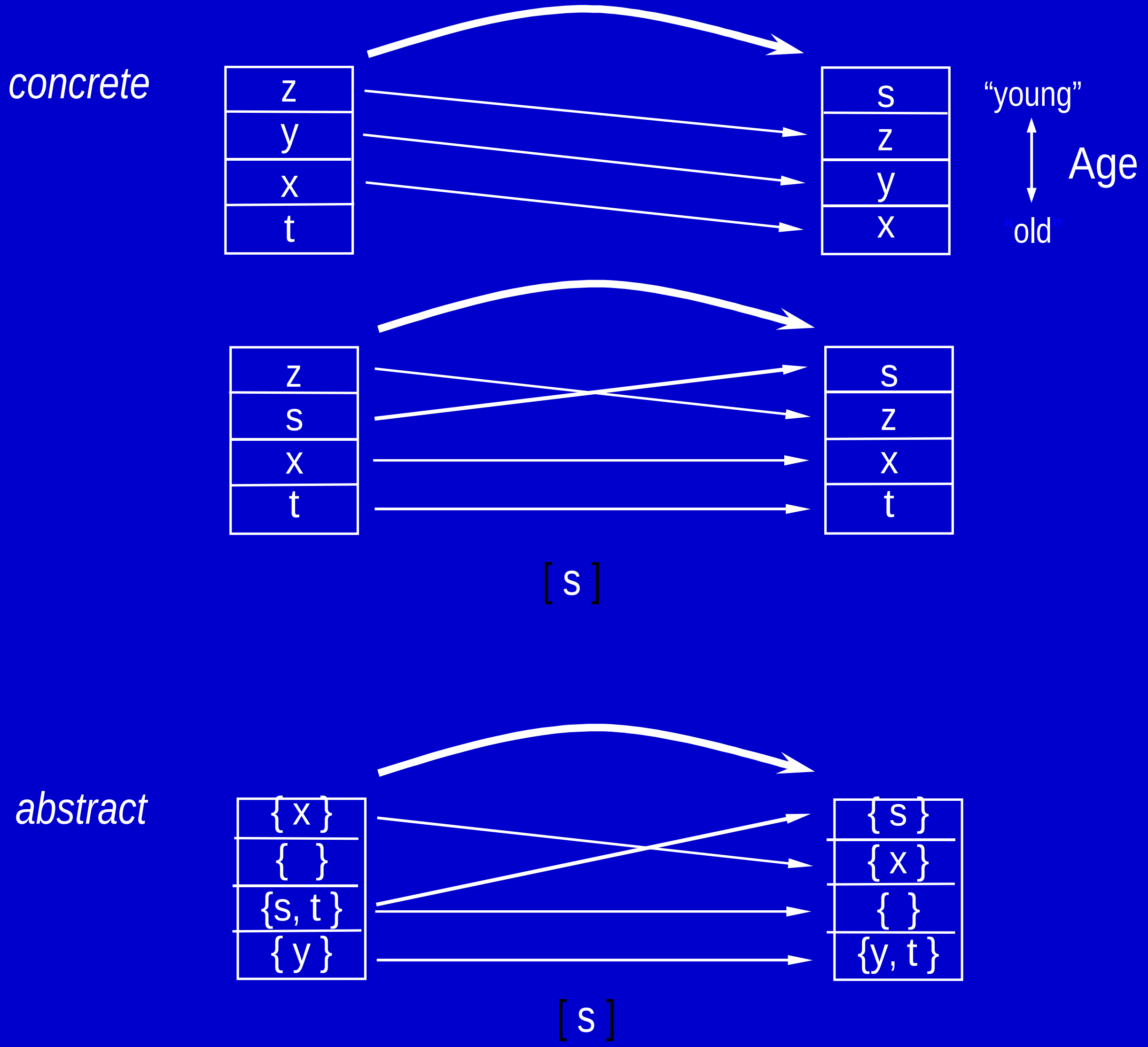


Abstract Domain: May Cache

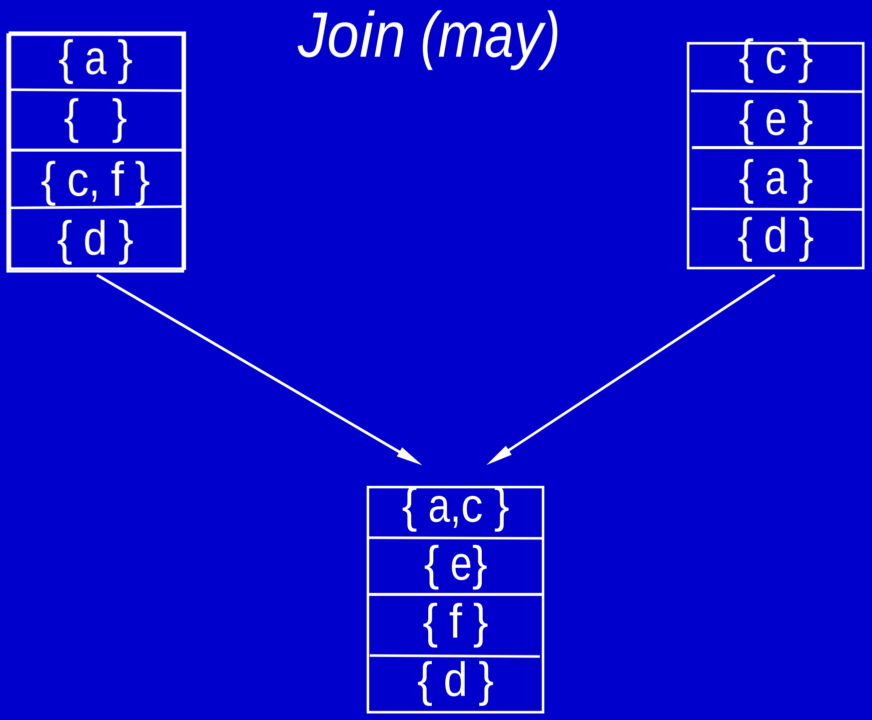
Concretization



Cache with LRU Replacement: Transfer for may



Cache Analysis: Join (may)



Join (may)

{ a }
{ }
{ c, f }
{ d }

{ c }
{ e }
{ a }
{ d }

{ a,c }
{ e}
{ f }
{ d }

"union
+ minimal age"

Cache Analysis

Approximation of the Collecting Semantics

$$\begin{tikzcd}
\text{the semantics} \arrow[r, "\text{determines}"] & \begin{array}{l}\text{set of all cache states}\\ \text{for each program point}\end{array} \arrow[d, phantom, sloped, "\supseteq"] \\
\text{“cache” semantics} \arrow[r, "\text{determines}"] & \begin{array}{l}\text{set of all cache states}\\ \text{for each program point}\end{array} \arrow[d, phantom, sloped, "\supseteq"] \arrow[d, phantom, near end, "\textit{conc}"] \\
\text{abstract semantics} \arrow[r, "\text{determines}", "\text{PAG}"'] & \begin{array}{l}\text{abstract cache states}\\ \text{for each program point}\end{array}
\end{tikzcd}$$

Deriving a Cache Analysis
- Reduction and Abstraction -

- Reducing the semantics (to what concerns caches)
  - e.g. from values to locations,
  - ignoring arithmetic.
  - obtain "auxiliary/instrumented" semantics
- Abstraction
  - Changing the domain: sets of memory blocks in single cache lines
- Design in these two steps is matter of engineering

Result of the Cache Analyses

Categorization of memory references

Category	Abb.	Meaning
always hit	**ah**	The memory reference will always result in a cache hit.
always miss	**am**	The memory reference will always result in a cache miss.
not classified	**nc**	The memory reference could neither be classified as **ah** nor **am**.

WCET: am
BCET: ah

Contribution to WCET

Information about cache contents sharpens timings.

```
while . . . do [max n]
  .
  .
  .
  ref to s
  .
  .
  .
od
```

time

t_{miss}

t_{hit}

loop time

$n * t_{miss}$

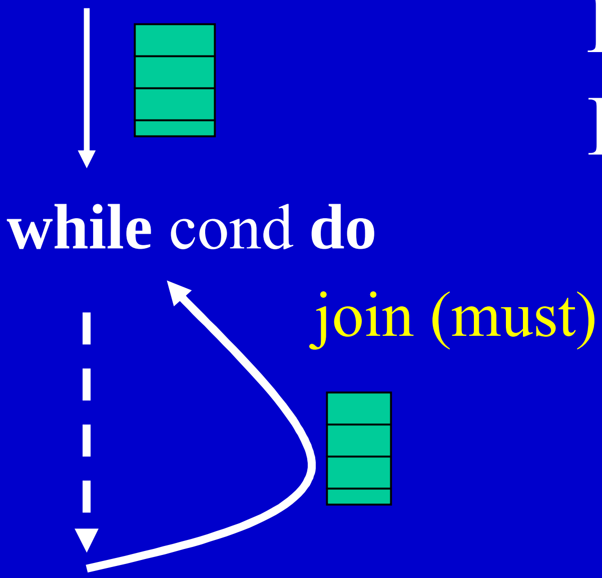
$n * t_{hit}$

$t_{miss} + (n - 1) * t_{hit}$

$t_{hit} + (n - 1) * t_{miss}$

Contexts

Cache contents depends on the Context, i.e. calls and loops



while cond **do**

join (must)

First Iteration loads the cache => Intersection looses most of the information!

Distinguish basic blocks by contexts

- Transform loops into tail recursive procedures
- Treat loops and procedures in the same way
- Use interprocedural analysis techniques, VIVU
  - virtual inlining of procedures
  - virtual unrolling of loops
- Distinguish as many contexts as useful
  - 1 unrolling for caches
  - 1 unrolling for branch prediction (pipeline)

Real-Life Caches

Processor	MCF 5307	MPC 750/755
Line size	16	32
Associativity	4	8
Replacement	Pseudo-round robin	Pseudo-LRU
Miss penalty	6 - 9	60 - 200

Real-World Caches I, the MCF 5307

- 128 sets of 4 lines each (4-way set-associative)
- Line size 16 bytes
- Pseudo Round Robin replacement strategy
- One! 2-bit replacement counter
- Hit or Allocate: Counter is neither used nor modified
- Replace:
  Replacement in the line as indicated by counter;
  Counter increased by 1 (modulo 4)

Example

Assume program accesses blocks 0, 1, 2, 3, …
starting with an empty cache
and block *i* is placed in cache set *i mod* 128

Accessing blocks 0 to 127:

counter = 0

Line 0	0	1	2	3	4	5	…	127
Line 1								
Line 2								
Line 3								

After accessing block 511: Counter still 0

Line 0	0	1	2	3	4	5	...	127
Line 1	128	129	130	131	132	133	...	255
Line 2	256	257	258	259	260	261	...	383
Line 3	384	385	386	387	388	389	...	511

After accessing block 639: Counter again 0

Line 0	512	1	2	3	516	5	...	127
Line 1	128	513	130	131	132	517	...	255
Line 2	256	257	514	259	260	261	...	383
Line 3	384	385	386	515	388	389	...	639

Lesson learned

- Memory blocks, even useless ones, may remain in the cache
- The worst case is not the empty cache, but a cache full of junk (blocks not accessed)!
- Assuming the cache to be empty at program start is unsafe!

Cache Analysis for the MCF 5307

- Modeling the counter: Impossible!
  - Counter stays the same or is increased by 1
  - Sometimes this is unknown
  - After 3 unknown actions: all information lost!
- May analysis: never anything removed! => useless!
- Must analysis: replacement removes all elements from set and inserts accessed block => set contains at most one memory block

Cache Analysis for the MCF 5307

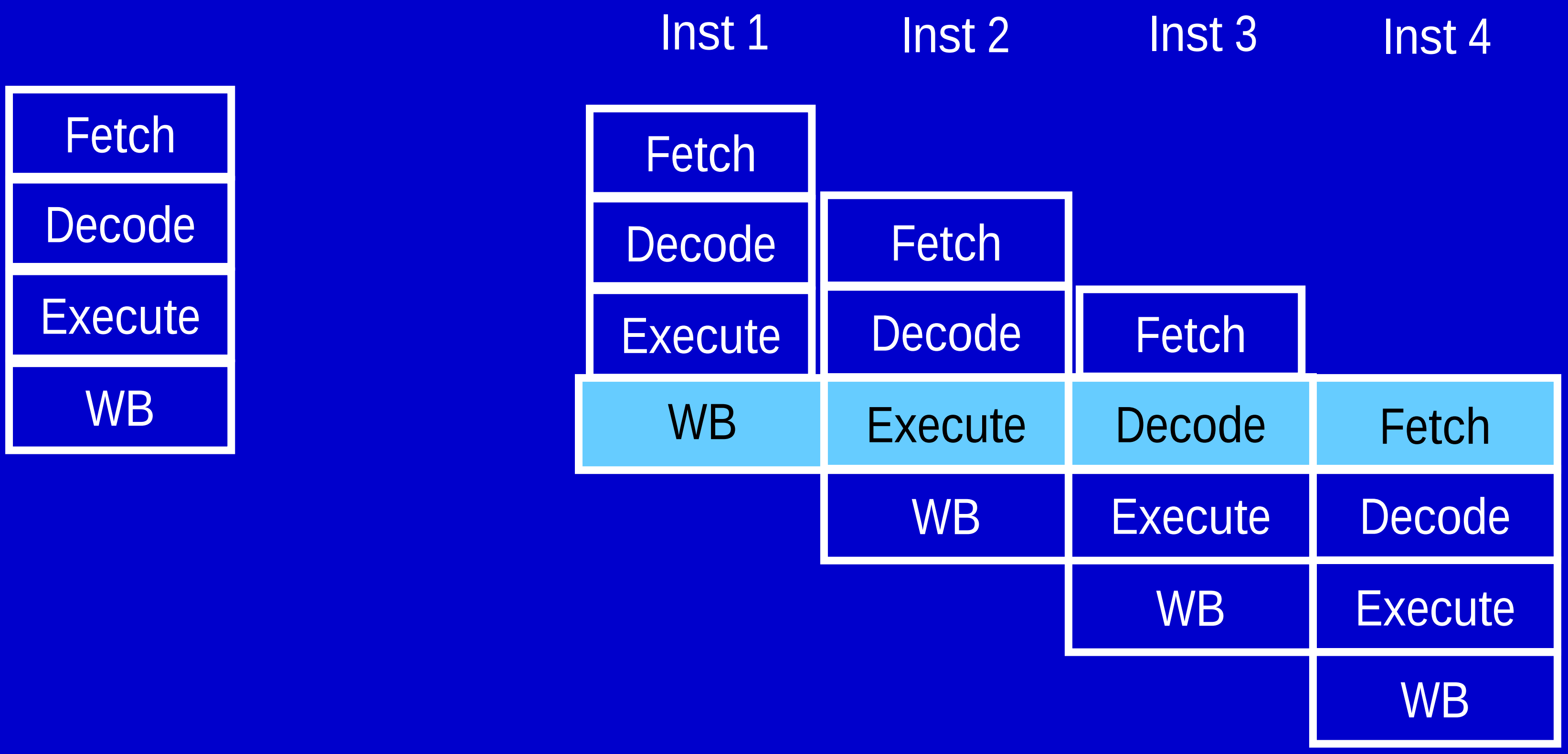
- Abstract cache contains at most one block per line
- Corresponds to direct mapped cache
- Only ¼ of capacity
- As for predictability, ¾ of capacity are lost!
- In addition: Uniform cache => instructions and data evict each other

Results of Cache Analysis

- Annotations of memory accesses (in contexts) with Cache Hit: Access will always hit the cache Cache Miss: Access will never hit the cache Unknown: We can’t tell

Pipelines

Hardware Features: Pipelines



Fetch
Decode
Execute
WB

Inst 1	Inst 2	Inst 3	Inst 4
Fetch			
Decode	Fetch		
Execute	Decode	Fetch	
WB	Execute	Decode	Fetch
	WB	Execute	Decode
		WB	Execute
			WB

Ideal Case: 1 Instruction per Cycle

Hardware Features: Pipelines II

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

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

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

IF	🟥	🟨					
EX		🟥	🟨	🟨			
M			🟥		🟨	🟨	🟨
F							

CPU as a (Concrete) State Machine

- Processor (pipeline, cache, memory, inputs) viewed as a *big* state machine, performing transitions every clock cycle
- Starting in an initial state for an instruction transitions are performed, until a final state is reached:
  - End state: instruction has left the pipeline
  - # transitions: execution time of instruction

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

An Abstract Pipeline Executing a Basic Block

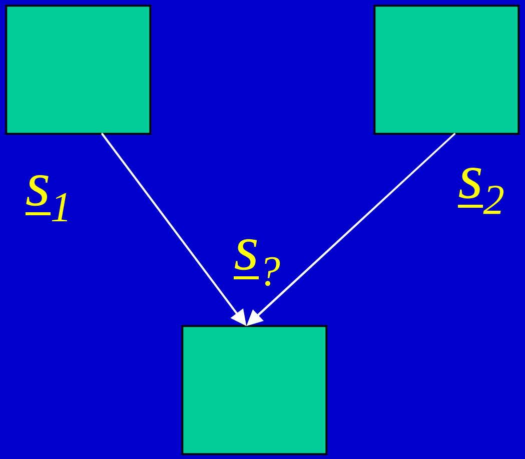
function exec (*b* : basic block, *s* : abstract pipeline state)
t: trace

interprets instruction stream of *b* (annotated with cache information) starting in state *s* producing trace *t*

length(t) gives number of cycles

What is different?

- 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).
- Starting state for successor basic block?
  In particular, if there are several predecessor blocks.



Alternatives:

- *sets of states*
- *combine by least upper bound*

(Concrete) Instruction Execution

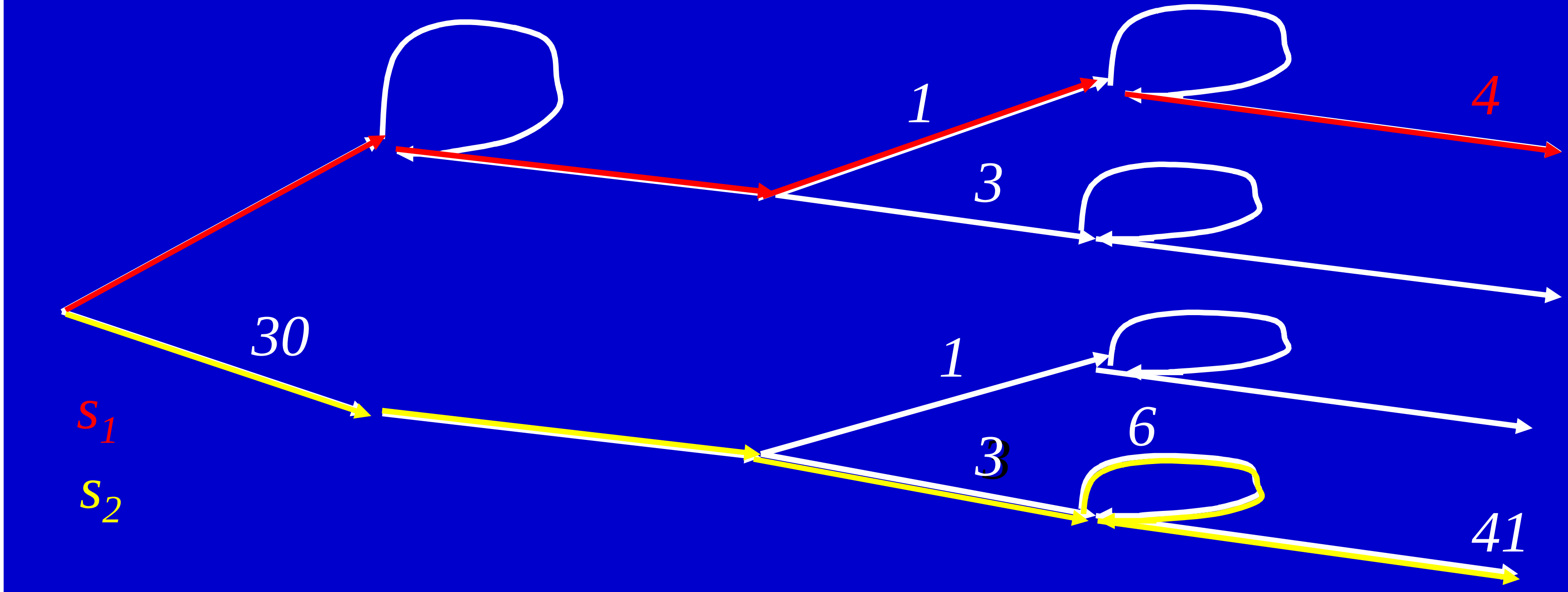
`mul`

Fetch
I-Cache miss?

Issue
Unit occupied?

Execute
Multicycle?

Retire
Pending instructions?



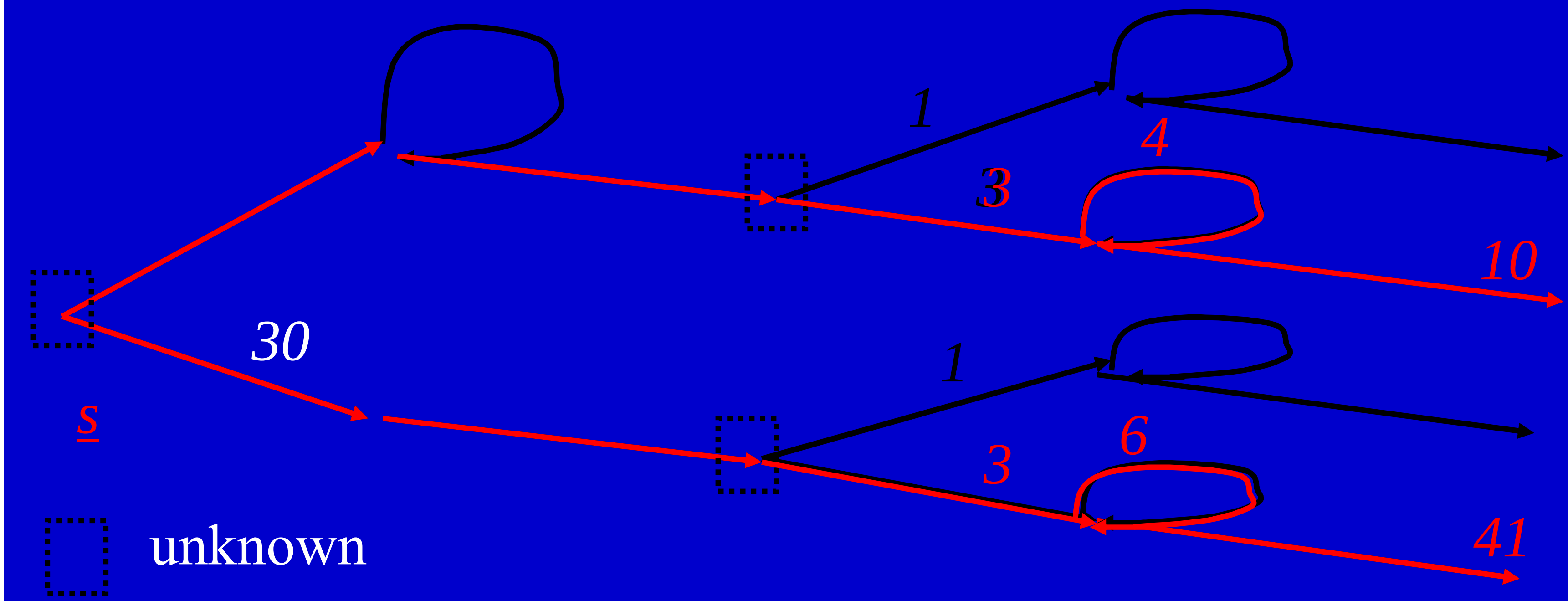
Abstract Instruction-Execution

Fetch
I-Cache miss?

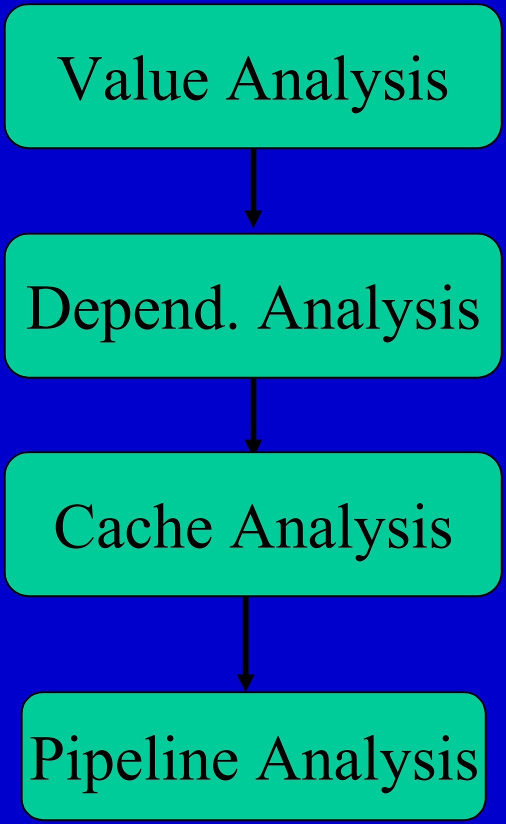
Issue
Unit occupied?

Execute
Multicycle?

Retire
Pending instructions?



A Modular Process



Value Analysis

Depend. Analysis

Cache Analysis

Pipeline Analysis

Static determ. of effective addresses

Elim. of true data dependences (for safe elim. of data hazards)

Annotation of instructions with Hit

Safe abstract execution based on the available static information

Corresponds to the Following Sequence of Steps

1. Value analysis
2. Cache analysis using statically computed effective addresses and loop bounds
3. 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!

Surprises may lurk in the Future!

- Interference between processor components produces Timing Anomalies:
  - Assuming local good case leads to higher overall execution time ⇒ risk for WCET
  - Assuming local bad case leads to lower overall execution time ⇒ risk for BCET

    Ex.: Cache miss preventing branch misprediction
- Treating components in isolation may be unsafe

Non-Locality of Local Contributions

- Interference between processor components produces Timing Anomalies: Assuming local best case leads to higher overall execution time.

  Ex.: Cache miss in the context of branch prediction
- Treating components in isolation maybe unsafe
- Implicit assumptions are not always correct:
  - Cache miss is not always the worst case!
  - The empty cache is not always the worst case

An Abstract Pipeline Executing a Basic Block - processor with timing anomalies -

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

Analysis states = $2^{\underline{PS} \times \underline{CS}}$

PS = set of abstract pipeline states

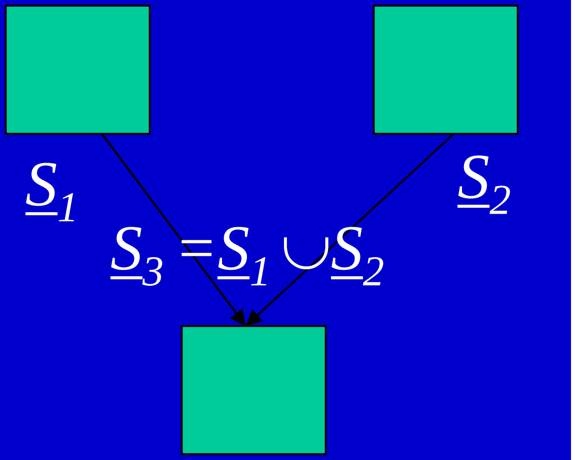
CS = set of abstract cache states

interprets instruction stream of *b* (annotated with cache information) starting in state *S* producing set of traces *T*

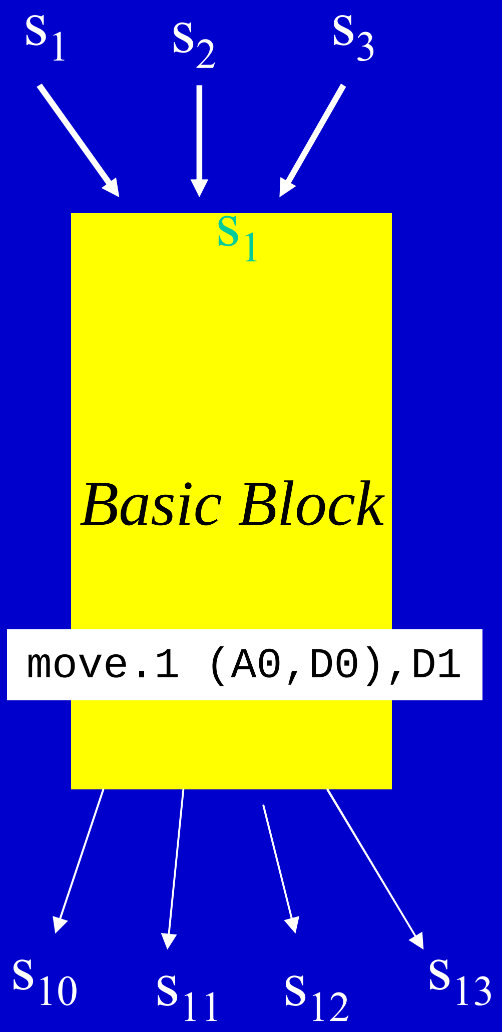
max(length(T)) - upper bound for execution time

last(T) - set of initial states for successor block

Union for blocks with several predecessors.



Integrated Analysis: Overall Picture

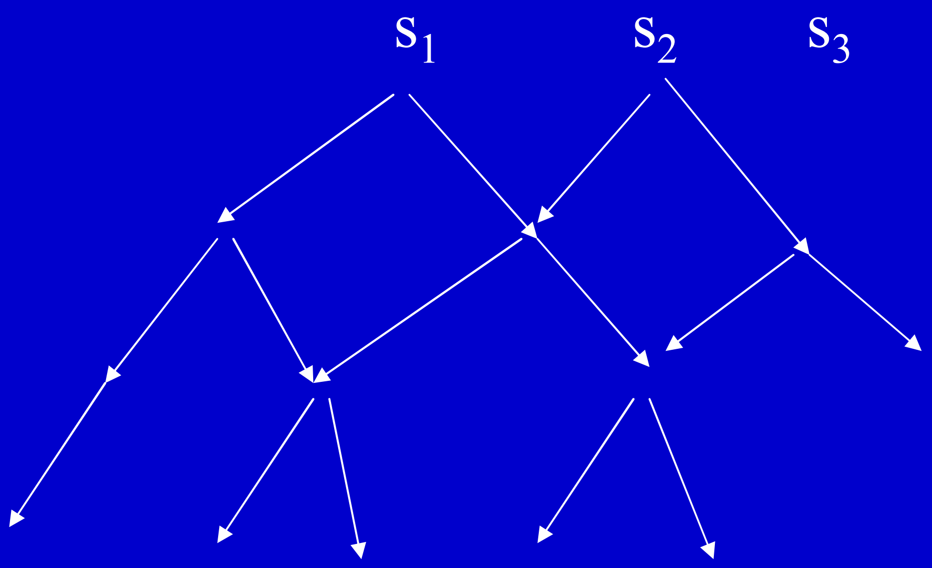


Basic Block

```
move.1 (A0,D0),D1
```

Fixed point iteration over Basic Blocks (in context) $\{s_1, s_2, s_3\}$ abstract state

Cyclewise evolution of processor model for instruction



Benchmarks

Interpretation

- Airbus' results obtained with legacy method:
  measurement for blocks, tree-based composition, added safety margin
- ~30% overestimation
- aiT's results were between real worst-case execution times and Airbus' results

MCF 5307: Results

- The value analyzer is able to predict around 70-90% of all data accesses precisely (Airbus Benchmark)
- The cache/pipeline analysis takes reasonable time and space on the Airbus benchmark
- The predicted times are close to or better than the ones obtained through convoluted measurements
- Results are visualized and can be explored interactively

Pipeline Modeling

How to Create a Pipeline Analysis?

- Starting point: Concrete model of execution
- First build reduced model
  - E.g. forget about the store, registers etc.
- Then build abstract timing model
  - Change of domain to abstract states, i.e. sets of (reduced) concrete states
  - Conservative in execution times of instructions

Defining the Concrete State Machine

How to define such a complex state machine?

- A state consists of (the state of) internal components
  (register contents, fetch/ retirement queue contents...)
- Combine internal components into units (modularisation, cf. VHDL/Verilog)
- Units communicate via signals
- (Big-step) Transitions via unit-state updates and signal sends and receives

An Example: MCF5307

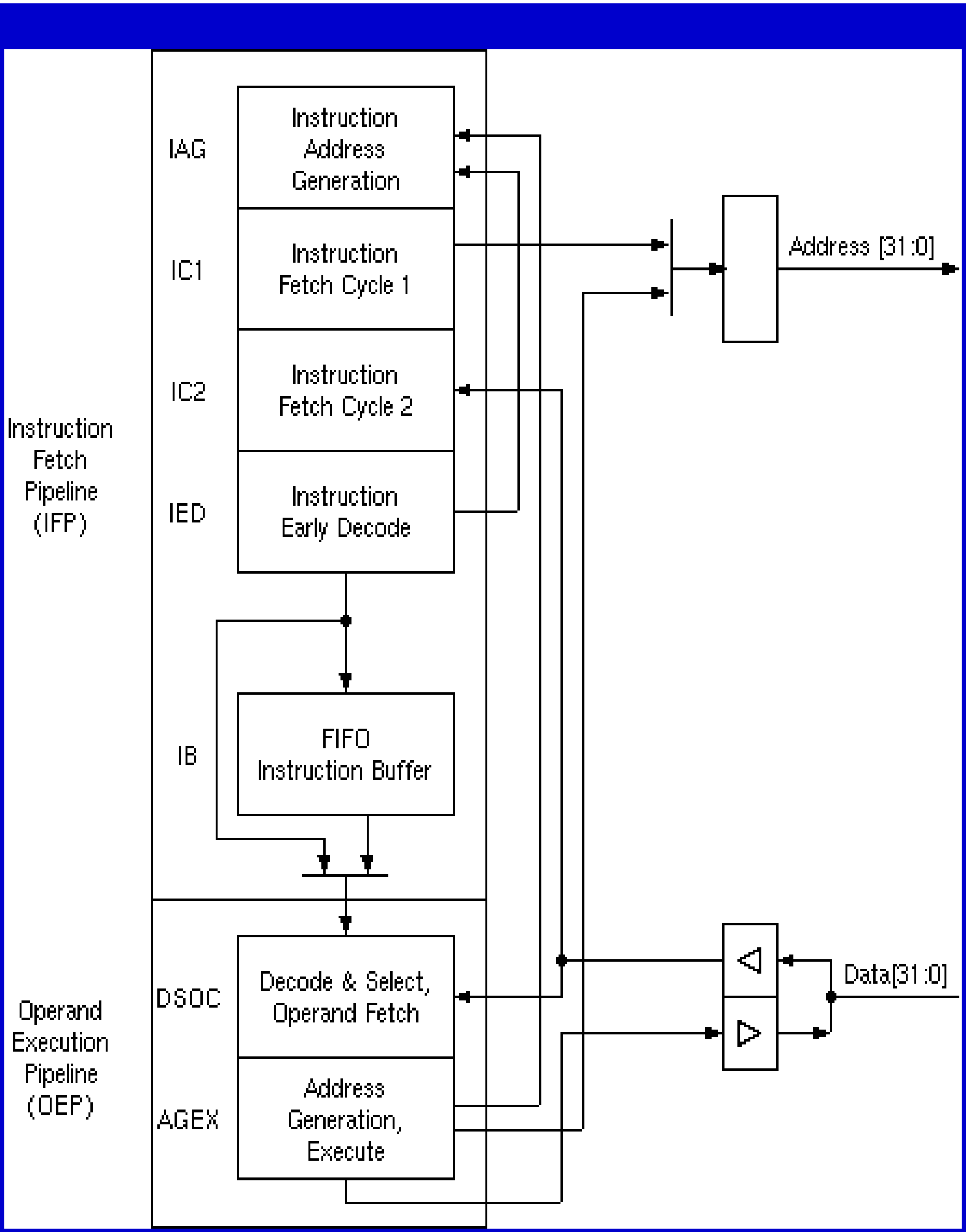
- MCF 5307 is a V3 Coldfire family member
- Coldfire is the successor family to the M68K processor generation
- Restricted in instruction size, addressing modes and implemented M68K opcodes
- MCF 5307: small and cheap chip with integrated peripherals
- Separated but coupled bus/core clock frequencies

ColdFire Pipeline

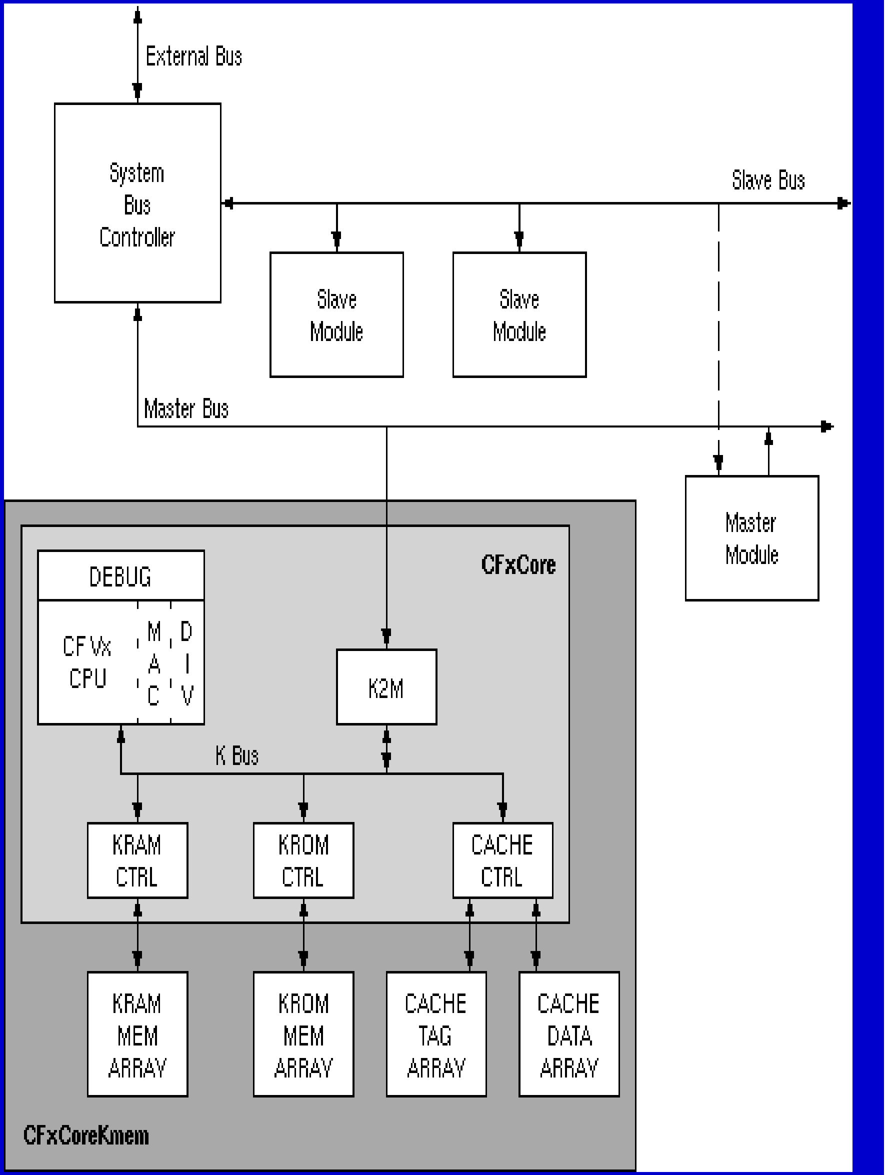
The ColdFire pipeline consists of

- a Fetch Pipeline of 4 stages
  - Instruction Address Generation (IAG)
  - Instruction Fetch Cycle 1 (IC1)
  - Instruction Fetch Cycle 2 (IC2)
  - Instruction Early Decode (IED)
- an Instruction Buffer (IB) for 8 instructions
- an Execution Pipeline of 2 stages
  - Decoding and register operand fetching (1 cycle)
  - Memory access and execution (1 – many cycles)

- Two coupled pipelines
- Fetch pipeline performs branch prediction
- Instruction executes in up two to iterations through OEP
- Coupling FIFO buffer with 8 entries
- Pipelines share same bus
- Unified cache

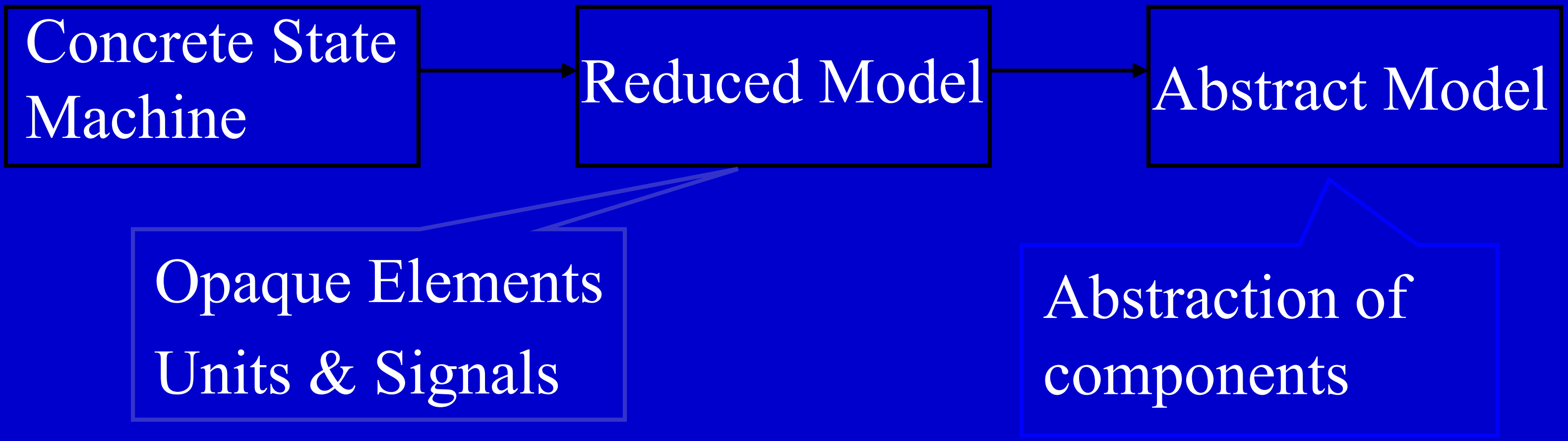

Instruction Fetch Pipeline (IFP)
Operand Execution Pipeline (OEP)
IAG
IC1
IC2
IED
IB
DSOC
AGEX
Instruction Address Generation
Instruction Fetch Cycle 1
Instruction Fetch Cycle 2
Instruction Early Decode
FIFO Instruction Buffer
Decode & Select, Operand Fetch
Address Generation, Execute
Address [31:0]
Data[31:0]

- Hierarchical bus structure
- Pipelined K- and M-Bus
- Fast K-Bus to internal memories
- M-Bus to integrated peripherals
- E-Bus to external memory
- Busses independent
- Bus unit: K2M, SBC, Cache

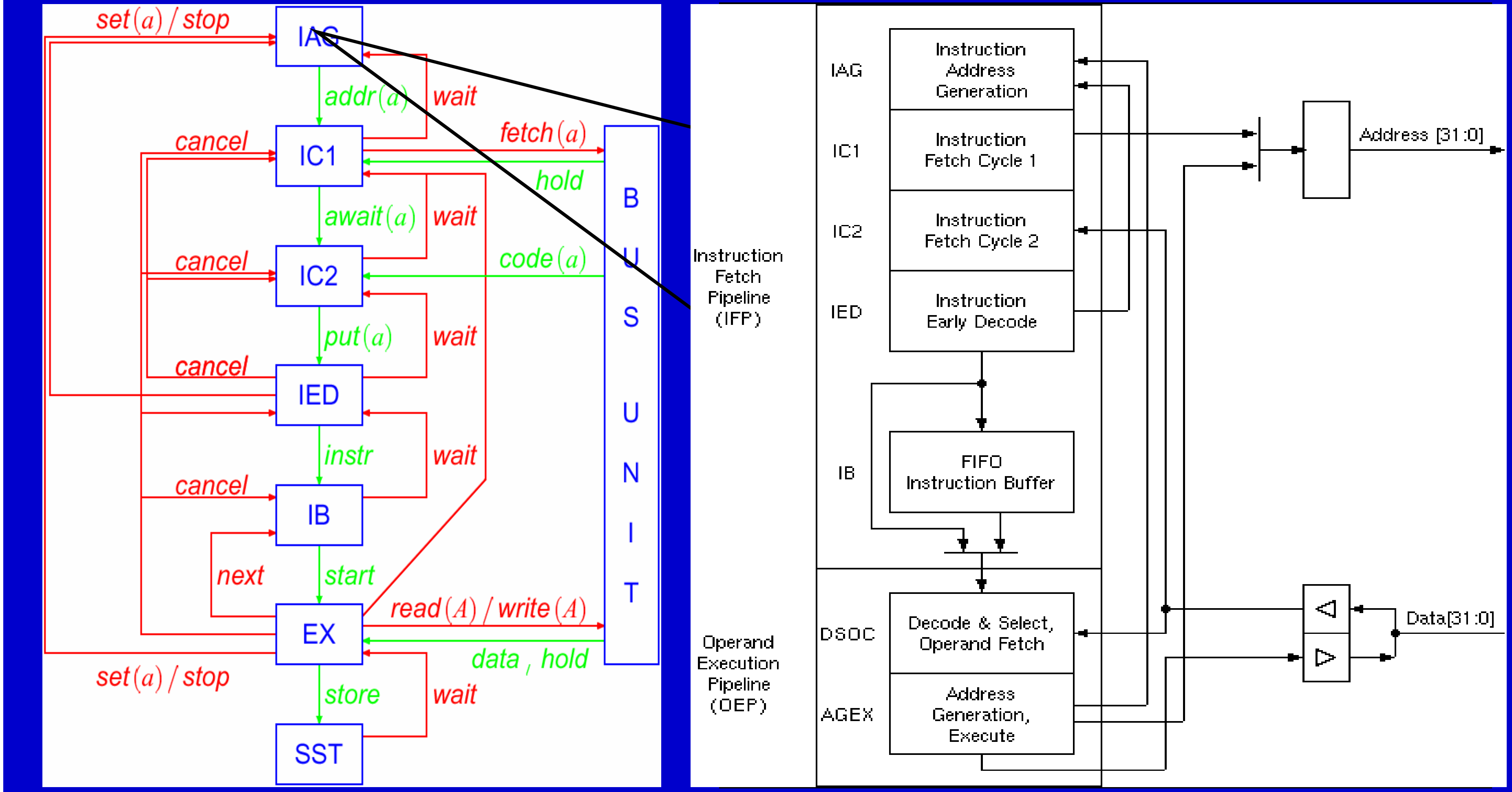


Model with Units and Signals

Opaque components - not modeled: thrown away in the analysis (e.g. registers up to memory accesses)



Model for the MCF 5307



Abstraction

- We abstract reduced states
  - Opaque components are thrown away
  - Caches are abstracted as described
  - Signal parameters: abstracted to memory address ranges or unchanged
  - Other components of units are taken over unchanged
- Cycle-wise update is kept, but
  - transitions depending on opaque components before are now non-deterministic
  - same for dependencies on unknown values

Nondeterminism

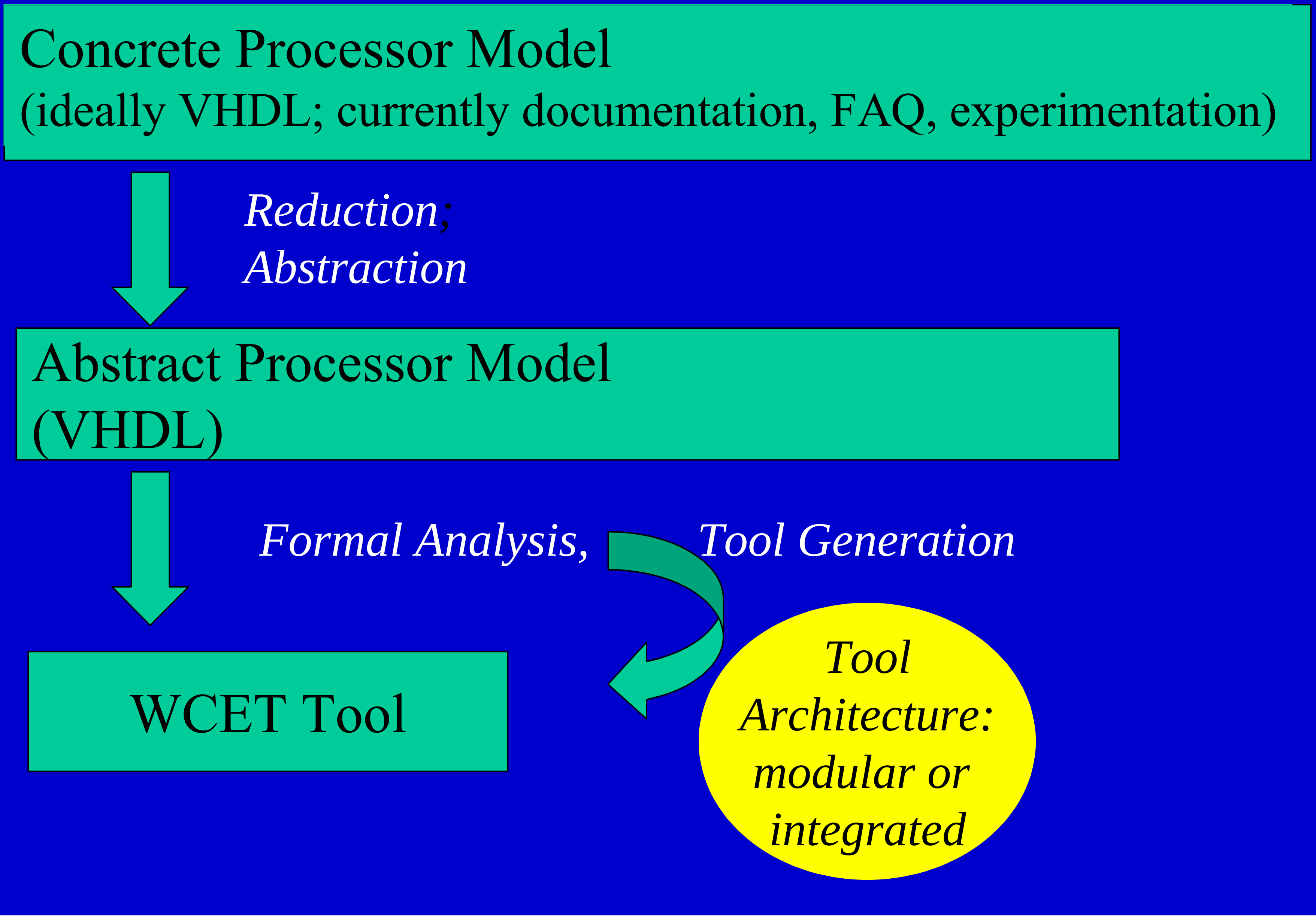
- In the reduced model, one state resulted in one new state after a one-cycle transition
- Now, one state can have several successor states
  - Transitions from set of states to set of states

Implementation

- Abstract model is implemented as a DFA
- Instructions are the nodes in the CFG
- Domain is powerset of set of abstract states
- Transfer functions at the edges in the CFG iterate cycle-wise updating each state in the current abstract value
- max { *# iterations for all states*} gives WCET
- From this, we can obtain WCET for basic blocks

Tool Architecture

The Tool-Construction Process



Concrete Processor Model
(ideally VHDL; currently documentation, FAQ, experimentation)

Reduction;
Abstraction

Abstract Processor Model
(VHDL)

Formal Analysis,

Tool Generation

WCET Tool

Tool Architecture: modular or integrated

Why integrated analyses?

- Simple modular analysis not possible for architectures with unbounded interference between processor components
- Timing anomalies (Lundquist/Stenström):
  - Faster execution locally assuming penalty
  - Slower execution locally removing penalty
- Domino effect: Effect only bounded in length of execution

Integrated Analysis

- Goal: calculate all possible abstract processor states at each program point (in each context) Method: perform a cyclewise evolution of abstract processor states, determining all possible successor states
- Implemented from an abstract model of the processor:
  the pipeline stages and communication between them
- Results in WCET for basic blocks

Timing Anomalies

Let Δ_{Tl} be an execution-time difference between two different cases for an instruction,

Δ_{Tg} the resulting difference in the overall execution time.

A Timing Anomaly occurs if either

- $\Delta_{Tl} < 0$: the instruction executes faster, and
  - $\Delta_{Tg} < \Delta_{T1}$: the overall execution is yet faster, or
  - $\Delta_{Tg} > 0$: the program runs longer than before.
- $\Delta_{Tl} > 0$: the instruction takes longer to execute, and
  - $\Delta_{Tg} > \Delta_{Tl}$: the overall execution is yet slower, or
  - $\Delta_{Tg} < 0$: the program takes less time to execute than before

Timing Anomalies

$\Delta_{Tl} < 0$ and $\Delta_{Tg} > 0$:

Local timing merit causes global timing penalty

is critical for WCET:

using local timing-merit assumptions is unsafe

$\Delta_{Tl} > 0$ and $\Delta_{Tg} < 0$:

Local timing penalty causes global speed up

is critical for BCET:

using local timing-penalty assumptions is unsafe

Timing Anomalies - Remedies

- For each local Δ_{Tl} there is a corresponding set of global Δ_{Tg}
  Add upper bound of this set to each local Δ_{Tl} in a

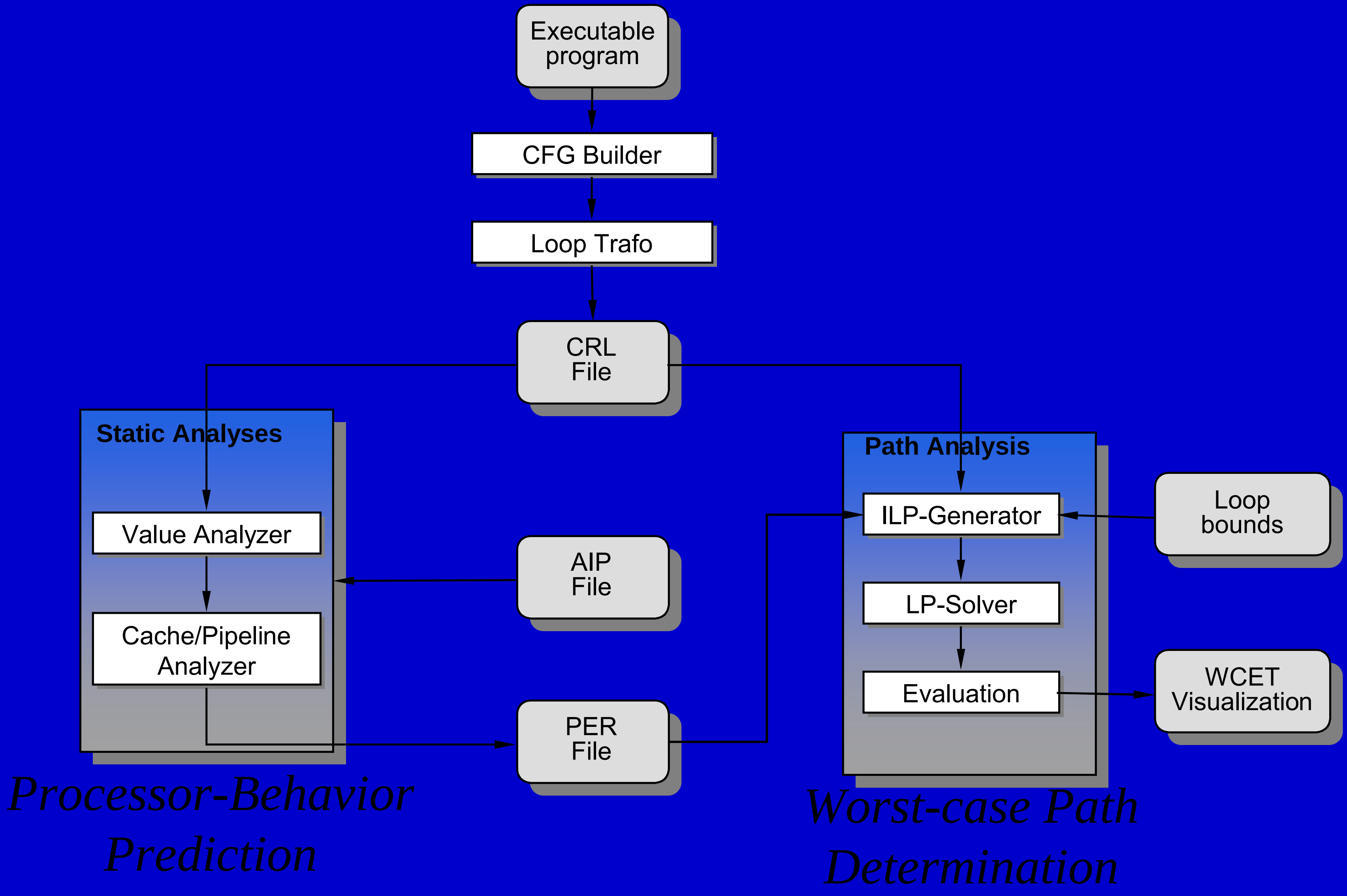
  Problem Bound may not exist
  Domino Effect: anomalous effect increases with the size of the program (loop).
  Domino Effect on PowerPC (Diss. J. Schneider)
- Follow all possible scenarios in an

Integrated Analysis II

is a set of (reduced) concrete processor states, computed:

- Sets are small, pipeline is not too history sensitive
- Joins are set union

Overall Structure



Path Analysis

by Integer Linear Programming (ILP)

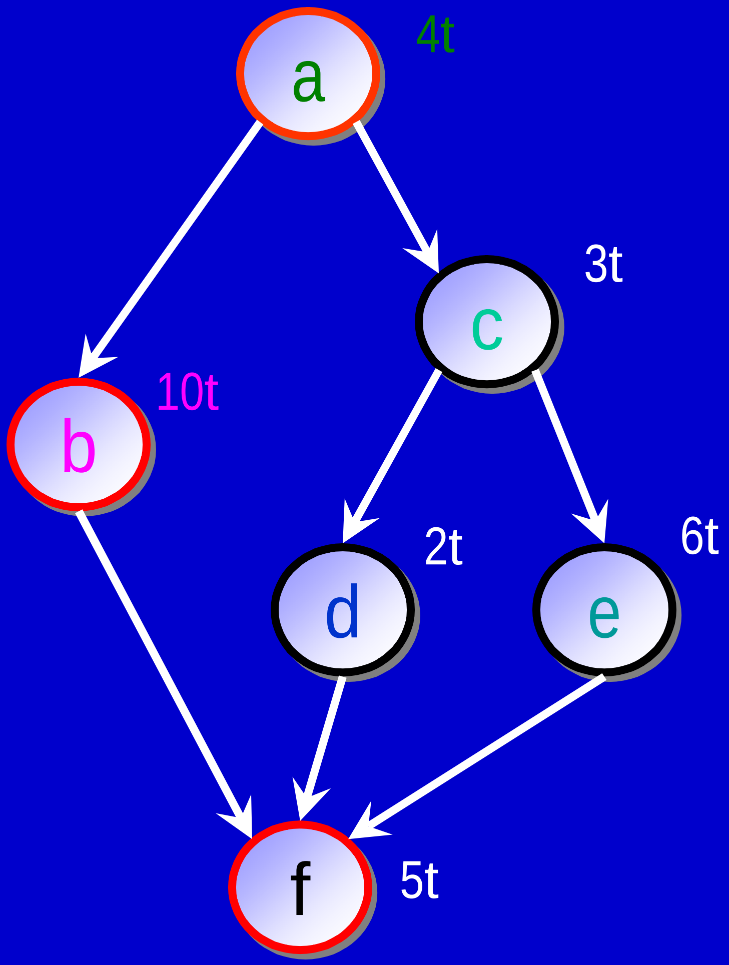
- Execution time of a program =

$$\sum_{\text{Basic\_Block } b} \text{Execution\_Time}(b) \times \text{Execution\_Count}(b)$$

- ILP solver maximizes this function to determine the WCET
- Program structure described by linear constraints
  - automatically created from CFG structure
  - user provided loop/recursion bounds
  - arbitrary additional linear constraints to exclude infeasible paths

Example (simplified constraints)

```
if  a  then
    b
elseif  c  then
    d
else
    e
endif
f
```



max: $4\,x_a + 10\,x_b + 3\,x_c + 2\,x_d + 6\,x_e + 5\,x_f$

where

$$x_a = x_b + x_c$$

$$x_c = x_d + x_e$$

$$x_f = x_b + x_d + x_e$$

$$x_a = 1$$

Value of objective function: 19

x_a	1
x_b	1
x_c	0
x_d	0
x_e	0
x_f	1

Analysis Results (Airbus Benchmark)

Task	Airbus' method	**aiT**'s results	precision improvement
1	6.11 ms	5.50 ms	10.0 %
2	6.29 ms	5.53 ms	12.0 %
3	6.07 ms	5.48 ms	9.7 %
4	5.98 ms	5.61 ms	6.2 %
5	6.05 ms	5.54 ms	8.4 %
6	6.29 ms	5.49 ms	12.7 %
7	6.10 ms	5.35 ms	12.3 %
8	5.99 ms	5.49 ms	8.3 %
9	6.09 ms	5.45 ms	10.5 %
10	6.12 ms	5.39 ms	11.9 %
11	6.00 ms	5.19 ms	13.5 %
12	5.97 ms	5.40 ms	9.5 %

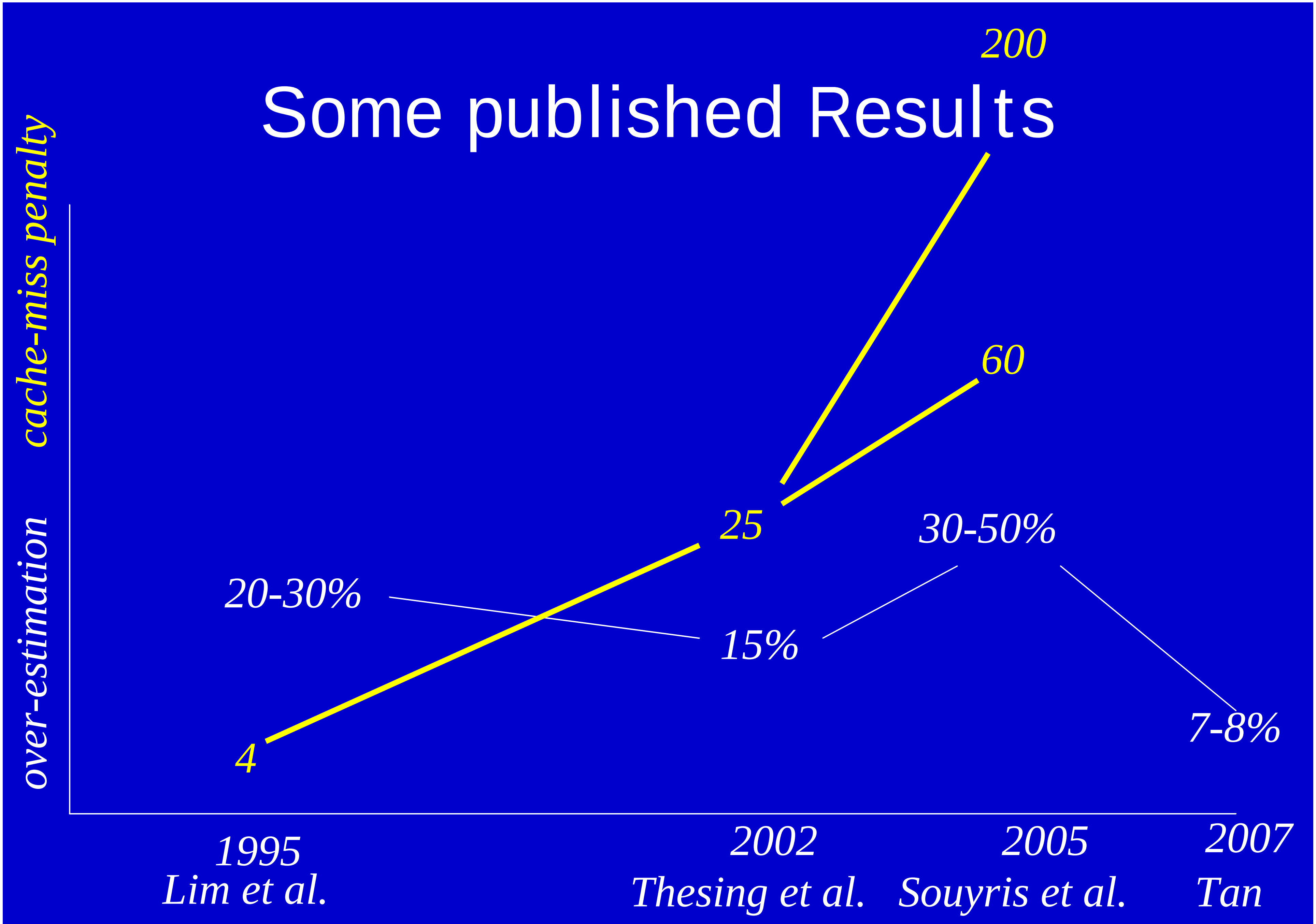
Interpretation

- Airbus’ results obtained with legacy method:
  measurement for blocks, tree-based composition, added safety margin
- ~30% overestimation
- aiT’s results were between real worst-case execution times and Airbus’ results

Reasons for Success

- C code synthesized from SCADE specifications
- Very disciplined code
  - No pointers, no heap
  - Few tables
  - Structured control flow
- However, very badly designed processor!

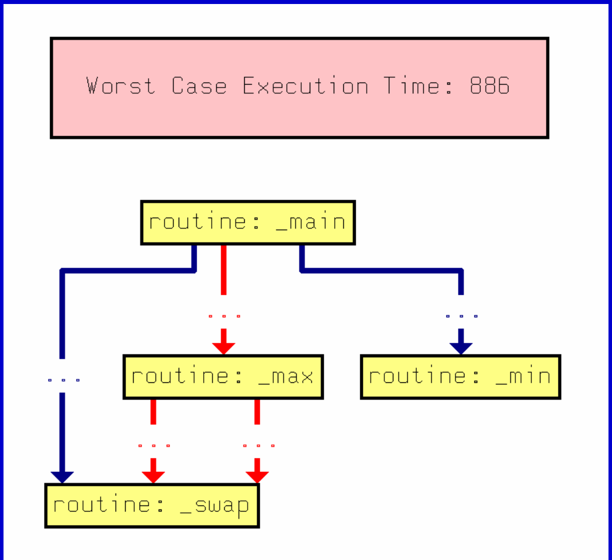
Some published Results



Conclusion II

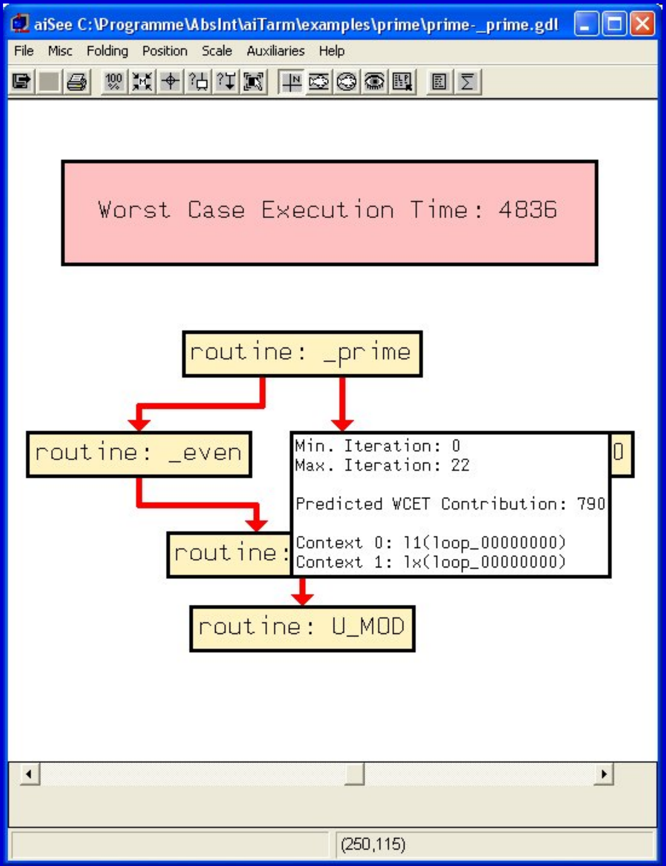
THE EUROPEAN IST PRIZE WINNER

- aiT enables development of complex hard real-time systems on state-of-the-art hardware.
- Increases safety.
- Saves development time.
- Precise timing predictions enable the most cost-efficient hardware to be chosen.



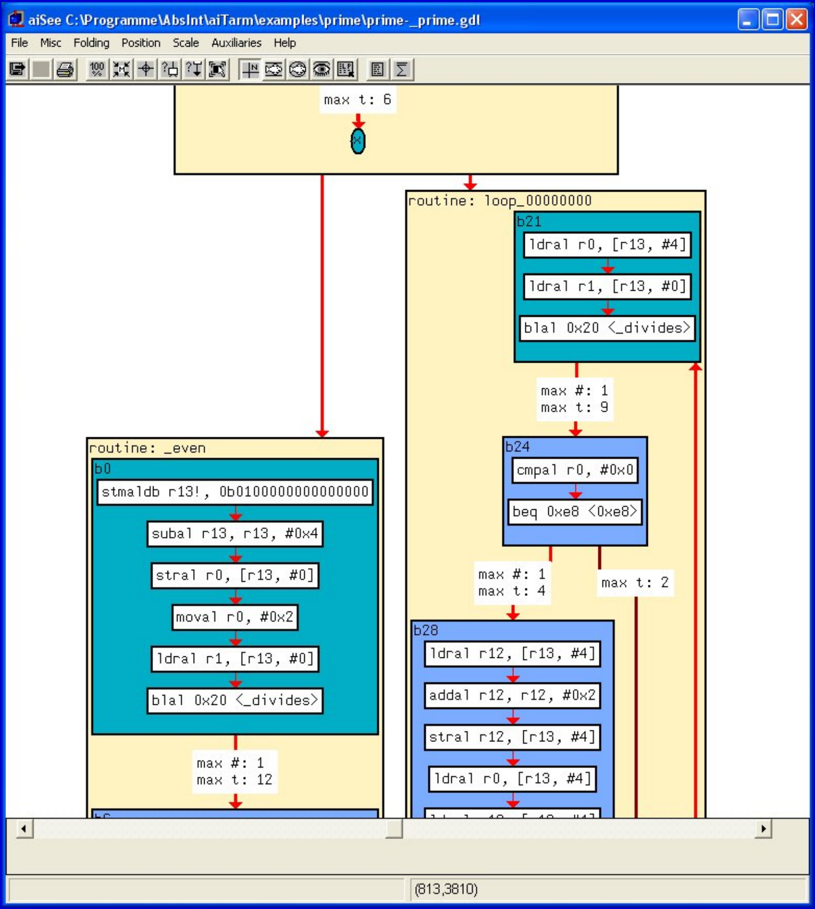
Worst Case Execution Time: 886

aiT: Timing Details



Worst Case Execution Time: 4836

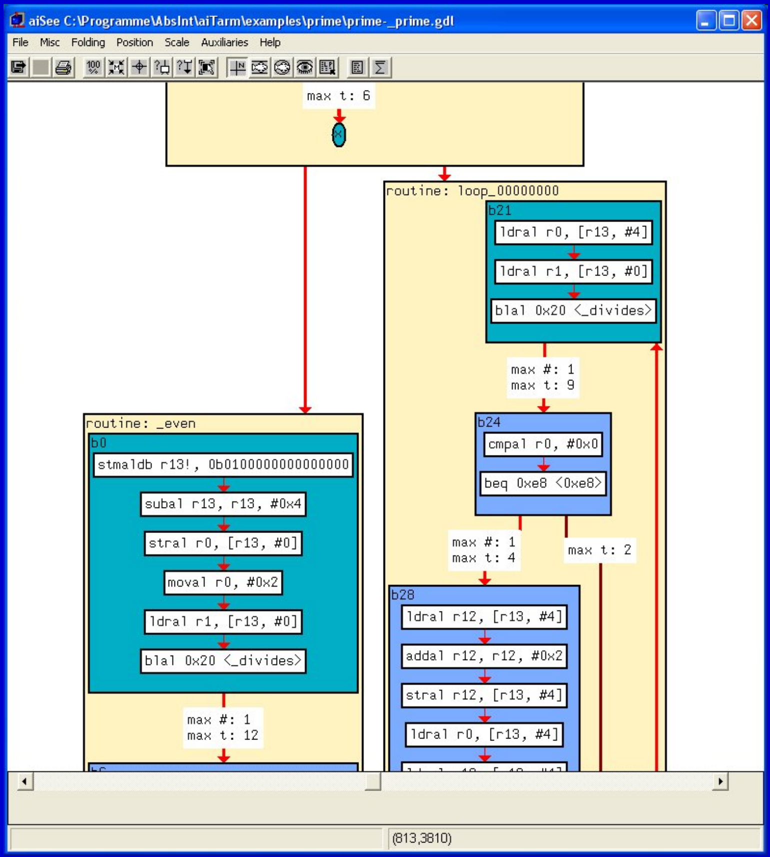
```
Min. Iteration: 0
Max. Iteration: 22

Predicted WCET Contribution: 790

Context 0: 1l(loop_00000000)
Context 1: 1x(loop_00000000)
```



```
b21
ldral r0, [r13, #4]
ldral r1, [r13, #0]
blal 0x20 <_divides>

b24
cmpal r0, #0x0
beq 0xe8 <0xe8>

b28
ldral r12, [r13, #4]
addal r12, r12, #0x2
stral r12, [r13, #4]
ldral r0, [r13, #4]

b0
stmaldb r13!, 0b0100000000000000
subal r13, r13, #0x4
stral r0, [r13, #0]
moval r0, #0x2
ldral r1, [r13, #0]
blal 0x20 <_divides>
```

aiT WCET Analyzer

Visualization

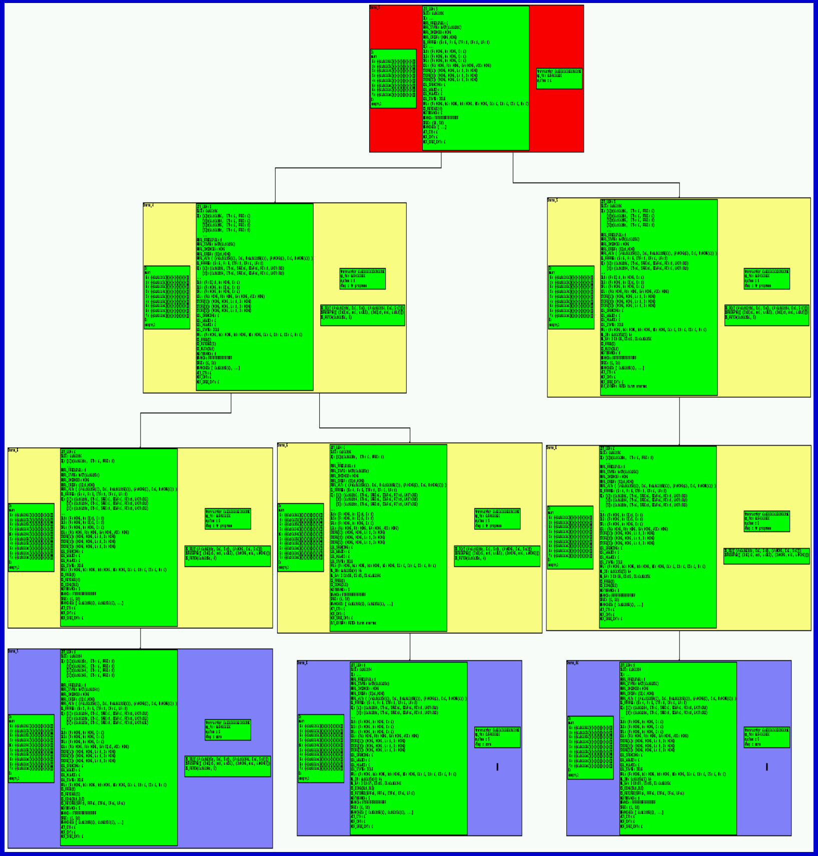


```
ldral r0, [r13, #4]
ldral r1, [r13, #0]
blal 0x20 <_divides>
```

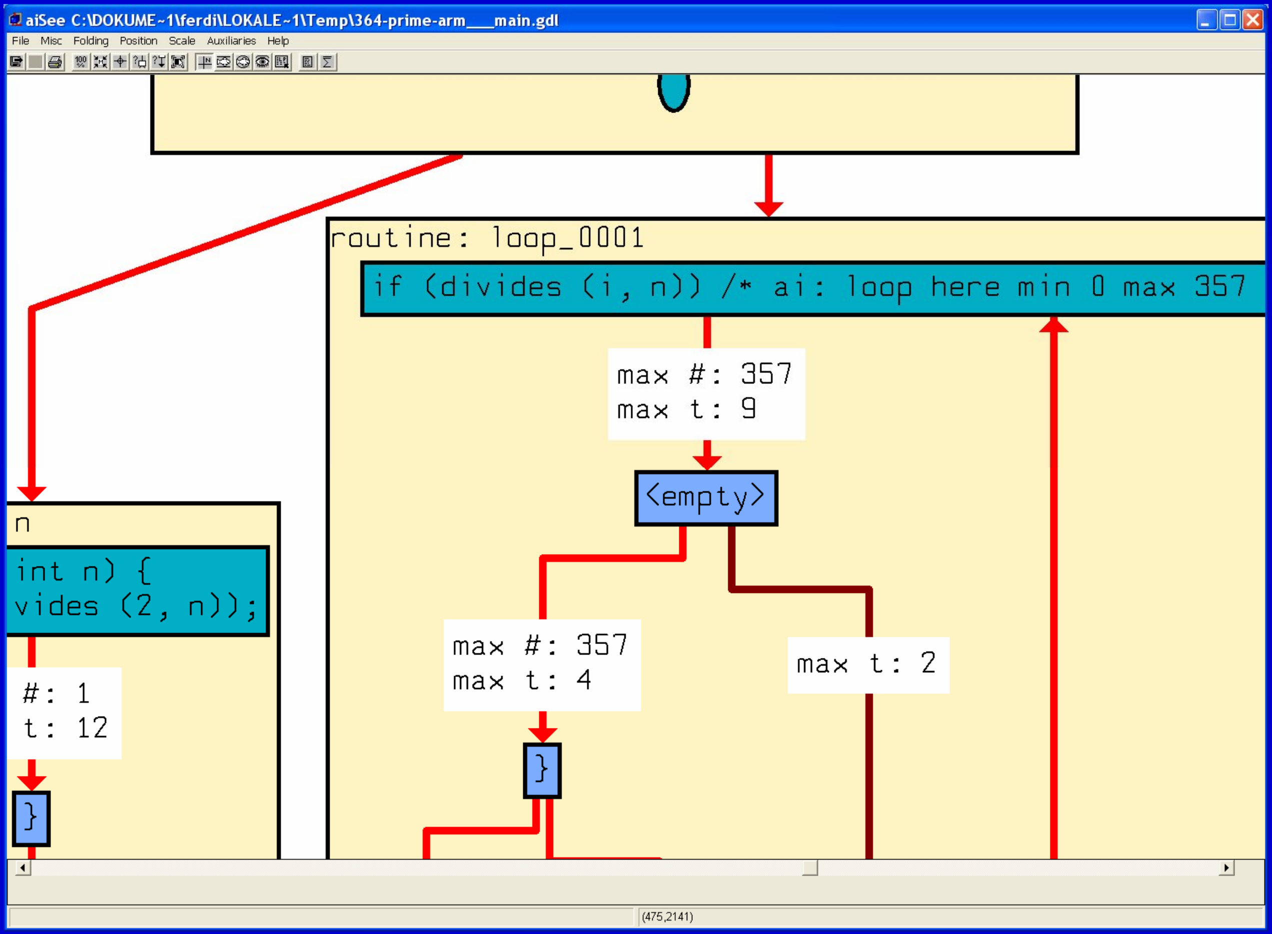
```
cmpal r0, #0x0
beq 0xe8 <0xe8>
```

```
ldral r12, [r13, #4]
addal r12, r12, #0x2
stral r12, [r13, #4]
ldral r0, [r13, #4]
```

```
stmaldb r13!, 0b0100000000000000
subal r13, r13, #0x4
stral r0, [r13, #0]
moval r0, #0x2
ldral r1, [r13, #0]
blal 0x20 <_divides>
```



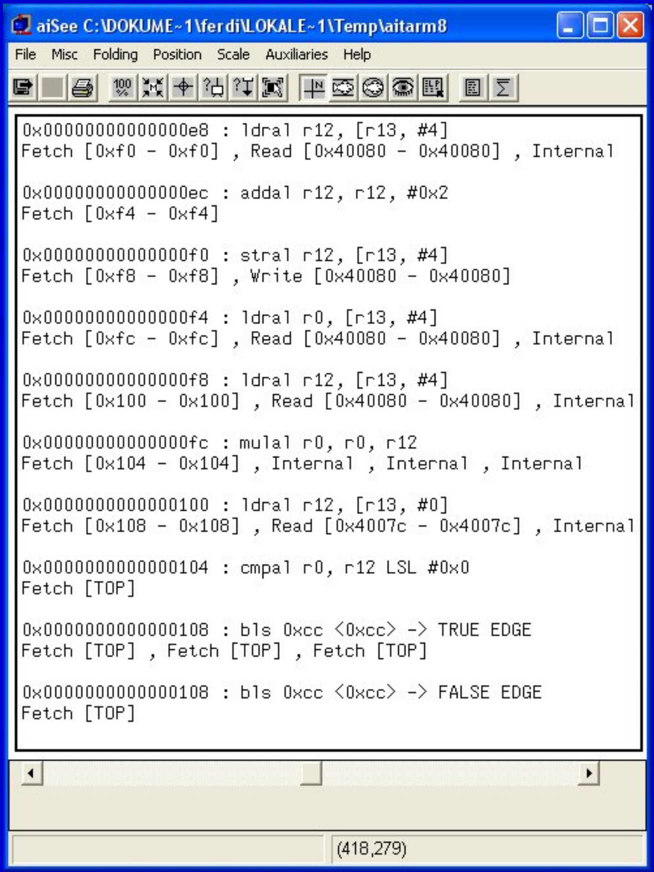
aiT: Timing Details


aiSee C:\DOKUME~1\ferdi\LOKALE~1\Temp\364-prime-arm___main.gdl
File
Misc
Folding
Position
Scale
Auxiliaries
Help
(475,2141)

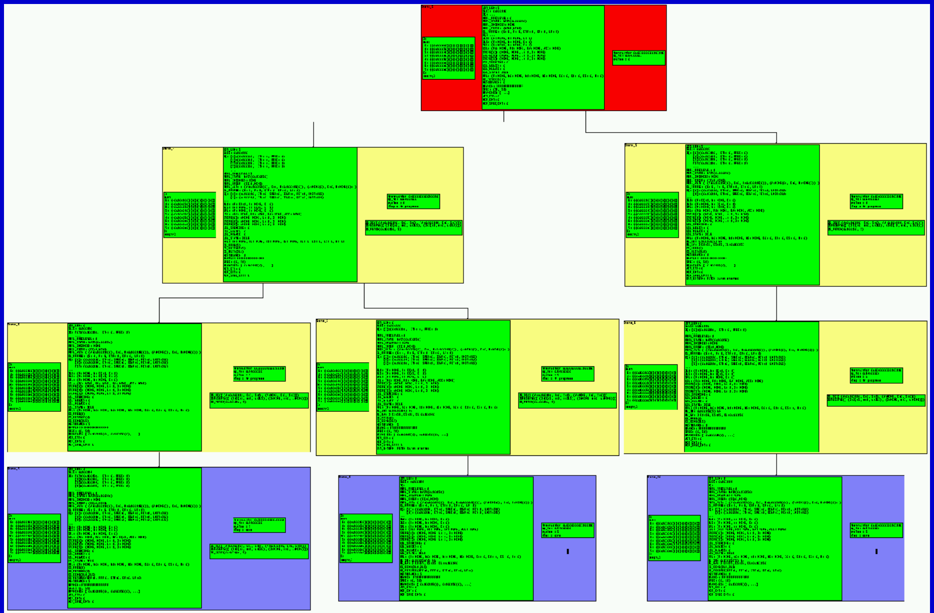
```
routine: loop_0001
if (divides (i, n)) /* ai: loop here min 0 max 357
```

```
int n) {
vides (2, n));
```

aiT: Timing Details

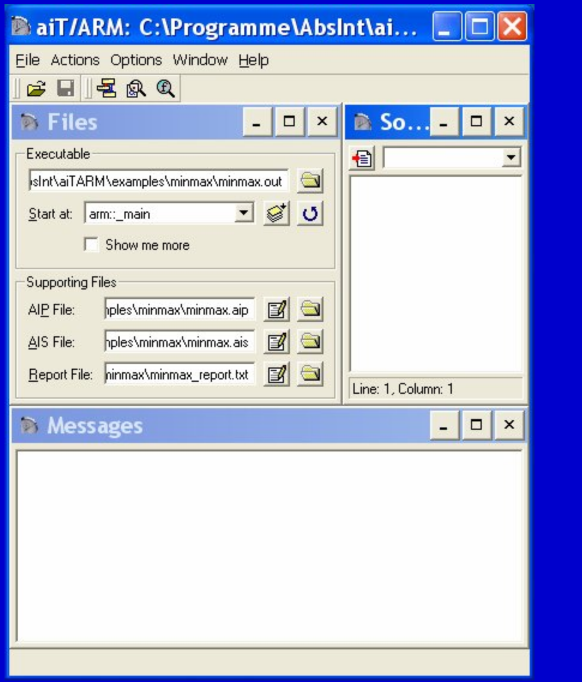


```
0x00000000000000e8 : ldral r12, [r13, #4]
Fetch [0xf0 - 0xf0] , Read [0x40080 - 0x40080] , Internal

0x00000000000000ec : addal r12, r12, #0x2
Fetch [0xf4 - 0xf4]

0x00000000000000f0 : stral r12, [r13, #4]
Fetch [0xf8 - 0xf8] , Write [0x40080 - 0x40080]

0x00000000000000f4 : ldral r0, [r13, #4]
Fetch [0xfc - 0xfc] , Read [0x40080 - 0x40080] , Internal

0x00000000000000f8 : ldral r12, [r13, #4]
Fetch [0x100 - 0x100] , Read [0x40080 - 0x40080] , Internal

0x00000000000000fc : mulal r0, r0, r12
Fetch [0x104 - 0x104] , Internal , Internal , Internal

0x0000000000000100 : ldral r12, [r13, #0]
Fetch [0x108 - 0x108] , Read [0x4007c - 0x4007c] , Internal

0x0000000000000104 : cmpal r0, r12 LSL #0x0
Fetch [TOP]

0x0000000000000108 : bls 0xcc <0xcc> -> TRUE EDGE
Fetch [TOP] , Fetch [TOP] , Fetch [TOP]

0x0000000000000108 : bls 0xcc <0xcc> -> FALSE EDGE
Fetch [TOP]
```



aiT WCET Analyzer

- Input: an executable program, starting points, loop iteration counts, call targets of indirect function calls, and a description of bus and memory speeds
- Computes Worst-Case Execu


aiT/ARM: C:\Programme\AbsInt\ai…
File Actions Options Window Help
Files
So…
Messages

Executable: sInt\aiTARM\examples\minmax\minmax.out

Start at: arm::_main

□ Show me more

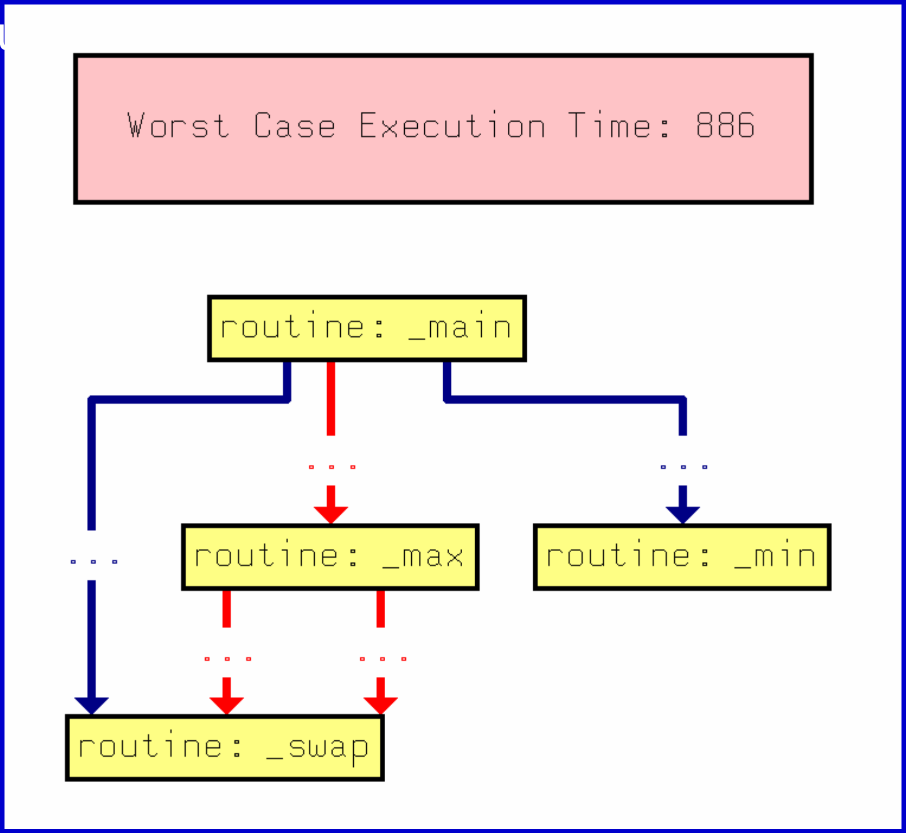
Supporting Files

AIP File: ples\minmax\minmax.aip

AIS File: nples\minmax\minmax.ais

Report File: ninmax\minmax_report.txt

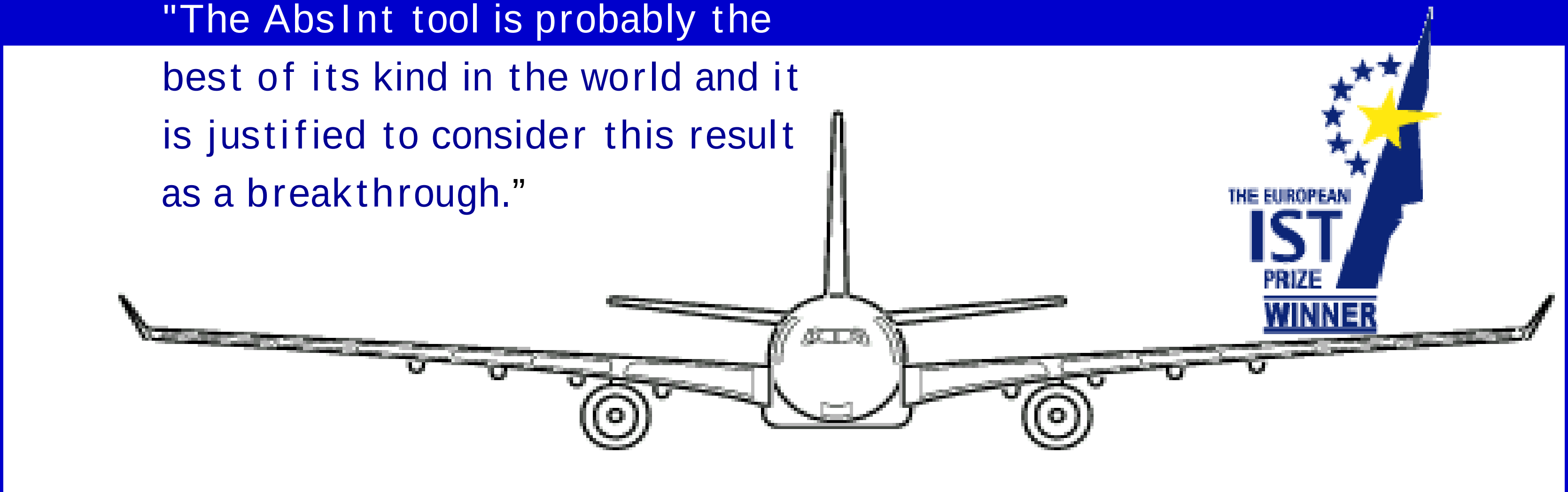
Line: 1, Column: 1


Worst Case Execution Time: 886

aiT WCET Analyzer

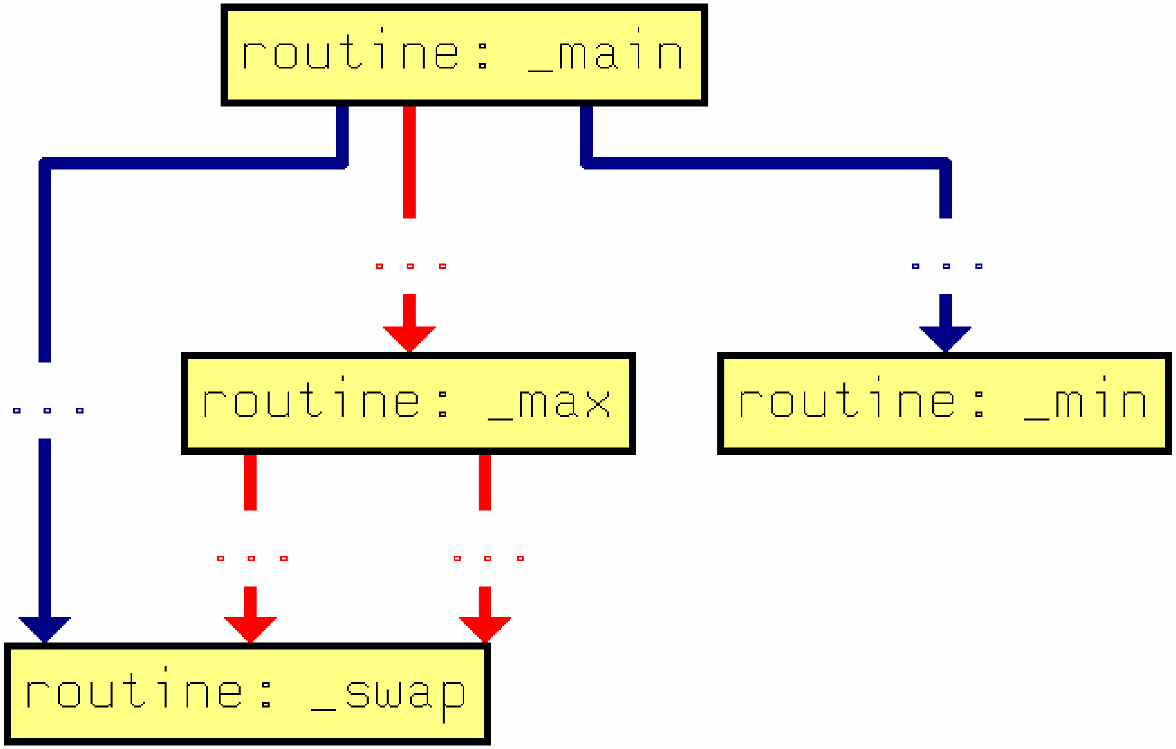
IST Project DAEDALUS final review report:

"The AbsInt tool is probably the best of its kind in the world and it is justified to consider this result as a breakthrough."



Several time-critical subsystems of the Airbus A380 have been certified using aiT

Worst Case Execution Time: 886

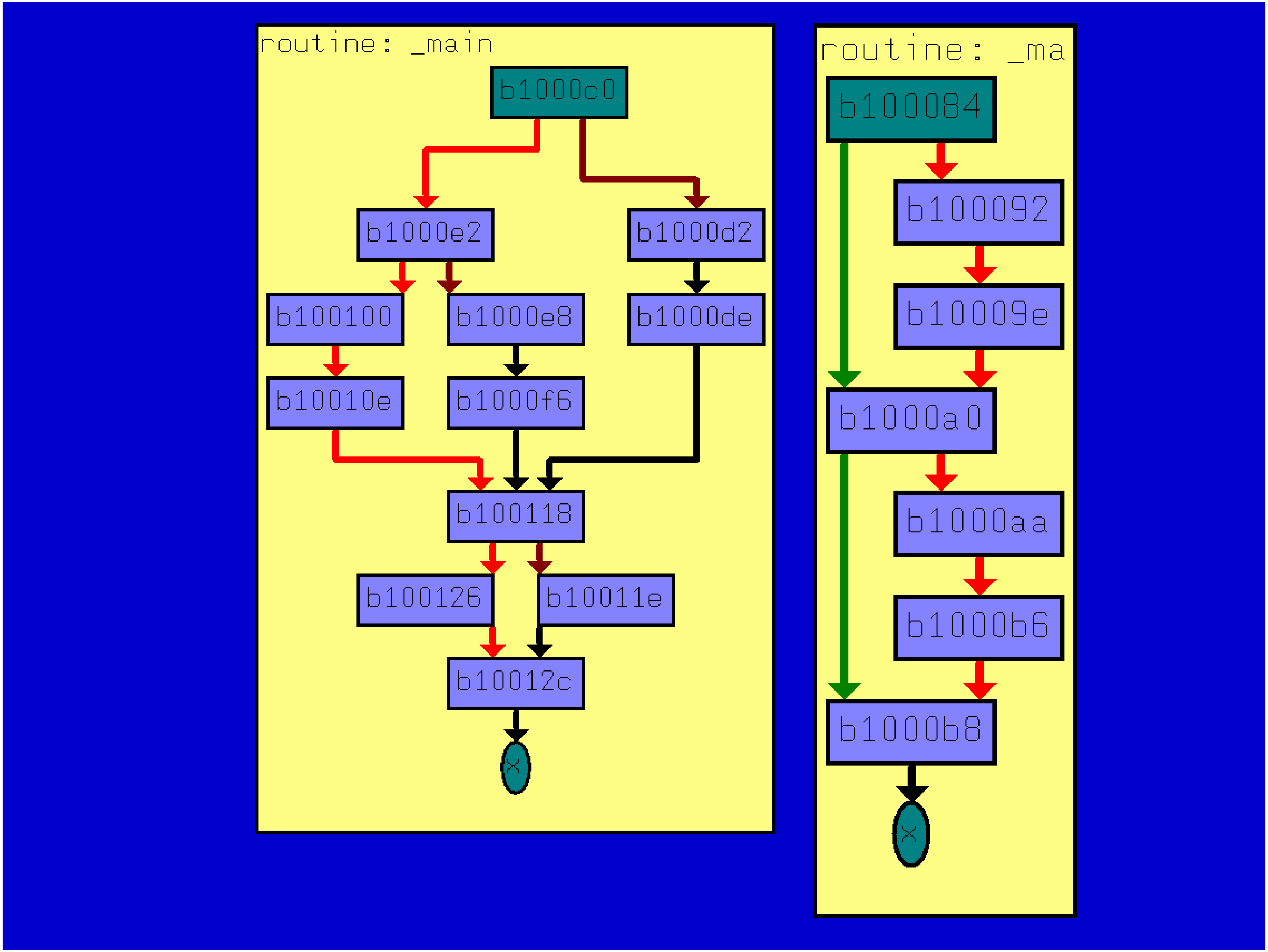


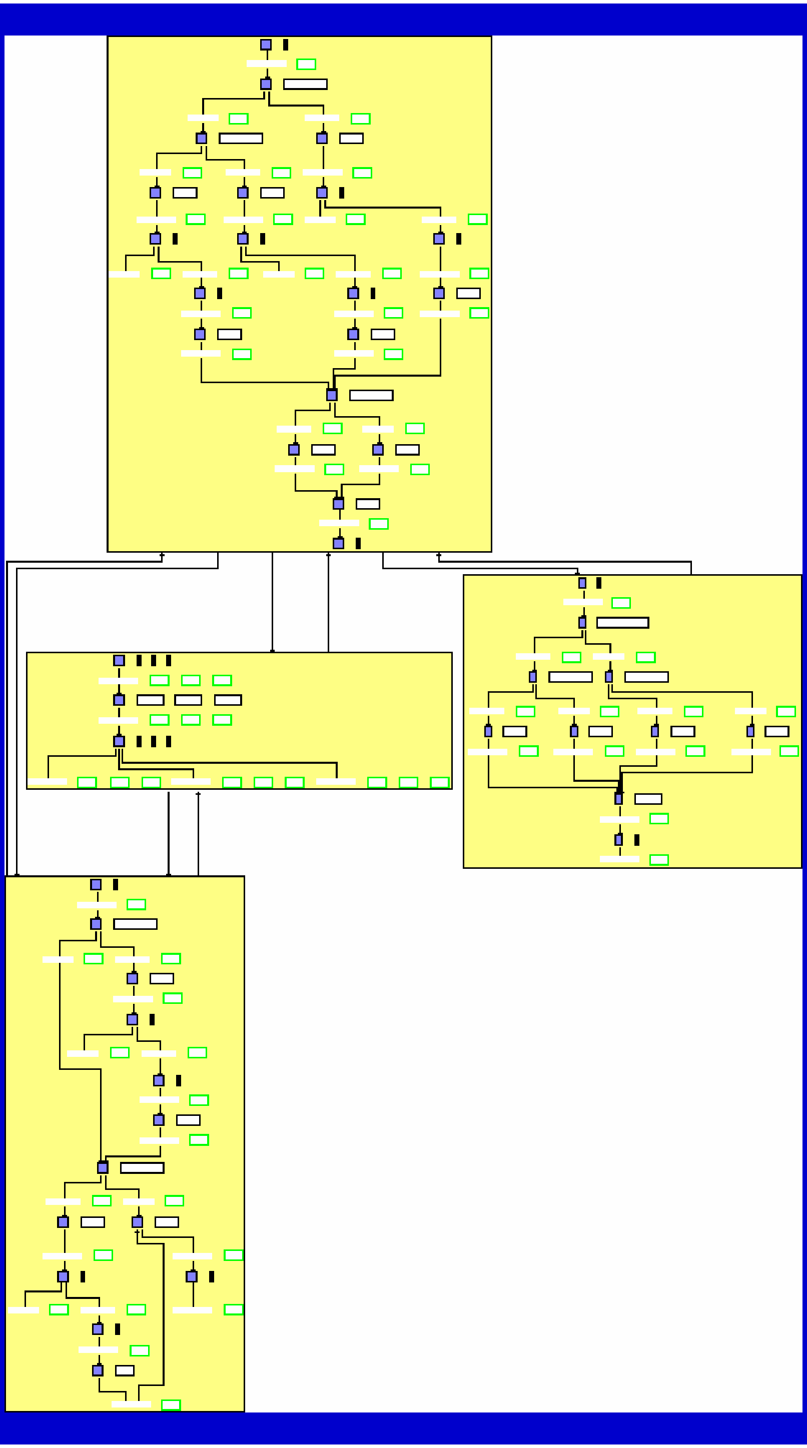
routine: _main

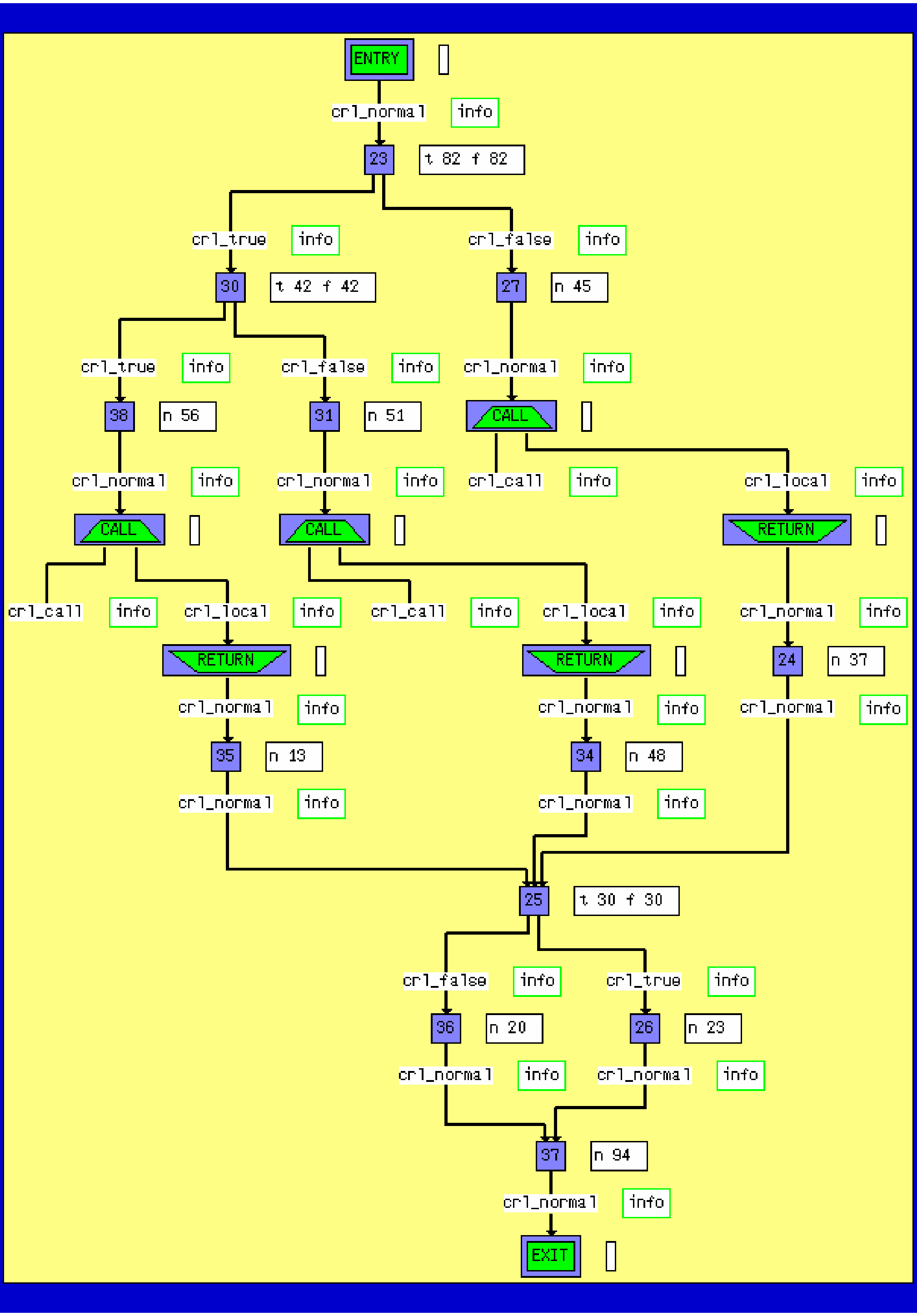
routine: _max

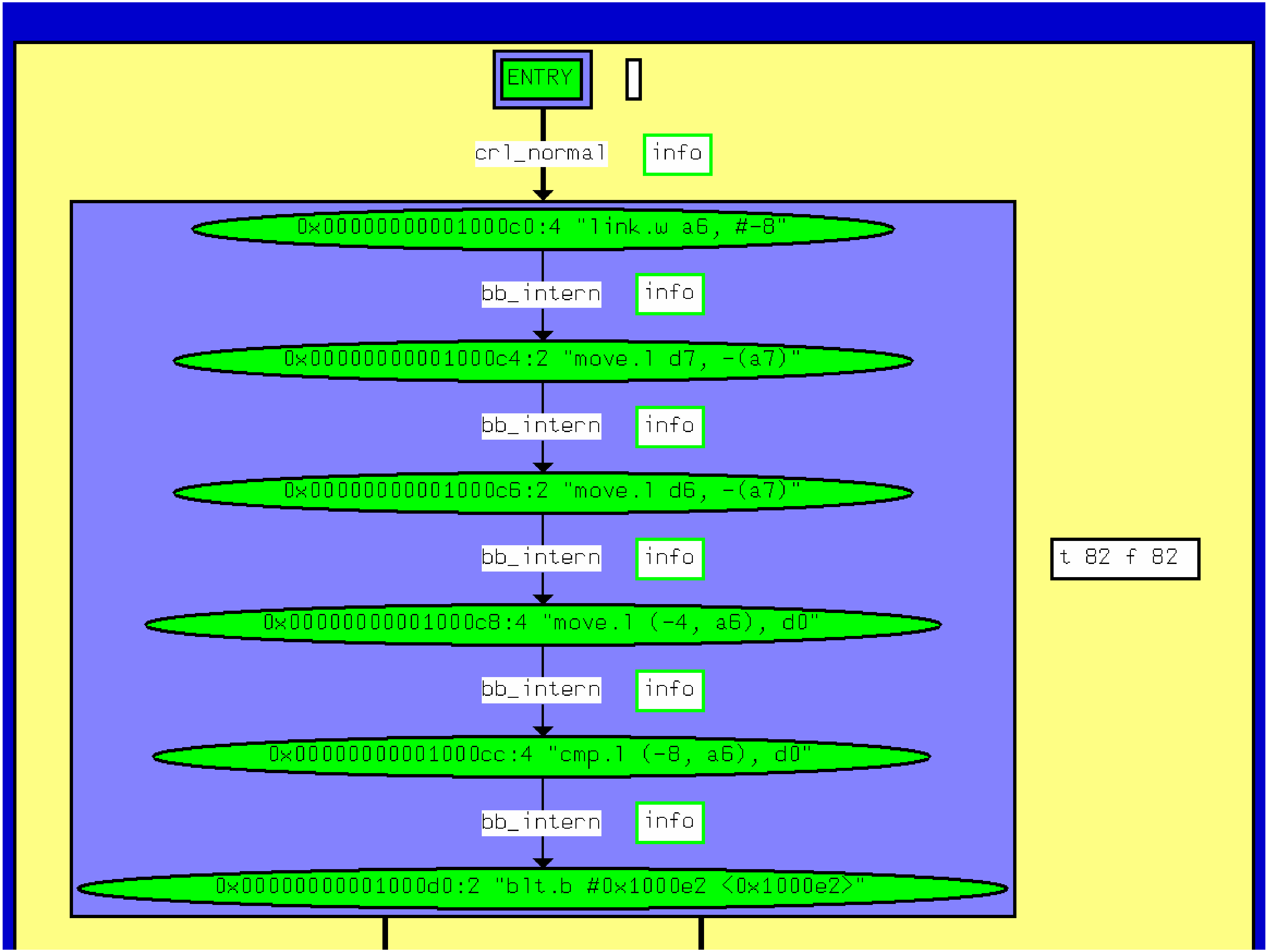
routine: _min

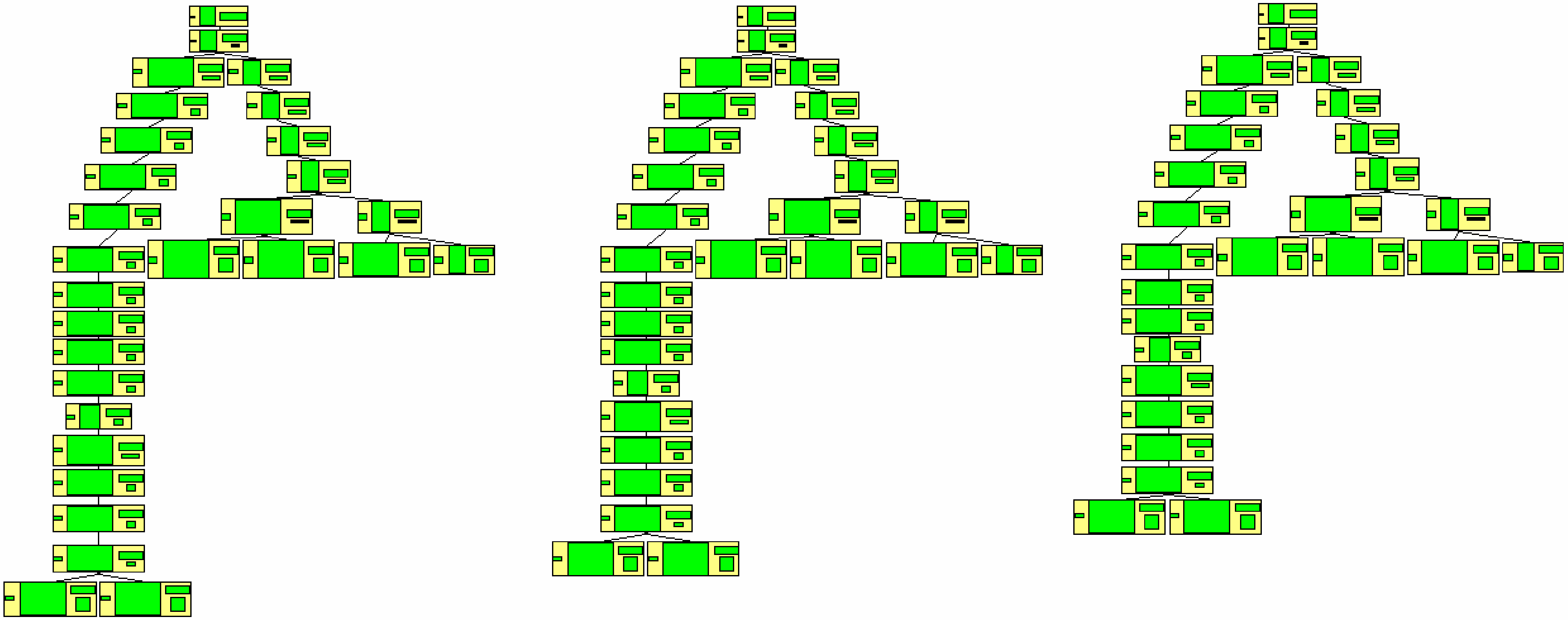
routine: _swap

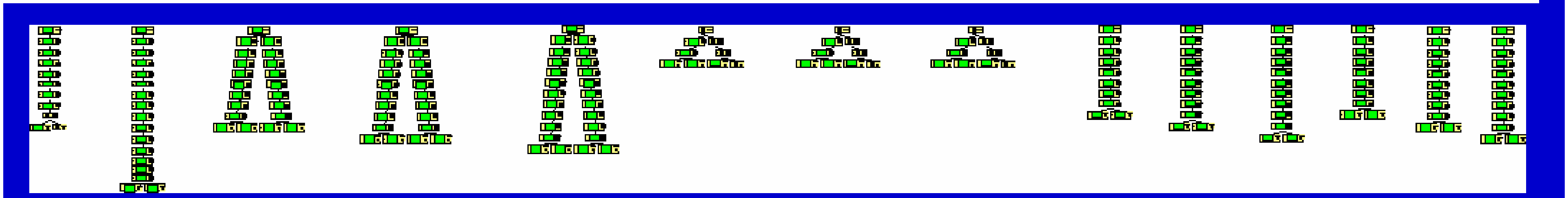

routine: _main
b1000c0
b1000e2
b1000d2
b100100
b1000e8
b1000de
b10010e
b1000f6
b100118
b100126
b10011e
b10012c
X
routine: _ma
b100084
b100092
b10009e
b1000a0
b1000aa
b1000b6
b1000b8
X

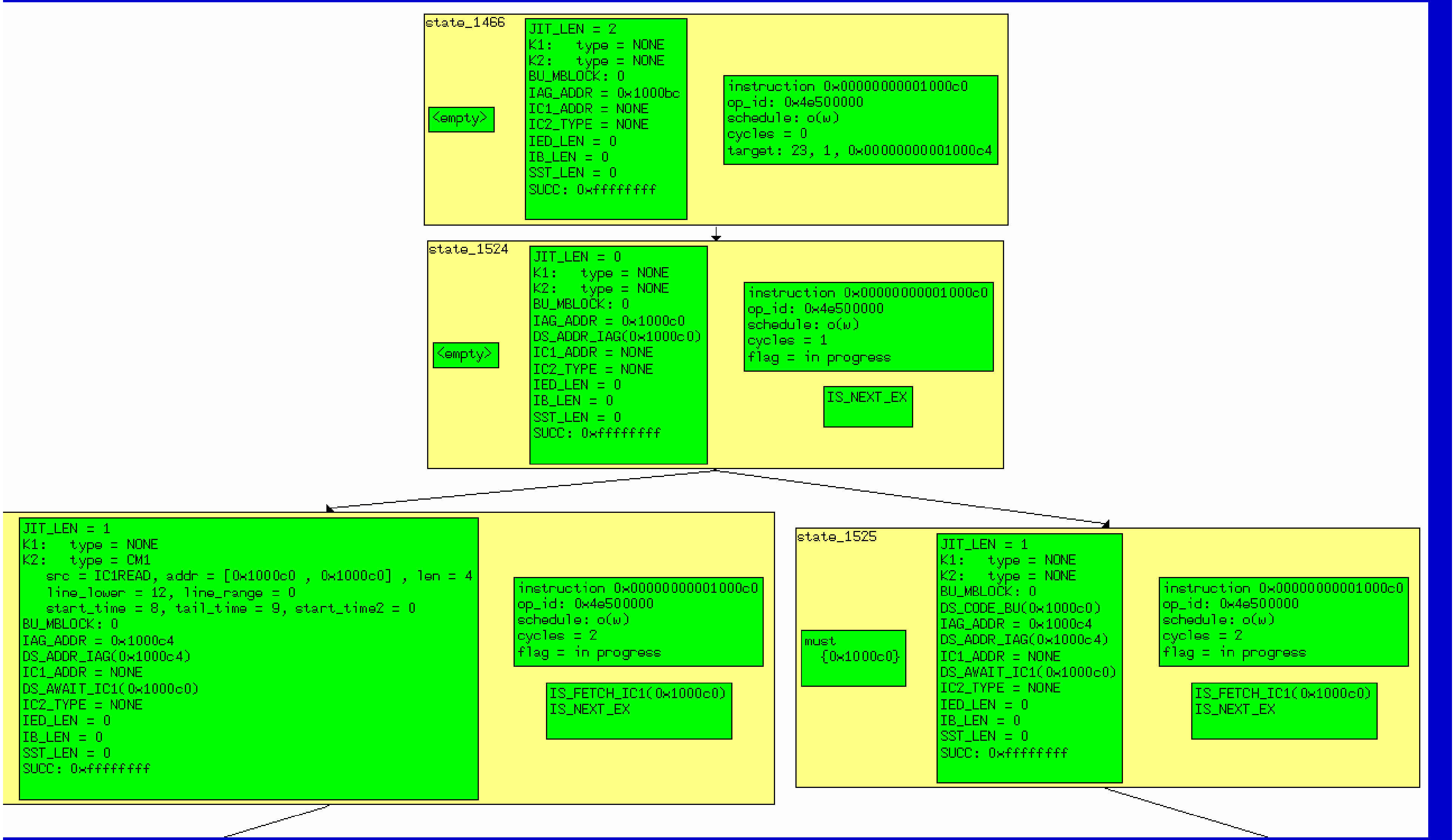



ENTRY
crl_normal
info
23
t 82 f 82
crl_true
info
crl_false
info
30
t 42 f 42
27
n 45
crl_true
info
crl_false
info
crl_normal
info
38
n 56
31
n 51
CALL
crl_normal
info
crl_normal
info
crl_call
info
crl_local
info
CALL
CALL
RETURN
crl_call
info
crl_local
info
crl_call
info
crl_local
info
crl_normal
info
RETURN
RETURN
24
n 37
crl_normal
info
crl_normal
info
crl_normal
info
35
n 13
34
n 48
crl_normal
info
crl_normal
info
25
t 30 f 30
crl_false
info
crl_true
info
36
n 20
26
n 23
crl_normal
info
crl_normal
info
37
n 94
crl_normal
info
EXIT


ENTRY
crl_normal
info
0x00000000001000c0:4 "link.w a6, #-8"
bb_intern
info
0x00000000001000c4:2 "move.l d7, -(a7)"
bb_intern
info
0x00000000001000c6:2 "move.l d6, -(a7)"
bb_intern
info
0x00000000001000c8:4 "move.l (-4, a6), d0"
bb_intern
info
0x00000000001000cc:4 "cmp.l (-8, a6), d0"
bb_intern
info
0x00000000001000d0:2 "blt.b #0x1000e2 <0x1000e2>"
t 82 f 82







state_1466

```
<empty>
```

```
JIT_LEN = 2
K1:   type = NONE
K2:   type = NONE
BU_MBLOCK: 0
IAG_ADDR = 0x1000bc
IC1_ADDR = NONE
IC2_TYPE = NONE
IED_LEN = 0
IB_LEN = 0
SST_LEN = 0
SUCC: 0xffffffff
```

```
instruction 0x00000000001000c0
op_id: 0x4e500000
schedule: o(w)
cycles = 0
target: 23, 1, 0x00000000001000c4
```

state_1524

```
<empty>
```

```
JIT_LEN = 0
K1:   type = NONE
K2:   type = NONE
BU_MBLOCK: 0
IAG_ADDR = 0x1000c0
DS_ADDR_IAG(0x1000c0)
IC1_ADDR = NONE
IC2_TYPE = NONE
IED_LEN = 0
IB_LEN = 0
SST_LEN = 0
SUCC: 0xffffffff
```

```
instruction 0x00000000001000c0
op_id: 0x4e500000
schedule: o(w)
cycles = 1
flag = in progress
```

```
IS_NEXT_EX
```

```
JIT_LEN = 1
K1:   type = NONE
K2:   type = CM1
   src = IC1READ, addr = [0x1000c0 , 0x1000c0] , len = 4
   line_lower = 12, line_range = 0
   start_time = 8, tail_time = 9, start_time2 = 0
BU_MBLOCK: 0
IAG_ADDR = 0x1000c4
DS_ADDR_IAG(0x1000c4)
IC1_ADDR = NONE
DS_AWAIT_IC1(0x1000c0)
IC2_TYPE = NONE
IED_LEN = 0
IB_LEN = 0
SST_LEN = 0
SUCC: 0xffffffff
```

```
instruction 0x00000000001000c0
op_id: 0x4e500000
schedule: o(w)
cycles = 2
flag = in progress
```

```
IS_FETCH_IC1(0x1000c0)
IS_NEXT_EX
```

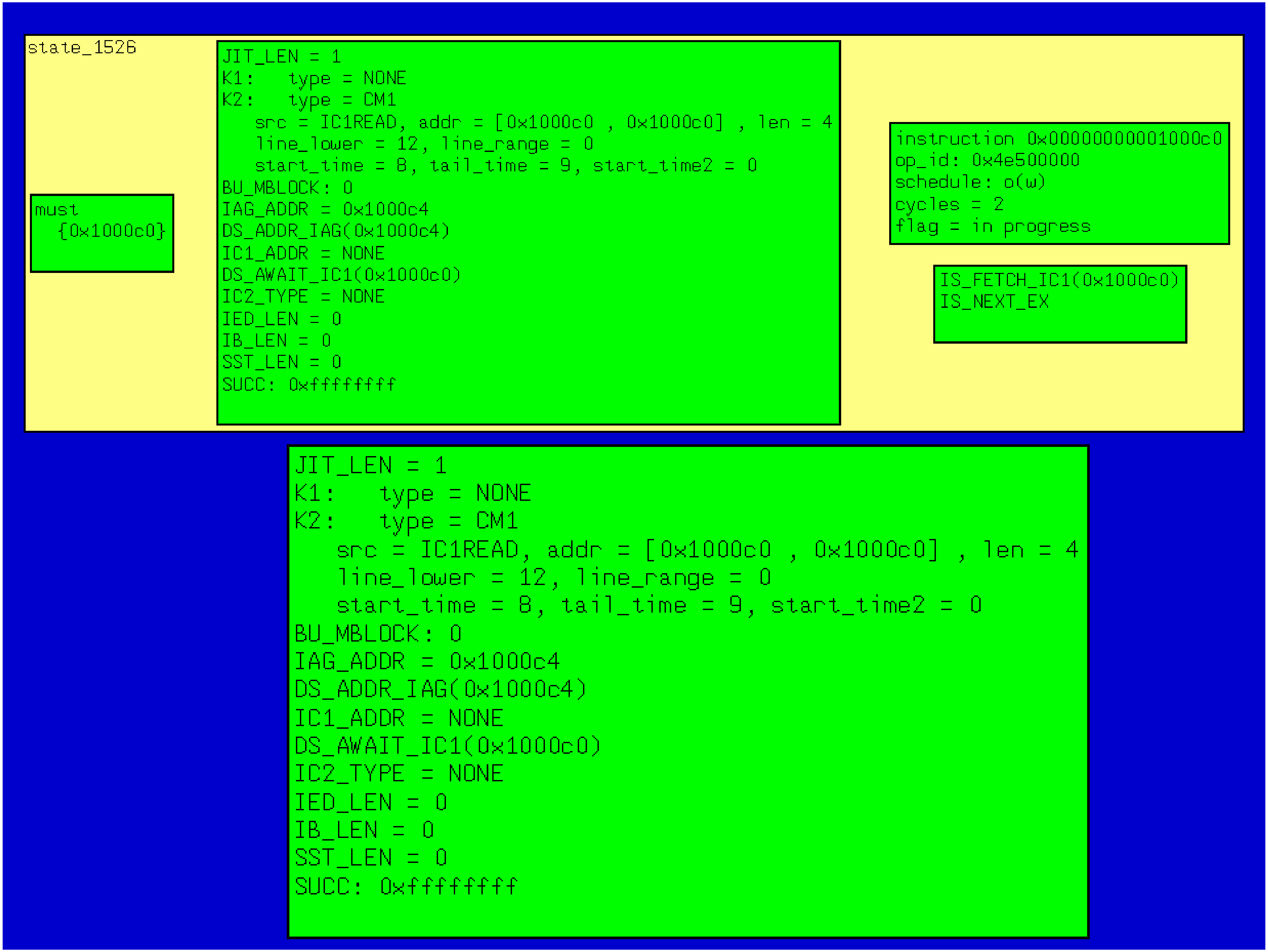
state_1525

```
must
  {0x1000c0}
```

```
JIT_LEN = 1
K1:   type = NONE
K2:   type = NONE
BU_MBLOCK: 0
DS_CODE_BU(0x1000c0)
IAG_ADDR = 0x1000c4
DS_ADDR_IAG(0x1000c4)
IC1_ADDR = NONE
DS_AWAIT_IC1(0x1000c0)
IC2_TYPE = NONE
IED_LEN = 0
IB_LEN = 0
SST_LEN = 0
SUCC: 0xffffffff
```

```
instruction 0x00000000001000c0
op_id: 0x4e500000
schedule: o(w)
cycles = 2
flag = in progress
```

```
IS_FETCH_IC1(0x1000c0)
IS_NEXT_EX
```



state_1526

```
must
  {0x1000c0}
```

```
JIT_LEN = 1
K1:   type = NONE
K2:   type = CM1
   src = IC1READ, addr = [0x1000c0 , 0x1000c0] , len = 4
   line_lower = 12, line_range = 0
   start_time = 8, tail_time = 9, start_time2 = 0
BU_MBLOCK: 0
IAG_ADDR = 0x1000c4
DS_ADDR_IAG(0x1000c4)
IC1_ADDR = NONE
DS_AWAIT_IC1(0x1000c0)
IC2_TYPE = NONE
IED_LEN = 0
IB_LEN = 0
SST_LEN = 0
SUCC: 0xffffffff
```

```
instruction 0x00000000001000c0
op_id: 0x4e500000
schedule: o(w)
cycles = 2
flag = in progress
```

```
IS_FETCH_IC1(0x1000c0)
IS_NEXT_EX
```

JIT_LEN = 1
K1:   type = NONE
K2:   type = CM1
   src = IC1READ, addr = [0x1000c0 , 0x1000c0] , len = 4
   line_lower = 12, line_range = 0
   start_time = 8, tail_time = 9, start_time2 = 0
BU_MBLOCK: 0
IAG_ADDR = 0x1000c4
DS_ADDR_IAG(0x1000c4)
IC1_ADDR = NONE
DS_AWAIT_IC1(0x1000c0)
IC2_TYPE = NONE
IED_LEN = 0
IB_LEN = 0
SST_LEN = 0
SUCC: 0xffffffff

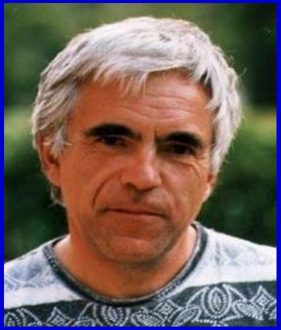
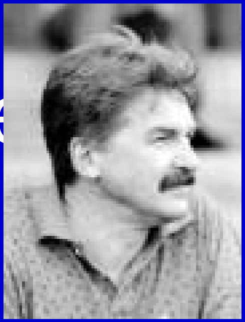
Current State and Future Work

- WCET tools available for the Motorola PowerPC MPC 555, 565, and 755, Motorola ColdFire MCF 5307, ARM7 TDMI, HCS12/STAR12, TMS320C33, C166/ST10, Renesas M32C/85, Infineon TriCore 1.3, ...
- Learned, how time-predictable architectures look like
- Adaptation effort still too big => automation
- Modeling effort error prone => formal methods
- Middleware, RTOS not treated => challenging!

All nice topics for future research!

Who needs aiT?

- TTA
- Synchronous languages
- Stream-oriented people
- UML real-time profile
- Hand coders

Acknowledgements

- Christian Ferdinand, whose thesis started all this
- Reinhold Heckmann, Mister Cache
- Florian Martin, Mister PAG
- Stephan Thesing, Mister Pipeline
- Michael Schmidt, Value Analysis
- Henrik Theiling, Mister Frontend + Path Analysis
- Jörn Schneider, OSEK

Recent Publications

- *C. Ferdinand et al.: CacheBehavior Prediction by Abstract Interpretation. Science of Computer Programming 35(2): 163-189 (1999)*
- *C. Ferdinand et al.: Reliable and Precise WCET Determination of a Real-Life Processor*, EMSOFT 2001
- *R. Heckmann et al.: The Influence of Processor Architecture on the Design and the Results of WCET Tools*, IEEE Proc. on Real-Time Systems, July 2003
- *St. Thesing et al.: An Abstract Interpretation-based Timing Validation of Hard Real-Time Avionics Software*, IPDS 2003
- *L. Thiele, R. Wilhelm: Design for Timing Predictability, 25th Anniversary edition of the Kluwer Journal* Real-Time Systems*, Dec. 2004*
- *R. Wilhelm: Determination of Execution Time Bounds, Embedded Systems Handbook, CRC Press, 2005*
- *R. Wilhelm et al.: The WCET Problem – Overview of Methods and Survey of Tools, to appear in ACM TECS, 2007*
- *J. Reineke et al.: Predictability of Cache Replacement Policies, AVACS TR, submitted for publication*