Run-Time Guarantees for Real-Time Systems

Reinhard Wilhelm
Saarbrücken

AbsInt
Angewandte Informatik GmbH

UNIVERSITÄT
DES
SAARLANDES

Structure of the Talk

1. WCET determination, introduction, architecture, static program analysis
2. Caches
   - must, may analysis
   - Real-life caches: Motorola ColdFire
3. Pipelines
   - Abstract pipeline models
   - Integrated analyses
4. Current State and Future Work

Structure of the Talk

1. WCET determination, introduction, architecture, static program analysis
2. Caches
   - must, may analysis
   - Real-life caches: Motorola ColdFire
3. Pipelines
   - Abstract pipeline models
4. Integrated analyses
5. Current State and Future Work in AVACS

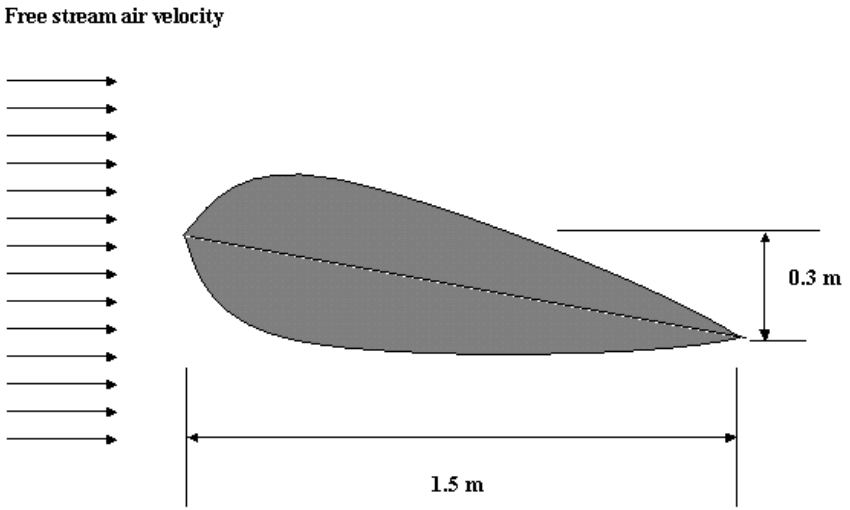
Industrial Needs

Hard real-time systems, often in safety-critical applications abound

- Aeronautics, automotive, train industries, manufacturing control

Sideairbag in car,
Reaction in <10 mSec

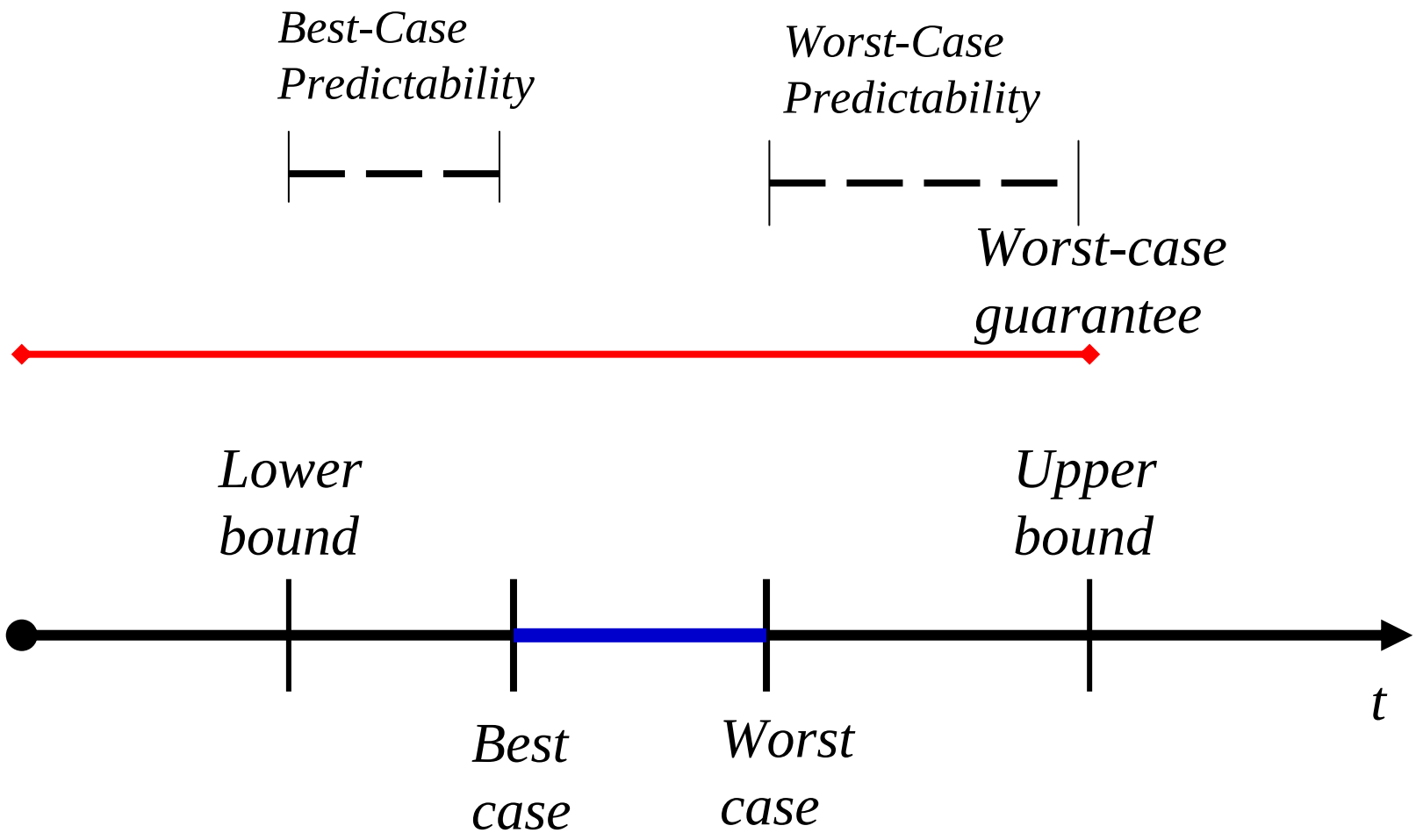
Wing vibration of airplane,
sensing every 5 mSec



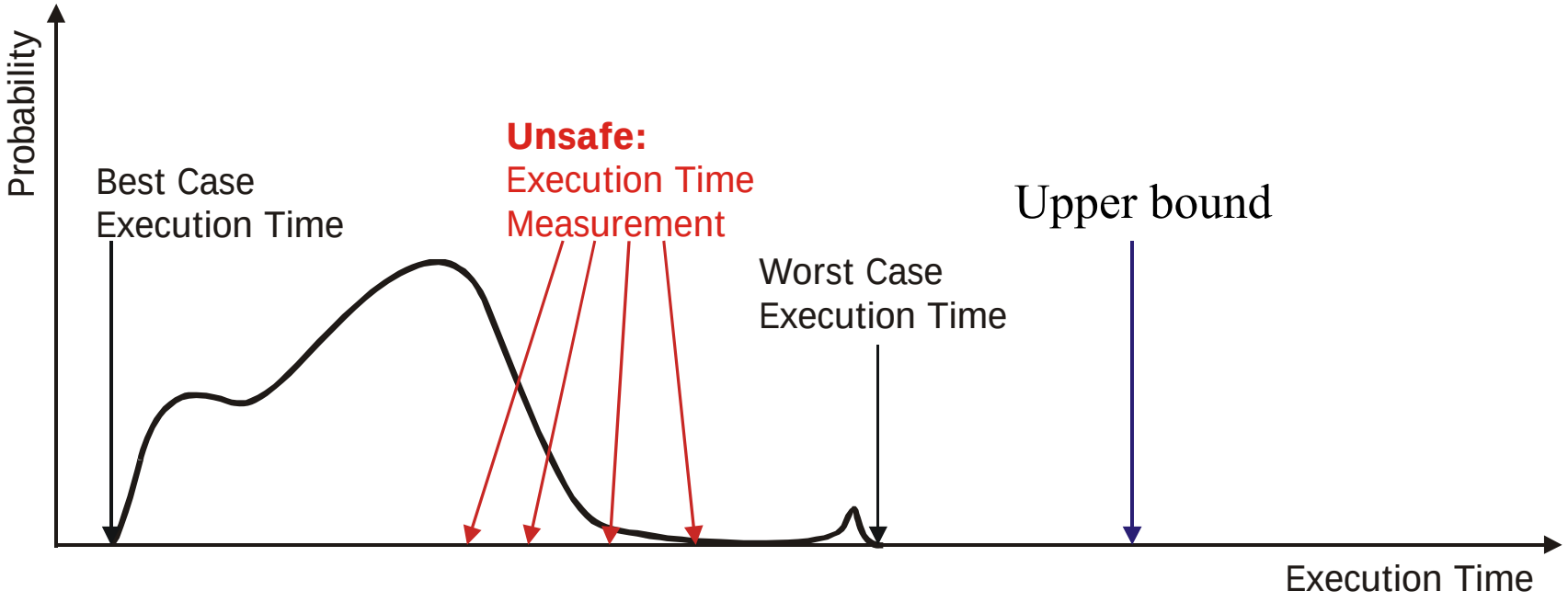
Hard Real-Time Systems

- 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



Measurement vs. Analysis



The Traditional Approaches

- Measurements: determine execution times directly by observing the execution.

  Does not guarantee an upper bound to all executions

- Structure-based: determine the maximum execution times according to the structure of the program.

  bound(**if** c **then** s_1 **else** s_2) =

  $$\text{bound}(c) + \max\{\text{bound}(s_1), \text{bound}(s_2)\}$$

  Requires compositionality – not given on modern hardware with caches/pipelines!

Modern Hardware Features

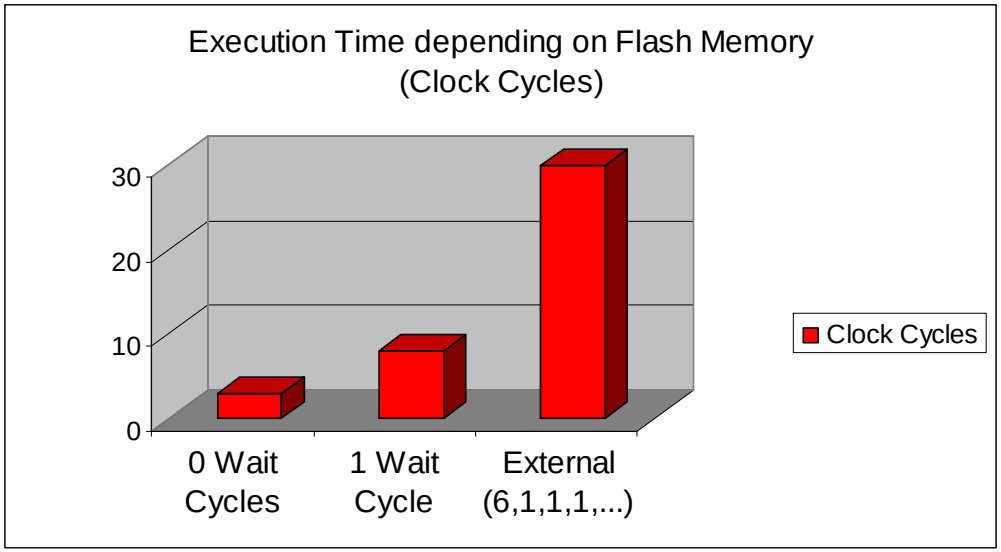
- Modern processors increase performance by using:
  Caches, Pipelines, Branch Prediction
- These features make WCET computation difficult:
  Execution times of instructions vary widely
  - Best case - everything goes smoothely: no cache miss, operands ready, needed resources free, branch correctly predicted
  - 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

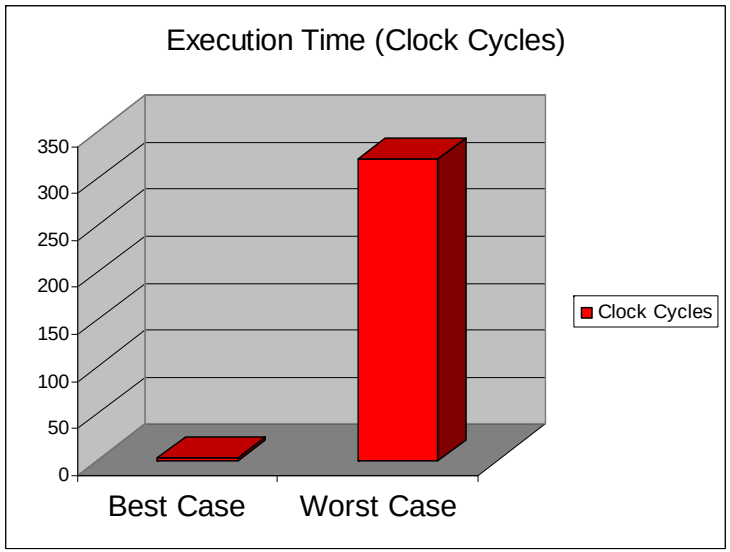
x = a + b;

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

MPC 5xx

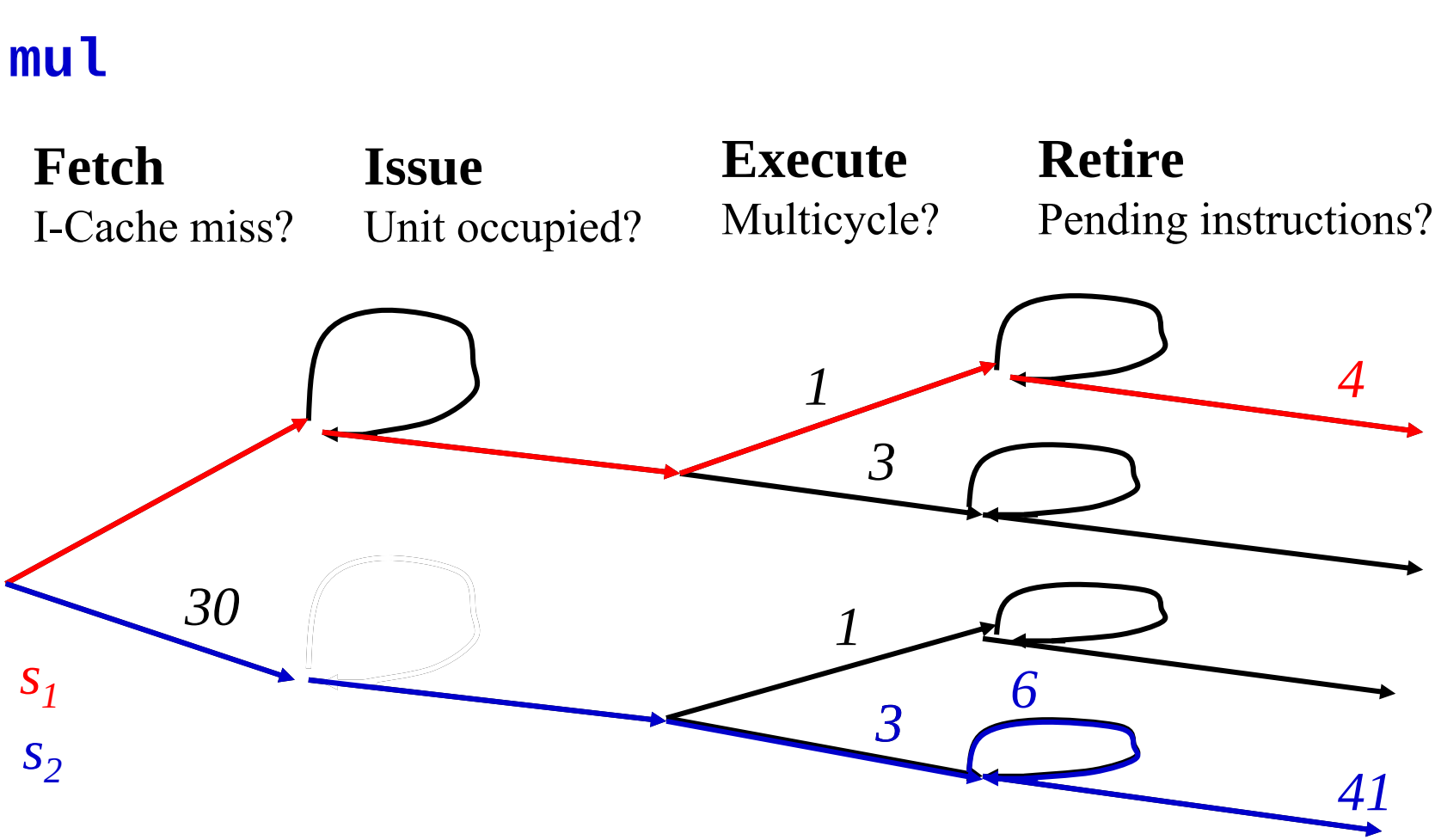


PPC 755



(Concrete) Instruction Execution

`mul`



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

Execution Time is History-Sensitive

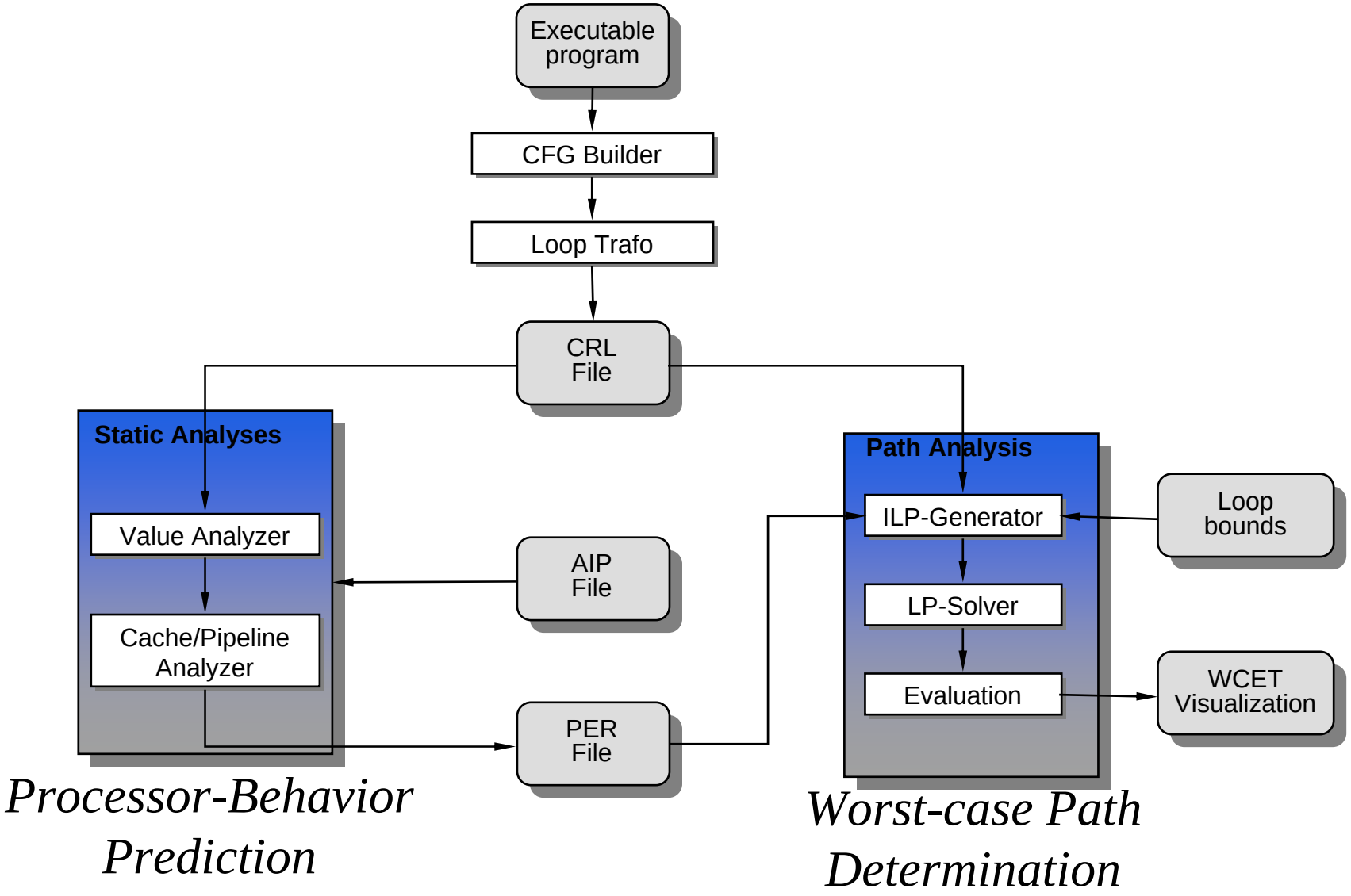
Contribution of the execution of an instruction to a program‘s execution time

- depends on the execution state, i.e., on the execution so far,
- i.e., cannot be determined in isolation

Overall Approach: Natural Modularization

1. Processor-Behavior Prediction:
   - 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



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)

First Attempt at Processor-Behavior Prediction

1. Abstractly interpret the program to obtain invariants about processor states
2. Derive safety properties, "timing accident *X* does not happen at instruction *I*"
3. Omit timing penalties, whenever a timing accident can be excluded;
   assume timing penalties, whenever
   - timing accident is predicted or
   - can not be safely 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 start!

Abstract Interpretation vs. Model Checking

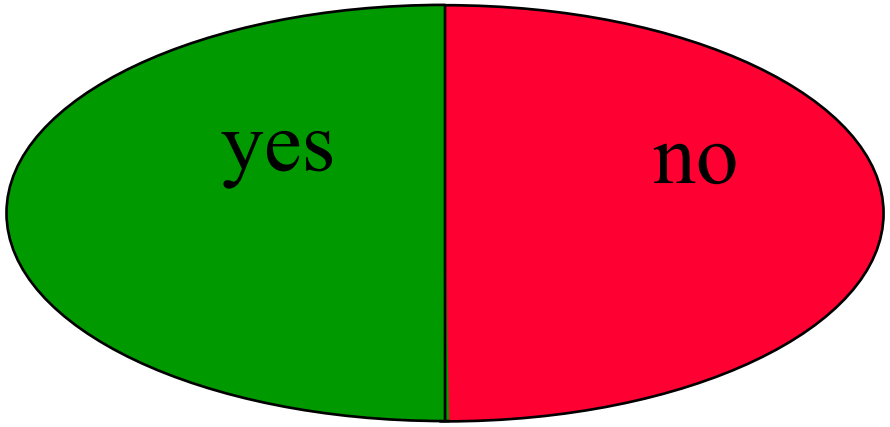
- Model Checking is good if you know the safety property that you want to prove
- A strong Abstract Interpretation verifies invariants at program points implying many safety properties
  - Individual safety properties need not be specified individually! ☺
  - They are encoded in the static analysis

Static Program Analysis

- Determination of invariants about program execution at compile time
- Most of the (interesting) properties are undecidable => approximations
- An approximate program analysis is safe, if its results can always be depended on. Results are allowed to be imprecise as long as they are on the safe side
- Quality of the results (precision) should be as good as possible

Approximation

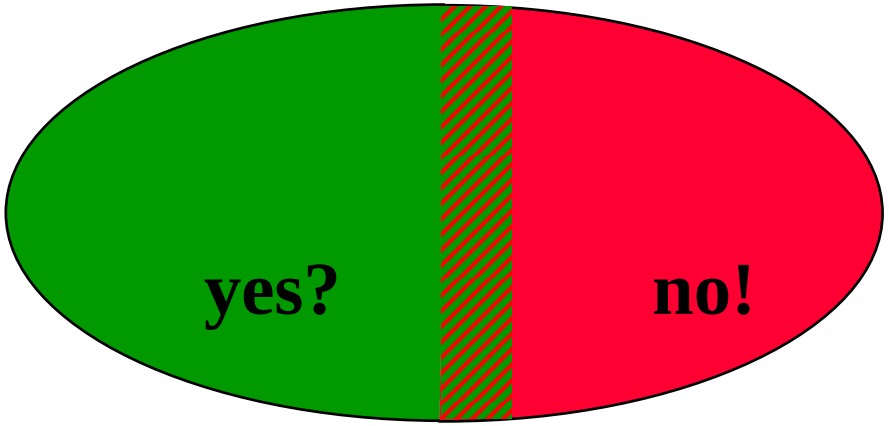
True Answers



Approximation

Safe

~~True~~ Answers



Precision

Safety and Liveness Properties

- Safety: „something bad will not happen“
  Examples: Division by 0,
  Array index not out of bounds
- Liveness: „something good will happen“
  Examples: Program will react to input,
  Request will be served

Analogies

- Rules-of-Sign Analysis σ: **VAR** –> {+,-,0, ⇖,◆}
  Derivable safety properties from invariant σ(x) = + :
  - `sqrt(x)` ⇒ No exception: sqrt of negative number
  - `a/x` ⇒ No exception: Division by 0
- Must-Cache Analysis mc: **ADDR –> CS x CL**
  Derivable safety properties:
  Memory access will always hit the cache

Example for Approximation

Rules of Sign: (Abstract) Addition $+^{\#}$

$+^{\#}$	0	$+$	$-$	$\nwarrow$
0	0	$+$	$-$	$\nwarrow$
$+$	$+$	$+$	$\nwarrow$	$\nwarrow$
$-$	$-$	$\nwarrow$	$-$	$\nwarrow$
$\nwarrow$	$\nwarrow$	$\nwarrow$	$\nwarrow$	$\nwarrow$

Example for Approximation

Abstract Multiplication *#

*#	0	+	–	⇖
0	0	0	0	0
+	0	+	–	⇖
–	0	–	+	⇖
⇖	0	⇖	⇖	⇖

Static Program Analysis Applied to WCET Determination

- WCET must be safe, i.e. not underestimated
- WCET should be tight, i.e. not far away from real execution times
- Analogous for BCET
- Effort must be tolerable

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

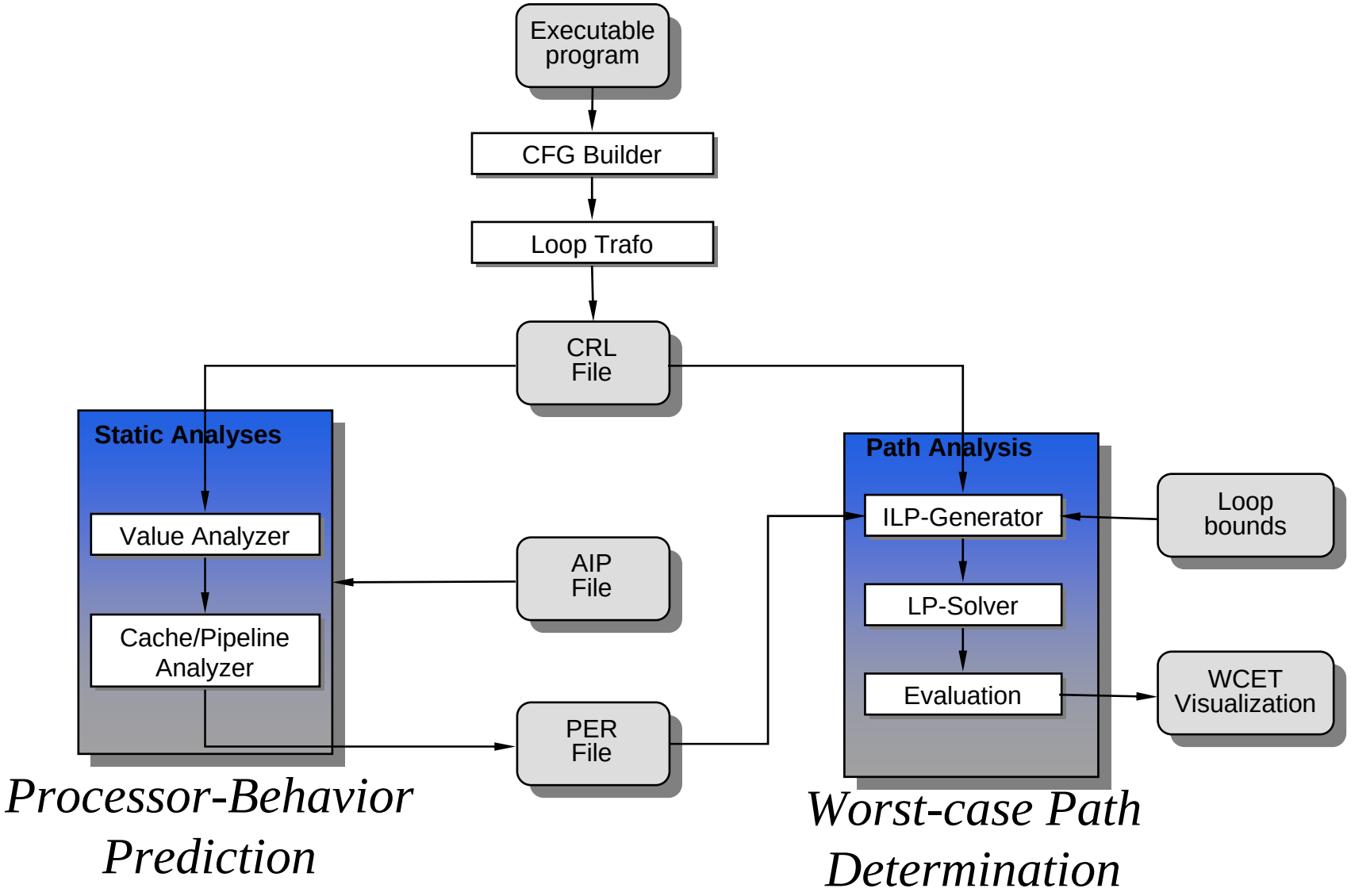
- C code synthesized from SCADE specifications
- Very disciplined code
  - No pointers, no heap
  - Few tables
  - Structured control flow
- However, very badly designed processor!

Abstract Interpretation (AI)

- AI: semantics based method for static program analysis
- Basic idea of AI: Perform the program's computations using value descriptions or abstract value in place of the concrete values
- Basic idea in WCET: Derive timing information from an approximation of the “collecting semantics” (for all inputs)
- AI supports correctness proofs
- Tool support (PAG)

Value Analysis

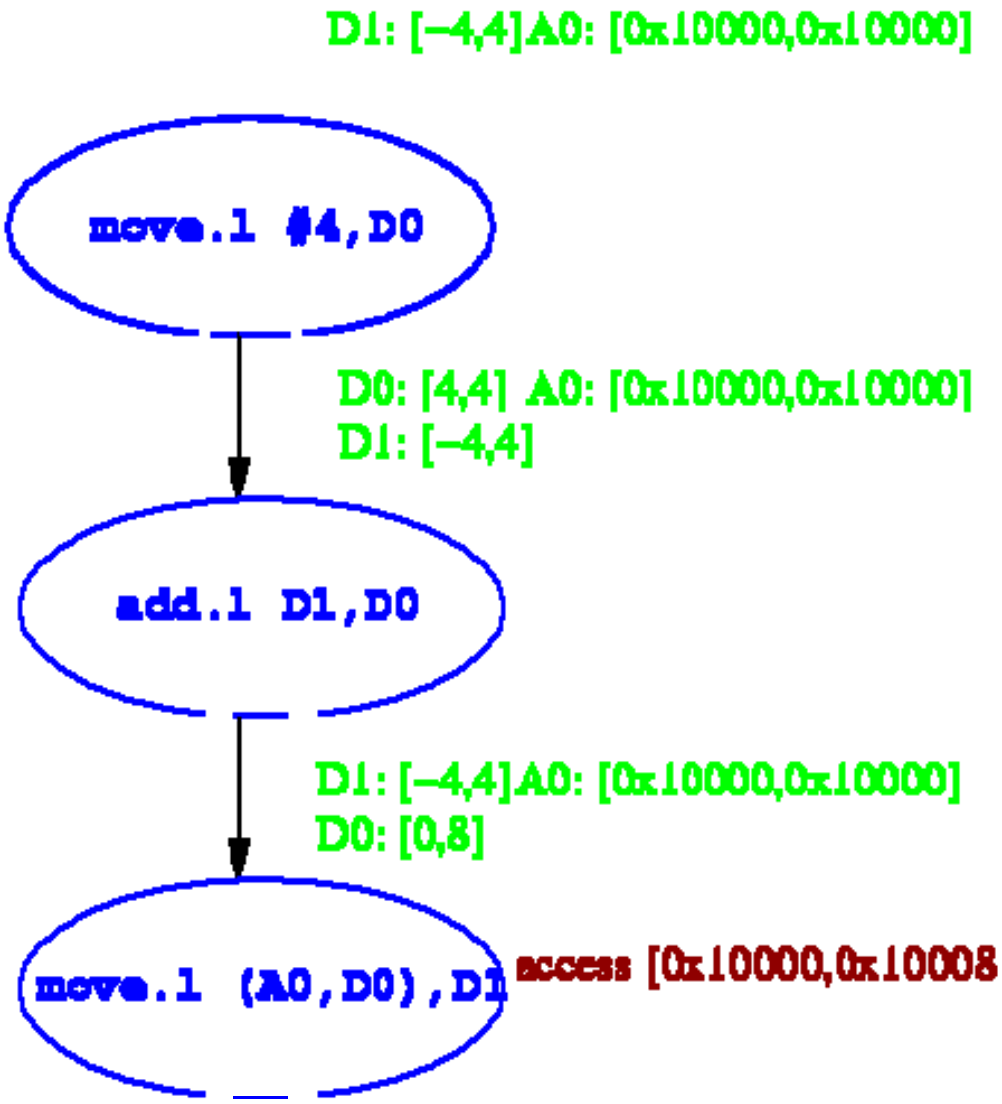
Overall Structure



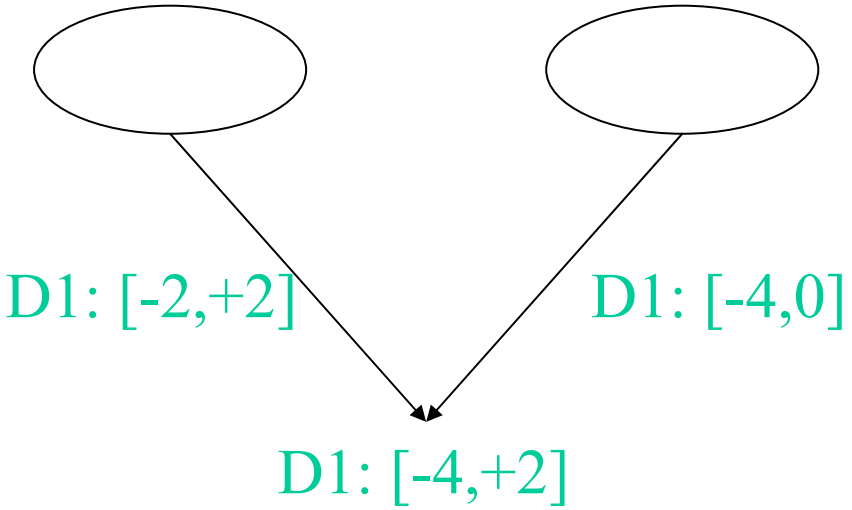
Value Analysis

- Motivation:
  - Provide exact access information to data-cache/pipeline analysis
  - Detect infeasible paths
- Method: calculate intervals, i.e. lower and upper bounds for the values occurring in the machine program (addresses, register contents, local and global variables)
- Method: Interval analysis
- Generalization of Constant Propagation $\Rightarrow$ Impossible/difficult to do by MC (c.f. Cousot against Manna paper)

Value Analysis II



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



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

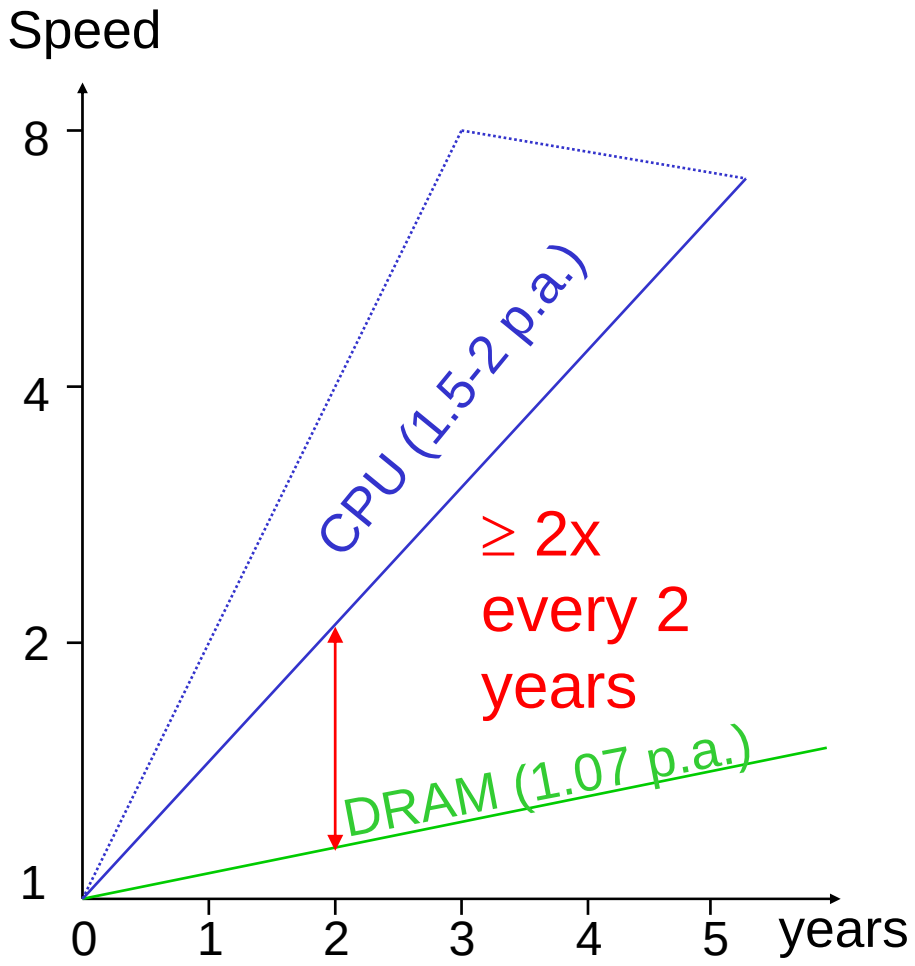
Good means less than 16 cache lines

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

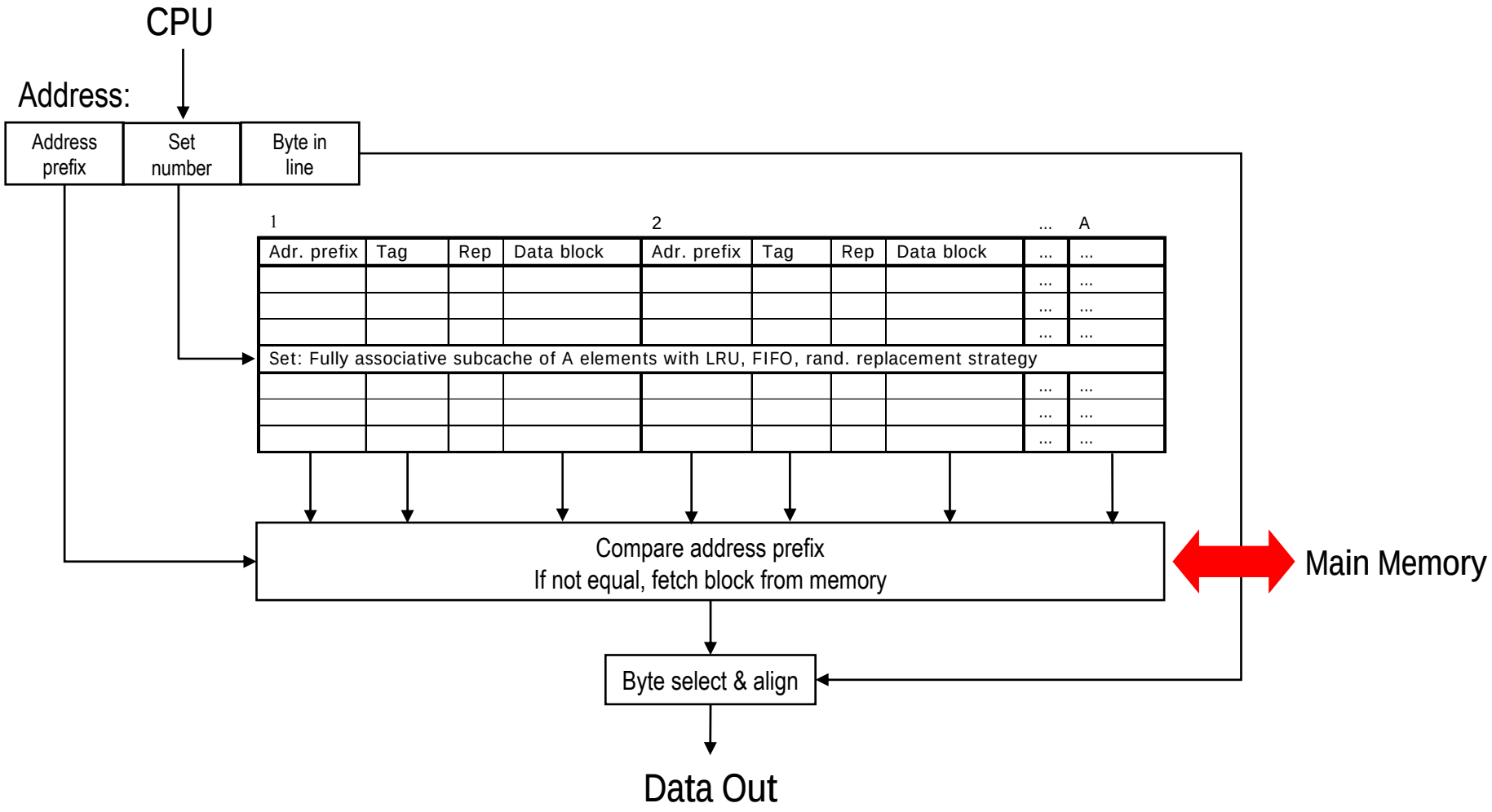
Caches: How the work

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



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

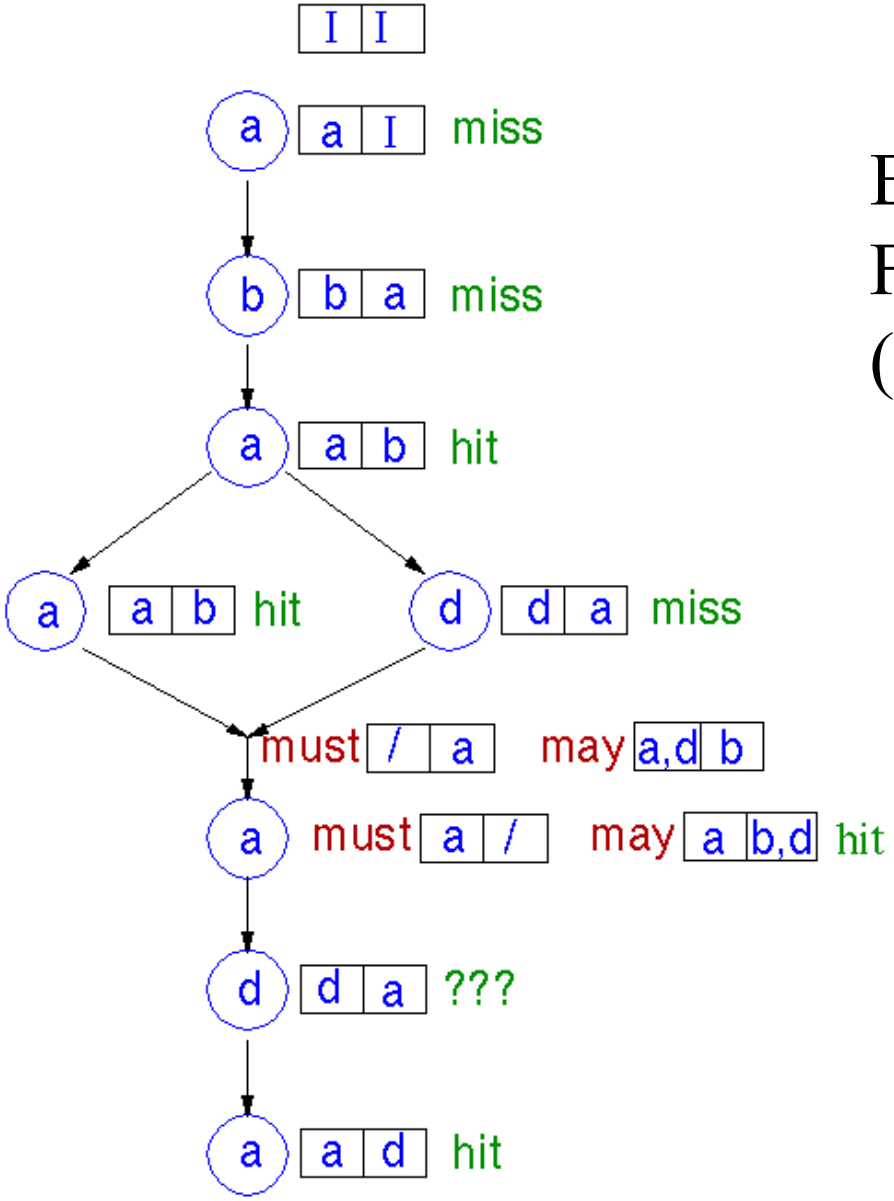
- May Analysis:

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

  Complement says what is not in the cache

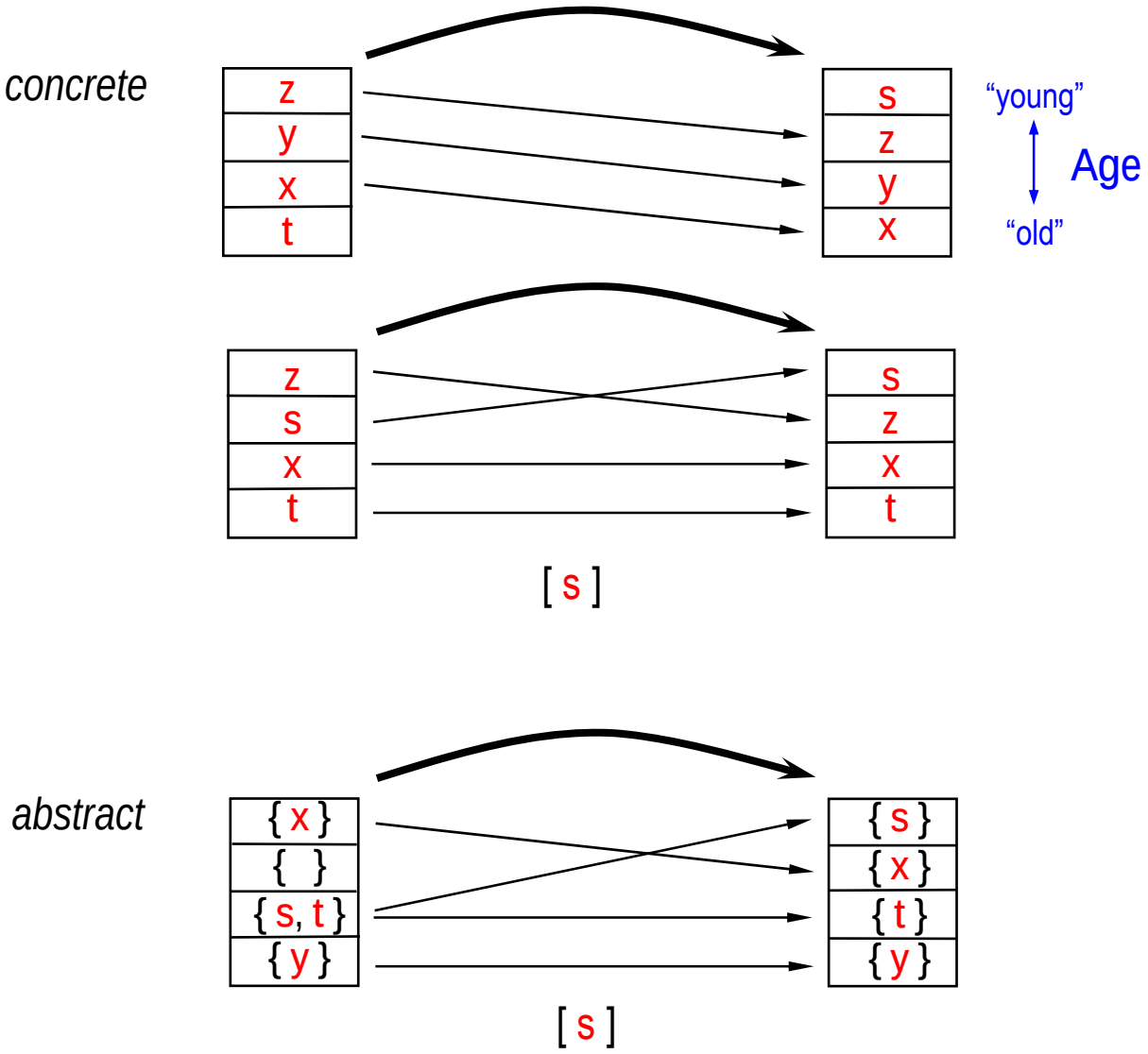
Must-Cache and May-Cache-Information

- 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

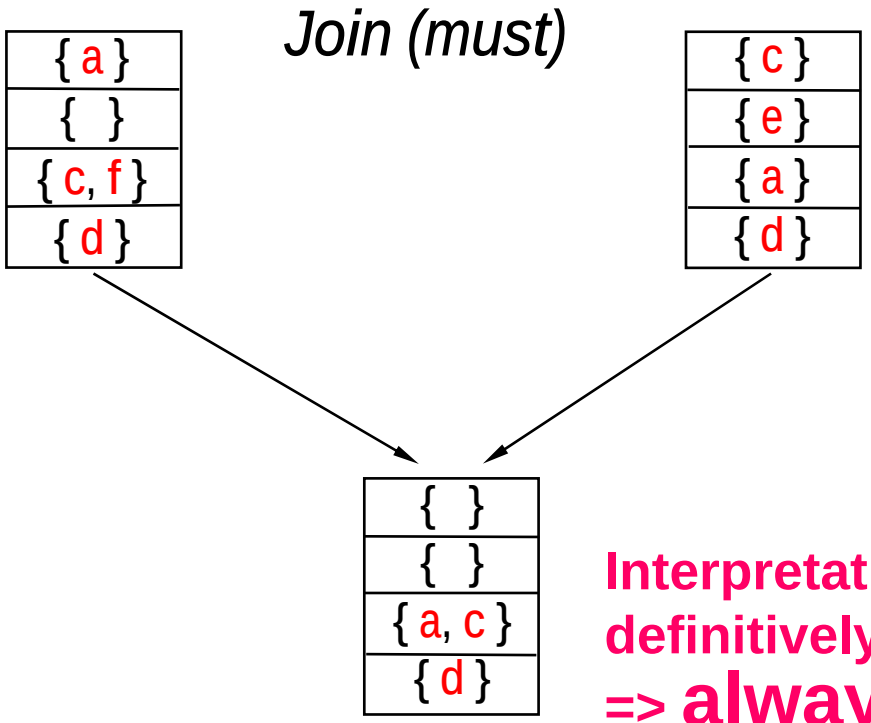


Example:
Fully Associative Cache
(2 Elements)

Cache with LRU Replacement: Transfer for must



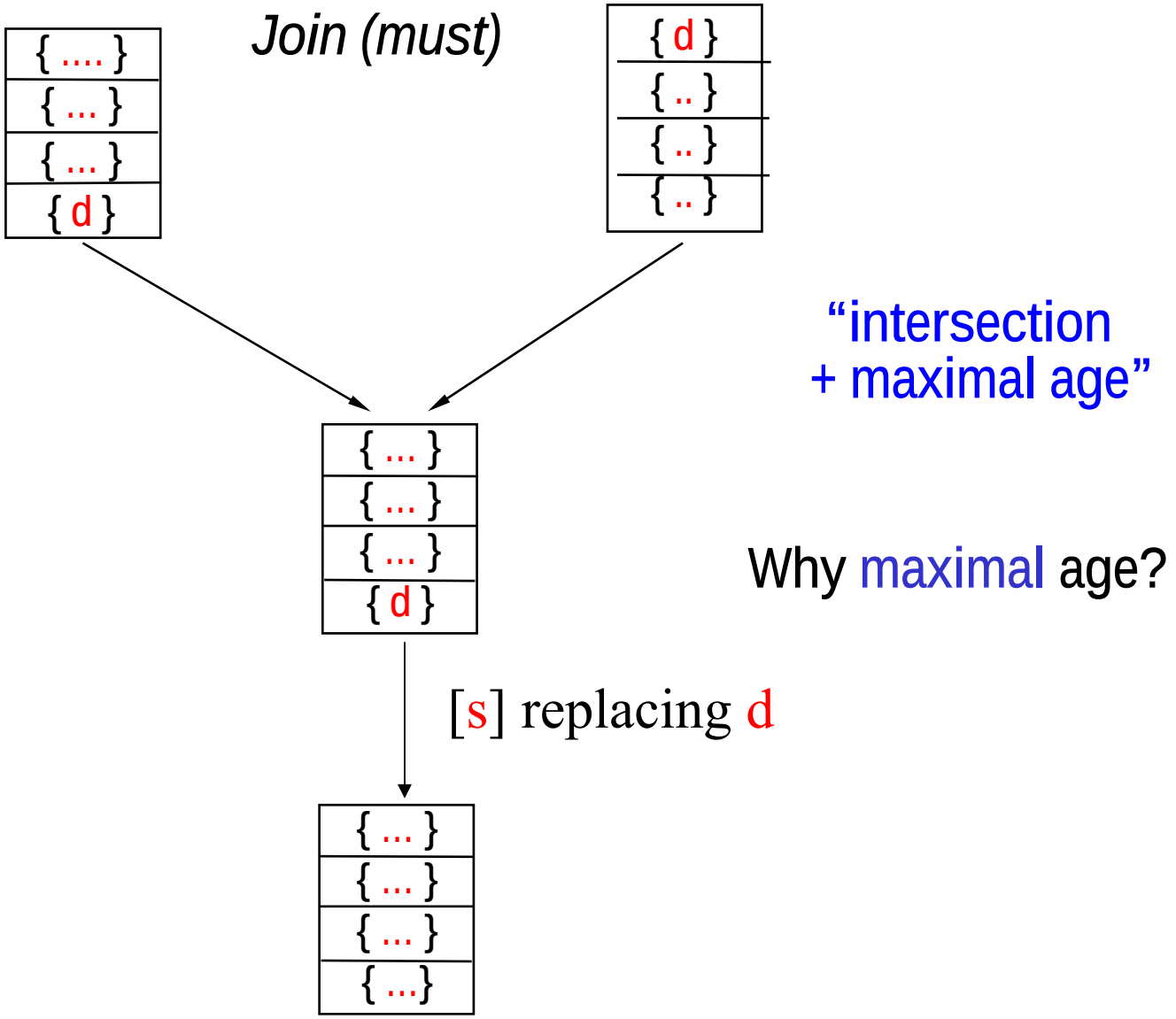
Cache Analysis: Join (must)



"intersection
+ maximal age"

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

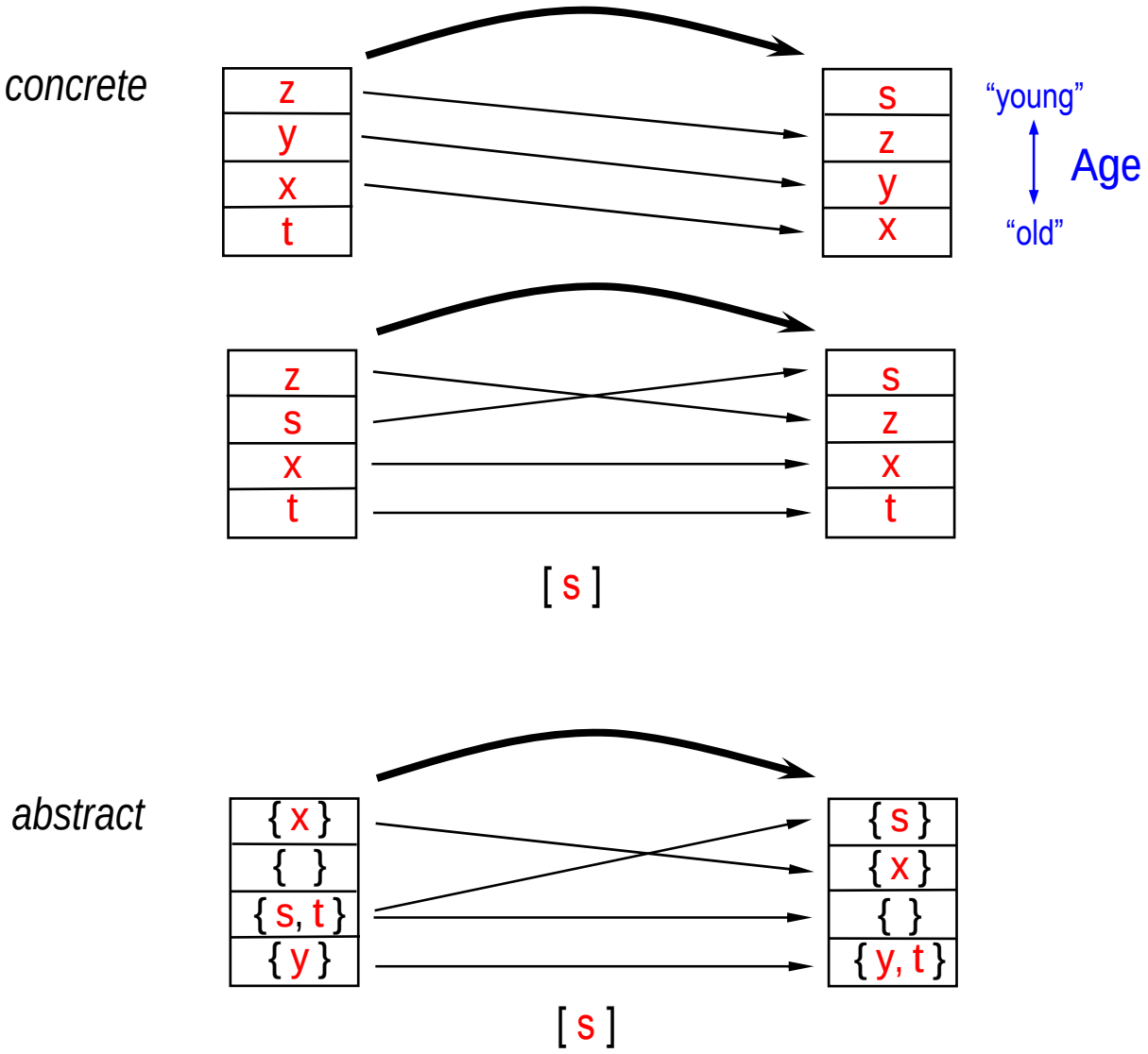
Cache Analysis: Join (must)



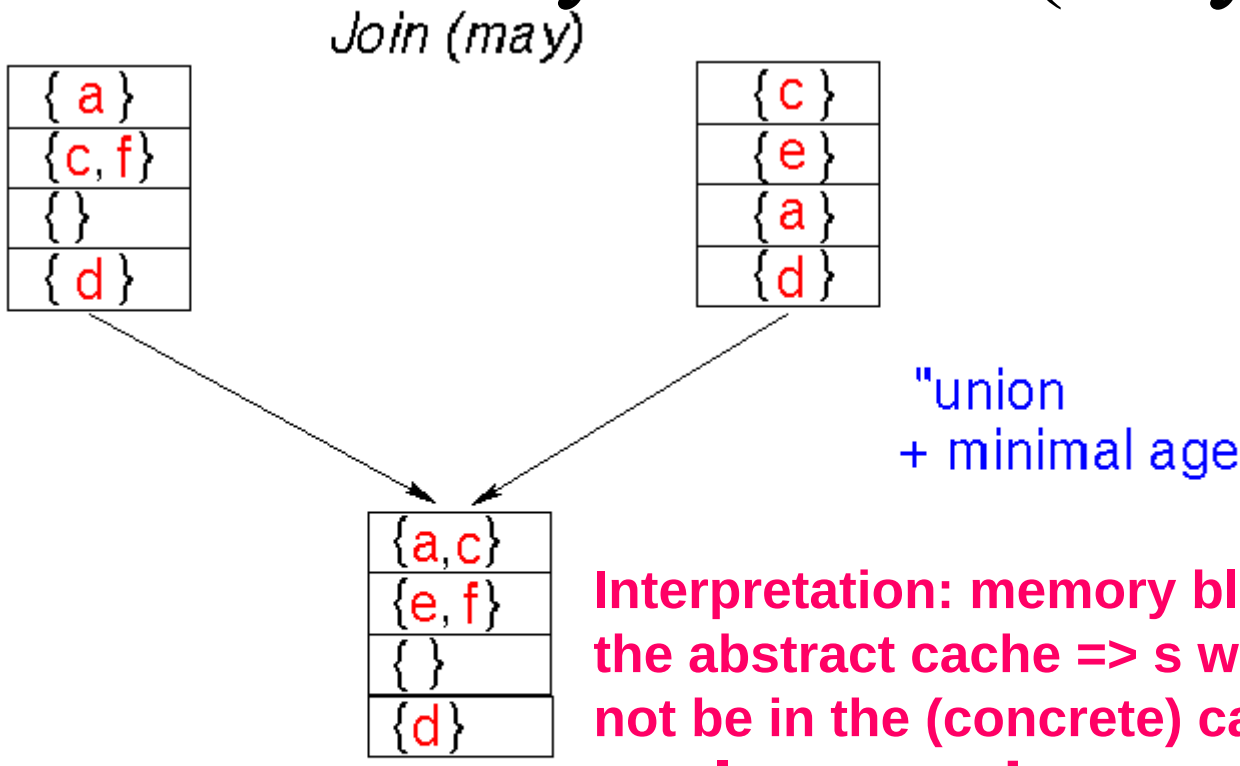
"intersection
+ maximal age"

Why maximal age?

Cache with LRU Replacement: Transfer for may



Cache Analysis: Join (may)



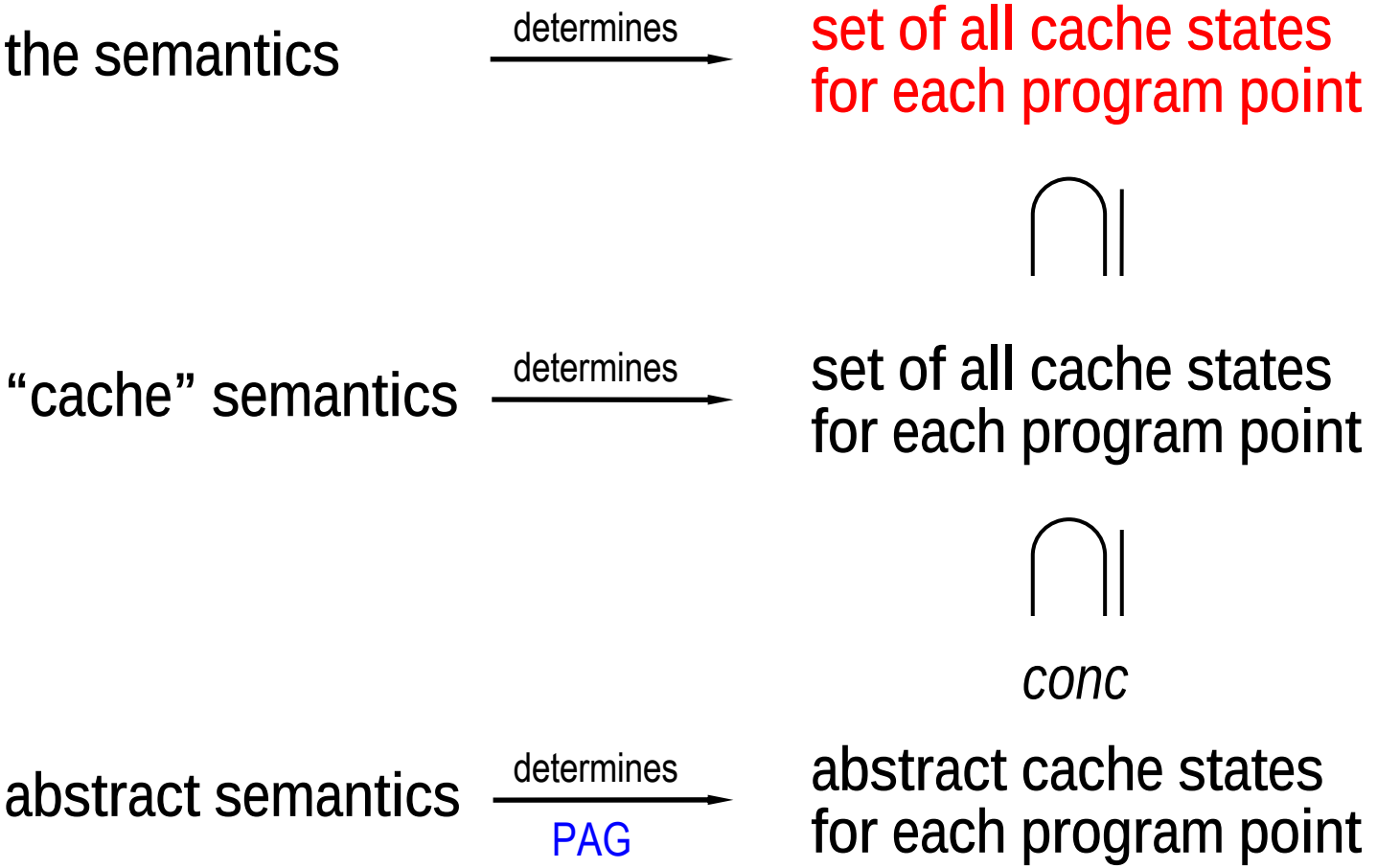
Interpretation: memory block s not in the abstract cache => s will definitively not be in the (concrete) cache

=> always miss

Question: How many references will a memory block maximally survive in the cache?

Cache Analysis

Approximation of the Collecting Semantics



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

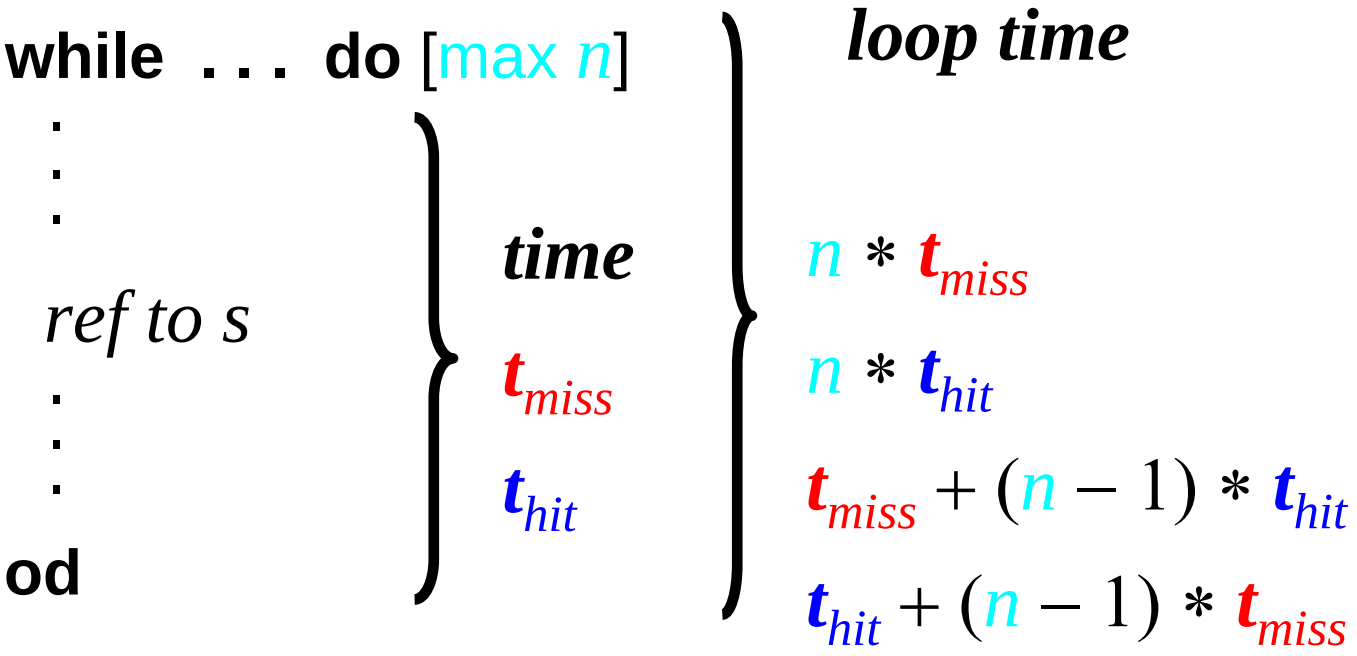
Categorization of memory references

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

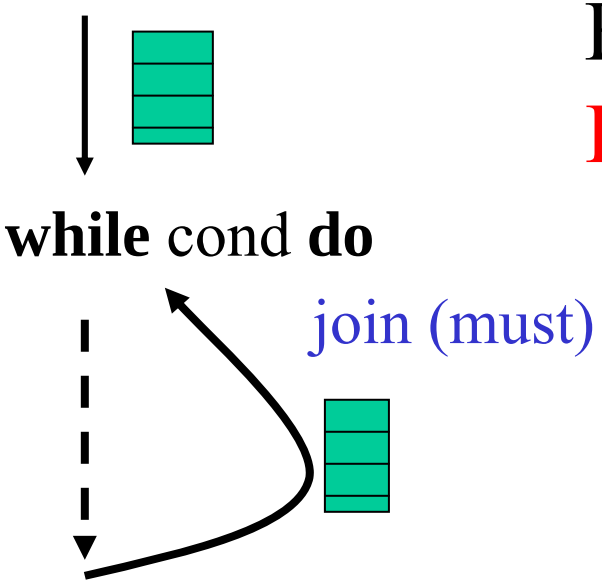
Contribution to WCET

Information about cache contents sharpens timings.



Contexts

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



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	32 - 45

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

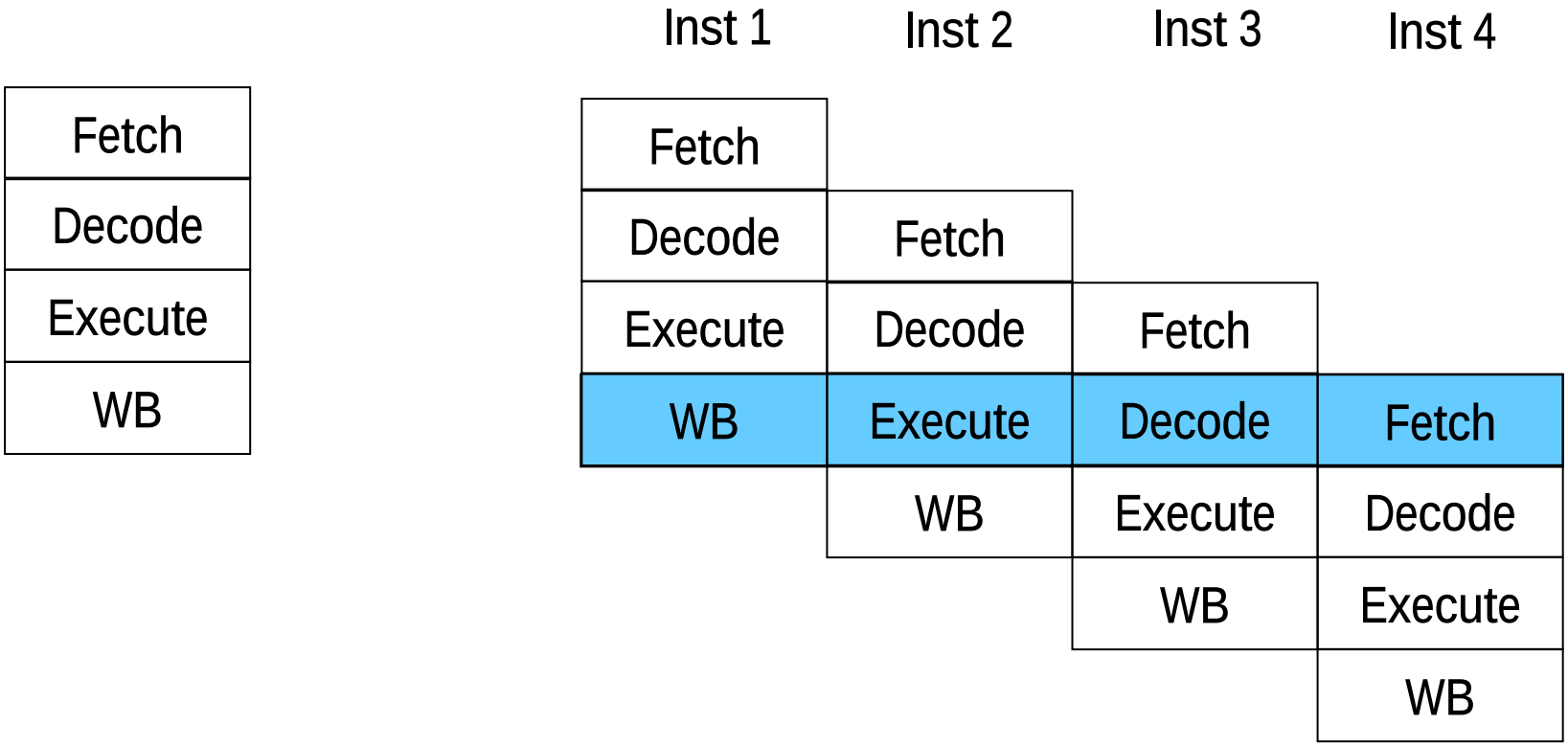
- Abstract cache contains at most one block per line
- Corresponds to direct mapped cache
- Only ¼ of capacity
- As for predictability, ¾ of capacity are lost!
- In addition: Uniform cache => instructions and data evict each other

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



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

Hardware Features: Pipelines III

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

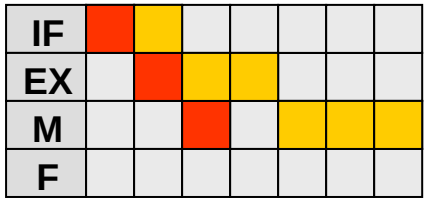
Cache analysis: prediction of cache hits on instruction or operand fetch or store

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



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

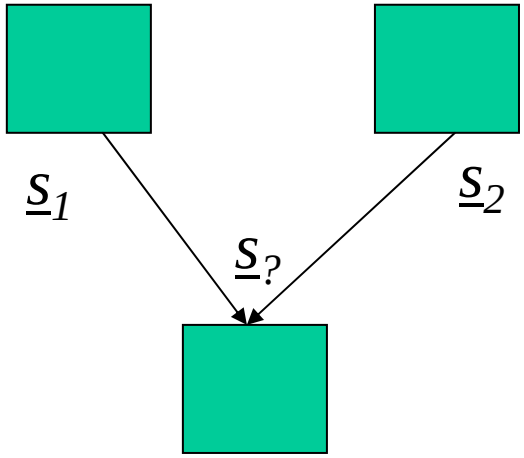
function exec (*b* : **basic block**, *s* : **abstract pipeline state**) *t*: **trace**

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

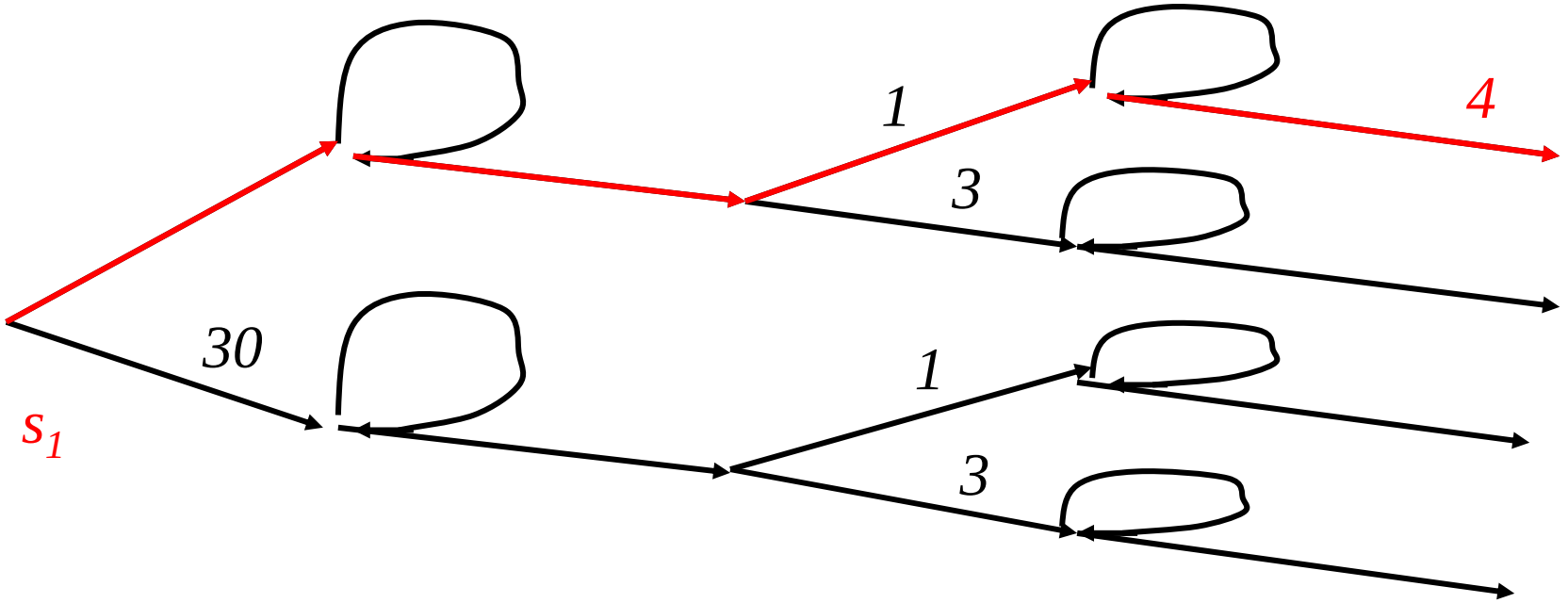
`mul`

Fetch
I-Cache miss?

Issue
Unit occupied?

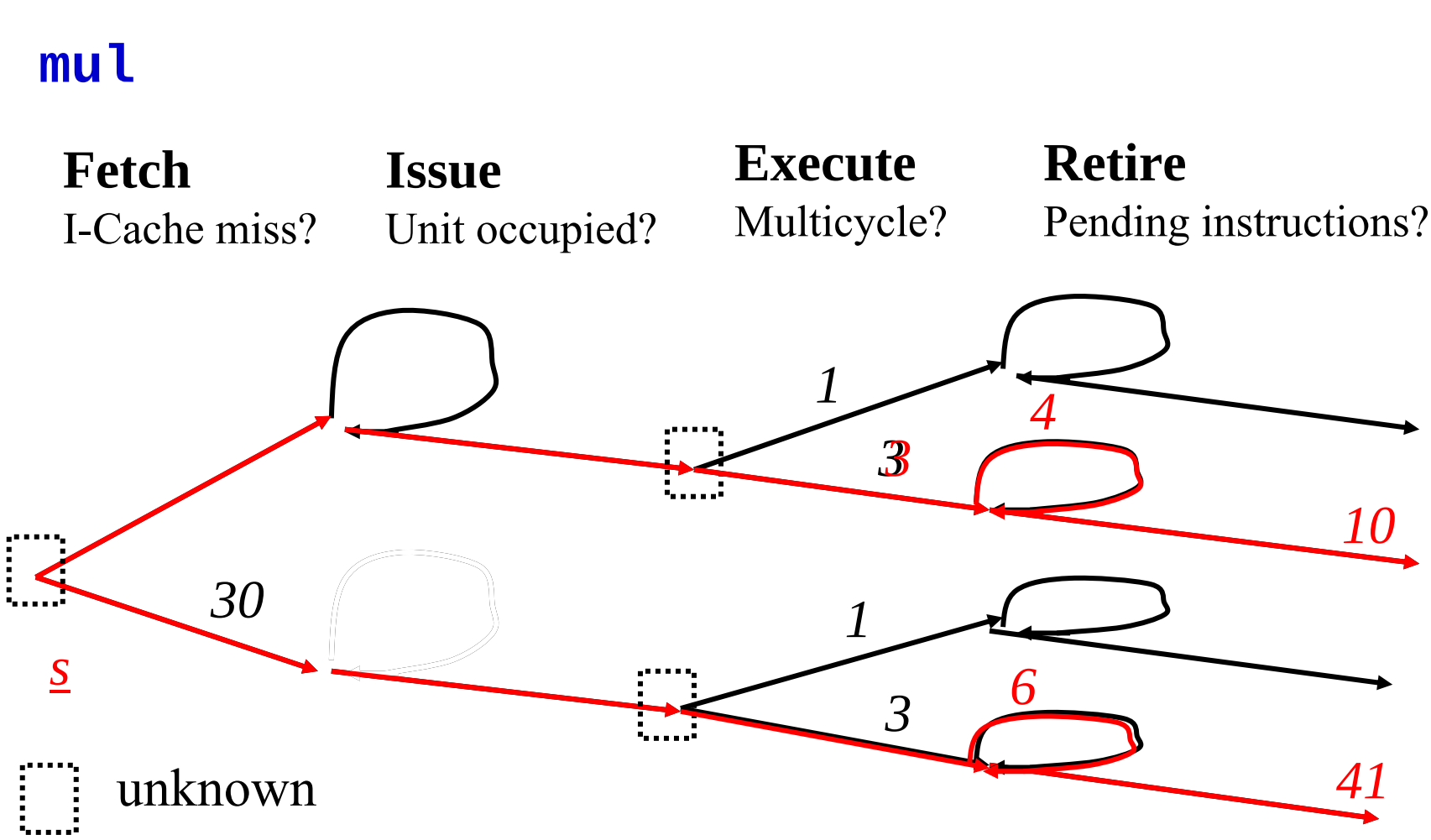
Execute
Multicycle?

Retire
Pending instructions?



Abstract Instruction-Execution

`mul`



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 = $2^{PS \times CS}$

PS = set of abstract pipeline states

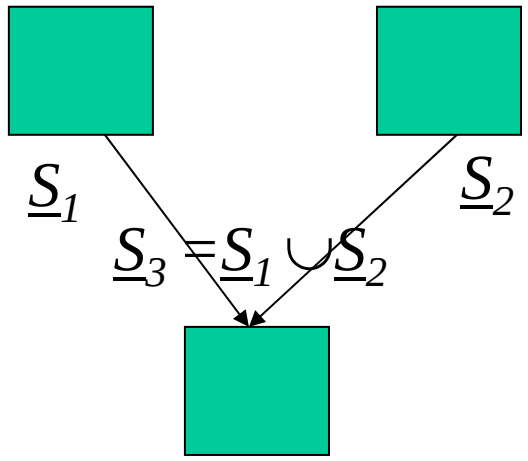
CS = set of abstract cache states

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

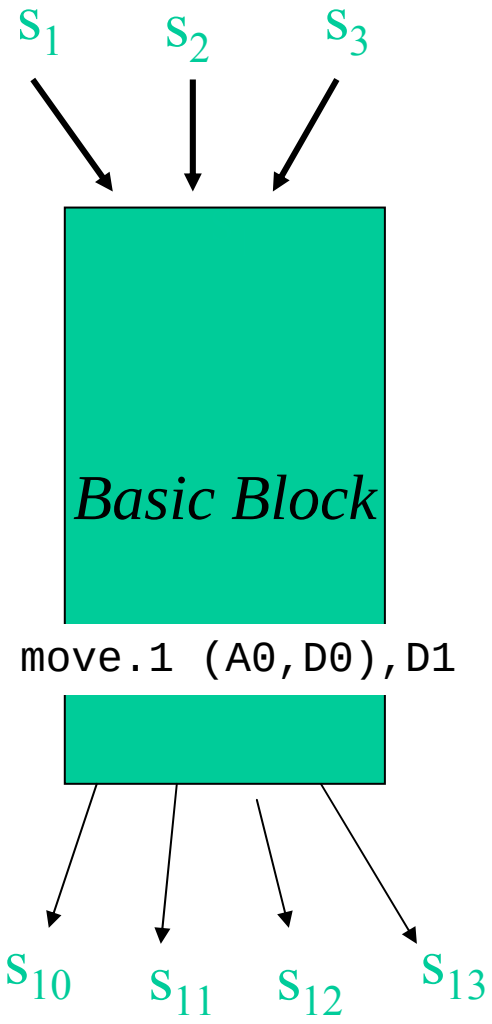
max(length(T)) - upper bound for execution time

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

Union for blocks with several predecessors.

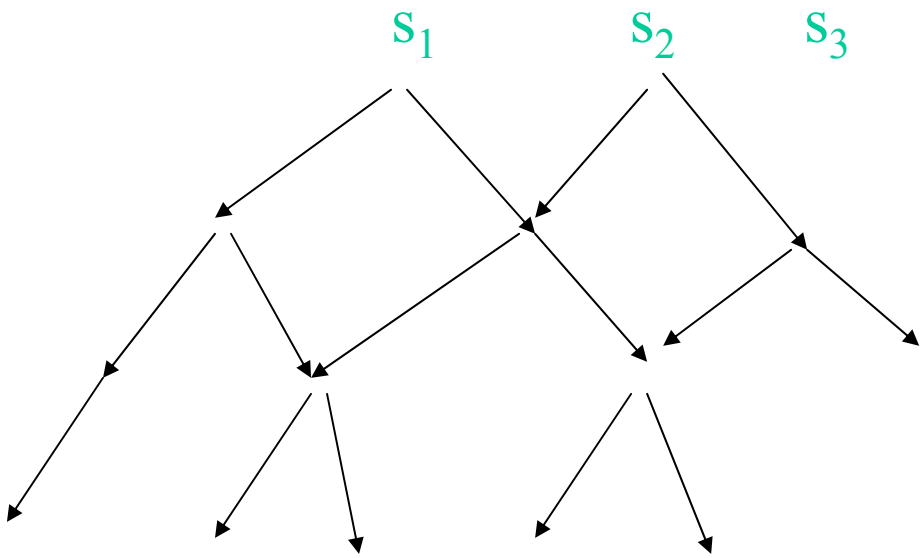


Integrated Analysis: Overall Picture



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

Cyclewise evolution of processor model for instruction



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

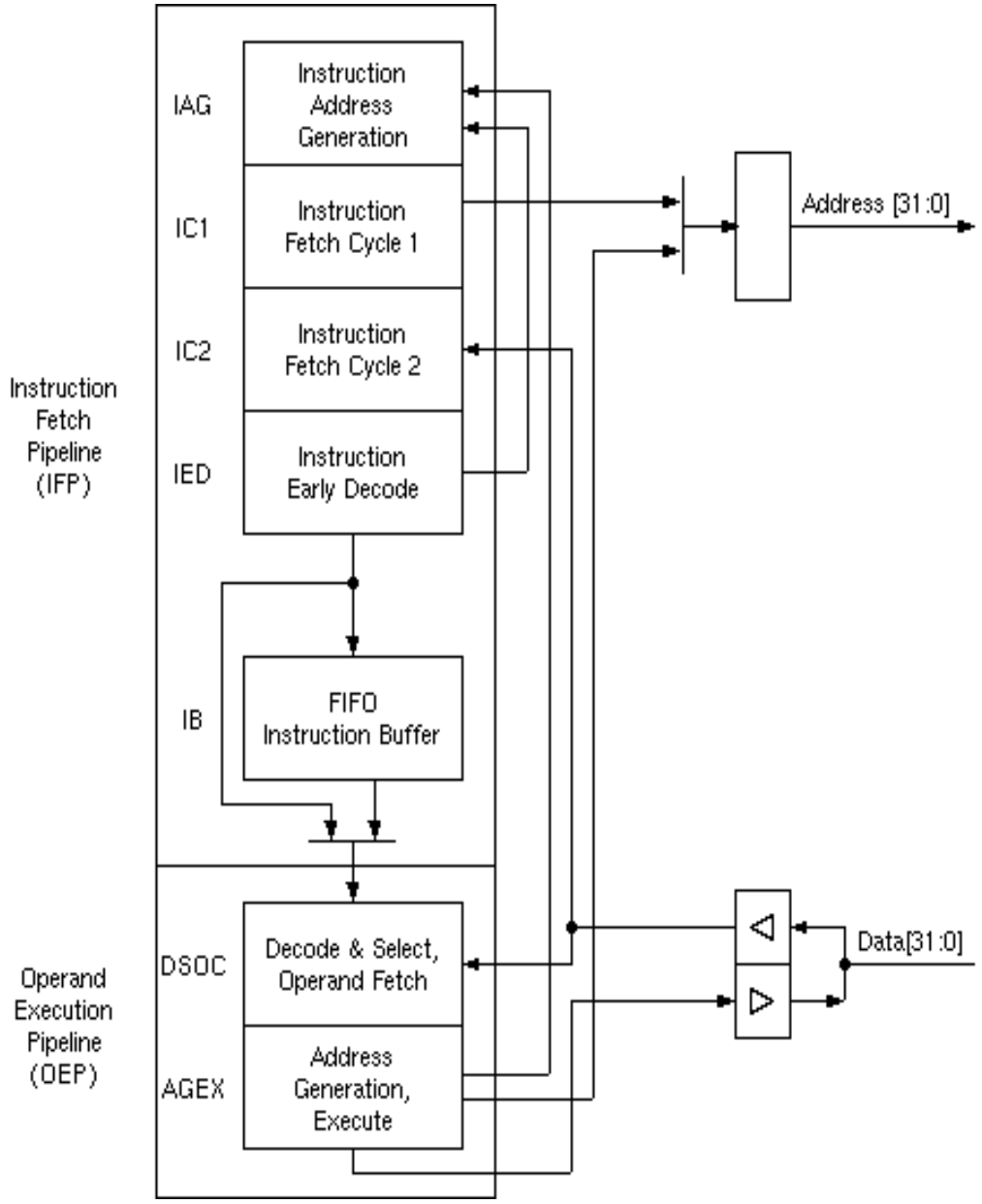
- MCF 5307 is a V3 Coldfire family member
- Coldfire is the successor family to the M68K processor generation
- Restricted in instruction size, addressing modes and implemented M68K opcodes
- MCF 5307: small and cheap chip with integrated peripherals
- Separated but coupled bus/core clock frequencies

ColdFire Pipeline

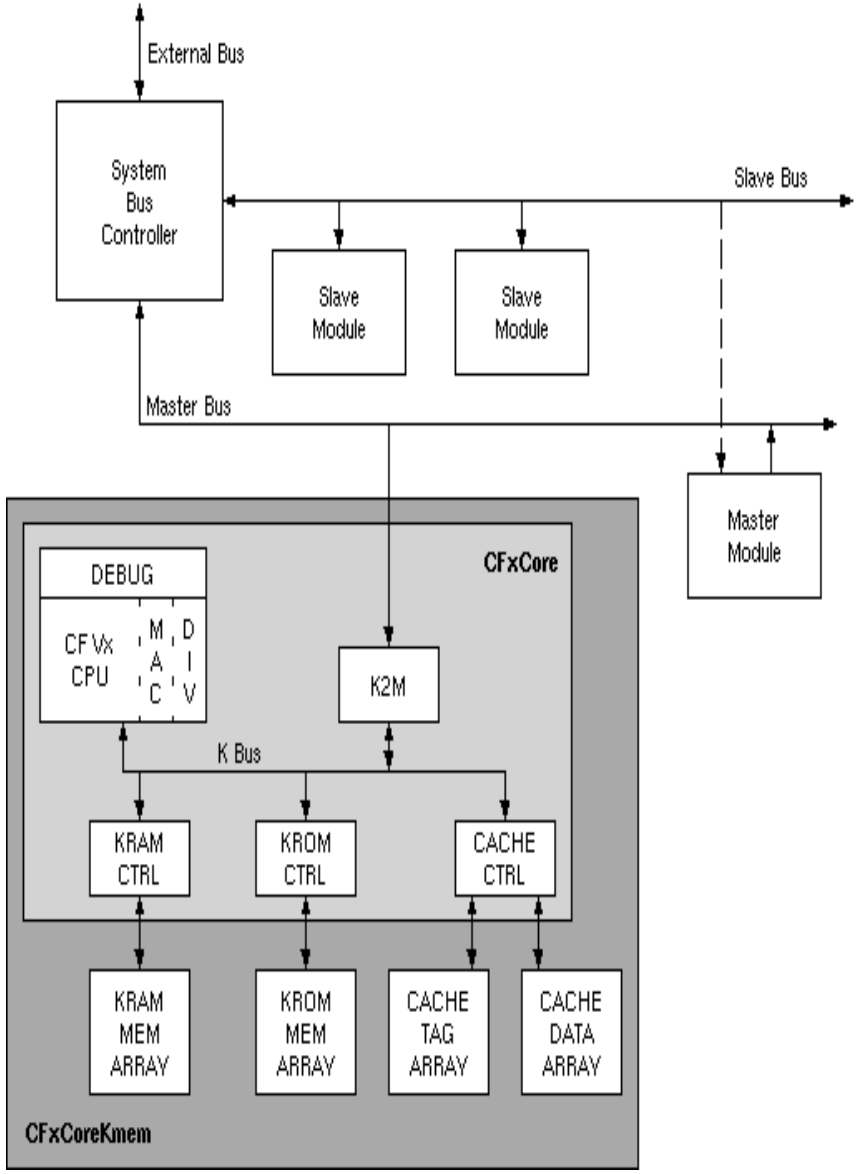
The ColdFire pipeline consists of

- 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

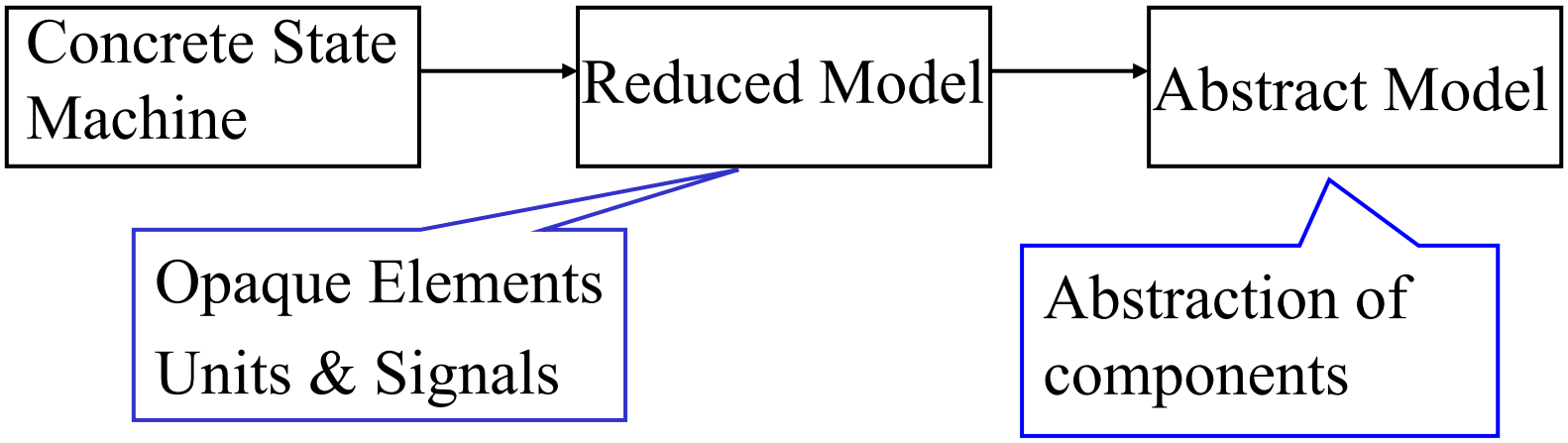


- Hierarchical bus structure
- Pipelined K- and M-Bus
- Fast K-Bus to internal memories
- M-Bus to integrated peripherals
- E-Bus to external memory
- Busses independent
- Bus unit: K2M, SBC, Cache

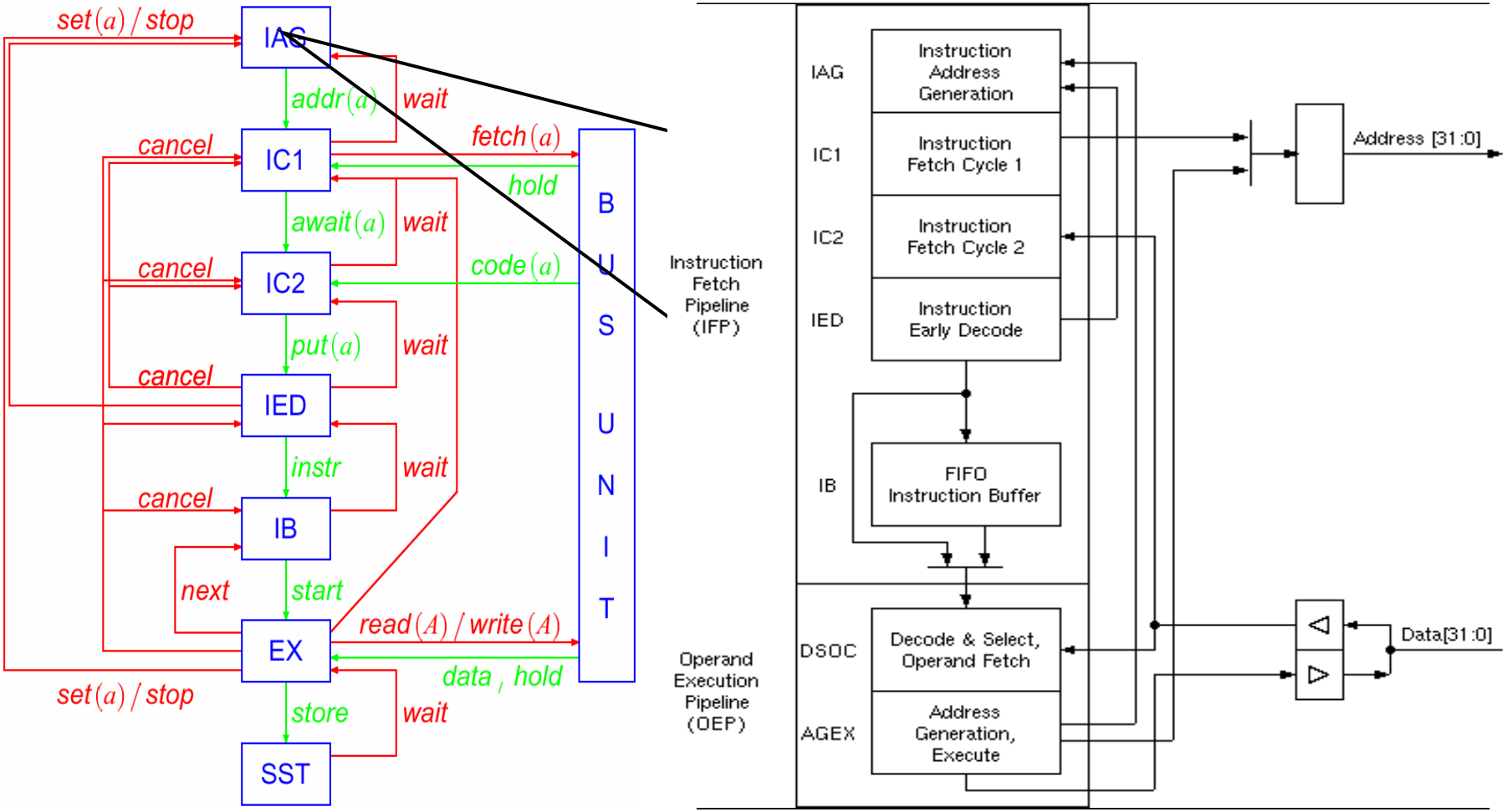


Model with Units and Signals

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

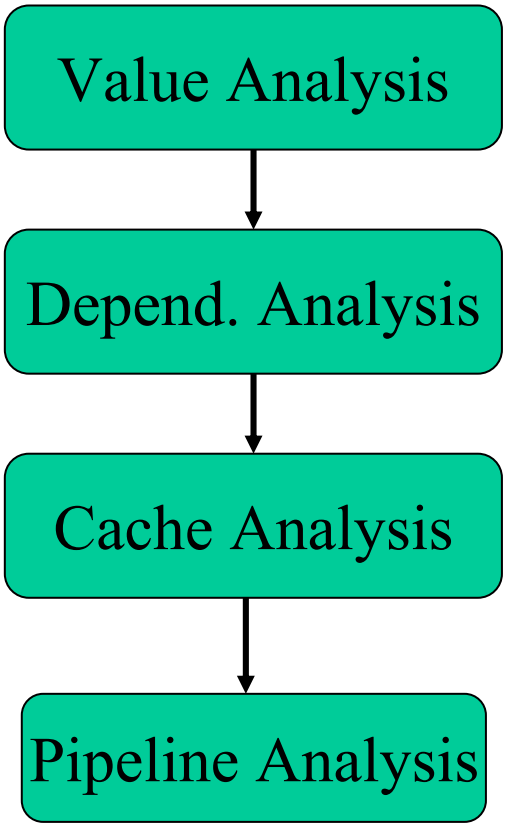
- In the reduced model, one state resulted in one new state after a one-cycle transition
- Now, one state can have several successor states
  - Transitions from set of states to set of states

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

A Simple Modular Structure



Static determ. of effective addresses

Elim. of true data dependences

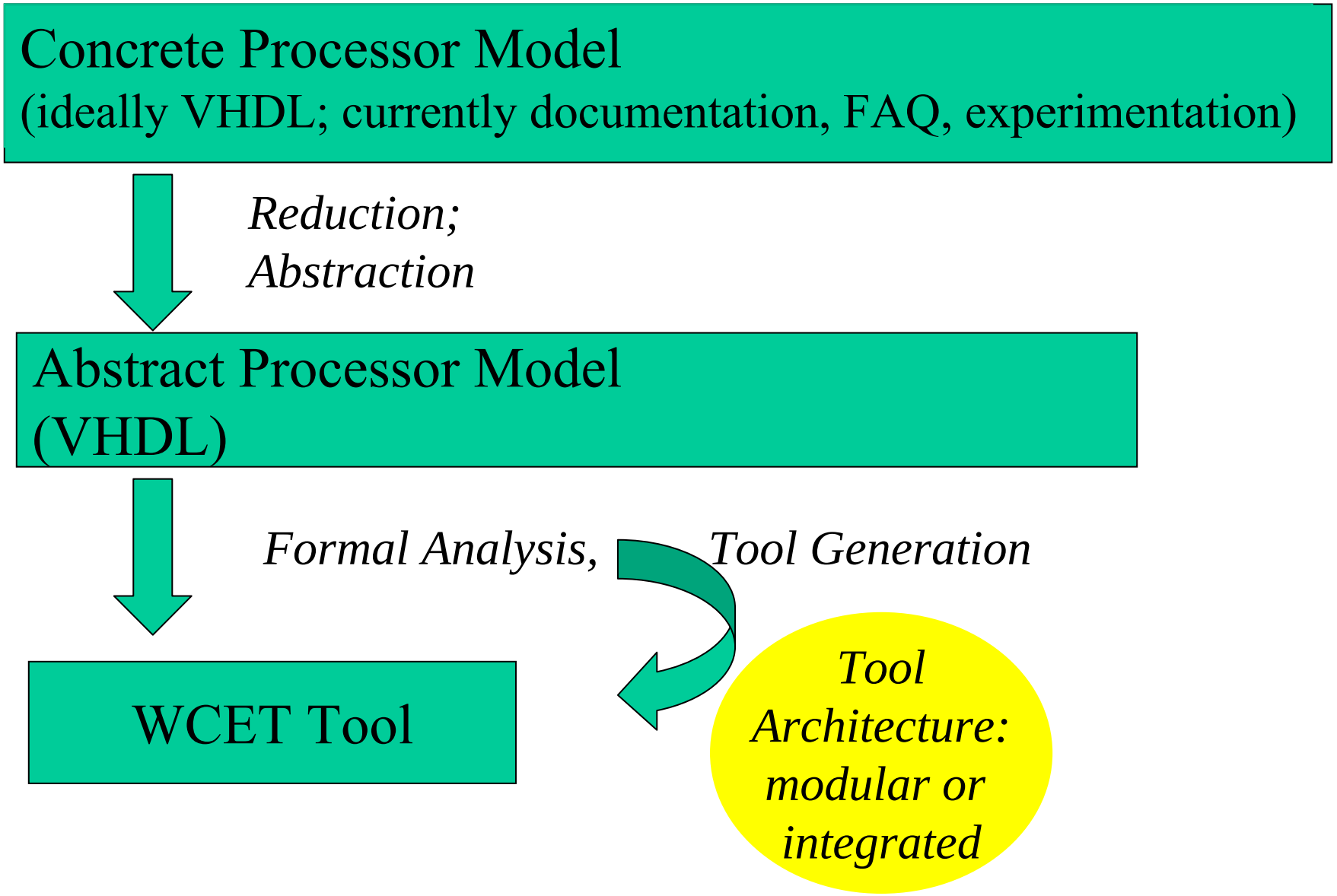
Annotation of instructions with Hit

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 “best” result states of an instruction need to be considered as input states for successor instructions! (no timing anomalies)

The Tool-Construction Process



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 modular analysis
  Problem: Bound may not exist $\Rightarrow$
  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

Examples

- ColdFire: Instruction cache miss preventing a branch misprediction
- PowerPC: Domino Effect (Diss. J. Schneider)

Challenges for the MC Community I

WCET determination by MC

- cf. VMCAI'04
- Campos/Clarke 2000 assume unit-time transitions
- Abstract Interpretation is losing information through abstraction
- MC could have complete information
- Experience shows that AI is not losing much information
- Are there cases, where AI's loss is too high, and where MC still terminates in acceptable time?

Challenges for the MC Community II

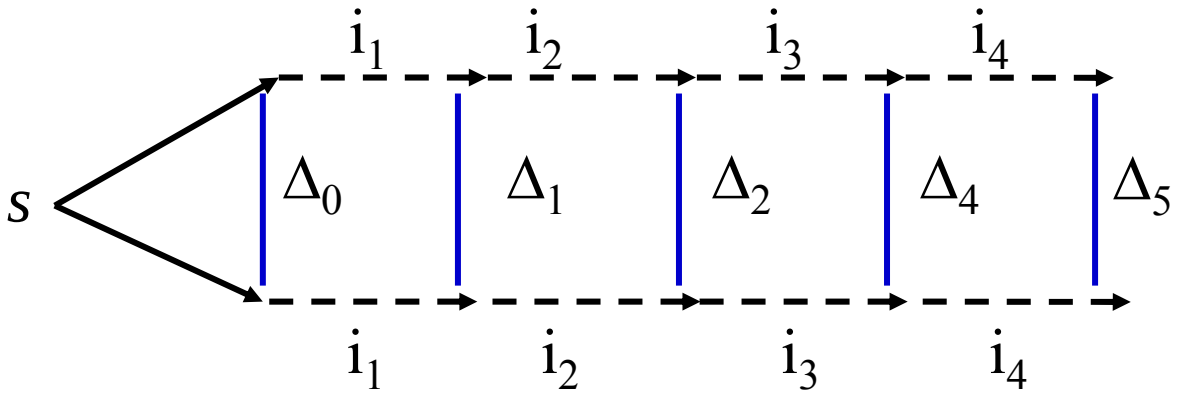
Partial-order reduction for out-of-order architectures

- Out-of-order execution of instructions increases #paths to consider
- Naïve PO reduction is unsafe: different orders may make a timing difference
- Can clever PO reduction still help?

MC for Architecture/Software Properties

- Checking for the potential of Timing Anomalies in a processor
- Checking for the potential of Timing Anomalies in a processor and a program
- Checking for the potential of Domino Effects in a processor
- Checking for the potential of Domino Effects in a processor and a program

Checking for Timing Anomalies



At each step, check for the conditions for TA

Note: Counting and comparing execution times is required!

Bounded Model Checking

- TA will occur on paths of bounded lengths
- Bounds depend on architectural parameters
  - Length of the pipeline
  - Length of queues, e.g., prefetch queues, instruction buffers
  - Maximal latency of instructions
- No TA condition satisfied inside bound $\Rightarrow$ no TA
- How to determine the bound is open

Checking for Domino Effects

- Identify cycle with TA (under equality of abstract states), (analogy to Pumping Lemma)
- Cycle will increase anomalous effect

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

Examples

- ColdFire: Instruction cache miss preventing a branch misprediction
- PowerPC: Domino Effect (Diss. J. Schneider)

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

Integrated Analysis II

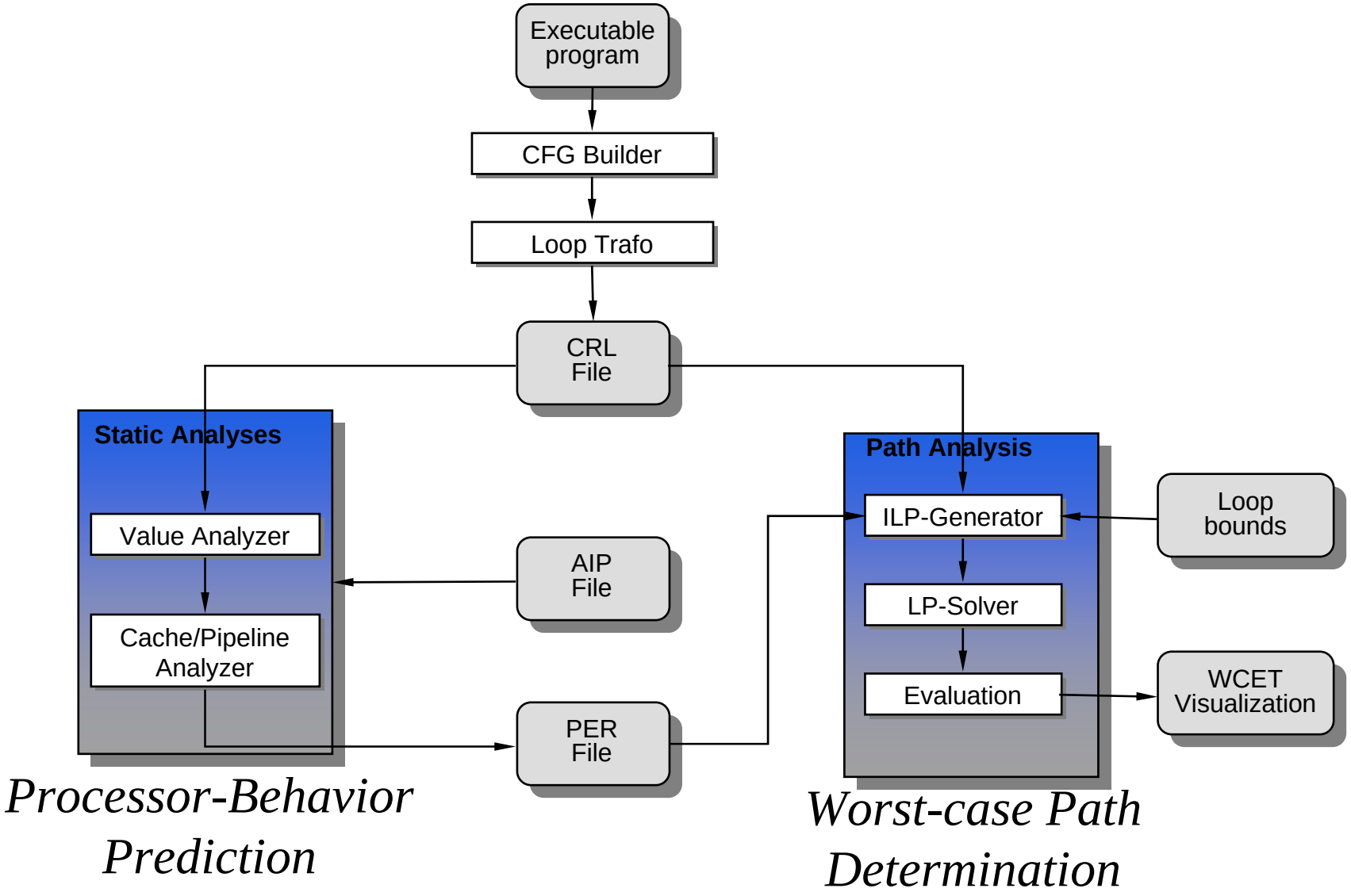
- Abstract state is a set of (reduced) concrete processor states,
  computed: superset of the collecting semantics
- Sets are small,
  pipeline is not too history sensitive
- Joins are set union

Loop Counts

- loop bounds have to be known
- user annotations are needed

```
# 0x0120ac34 -> 124 routine _BAS_Se_RestituerRamCritique
 0x0120ac9c 20
```

Overall Structure



Path Analysis

by Integer Linear Programming (ILP)

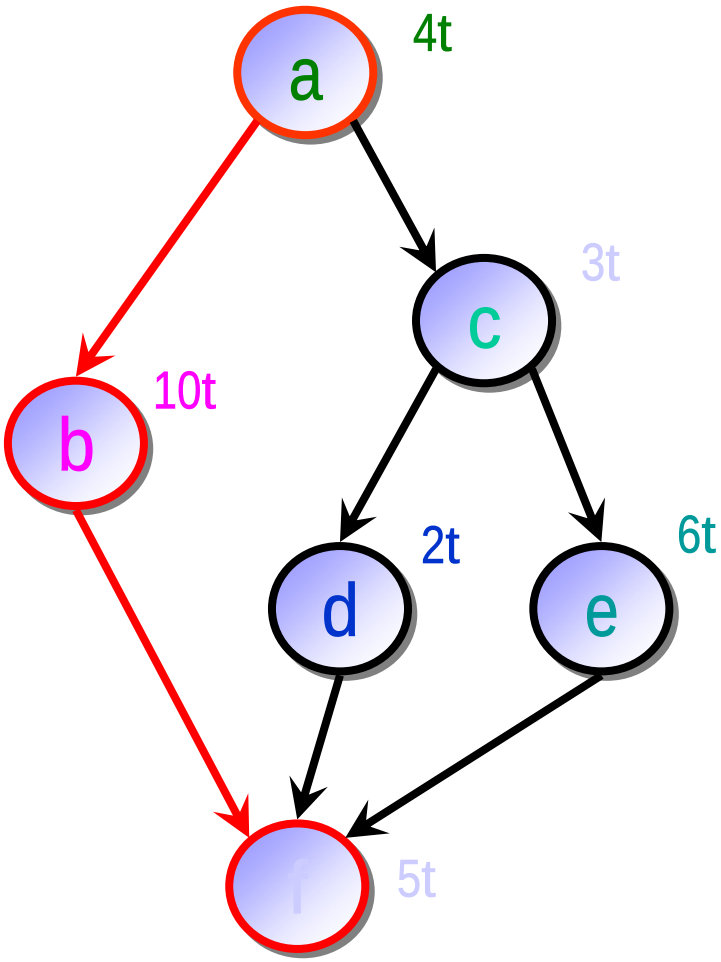
- Execution time of a program =

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

- 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

MCF 5307: Results

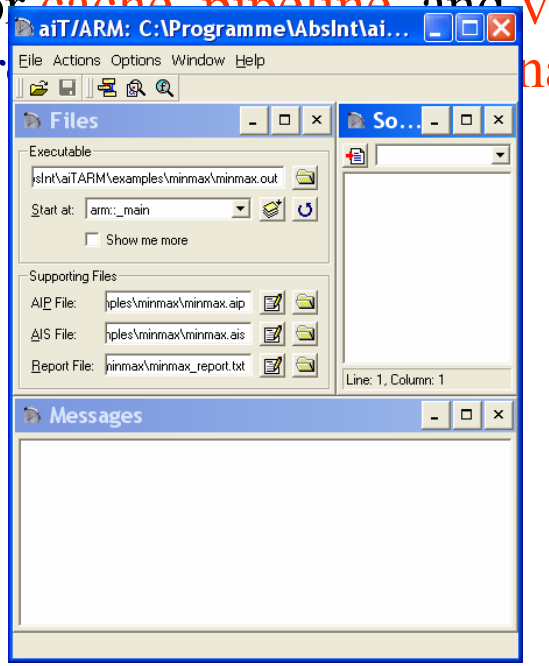
- The value analyzer is able to predict around 70-90% of all data accesses precisely (Airbus Benchmark)
- The cache/pipeline analysis takes reasonable time and space on the Airbus benchmark
- The predicted times are close to or better than the ones obtained through convoluted measurements
- Results are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

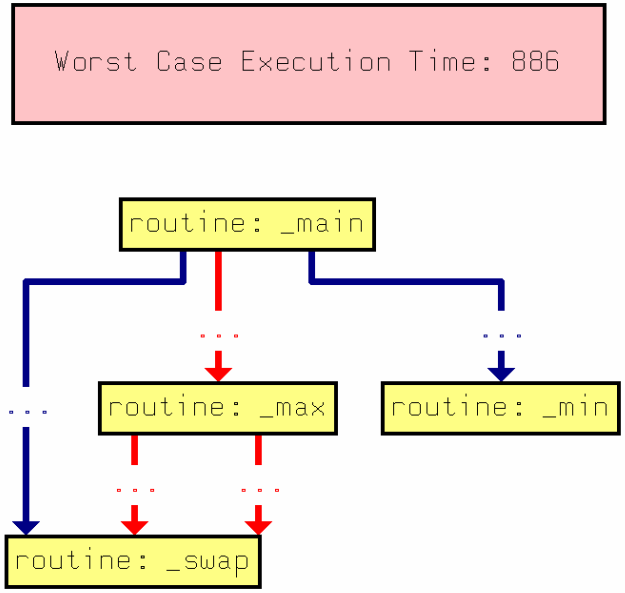
Timing-Analysis Tool aiT

aiT WCET Analyzer

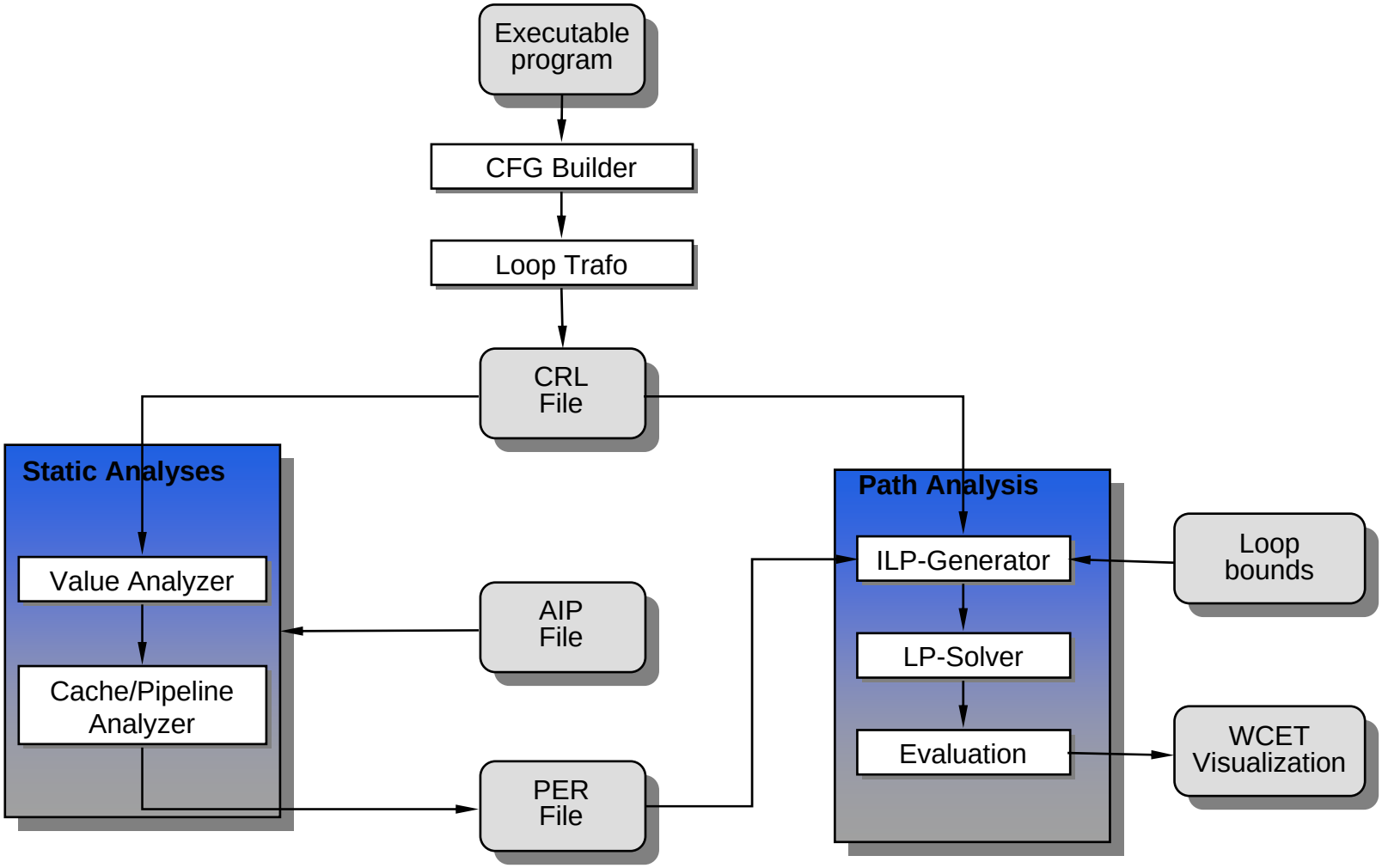
A Solution to the Timing Problem

- Combines global program analysis by abstract interpretation for cache, pipeline, and value analysis with integer linear pr… …nalysi…





aiT WCET Analyzer Structure



Input/Output

Application Code

```
void Task (void)
{
   variable++;
   function();
   next++;
   if (next)
      do this;
   terminate()
}
```

Parameters (.aip)*

```
INTERPROC: vivu4
MEMORY_AREA: 0 0xFFFFFFFE 1:1 1 READ&WRITE CODE&DATA
```

Specifications (.ais)*

```
clock 10200 kHz ;
loop "_codebook" + 1 loop exactly 16 end ;
recursion "_fac" max 6;
SNIPPET "printf" IS NOT ANALYZED AND TAKES MAX 333 CYCLES;
flow "U_MOD" + 0xAC bytes / "U_MOD" + 0xC4 bytes is max 4;
area from 0x20 to 0x497 is read-only;
```

Compiler
Linker

Executable (.elf / *.out)*

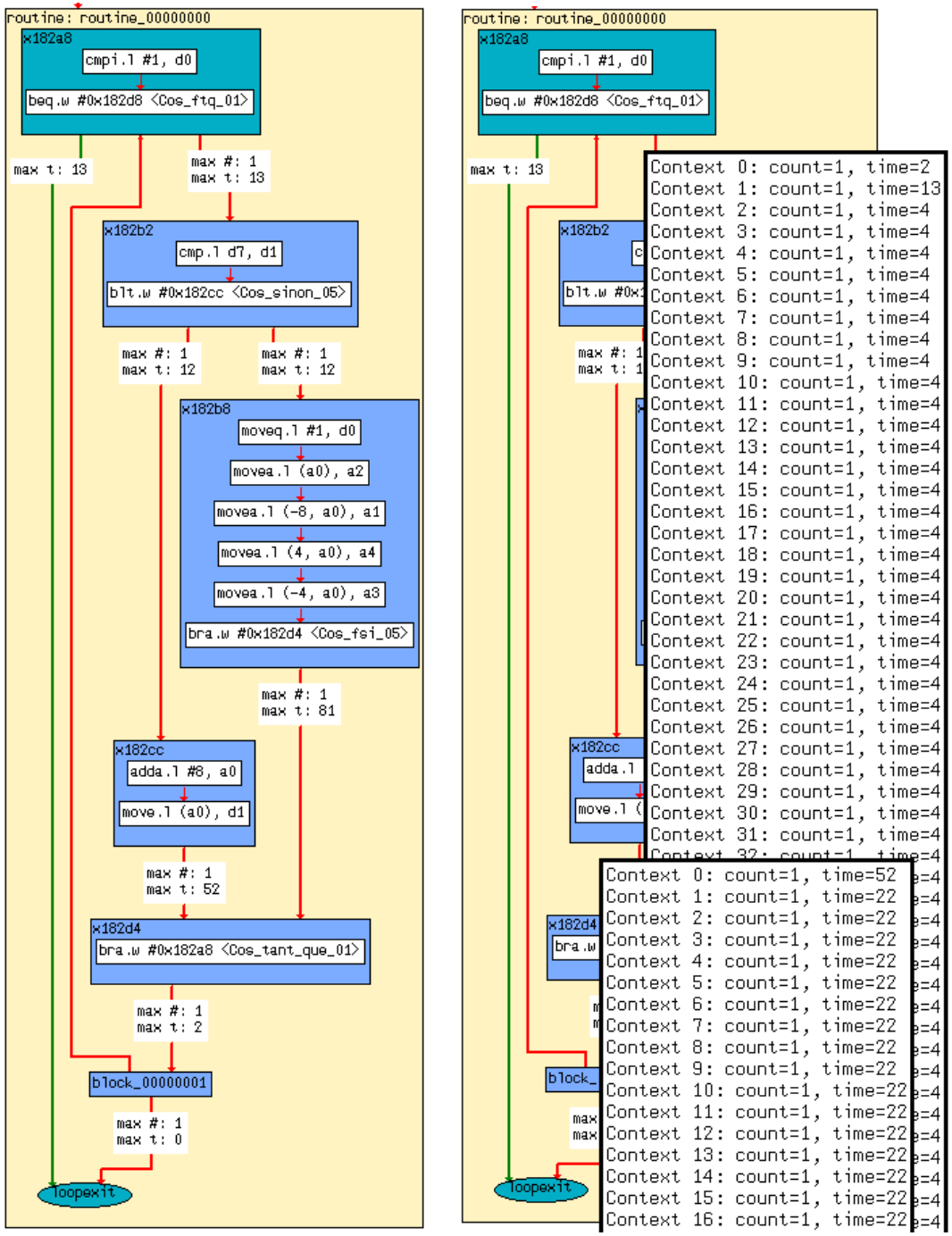
aiT

Entry Point

- Worst Case Execution Time
- Visualization, Documentation

Interprocedural Analysis/ Analysis of Loops

- Loops are analyzed like procedures
- This allows for
  - Virtual inlining
  - Virtual unrolling
  - Better address resolution
  - Burst accesses
  - Selectable precision
- Optional user constraints



Challenge: Reconstruction of CFG

- Indirect Jumps
  - Case/Switch statements as compiled by the C-compiler are automatically recognized
  - For hand-written assembly code annotations might be necessary

    INSTRUCTION *ProgramPoint* BRANCHES TO *Target*$_1$, …, *Target*$_n$
- Indirect Calls
  - Can often be recognized automatically if a static array of function pointers is used
  - For other cases

    INSTRUCTION *ProgramPoint* CALLS *Target*$_1$, …, *Target*$_n$

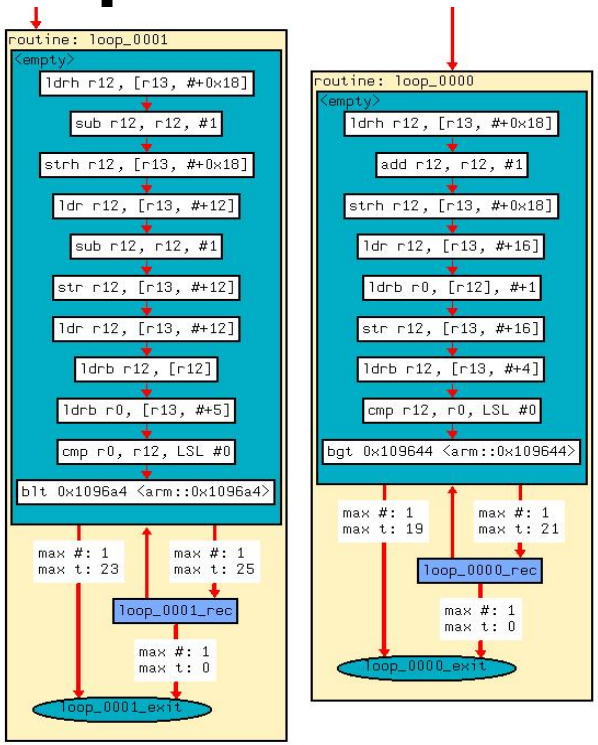
Loops

- aiT includes a loop bound analysis based on interval analysis and pattern matching that is able to recognize the iteration count of many „simple“ FOR loops automatically

- Other loops need to be annotated
  - Example:
    ```
    loop "_prime" + 1 loop end max 10;
    ```

TargetLink Loops

```
/* X-vector: ascending linear search. */
      ...
        while ( x > *(x_table++) ) low++;
      ...
 /* Y-vector: descending linear search. */
      ...
        do {
            --y_low;
            --y_table;
        } while ( y < *y_table );
     ...
```



aiT produces automatic „Proposals“ for loop iterations:

Data dependent loop bound of [0,5] iterations detected. Please verify this!

Data dependent loop bound of [1,5] iterations detected. Please verify this!

ASCET-SD Loops

```
function(map m) {
 ...
 for((map->ctr1->low = 1); (map->ctr1->low <= 3); (map->ctr1->low++)) {
   for((map->ctr2->low = 1); (map->ctr2->low <= 6); (map->ctr2->low++)) {
     /* code with memory accesses via pointers.... */
   }
 }
...
```

aiT produces automatic „Proposals“ for loop iterations:

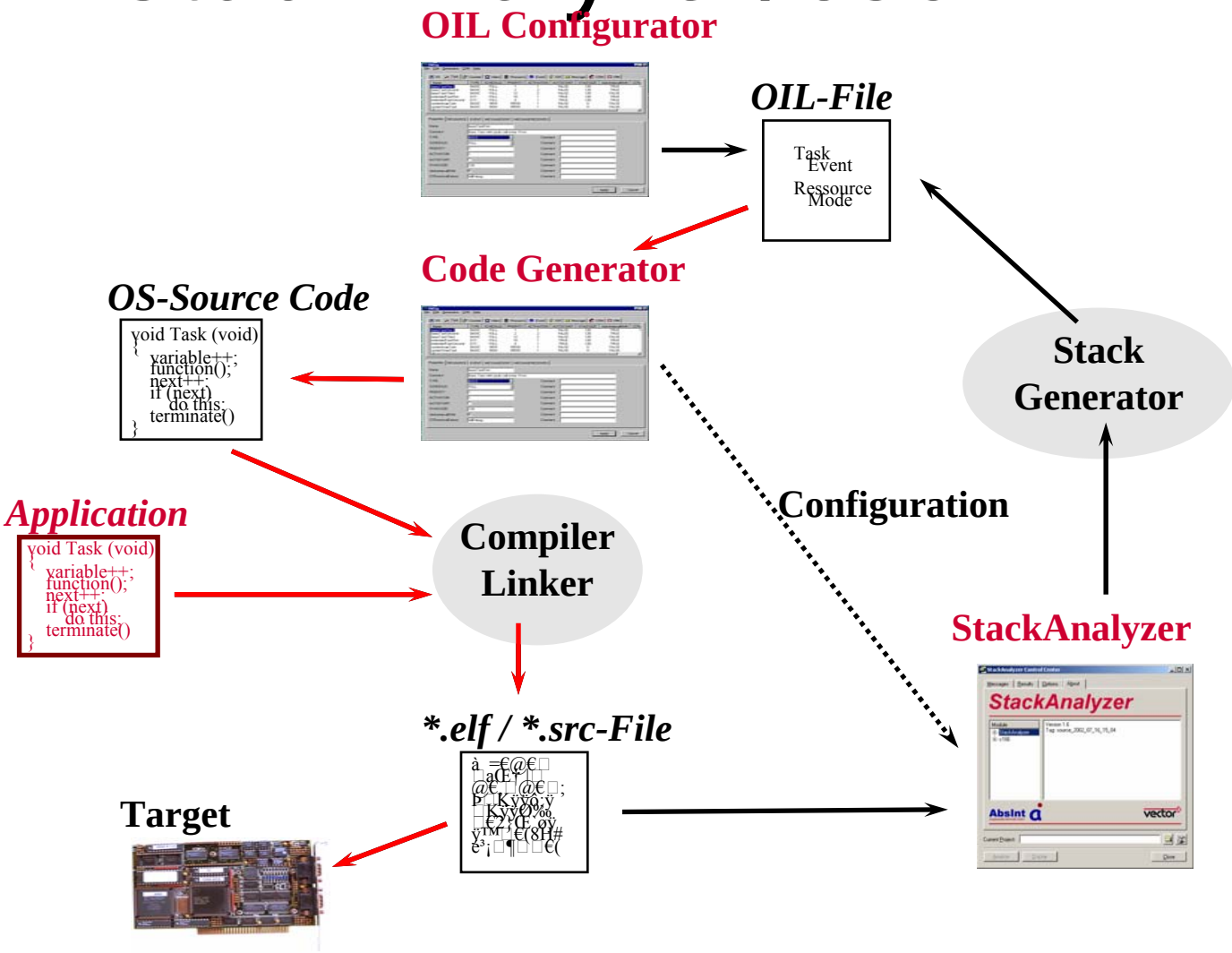
```
Data dependent loop bound of [3] iterations detected. Please verify this!
Data dependent loop bound of [6] iterations detected. Please verify this!
```

Source Level Annotations

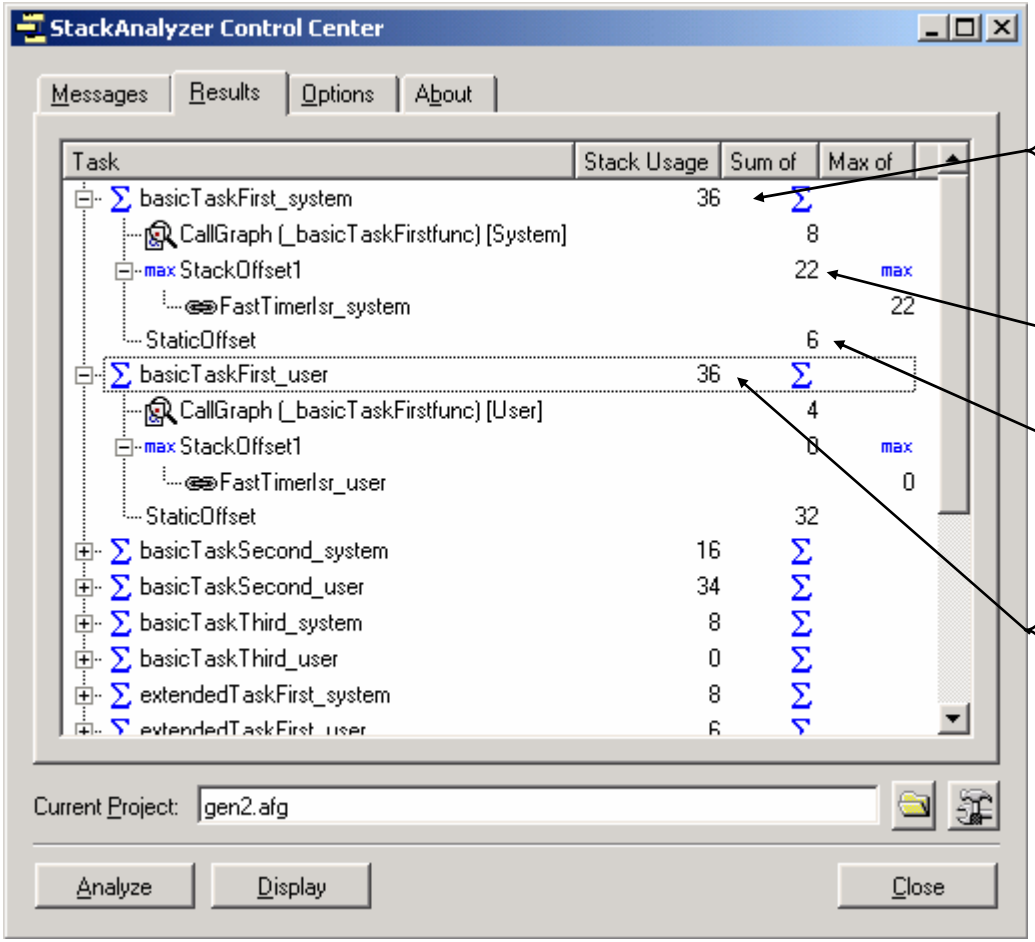
```
bool divides (uint n, uint m) {
/* ai: SNIPPET HERE NOT ANALYZED, TAKES MAX 173 CYCLES; */
  return (m % n == 0);
}

bool prime (uint n) {
  uint i;
  if (even (n))
  /* ai: SNIPPET HERE INFEASIBLE; */
      return (n == 2);
  for (i = 3; i * i <= n; i += 2) {
  /* ai: LOOP HERE MAX 20; */
      if (divides (i, n))
          return 0;
  }
  return (n > 1);
}
```

Integration Example: StackAnalyzer/osCAN



StackAnalyzer Results



Maximum system stack usage

Category 1 ISR

Task context offset

Maximum User stack usage

(osCAN C16x/ST10)

Conclusion I

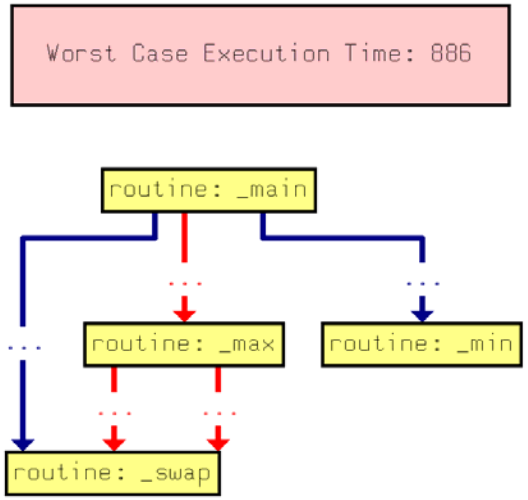
- Automatic loop bound and „array call“ recognition by static analysis reduces the amount of user annotations dramatically
- For situations where automatic analysis fails, convenient specification and annotation formats are available
- Annotations for library functions (RT, communication) and RTOS functions can be provided in separate files by the respective developers (on source level or separately)

Preliminary feedback from automotive users

- Providing the annotations (targets of indirect function calls and loop bounds) can require some effort
- Precision ✓
  - "Arbitrary ILP constraints" basically not used
- Analysis speed ✓
- Integration into development process still to be done

Conclusion II

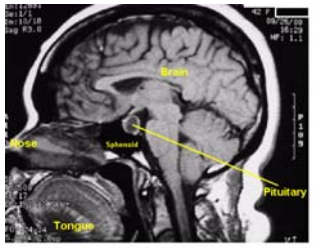
- **ai**T enables development of complex hard real-time systems on state-of-the-art hardware.
- Increases safety.
- Saves development time.
- Precise timing predictions enable the most cost-efficient hardware to be chosen.

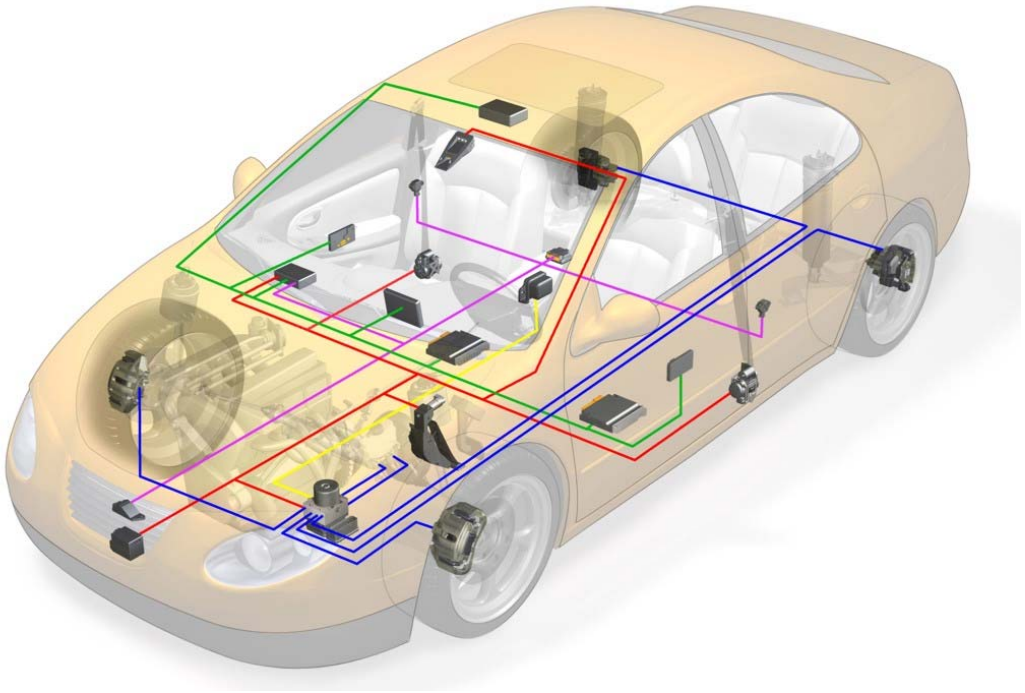


email: info@AbsInt.com
http://www.AbsInt.com

aiT WCET Analyzer
The Product

- **ai**T WCET Analyzer helps developers of safety-critical applications to verify that their programs will always react fast enough.





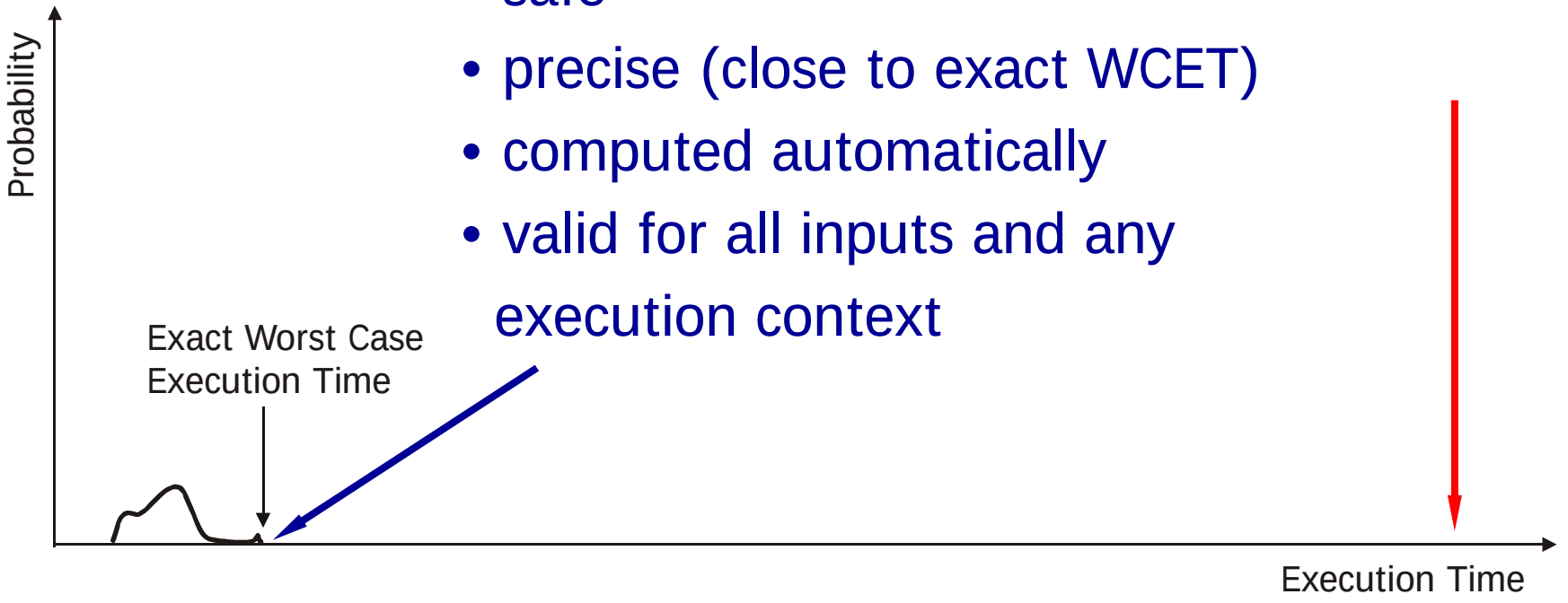
aiT WCET Analyzer

The Solution to the Timing Problem

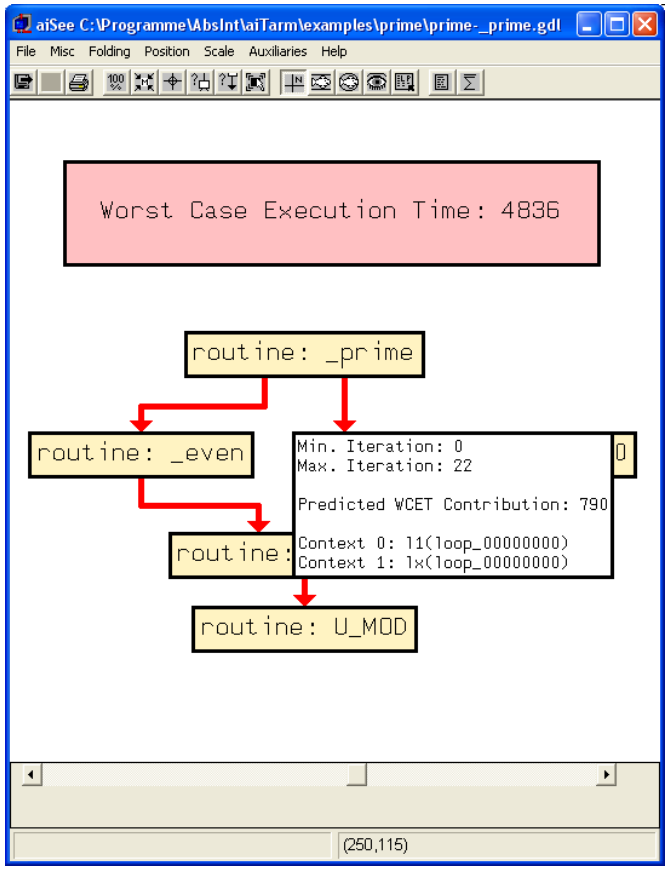
aiT WCET Analyzer results:

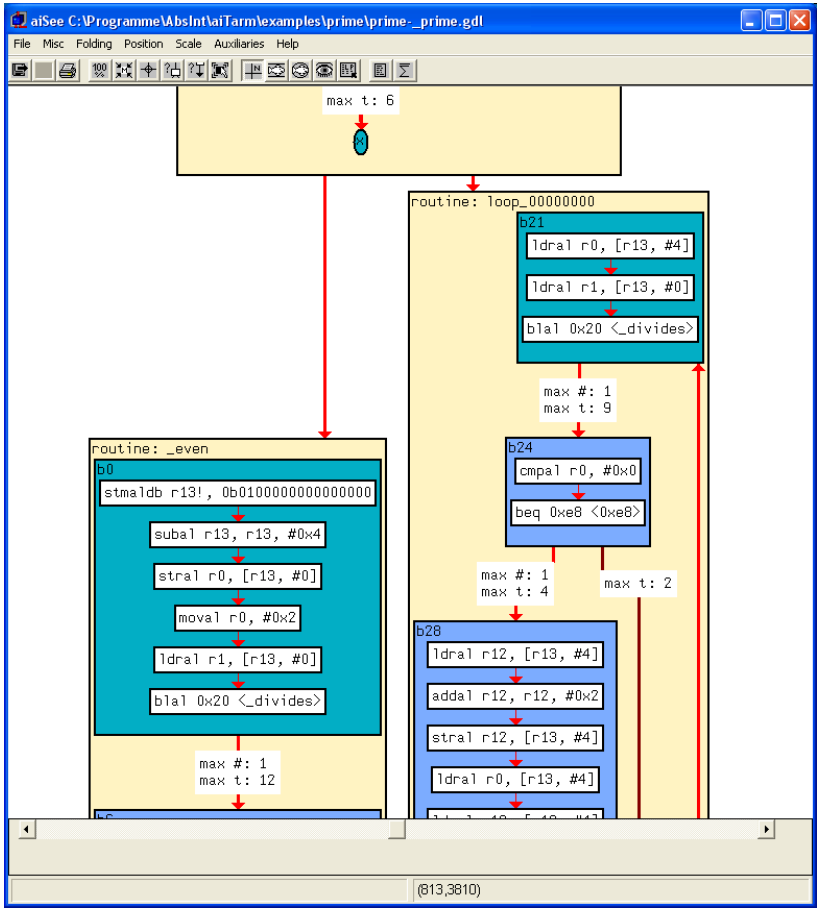
- safe
- precise (close to exact WCET)
- computed automatically
- valid for all inputs and any execution context

HUGE overestimation by naive WCET methods



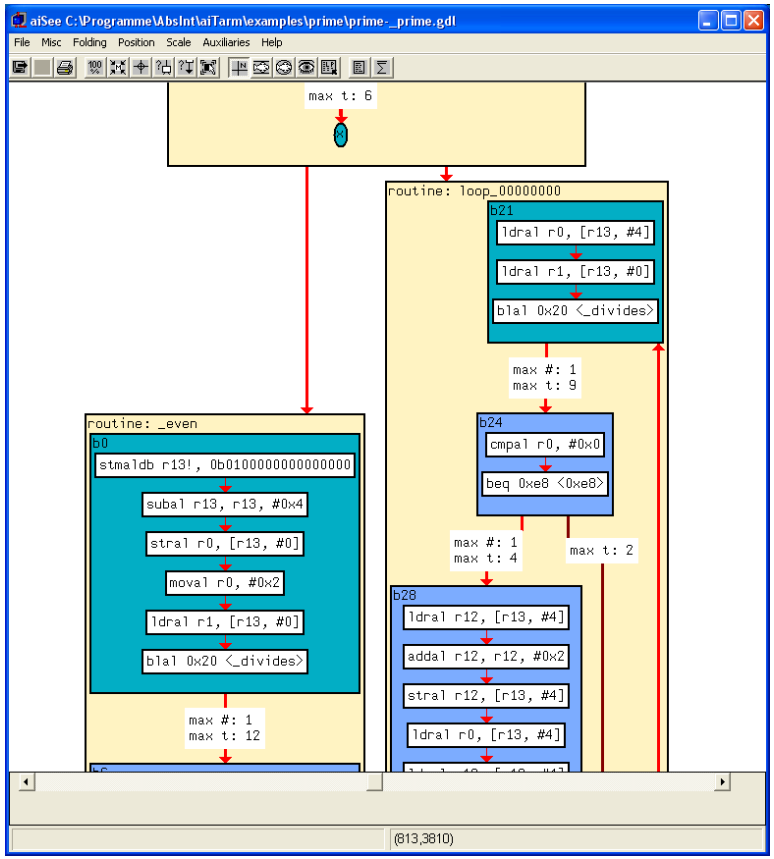
aiT: Timing Details

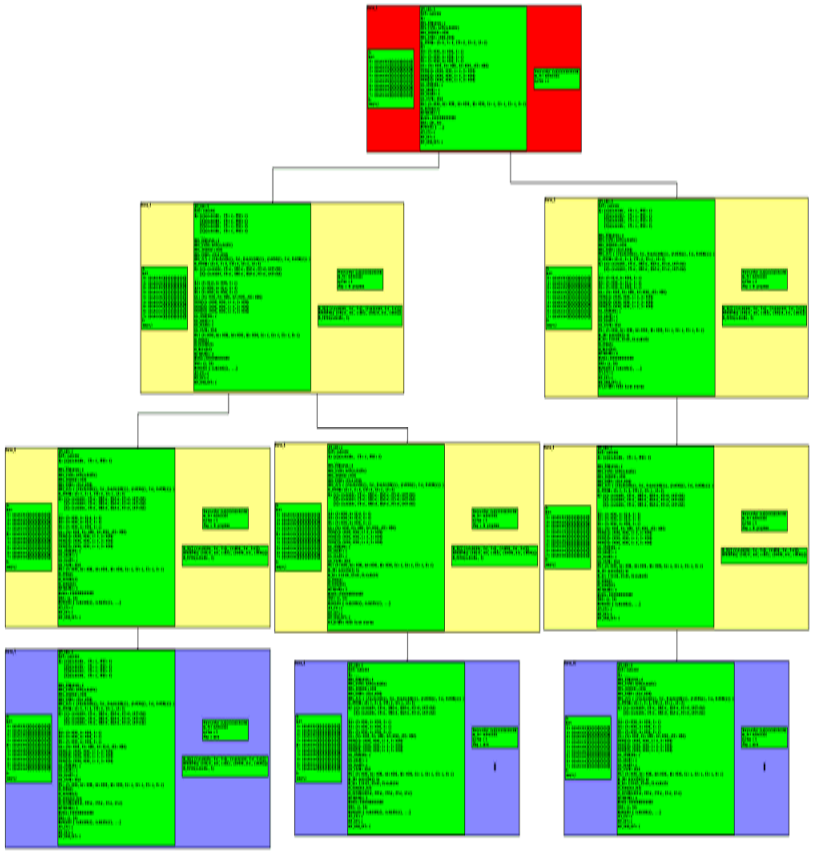




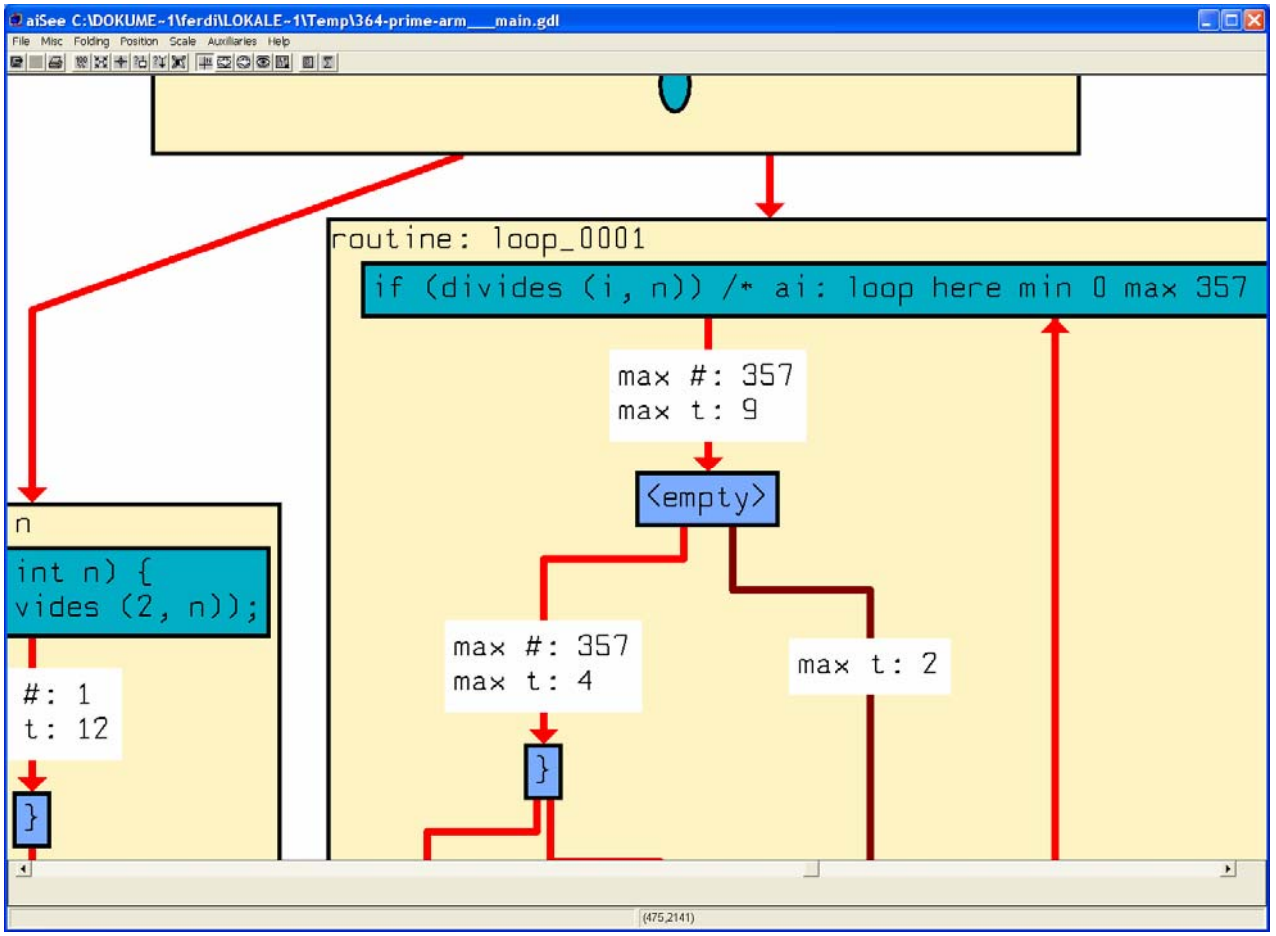
aiT WCET Analyzer

Visualization

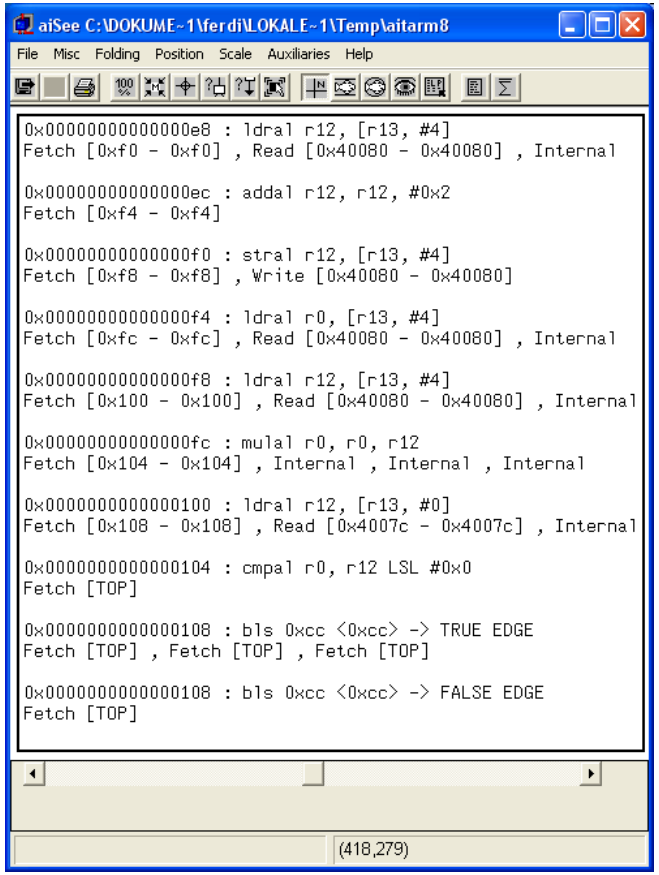


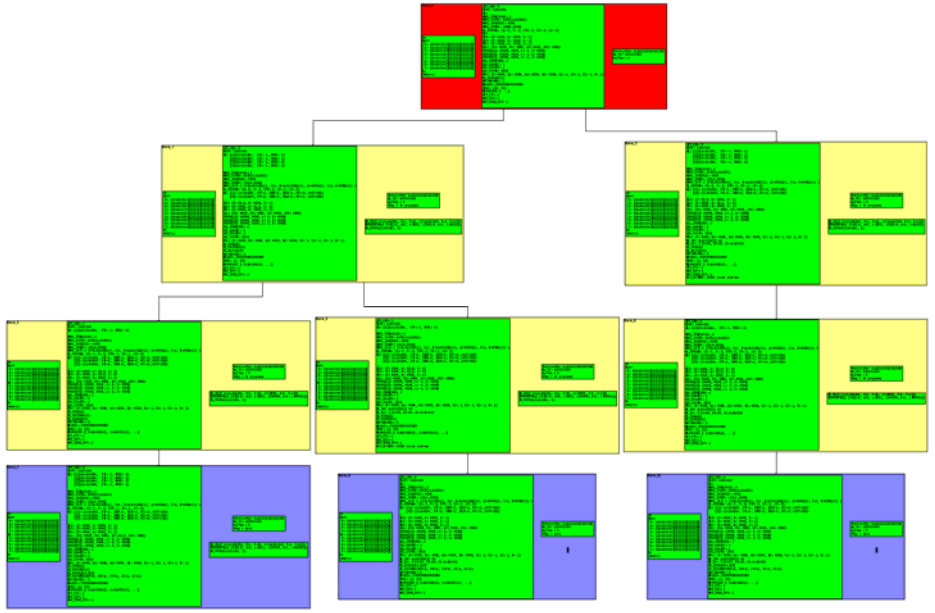


aiT: Timing Details



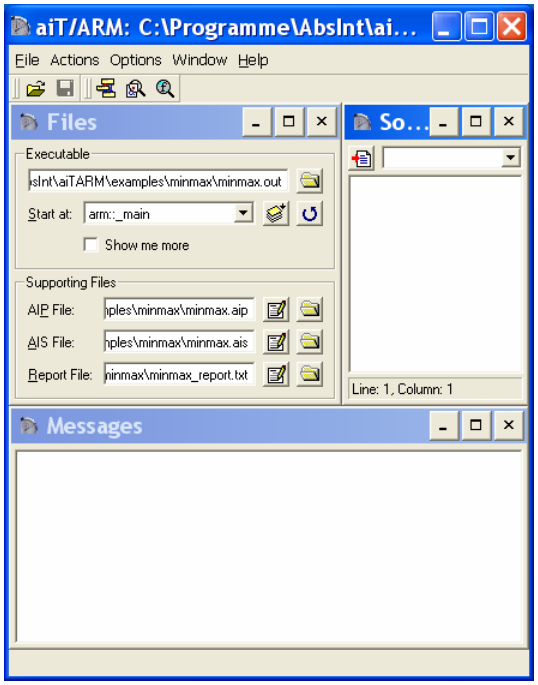
aiT: Timing Details

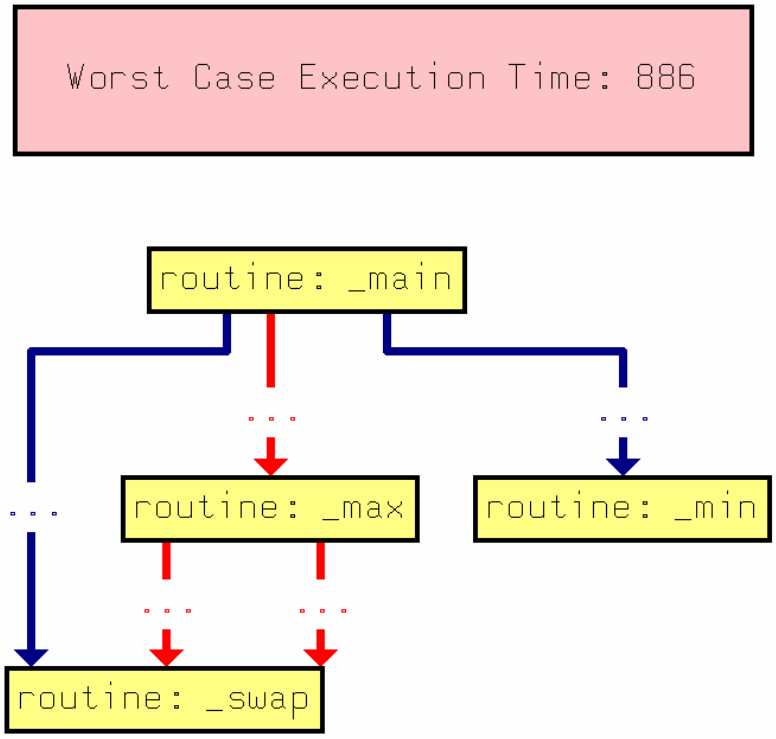




aiT WCET Analyzer

- 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 Execution Time** bounds of tasks





Example

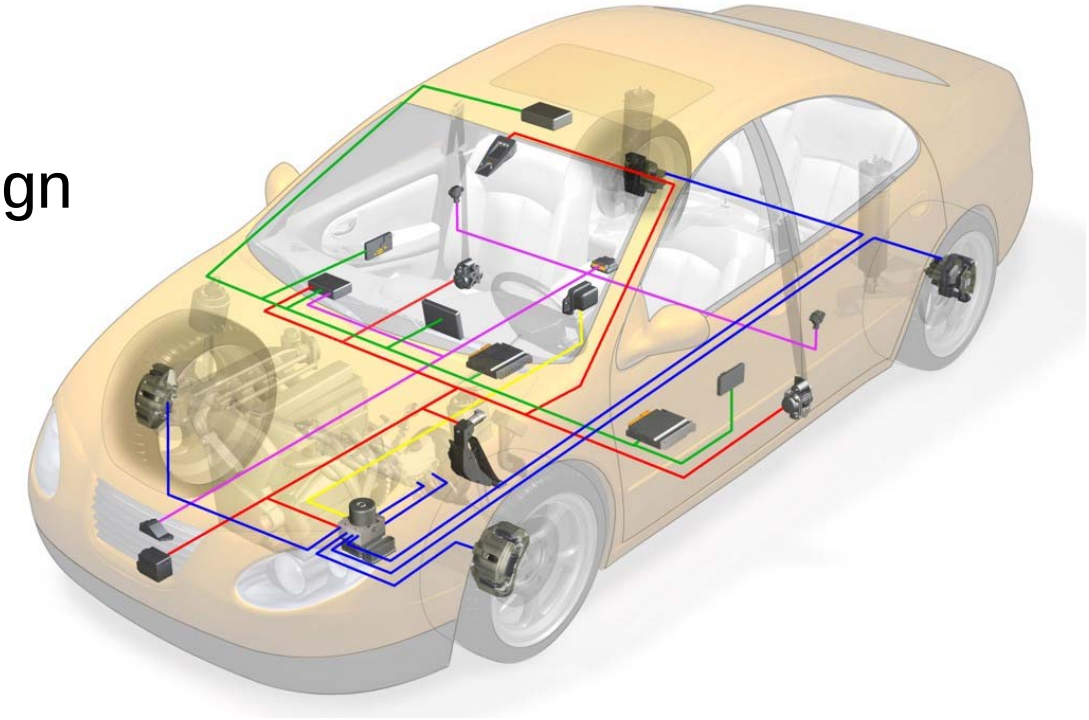
- Annotations for memcopy() of C-runtime library (handwritten assembly code)
  - `instruction "C_MEMCPY" + 1 computed branches to pc + 0x04 bytes, pc + 0x14 bytes, pc + 0x24 bytes;`
  - `instruction "C_MEMCPY" + 2 computed branches to pc + 0x10 bytes, pc + 0x20 bytes;`

Other Annotations

- Upper bounds for the recursion depths of all recursive routines.
- Flow constraints relate the execution counts of any two basic blocks. Example : **`flow 0x100 / 0x200 is max 4;`**
- Values of registers
- Memory area is read-only
- WCET of routines (excluded from WCET analysis)
- Never executed (excludes a path from WCET analysis)

aiT WCET Analyzer
The Future

- From verification to design
- From avionics to
  - automotive
  - consumer electronics
  - communication



aiT WCET Analyzer Advantages

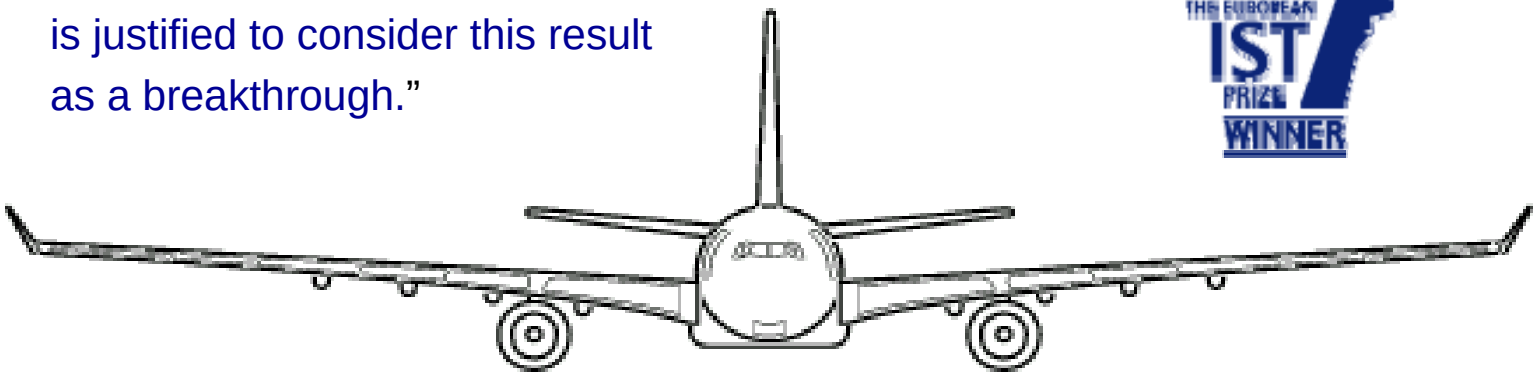
- **ai**T WCET analyzer allows you to:
  - inspect the timing behavior of (timing critical parts of) your code
- The analysis results
  - are determined without the need to change the code
  - hold for all executions (for the intrinsic cache and pipeline behavior)

aiT WCET Analyzer Advantages

- The results are **precise**
- The computation is **fast**
- **ai**T WCET analyzer is **easy to use**
- **ai**T WCET analyzer works on **optimized code**
- **ai**T WCET analyzer **saves development time** by avoiding the tedious and time consuming (instrument and) execute (or emulate) and measure cycle for a set of inputs over and over again

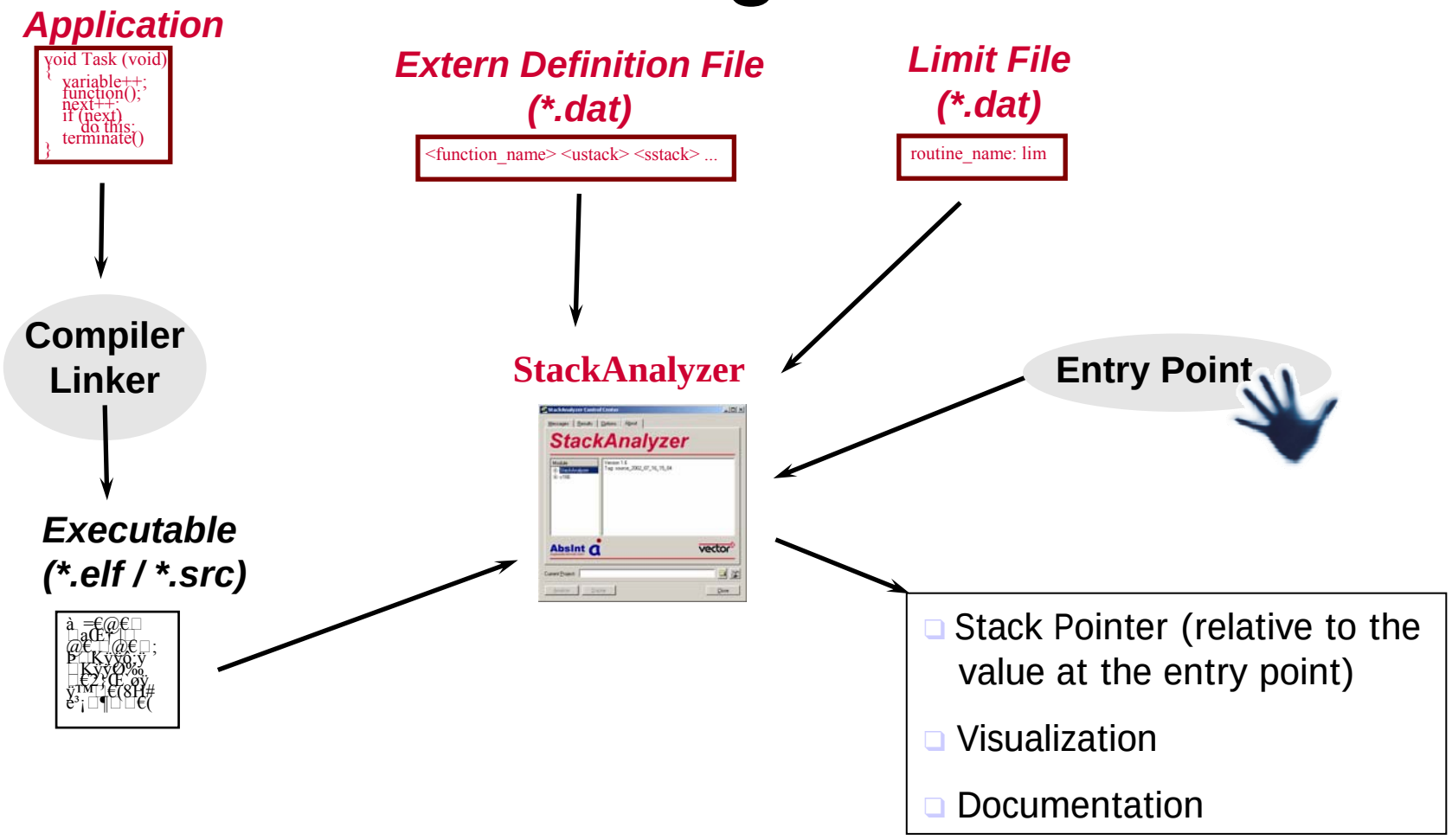
aiT WCET Analyzer
European Perspective

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."

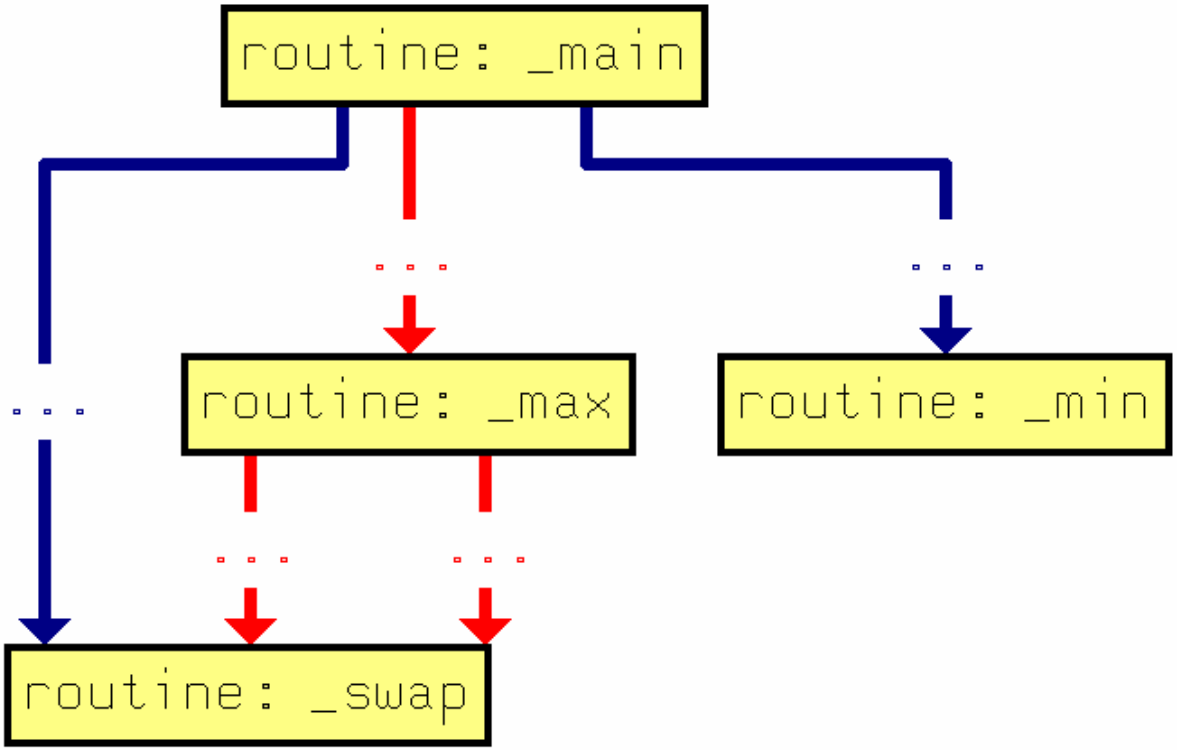


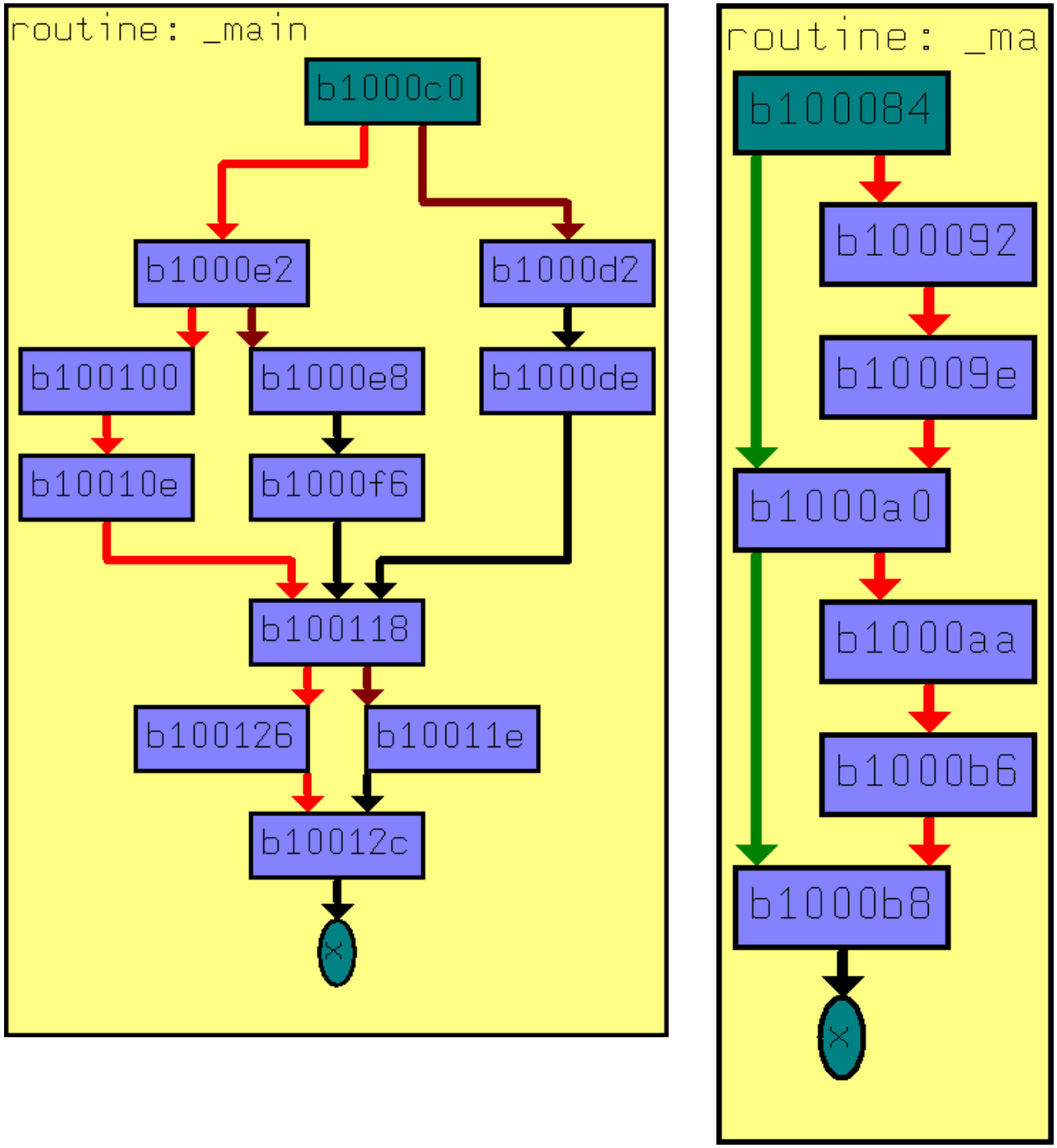
- Europe is leading in program analysis and WCET research.
- AbsInt actively participates in the definition of a European WCET research framework (NoE).

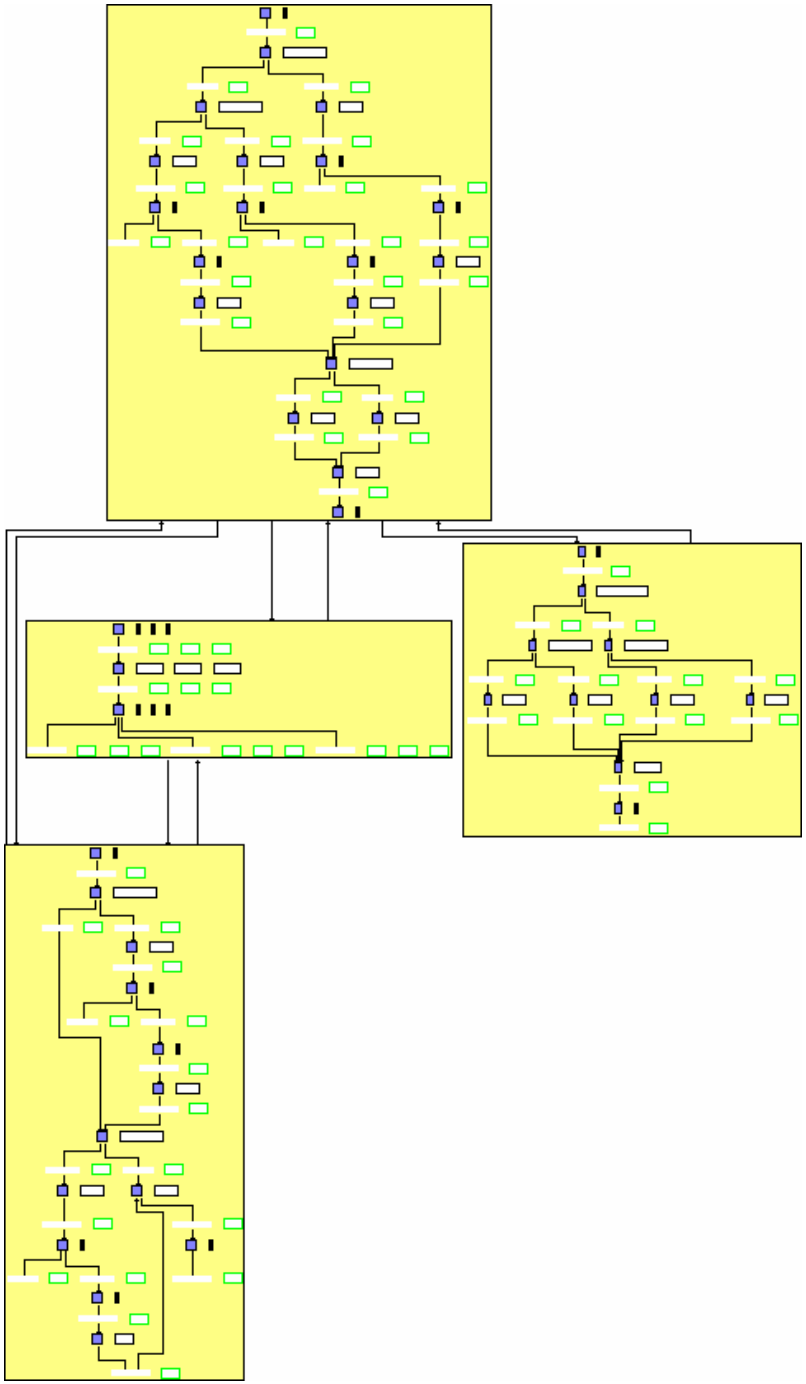
Analysis of the Stack Memory Usage

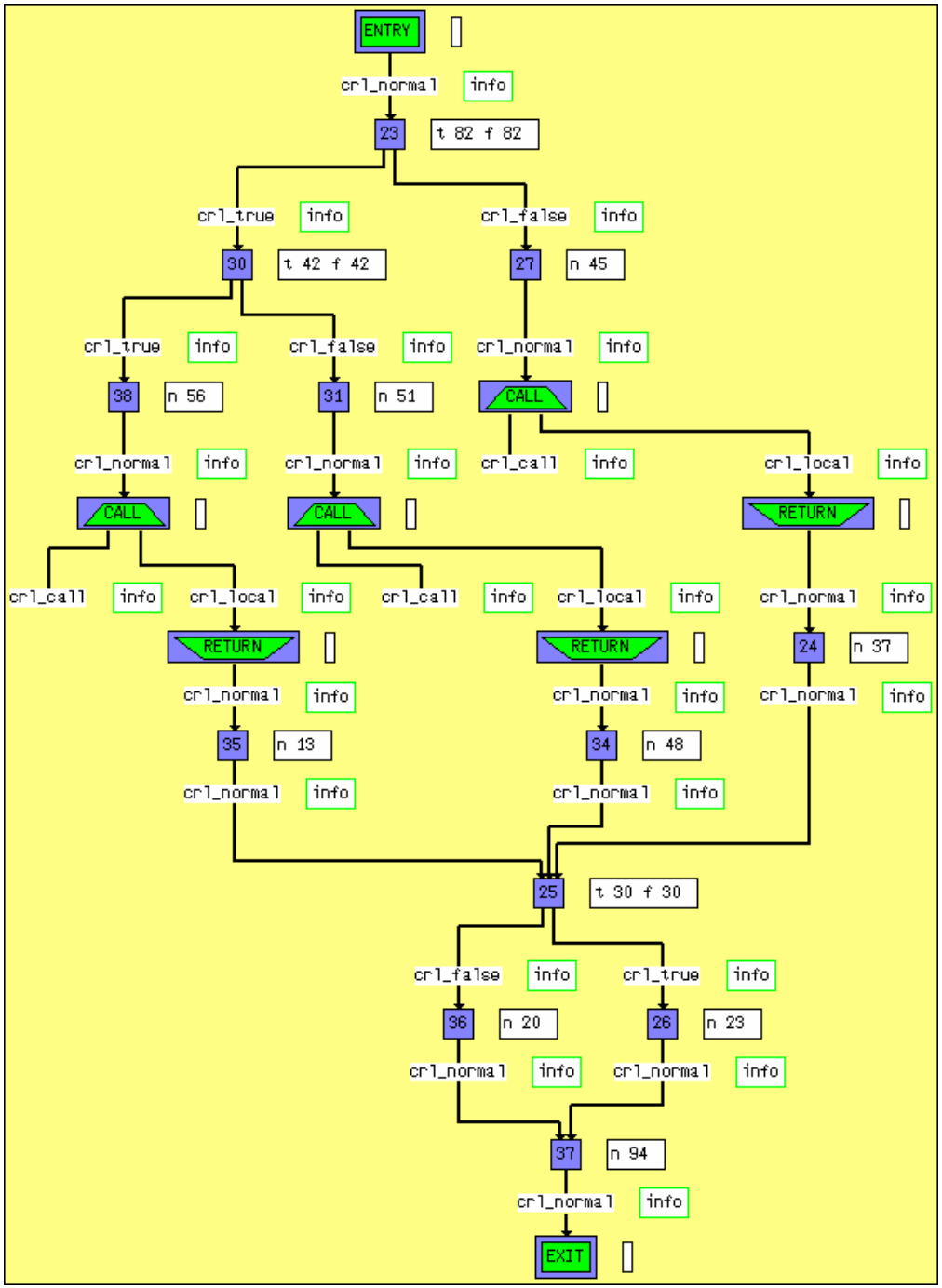


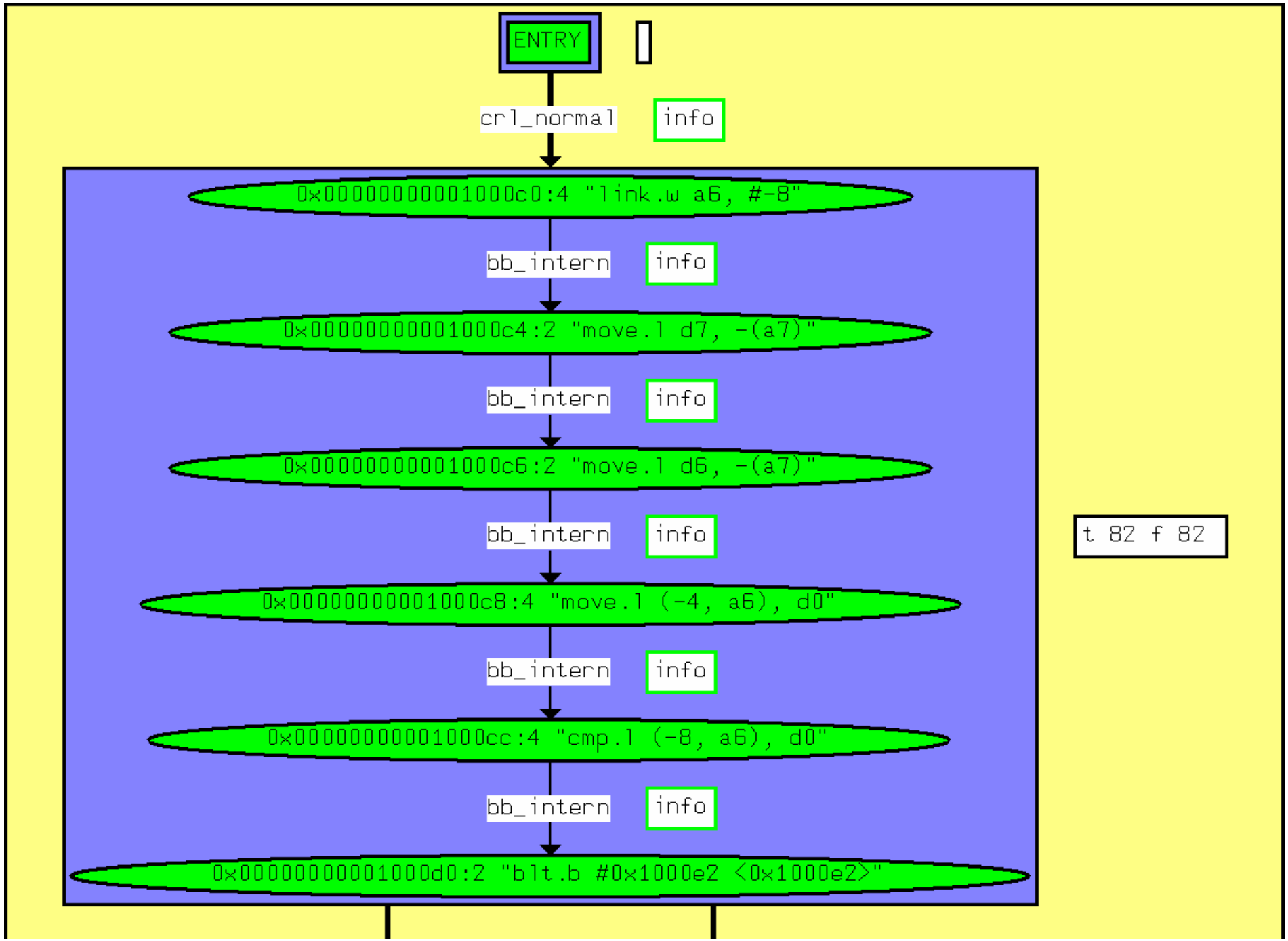
Worst Case Execution Time: 886

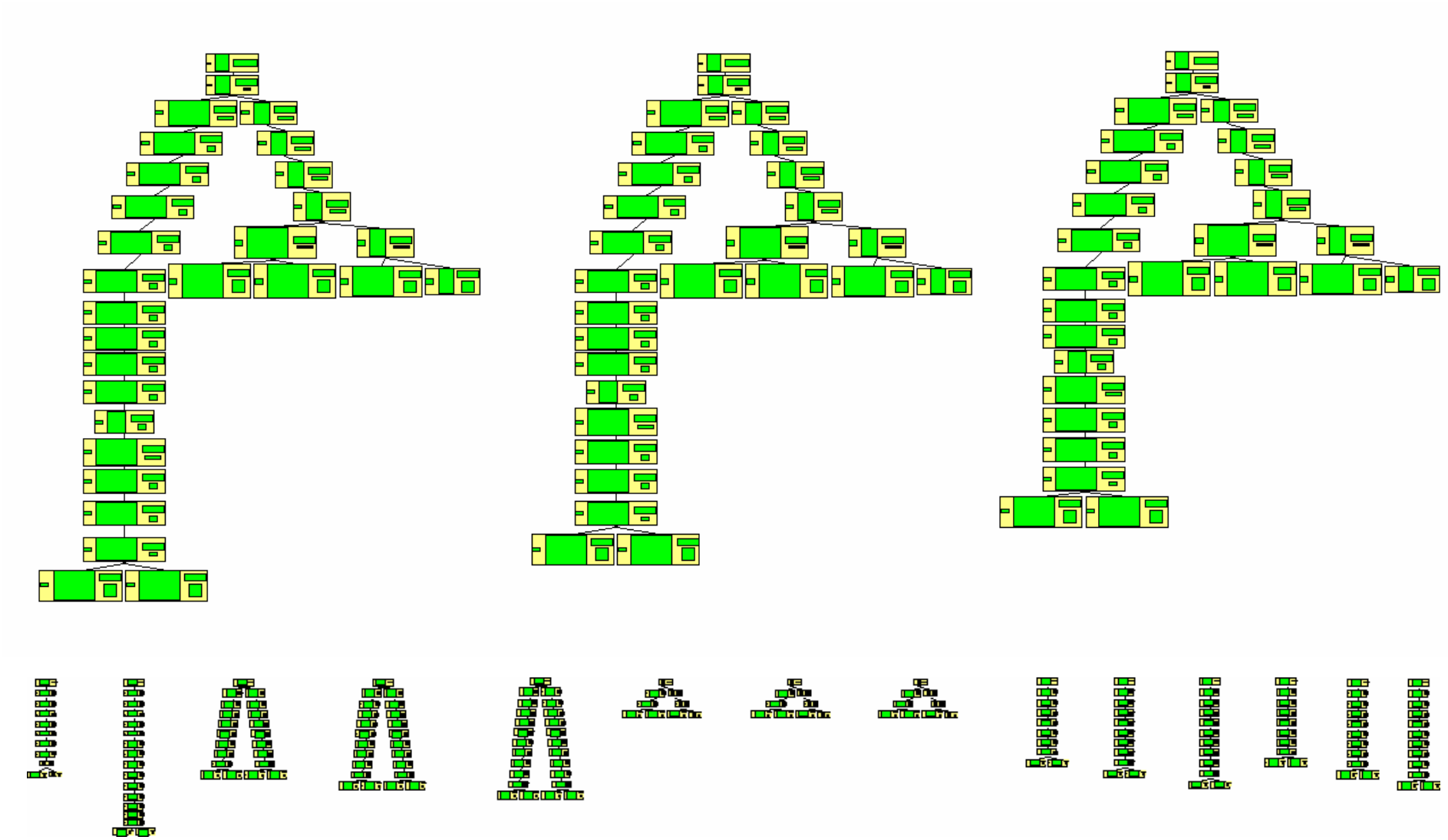



routine: _main
b1000c0
b1000e2
b1000d2
b100100
b1000e8
b1000de
b10010e
b1000f6
b100118
b100126
b10011e
b10012c
x
routine: _ma
b100084
b100092
b10009e
b1000a0
b1000aa
b1000b6
b1000b8
x

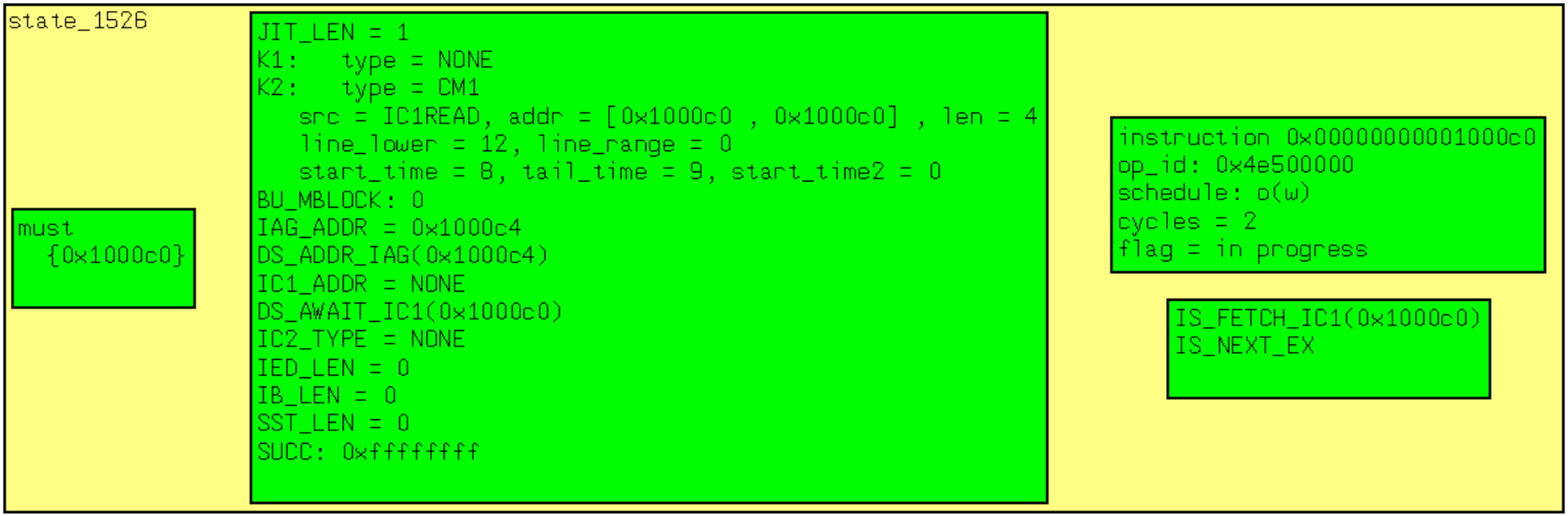



ENTRY
crl_normal
info
23
t 82 f 82
crl_true
info
crl_false
info
30
t 42 f 42
27
n 45
crl_true
info
crl_false
info
crl_normal
info
38
n 56
31
n 51
CALL
crl_normal
info
crl_normal
info
crl_call
info
crl_local
info
CALL
CALL
RETURN
crl_call
info
crl_local
info
crl_call
info
crl_local
info
crl_normal
info
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
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
<empty>
instruction 0x00000000001000c0
op_id: 0x4e500000
schedule: o(w)
cycles = 0
target: 23, 1, 0x00000000001000c4
state_1524
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
<empty>
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



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

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 %

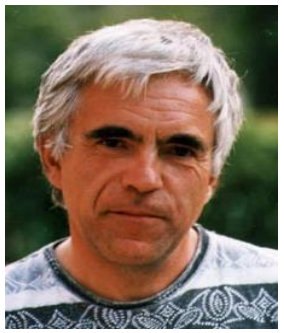
Current State and Future Work

- WCET tools available for the ColdFire 5307, the PowerPC 755, and the ARM7
- 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 AVACS!

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
- Marc Langenbach, trying to automatize

Recent Publications

- *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
- *C. Ferdinand et al.: Reliable and Precise WCET Determination of a Real-Life Processor*, EMSOFT 2001
- *H. Theiling: Extracting Safe and Precise Control Flow from Binaries*, RTCSA 2000
- *M. Langenbach et al.: Pipeline Modeling for Timing Analysis*, SAS 2002
- *St. Thesing et al.: An Abstract Interpretation-based Timing Validation of Hard Real-Time Avionics Software*, IPDS 2003
- *R. Wilhelm: AI + ILP is good for WCET, MC is not, nor ILP alone*, VMCAI 2004
- *O. Parshin et al.: Component-wise Data-cache Behavior Prediction*, ATVA 2004
- *L. Thiele, R. Wilhelm: Design for Timing Predictability, 25th Anniversary edition of the Kluwer Journal* Real-Time Systems, *Dec. 2004*