Modular WCET Analysis of ARM Processors

Andreas Engelbredt Dalsgaard
Mads Christian Olesen
Martin Toft
René Rydhof Hansen
Kim Guldstrand Larsen

The Problem

Problem

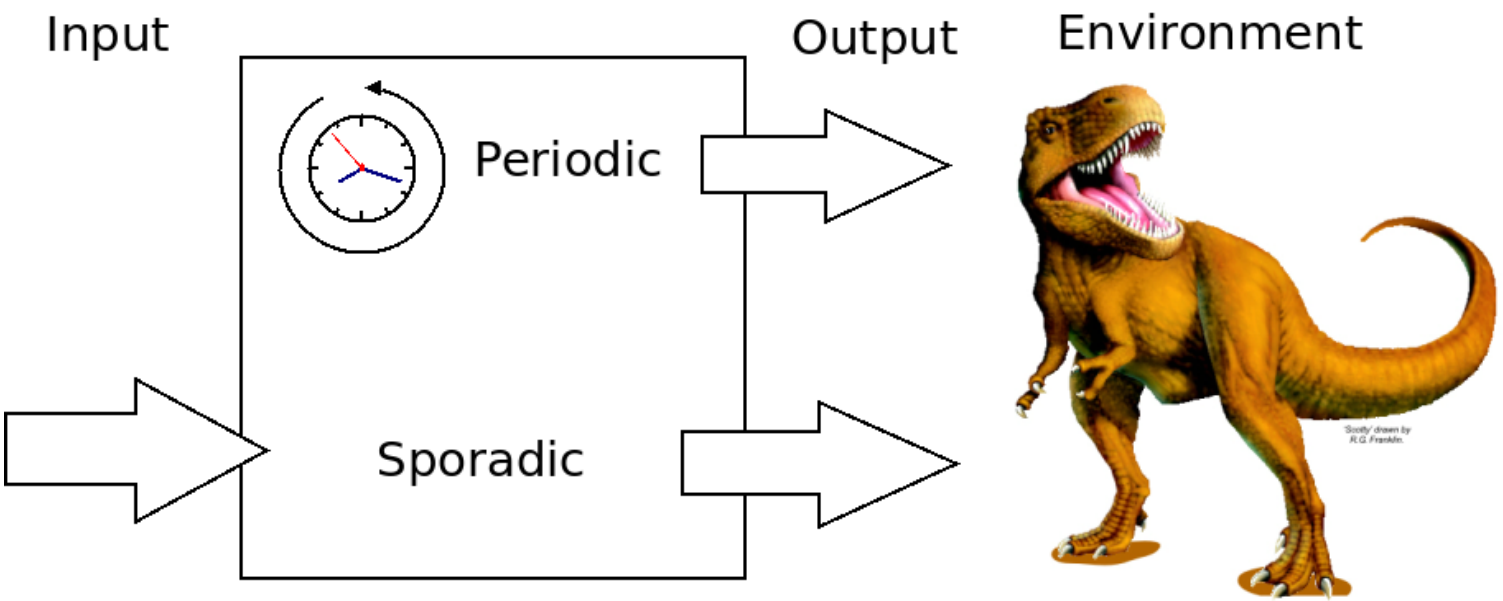
Given a program in executable form, for an ARM9 processor, determine a safe and tight worst-case execution time (WCET)

Goals:

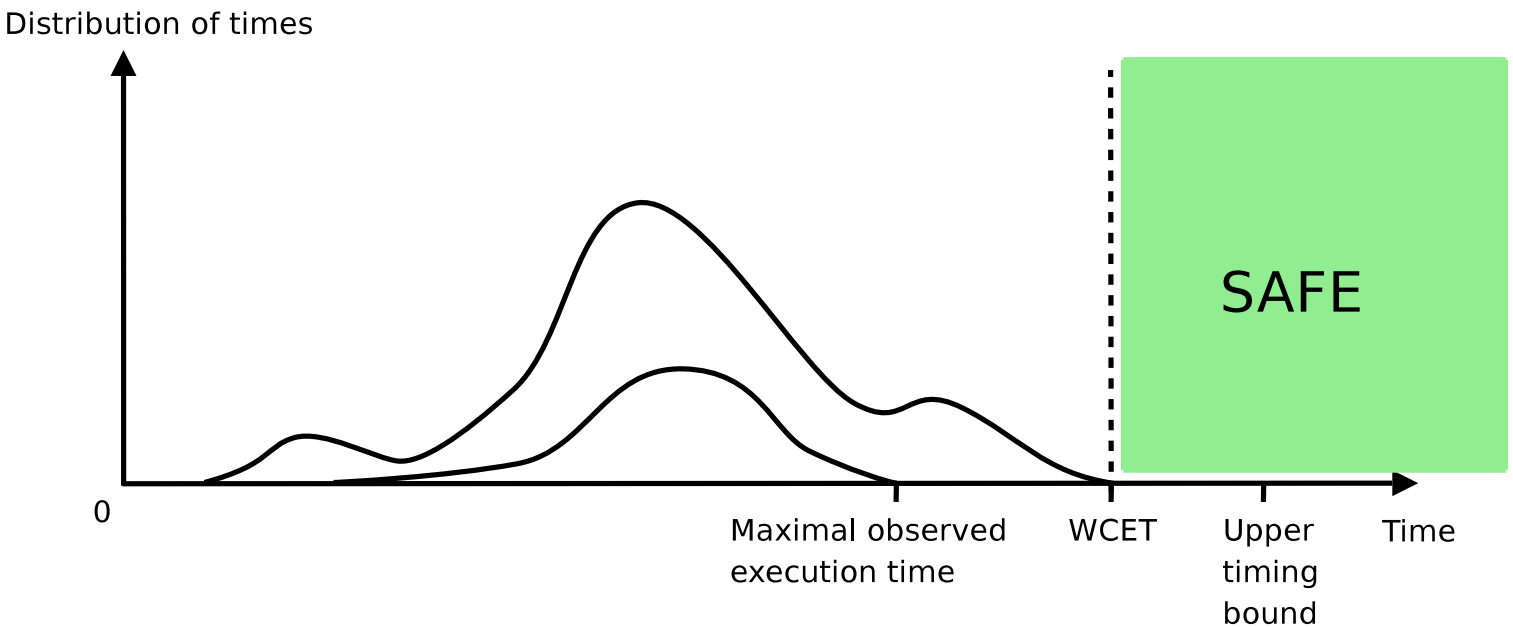
- Model the pipeline and cache(s) of the ARM9 in a precise manner
- Make the model modular, such that other ARM9 processors can easily be modelled

Real-Time Systems

- Real-time systems (RTS) are systems that need to respond to real-life events in a timely manner
- A number of processes with associated WCETs and deadlines
- Tasks are periodic or sporadic



WCET Distribution

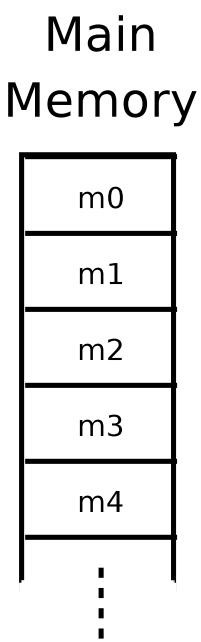


- Estimates should be on the safe side!
- However, too much on the safe side $\Rightarrow$ inefficient system

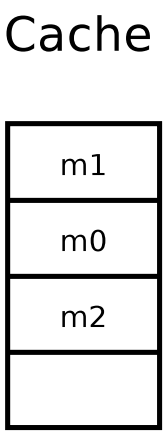
Challenge I: Modern Processors

Modern processors optimise for the **average case**, using:

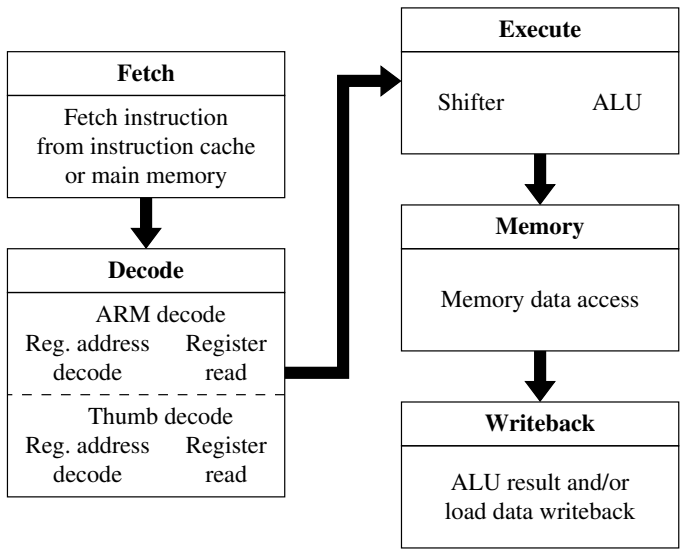
- **Caching:** allowing quick access to recently used memory items
- **Pipelining:** executing instructions in parallel



Size: ~128 Mb
Access-time: ~33 cycles



Size: ~16-32 Kb
Access-time: 1 cycle



Can we be ignorant?

No!

- Some processors have "timing anomalies", i.e. local worst-case $\not\Rightarrow$ global worst-case
- Even without "timing anomalies" assuming the local worst-case can give an over-approximation by a **factor 30**

The ARM9 processor does not exhibit "timing anomalies"

- Quicker analysis, less overapproximation
- Processors without "timing anomalies" are sufficient for most real-time systems

Challenge II: Making the Analysis Modular

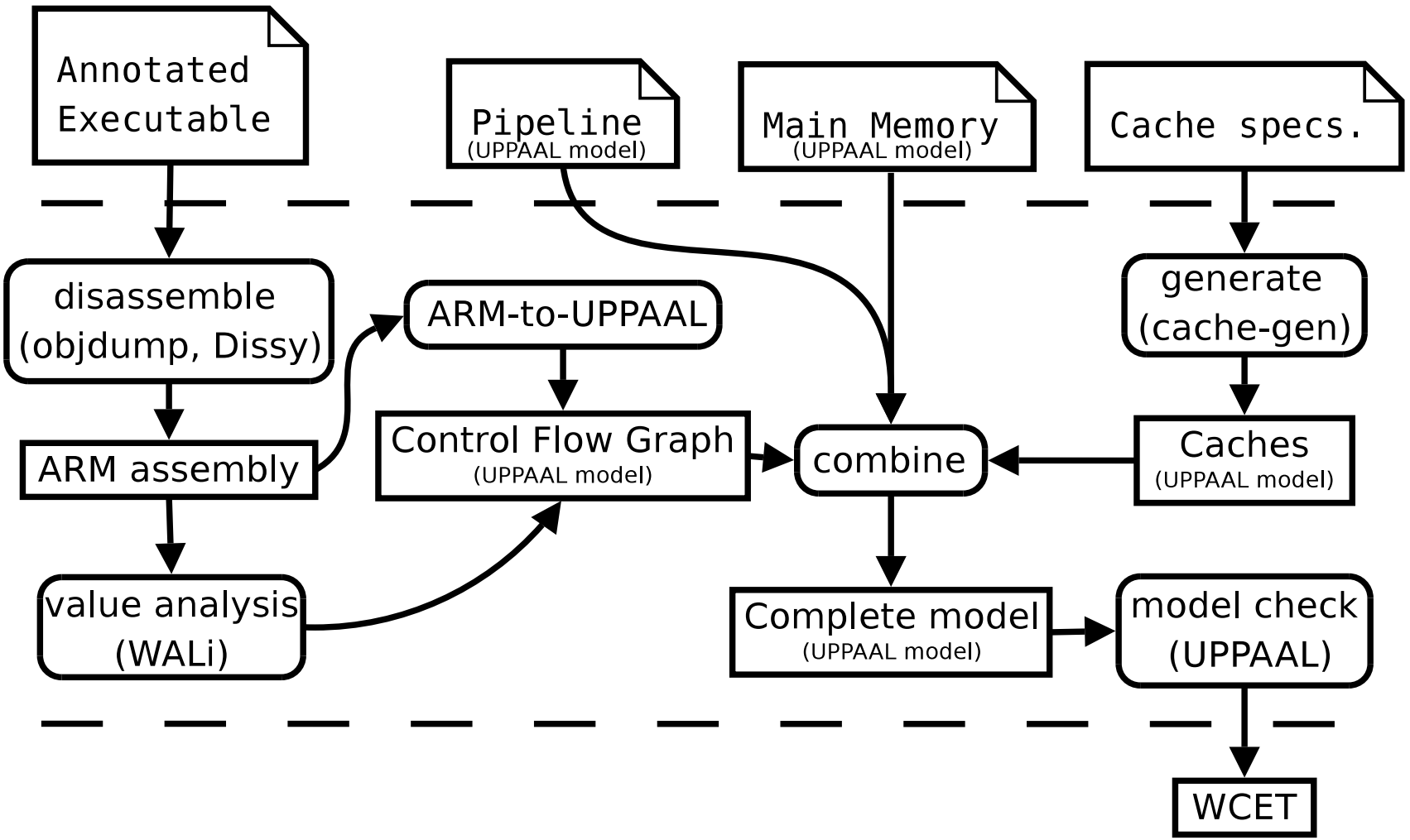
A modular analysis allows **more flexibility**, e.g. how would this program perform:

- . . . if the cache was larger?
- . . . with an extra processor core?
- . . . on an entirely different processor?

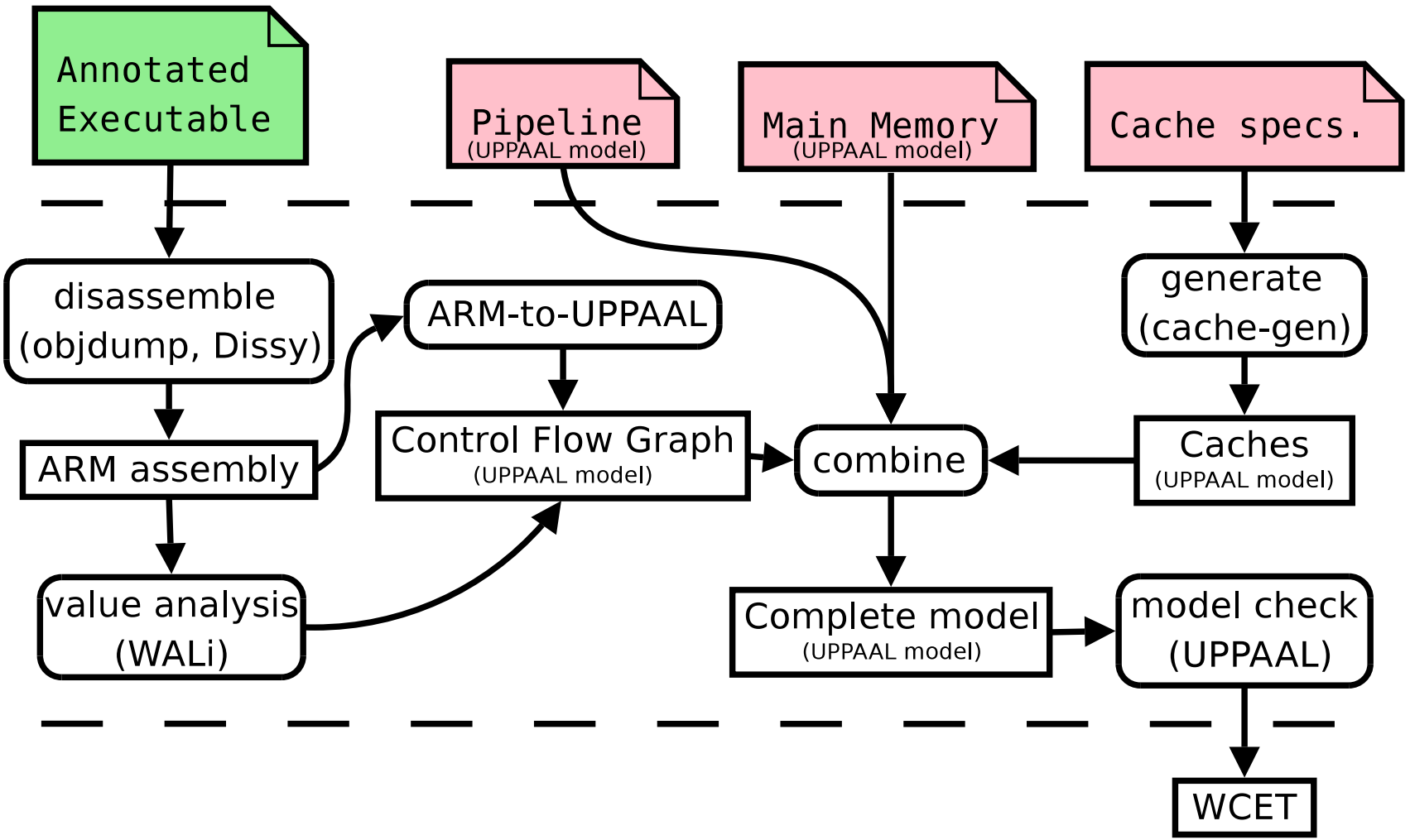
And different **accuracy/performance tradeoffs**:

- Abstract interpretation for (abstract) cache analysis
- Model checking for (concrete) cache analysis
- Use simple always-miss cache, if no need to do more precise analysis

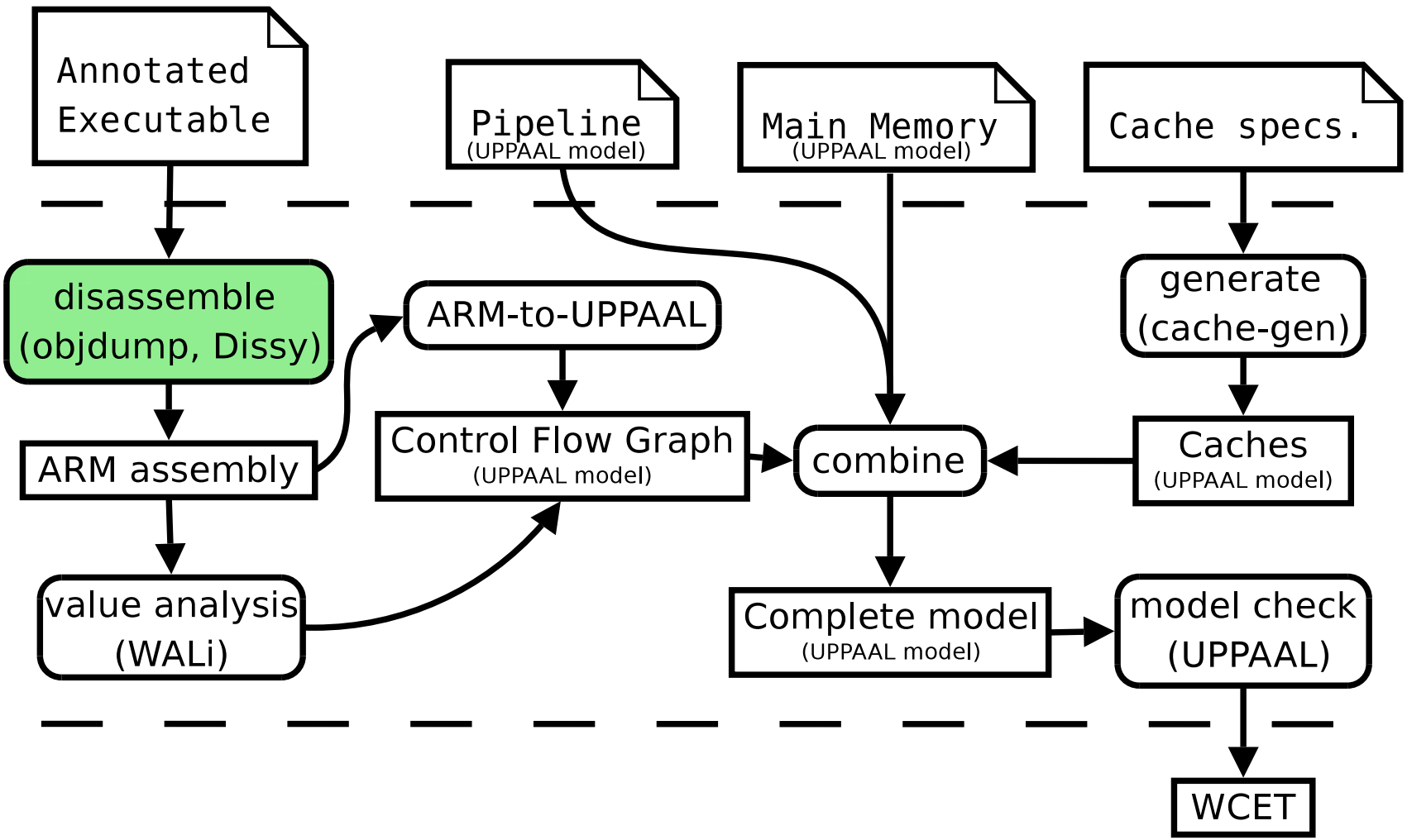
Tool-Chain Overview



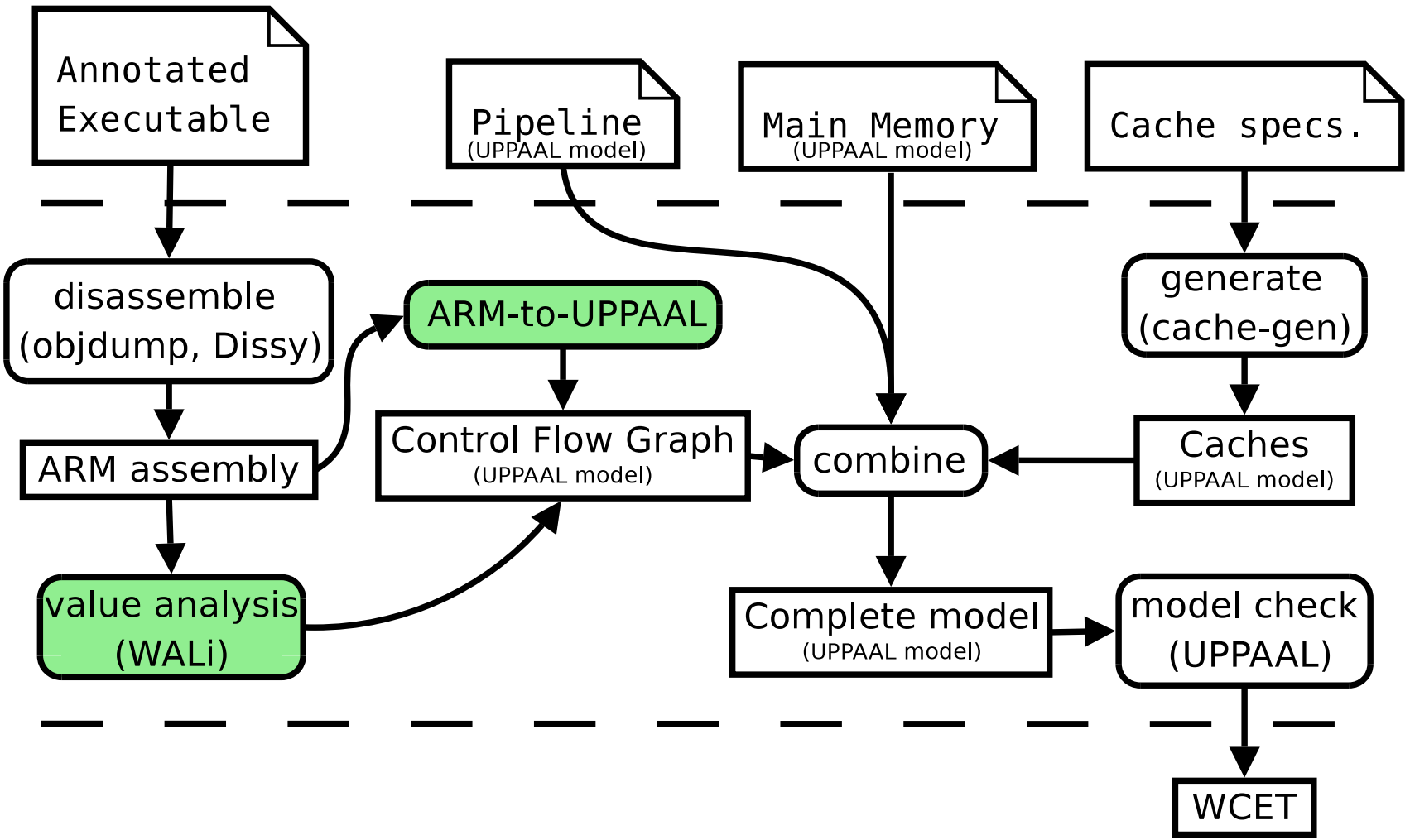
Tool-Chain Overview



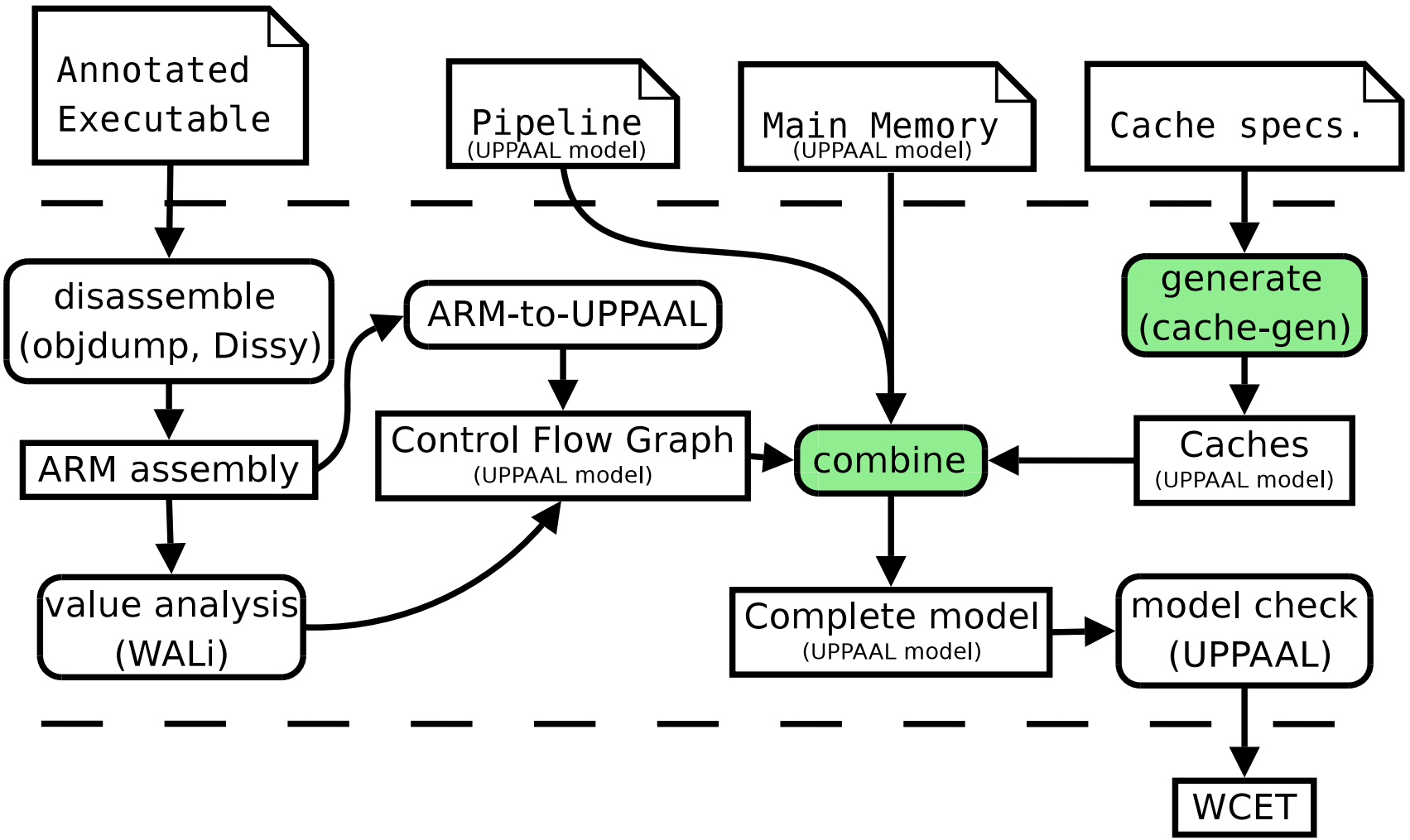
Tool-Chain Overview



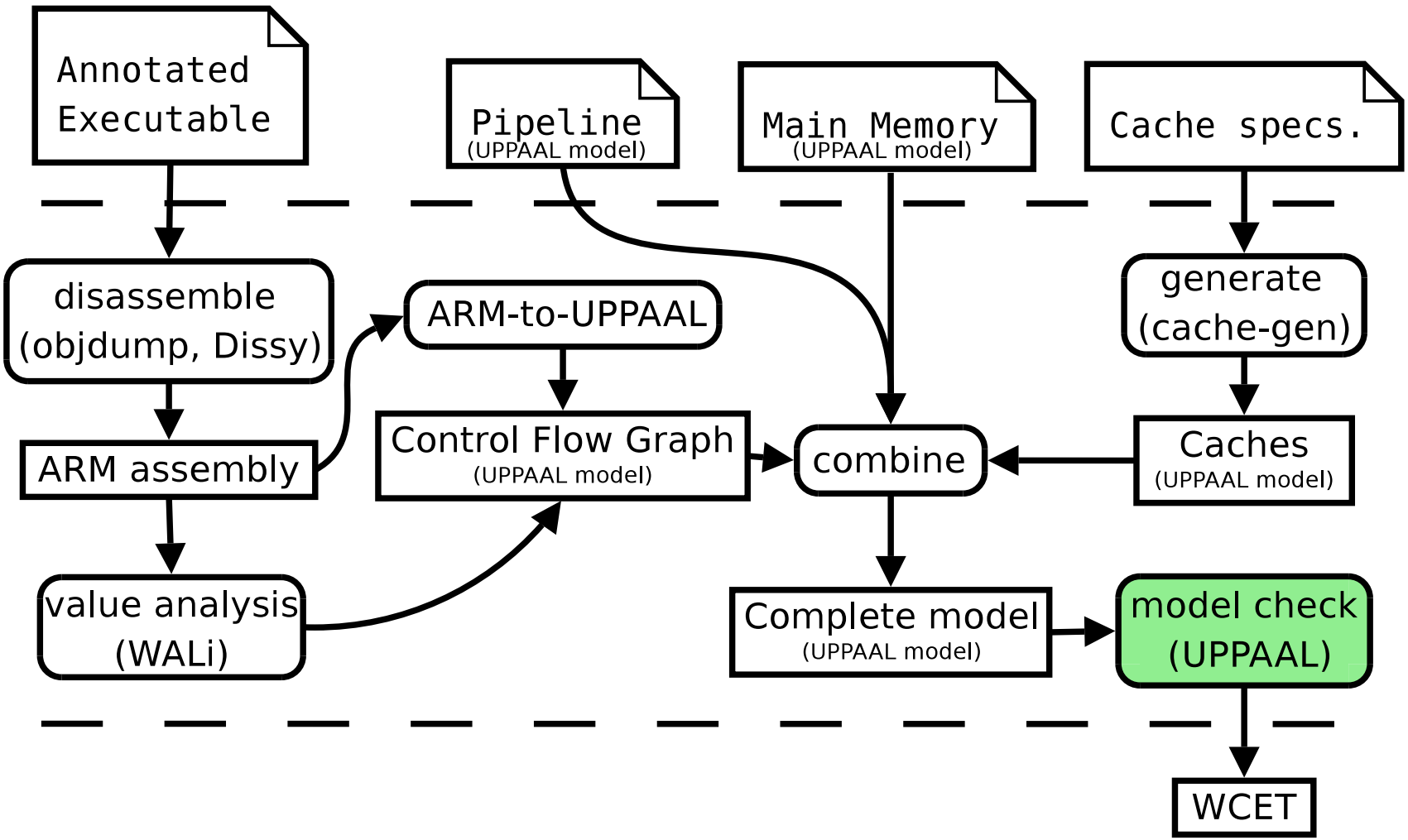
Tool-Chain Overview



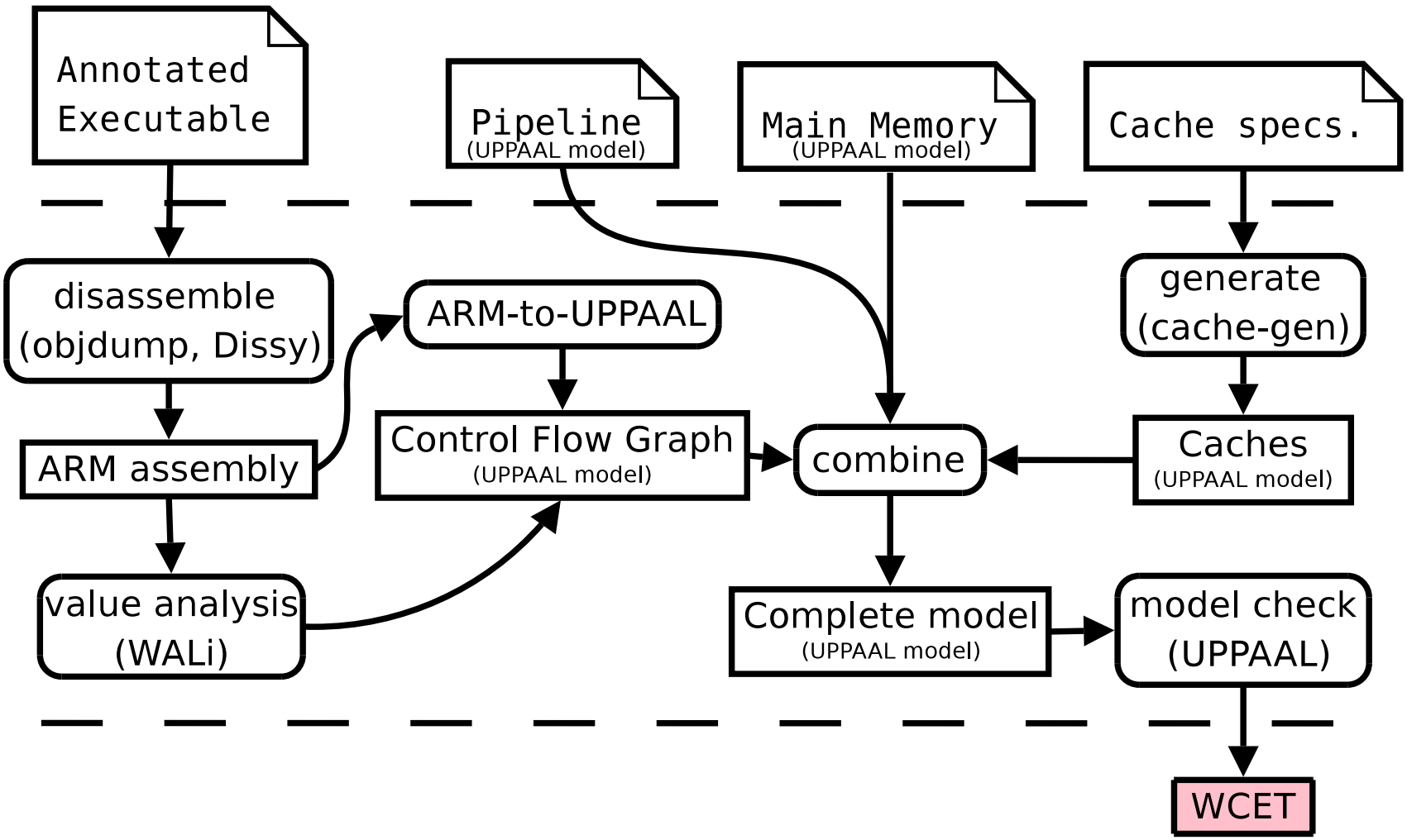
Tool-Chain Overview



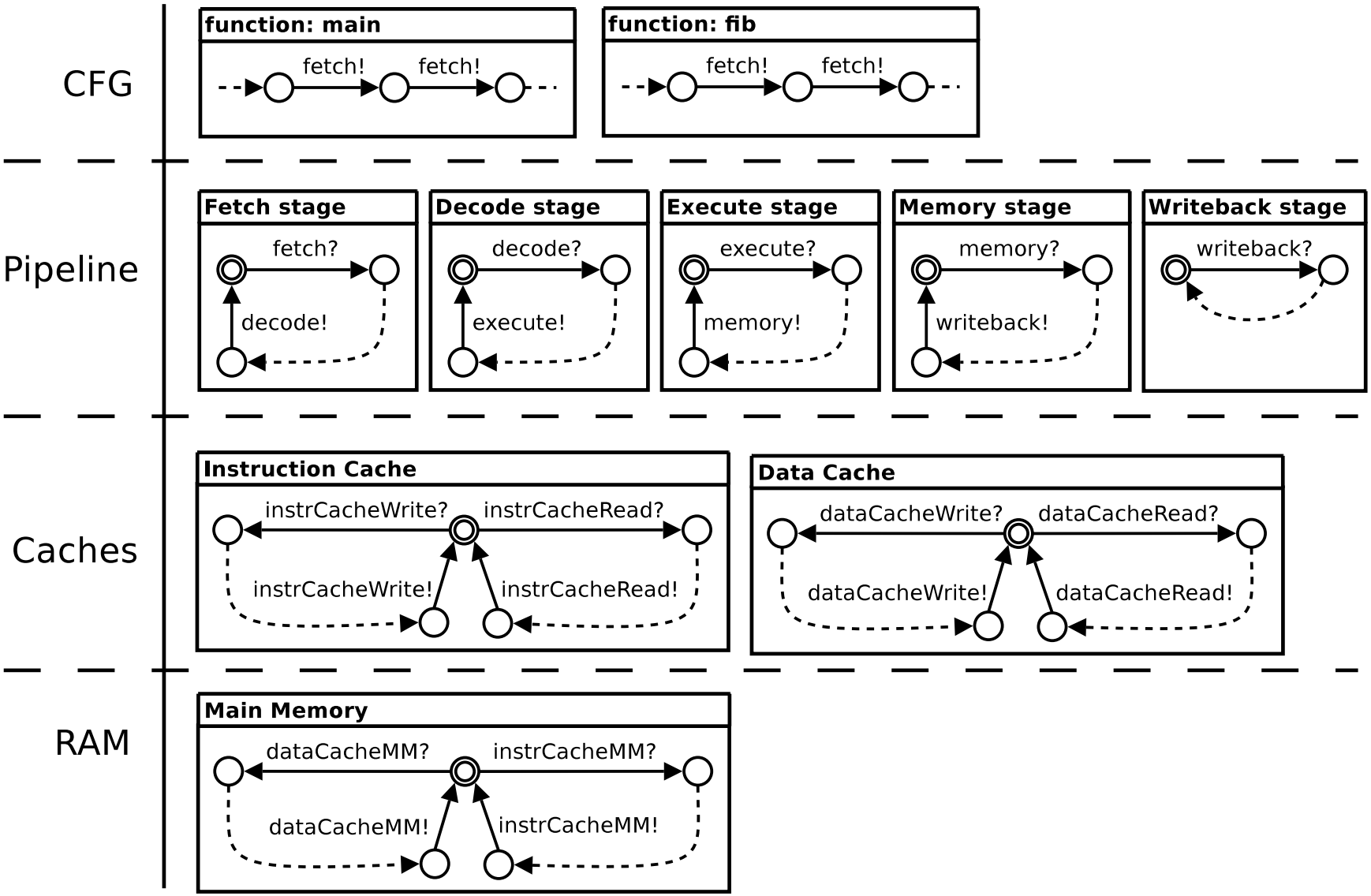
Tool-Chain Overview



Tool-Chain Overview



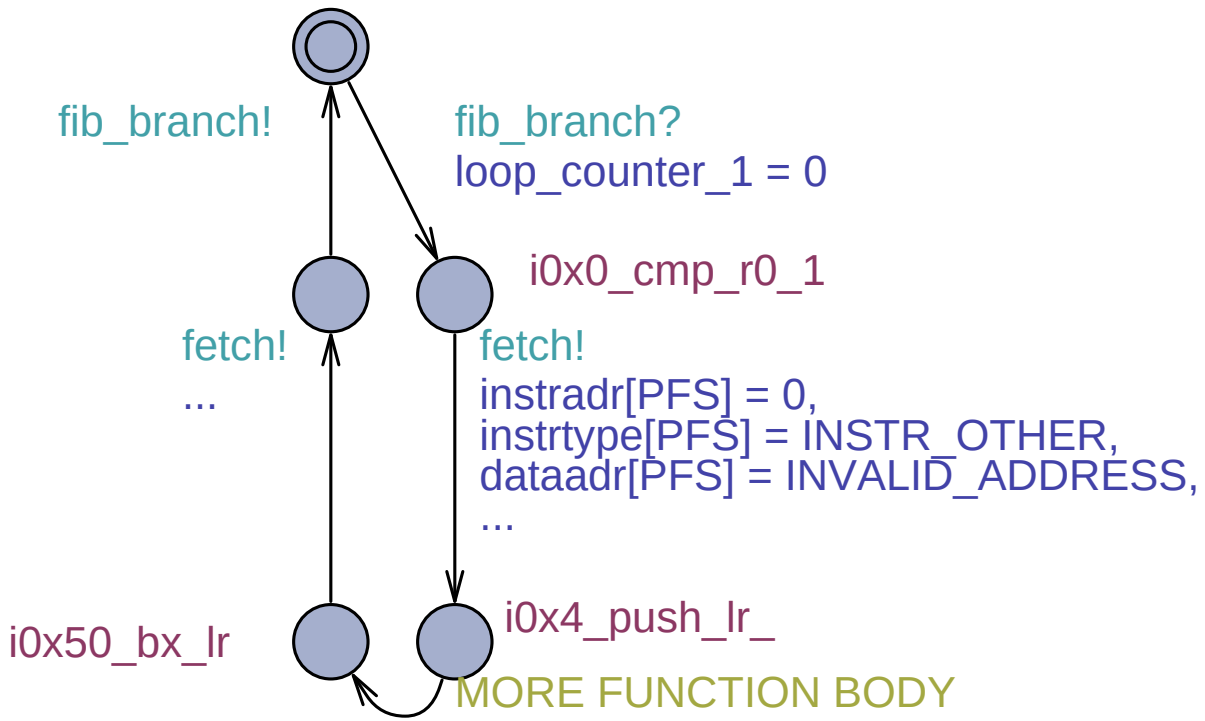
Overview of Our Model



Path Analysis

- Timed automaton for every function
- Transitions emulate instruction execution

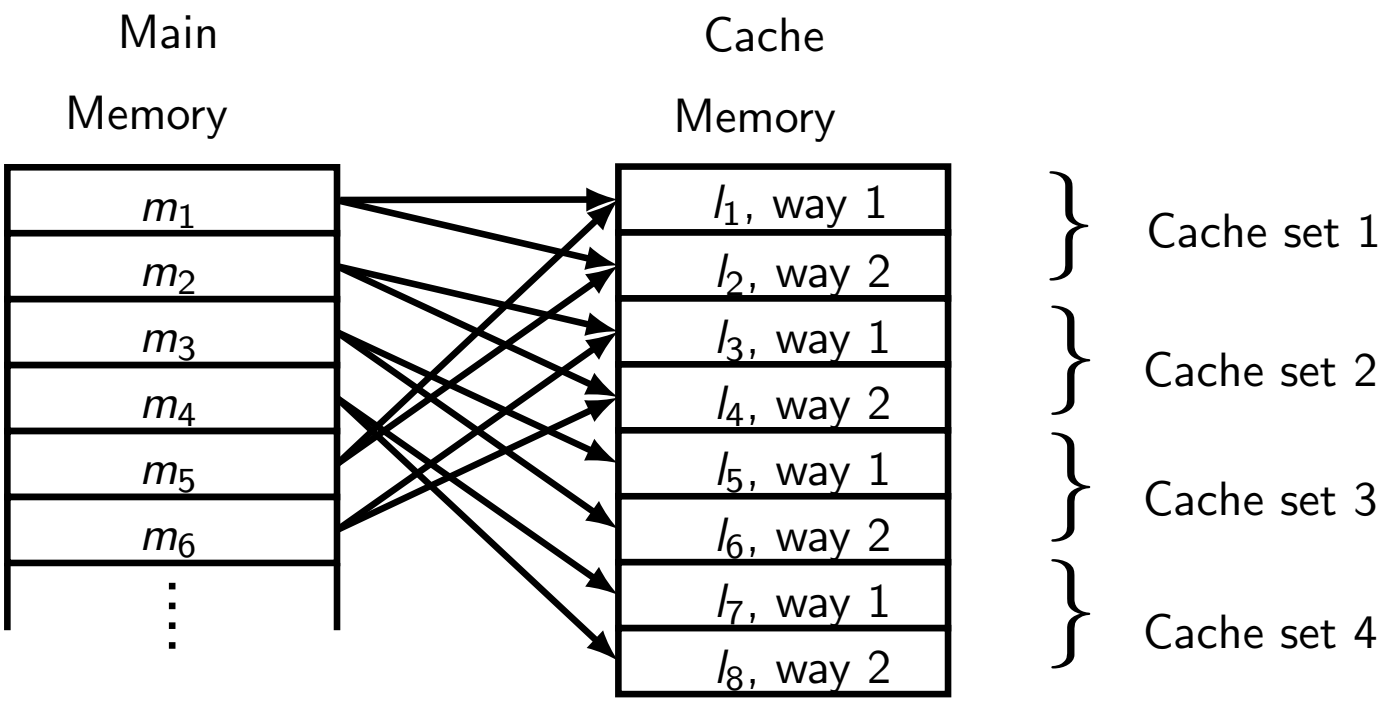
```
0x00   cmp r0, 1
0x04   push lr
...    ...
0x50   bx lr
```



- Functions handled flow-sensitively

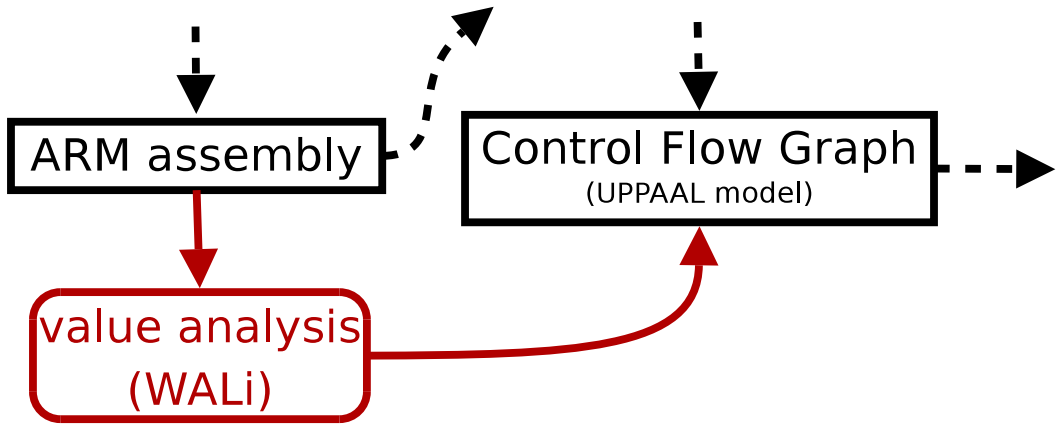
Cache Analysis

- ARM9: Separate data and instruction caches
  - 16 kB in size, 64-way associative, 8 words (32 byte) per line
  - Write-through and write-back policies
  - Pseudo-random and round-robin replacement policies
- Modelled concretely as timed automata in UPPAAL



Value Analysis

- The cache analysis needs concrete memory addresses
- Registers are used as base and offset in all memory accesses



- Value analysis:
  Find an over-approximation of possible register values at all execution points of a process

- Weighted push-down systems (WPDSs) used for inter-procedural, control-flow sensitive value analysis
- Presented by Reps et al. in **Program Analysis using Weighted Push-Down Systems**

Weighted Push-Down Systems

Use the PDS-stack as call-stack:

- Sequential: $\langle p, n_{main} \rangle \hookrightarrow \langle p, n_2 \rangle$
- Function call: $\langle p, n_4 \rangle \hookrightarrow \langle p, n_8 n_5 \rangle$
- Function return: $\langle p, n_{12} \rangle \hookrightarrow \langle p, \epsilon \rangle$

Each rule has an associated weight, describing the effect of the transition. Weights can be:

- Combined ("join"): $w_1 \oplus w_2 = w_3$
- Extended (sequential progression): $w_1 \otimes w_2 = w_3$

The effect of executing a program to a set of configurations (T) ("Meet over all paths"):
$\bigoplus \{ w_1 \otimes \ldots \otimes w_n | w_1, \ldots, w_n$ is the weights associated with a path of rules leading to a configuration in T $\}$

Our Value Analysis

Implemented simple value analysis, using:

- Loop unrolling
- Simple (syntactical) register-value tracking
- No tracking of values in memory
- Finds good amount of values for some programs, but could be much better

Our Value Analysis

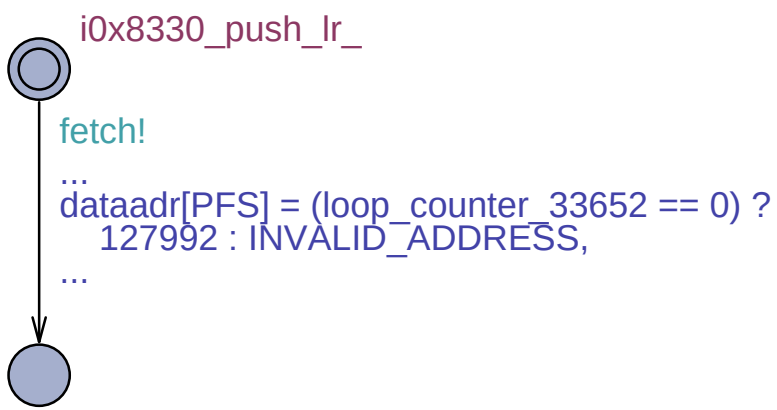
- Weights = functions representing the effect of an instruction or a sequence of instructions, e.g.:

$$\mathbf{w_1}\begin{pmatrix} r_0 \\ r_1 \end{pmatrix} = \begin{pmatrix} \text{“}r_1 + 2\text{”} \\ \mathbf{id} \end{pmatrix}, \quad \mathbf{w_2}\begin{pmatrix} r_0 \\ r_1 \end{pmatrix} = \begin{pmatrix} \mathbf{id} \\ \text{“}r_0 * 2 + r_1 << 3\text{”} \end{pmatrix}$$

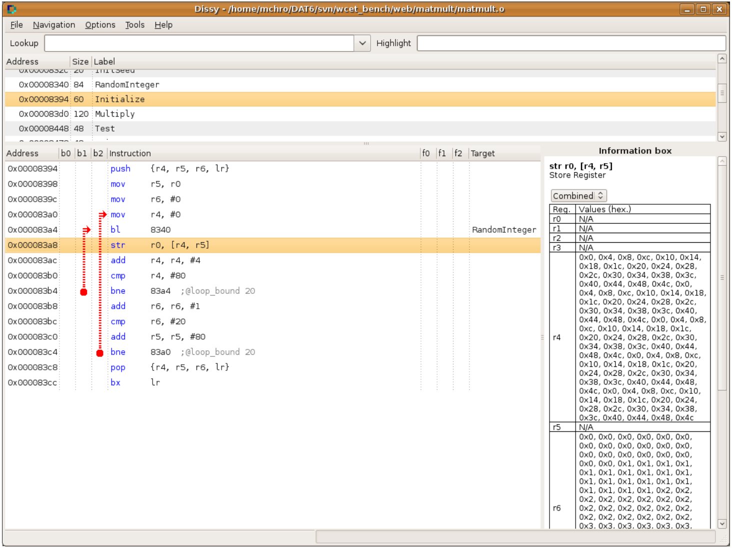
- Special values: **id**, $\bot$ and $\top$
- Combine and extend handled syntactically (string equality, and string replacement)

Implementation = WALi + Python

- The open source Weighted Automata Library (WALi) implements a number of WPDS algorithms
- Easy to extend with e.g. new weight domains
- Our weights are, very conveniently, valid Python expressions
- Process automata are annotated with the results



Disassembler — Dissy



WCET Guarantee in Three Easy Steps

Demo

Experiments

- Evaluated on the Mälardalen WCET benchmarks

Experiments

- Evaluated on the Mälardalen WCET benchmarks
- The most interesting findings:
  - Taking the **instruction cache** into account yields WCETs that are up to 97% sharper (78% on average at -O2)
  - Taking the **data cache** into account yields WCETs that are up to 68% sharper (31% on average at -O2)
  - Almost all results are obtained within five minutes

Experiments

- Evaluated on the Mälardalen WCET benchmarks
- The most interesting findings:
  - Taking the **instruction cache** into account yields WCETs that are up to 97% sharper (78% on average at -O2)
  - Taking the **data cache** into account yields WCETs that are up to 68% sharper (31% on average at -O2)
  - Almost all results are obtained within five minutes
- Some programs fail due to
  - State space explosion (6)
  - Write to program counter (2)
  - Floating point operations [1]
  - Value analysis problems

[1] need to manually find good loop-bounds for **very** optimised assembler

Experiments

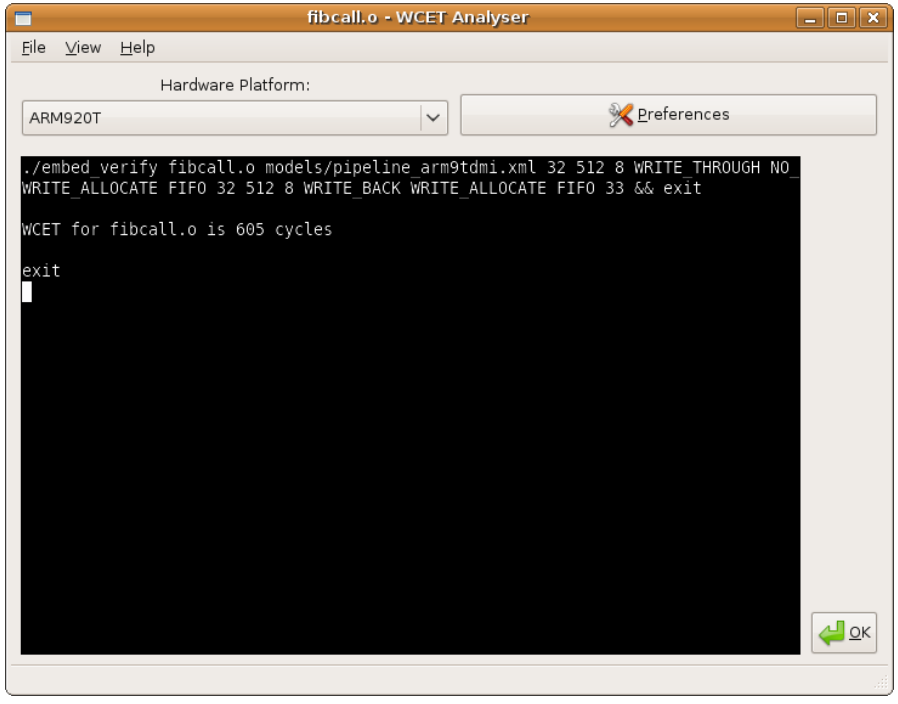
- Evaluated on the Mälardalen WCET benchmarks
- The most interesting findings:
  - Taking the **instruction cache** into account yields WCETs that are up to 97% sharper (78% on average at -O2)
  - Taking the **data cache** into account yields WCETs that are up to 68% sharper (31% on average at -O2)
  - Almost all results are obtained within five minutes
- Some programs fail due to
  - State space explosion (6)
  - Write to program counter (2)
  - Floating point operations [1]
  - Value analysis problems
- **We are able to analyse 17 out of the 25 non-floating point benchmarks!**

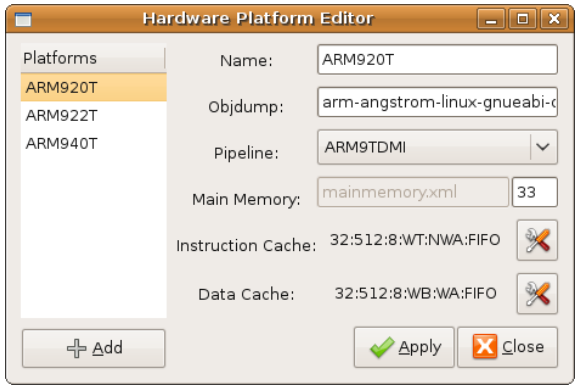
[1] need to manually find good loop-bounds for **very** optimised assembler

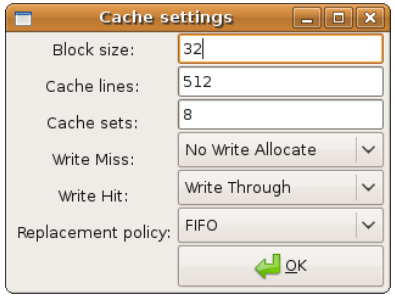
Conclusion - Future Work

- Prototype implementation successful
- UPPAAL provides a good framework for modularising the models
- The analysis times seem acceptable

- Better path analysis
- Precise value analysis essential for tight bounds (work in progress)
  - Modelling the stack
  - Modelling memory regions
- Support other typical embedded processors:
  - ARM7 (3-stage pipeline, cache)
  - Atmel AVR 8bit (3-stage pipeline, no cache, 1-2 cycle instructions)
  - H8/300 (old Lego Mindstorms)
- Modelling the cache abstractly







- Our master's thesis, the accompanying source code, and these slides are available at

  `http://metamoc.martintoft.dk`

Thank you for your attention!

- Implementation = WALi + Python
- Disassembler — Dissy

5 Demo
- WCET Guarantee in Three Easy Steps

6 Conclusion
- Experiments
- Conclusion - Future Work