Formal models of timed systems: WCET analysis in single-core systems, and some ideas for multi-core systems

Jean-Luc Béchennec Sébastien Faucou

CAPITAL Workshop - 4th of June 2021

Université de Nantes, CNRS, LS2N
F-44000 Nantes, France

About this talk

This talk is about a work carried out in our group since a few years concerning the use of real-time model-checking to estimate the WCET of programs

- The work was initiated by Franck Cassez (now with ConsenSys Software R&D)
- Quickly joined by Jean-Luc Béchennec
- And a bit later by Mikäel Briday, Sébastien Faucou and Armel Mangean

About this talk

Talk is split in 2 parts

- Review of past work concerning single-core systems
  → Sébastien Faucou
- Demo of on-going works concerning multi-core systems
  → Jean-Luc Béchennec

Real-time model-checking for WCET analysis: motivations and overview

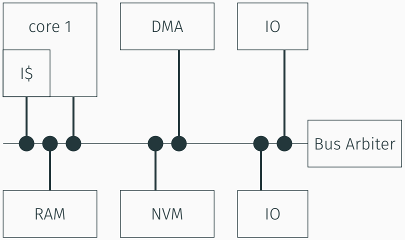
The WCET problem

Given a system $\mathcal{S}$ composed of:

A program $\mathcal{P}$

```
00003000 <_start>:
    3000:  li    r1,1         ;r1  <- 1
    3004:  ori   r1,r1,49296 ;r1  <- r1 | 49296
    3008:  bl    3010         ;call main
0000300c <loop>:
    300c:  b     300c         ;branch
00003010 <main>:
    3010:  li    r8,29        ;r8  <- 29
    3014:  li    r10,1        ;r10 <- 1
    3018:  mtctr r8           ;ctr <- r8
    301c:  li    r9,1         ;r9  <- 1
    3020:  b     3028         ;branch
    3024:  mr    r9,r3        ;r9  <- r3
    3028:  add   r3,r9,r10    ;r3  <- r9+r10
    302c:  mr    r10,r9       ;r10 <- r9
    3030:  bdnz  3024         ;ctr--,
                              ;branch if ctr!=0
    3034:  blr                ;return
```

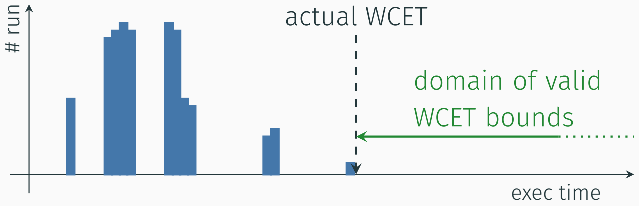
A micro-architecture $\mathcal{A}$



Find an upper-bound on the execution time of $\mathcal{P}$ on $\mathcal{A}$

The WCET problem (cont'd)

The WCET bound does not necessarily corresponds to a run of $\mathcal{S}$: any value greater than or equal to the actual WCET is valid



The WCET problem (cont'd)

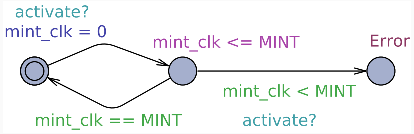
To derive a WCET bound, one needs to combine:

- Program analysis
  - which instructions are executed? how many times?
- Architecture analysis
  - how long does it take to *execute* each instruction?

Real-time model-checking

Real-time model-checking =
automated verification of timed models.

Timed models: discrete event formalism extended with real-valued clocks, *e.g.*, timed automata, or time Petri nets.



Ex: monitoring of a sporadic task with a minimal inter-arrival time

Real-time model-checking (cont'd)

Real-time model checkers:

- Powerful abstractions to represent and manipulate the dense-time part of the state (*e.g.*, DBM, zone)
- But have to relie on an explicit representation of the discrete part (no BDD/ZDD, no efficient partial order)

We have experimented with 2 tools:

- UPPAAL[1] based on timed automata
- Roméo[2] based on time Petri nets

Both offer:

- modular models with synchronization between processes
- finite variables to model the discrete part of the state
- a C-like language to manipulate the discrete part of the state

[1] `https://uppaal.org/`
[2] `http://romeo.rts-software.org`

Motivation for investigating real-time MC for WCET analysis

Pipeline stages, cache, memory controllers, buses are concurrent components that evolve and synchronize in real-time

WCET analysis asks to analyse their timing behavior.

At first sight, real-time model-checking is precisely done for this type of job. It seems an interesting direction to explore for WCET analysis.

Expected advantages and disadvantages

With real-time model-checking, the analysis is based on the exploration of traces

- When an instruction is *executed*, its actual execution time is defined by the current state

→ Thanks to context-sensitive execution times, we expect to obtain accurate bounds

- In case of missing/unknown information, the trace is split to account for the different cases.

initial cache content, input data, contention latency, …

→ We cannot expect to support too much missing/unknown information

Open questions that we wanted to explore

- It will certainly face scalability issues
  → is it even usable?
- Since the analysis is based on traces, it should be ”closer” to the real system
  → how accurate is it ?
- Exhaustivity ensures correctness in the presence of so-called timing anomalies (close to non sustainability in scheduling)
  → is it a golden bullet?

Overview of our approach

$$\begin{tikzcd}
 & \mathcal{S} \arrow[dl, dotted] \arrow[dr, dotted] & \\
\mathcal{P} \arrow[d] & & \mathcal{A} \arrow[d, dashed] \\
\hat{\mathcal{P}} \arrow[r] & \times \arrow[d] & \hat{\mathcal{A}} \arrow[l] \\
 & \hat{\mathcal{S}} \arrow[dd] & \\
 & & \text{WCET} \\
 & \boxed{\text{Model-checker}} \arrow[ur] \arrow[r] \arrow[dr] & \text{Trace} \\
 & & \text{Counters}
\end{tikzcd}$$

Given $\mathcal{S} = \mathcal{P} \times \mathcal{A}$

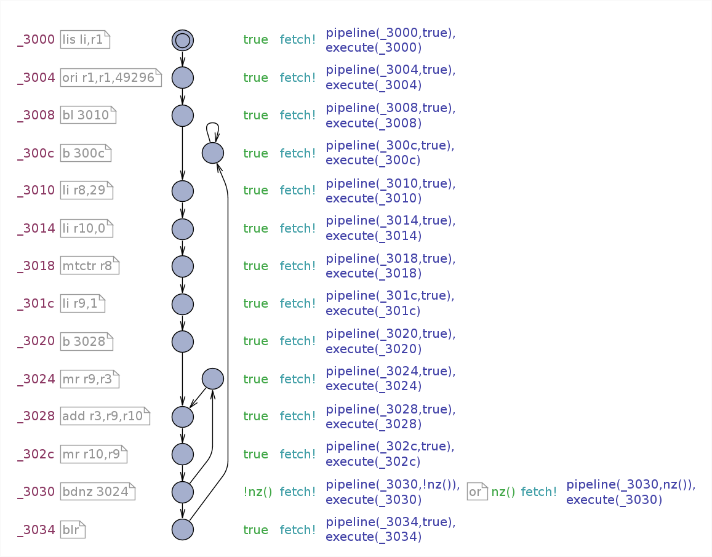
1. Compute an abstract model $\hat{\mathcal{P}}$ of $\mathcal{P}$ (fully automated)
2. Build an abstract model $\hat{\mathcal{A}}$ of $\mathcal{A}$ (not automated but needs to be done only one time)
3. Compute an abstract model $\hat{\mathcal{S}} = \hat{\mathcal{P}} \times \hat{\mathcal{A}}$ of $\mathcal{S}$ (fully automated)
4. Search for the WCET of $\hat{\mathcal{S}}$ with a model-checker (fully automated)

Modeling a program

Basic intuition of program models

A program is a sequence of instructions + a set of memory locations

- We are only interested in binary programs.
- An intuitive representation is an automata/Petri net such that:
  - each instruction is associated with a location/place
  - a control flow between two instructions is denoted by an edge/transition
- This is an untimed model: a program is inactive
- This intuitive representation proved to be relevant for visualization and debugging
- Memory locations are represented by variables
  warning: explicit representation in the state



_3000 lis li,r1 true fetch! pipeline(_3000,true), execute(_3000)
_3004 ori r1,r1,49296 true fetch! pipeline(_3004,true), execute(_3004)
_3008 bl 3010 true fetch! pipeline(_3008,true), execute(_3008)
_300c b 300c true fetch! pipeline(_300c,true), execute(_300c)
_3010 li r8,29 true fetch! pipeline(_3010,true), execute(_3010)
_3014 li r10,0 true fetch! pipeline(_3014,true), execute(_3014)
_3018 mtctr r8 true fetch! pipeline(_3018,true), execute(_3018)
_301c li r9,1 true fetch! pipeline(_301c,true), execute(_301c)
_3020 b 3028 true fetch! pipeline(_3020,true), execute(_3020)
_3024 mr r9,r3 true fetch! pipeline(_3024,true), execute(_3024)
_3028 add r3,r9,r10 true fetch! pipeline(_3028,true), execute(_3028)
_302c mr r10,r9 true fetch! pipeline(_302c,true), execute(_302c)
_3030 bdnz 3024 !nz() fetch! pipeline(_3030,!nz()), execute(_3030) or nz() fetch! pipeline(_3030,nz()), execute(_3030)
_3034 blr true fetch! pipeline(_3034,true), execute(_3034)

Objectives of program abstraction

Execution of an instruction is split in two parts:

- interaction with the micro-architecture, *e.g.*,
  - impact on the state of caches
  - traversal of pipeline
  - or memory accesses
- semantics: updates the memory locations

Observation

For WCET analysis, an update to a memory location can be discarded if it does not impact the timing behavior.

Corrolary: the content of a memory location does not need to be tracked if all its updates can be discarded.

Example

```
00003000 <_start>:
    3000:  li    r1,1         ;r1  <- 1
    3004:  ori   r1,r1,49296 ;ri  <- r1 | 49296
    3008:  bl    3010         ;call main
0000300c <loop>:
    300c:  b     300c         ;branch
00003010 <main>:
    3010:  li    r8,29        ;r8  <- 29
    3014:  li    r10,1        ;r10 <- 1
    3018:  mtctr r8           ;ctr <- r8
    301c:  li    r9,1         ;r9  <- 1
    3020:  b     3028         ;branch
    3024:  mr    r9,r3        ;r9  <- r3
    3028:  add   r3,r9,r10    ;r3  <- r9+r10
    302c:  mr    r10,r9       ;r10 <- r9
    3030:  bdnz  3024         ;ctr--,
                              ;branch if ctr!=0
    3034:  blr                ;return
```

Which registers do we need to track to compute the value of `ctr` at instruction `3030`?

Example

```
00003000 <_start>:
    3000:  li    r1,1         ;r1  <- 1
    3004:  ori   r1,r1,49296  ;ri  <- r1 | 49296
    3008:  bl    3010         ;call main
0000300c <loop>:
    300c:  b     300c         ;branch
00003010 <main>:
    3010:  li    r8,29        ;r8  <- 29
    3014:  li    r10,1        ;r10 <- 1
    3018:  mtctr r8           ;ctr <- r8
    301c:  li    r9,1         ;r9  <- 1
    3020:  b     3028         ;branch
    3024:  mr    r9,r3        ;r9  <- r3
    3028:  add   r3,r9,r10    ;r3  <- r9+r10
    302c:  mr    r10,r9       ;r10 <- r9
    3030:  bdnz  3024         ;ctr--,
                              ;branch if ctr!=0
    3034:  blr                ;return
```

Which registers do we need to track to compute the value of `ctr` at instruction `3030`?

Program slicing to the rescue

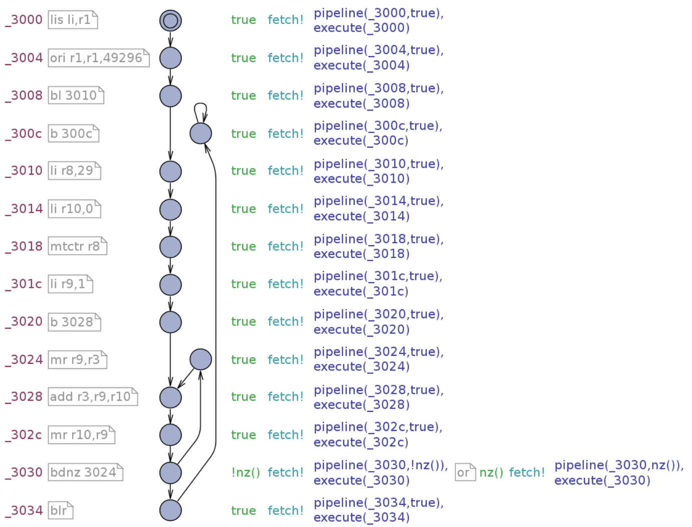
Program slicing = techniques to compute a subprogram which is equivalent to a program wrt. a set of variables and a set of locations[3].

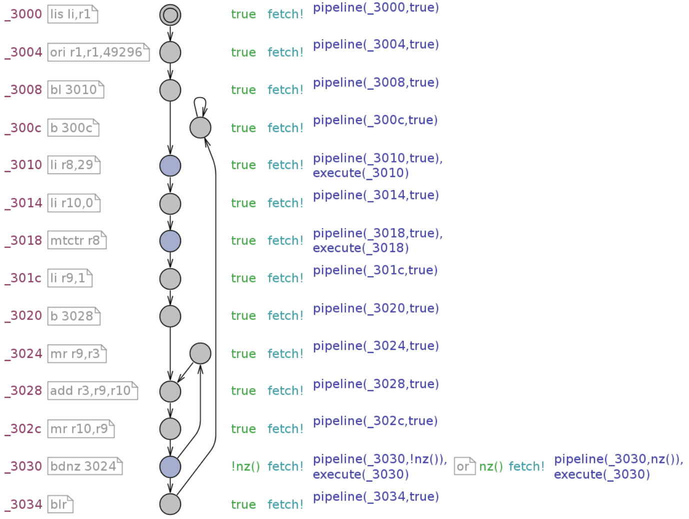
For WCET analysis:

1. Find a subprogram that reaches the end node with the same control flow.
2. Build a model that tracks the content of a memory location iff it appears in this subprogram.

Interactions with the micro-architecture (incl. memory accesses) are not modified.

[3] Kiss *et a.*, *Interprocedural Static Slicing of Binary Executables*. In Int. Work. on Source Code Analysis and Manipulation, 2003.

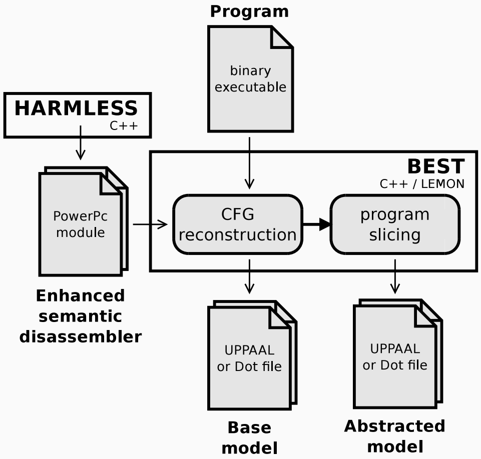

```
_3000 lis li,r1        true  fetch!  pipeline(_3000,true),
                                     execute(_3000)
_3004 ori r1,r1,49296  true  fetch!  pipeline(_3004,true),
                                     execute(_3004)
_3008 bl 3010          true  fetch!  pipeline(_3008,true),
                                     execute(_3008)
_300c b 300c           true  fetch!  pipeline(_300c,true),
                                     execute(_300c)
_3010 li r8,29         true  fetch!  pipeline(_3010,true),
                                     execute(_3010)
_3014 li r10,0         true  fetch!  pipeline(_3014,true),
                                     execute(_3014)
_3018 mtctr r8         true  fetch!  pipeline(_3018,true),
                                     execute(_3018)
_301c li r9,1          true  fetch!  pipeline(_301c,true),
                                     execute(_301c)
_3020 b 3028           true  fetch!  pipeline(_3020,true),
                                     execute(_3020)
_3024 mr r9,r3         true  fetch!  pipeline(_3024,true),
                                     execute(_3024)
_3028 add r3,r9,r10    true  fetch!  pipeline(_3028,true),
                                     execute(_3028)
_302c mr r10,r9        true  fetch!  pipeline(_302c,true),
                                     execute(_302c)
_3030 bdnz 3024        !nz() fetch!  pipeline(_3030,!nz()),  or  nz() fetch!  pipeline(_3030,nz()),
                                     execute(_3030)                             execute(_3030)
_3034 blr              true  fetch!  pipeline(_3034,true),
                                     execute(_3034)
```



```
_3000  lis li,r1          true   fetch!  pipeline(_3000,true)
_3004  ori r1,r1,49296    true   fetch!  pipeline(_3004,true)
_3008  bl 3010            true   fetch!  pipeline(_3008,true)
_300c  b 300c             true   fetch!  pipeline(_300c,true)
_3010  li r8,29           true   fetch!  pipeline(_3010,true),
                                         execute(_3010)
_3014  li r10,0           true   fetch!  pipeline(_3014,true)
_3018  mtctr r8           true   fetch!  pipeline(_3018,true),
                                         execute(_3018)
_301c  li r9,1            true   fetch!  pipeline(_301c,true)
_3020  b 3028             true   fetch!  pipeline(_3020,true)
_3024  mr r9,r3           true   fetch!  pipeline(_3024,true)
_3028  add r3,r9,r10      true   fetch!  pipeline(_3028,true)
_302c  mr r10,r9          true   fetch!  pipeline(_302c,true)
_3030  bdnz 3024          !nz()  fetch!  pipeline(_3030,!nz()),    or  nz()  fetch!  pipeline(_3030,nz()),
                                         execute(_3030)                                 execute(_3030)
_3034  blr                true   fetch!  pipeline(_3034,true)
```

BEST

BEST: a Binary Executable Slicing Tool[4]



[4]Mangean *et al.*, *BEST: a Binary Executable Slicing Tool.* In Int. Work. on Worst-Case Execution Time Analysis, 2016.

Efficiency of slicing for WCET analysis (cont'd)

Binary programs compiled from Mälardalen benchmarks to PowerPC with GCC and COSMIC C, sliced with BEST.

Evaluation of registers whose contents do not need to be tracked

Source file	Gcc				Cosmic C	
	-O0	-O1	-O2	-O3	-no	*default*
`adpcm.c`	11/17, 35%	28/32, 13%	26/28, 7%	33/36, 8%	22/37, 41%	22/37, 41%
`bs.c`	7/11, 36%	10/13, 23%	9/10, 10%	9/10, 10%	10/14, 29%	11/13, 15%
`bsort100.c`	9/12, 25%	13/18, 28%	11/16, 31%	11/16, 31%	13/15, 13%	13/15, 13%
`cnt.c`	10/15, 33%	13/18, 28%	10/16, 38%	10/18, 44%	10/37, 73%	10/37, 73%
`compress.c`	15/19, 21%	26/31, 16%	30/33, 9%	32/35, 9%	21/37, 43%	21/37, 43%
`crc.c`	8/17, 53%	14/23, 39%	10/19, 47%	9/19, 53%	18/37, 51%	18/37, 51%
`expint.c`	8/13, 38%	16/26, 38%	4/11, 64%	4/11, 63%	14/37, 62%	14/37, 62%
`fdct.c`	6/13, 54%	4/21, 81%	4/30, 87%	3/33, 91%	3/35, 91%	3/35, 91%
`fibcall.c`	7/11, 36%	7/12, 42%	3/7, 57%	3/7, 57%	6/12, 50%	6/10, 40%
`fir.c`	7/16, 56%	13/22, 41%	14/21, 33%	14/21, 33%	15/37, 59%	15/37, 59%
`janne_complex.c`	7/12, 42%	6/9, 33%	6/8, 25%	7/9, 22%	7/36, 81%	7/8, 13%
`jfdctint.c`	8/11, 27%	3/15, 80%	4/25, 84%	4/33, 88%	3/35, 91%	3/34, 91%
`matmult.c`	10/19, 47%	15/20, 25%	15/19, 21%	13/19, 32%	8/37, 78%	8/37, 78%
`ndes.c`	9/17, 47%	21/27, 22%	23/26, 12%	27/28, 4%	16/37, 57%	15/37, 59%
`ns.c`	9/14, 36%	13/17, 24%	13/15, 13%	9/12, 25%	14/37, 62%	14/36, 61%
`prime.c`	10/13, 23%	6/9, 33%	6/9, 33%	6/8, 25%	11/36, 69%	12/36, 67%
Average	38%	35%	36%	37%	59%	54%
	37%				57%	

# of registers in the slice / total # of register used in program, gain in percentage (the higher the better).

Modeling the micro-architecture

Requirements

The model of the architecture:

- Should allow cycle accurante *execution* of instruction
  $\Longrightarrow$ it is a timed model
- But should not mimic the actual design of the micro-architecture
- In particular, semantics of an instruction should be executed independantly from the interaction with the micro-architecture

Any transformation that preserves the behavior and decrease either the size of the discrete state or the number of state is welcome!

Modeling style

After several trials, we have adopted the following modeling style:

- the C-like language is used to define and manipulate the discrete part of the state
- the TA/TPN part is used for handling clocks and synchronizations

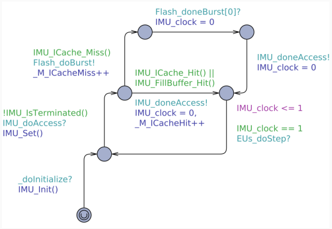
Our early models integrated too much timing and functional aspects. A clear separation offers more possibilities of abstraction, *e.g.*,

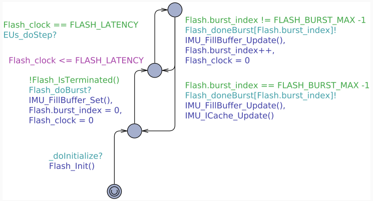
- actual content of the cache is not needed (only the tag & valid bit)
- ALU does no computation, it is just a stage to add a delay according to the class of the instruction
- speculation and rollbacks can be pre-computed offline and integrated in the model of the program
- …

Example: UPPAAL model of memory hierarchy

Part of the model inspired from PowerPC e200z4 micro-architecture

- Instruction cache only
- On cache-hit, instruction is sent to the pipeline in 1 cycle
- On cache-miss, a burst access is required (8 4-bytes words)
- Burst is received in a FillBuffer
- Burst starts with the requested word





Example: UPPAAL model of cache update (pseudo-RR)

```
void IMU_ICache_Update() {
  // on a miss, insert the current instruction on the instruction cache
  int                          addr = _INSTS[IMU.FillBuffer.index].addr;
  int[0, IMU_WAYS_MAX]         way;
  int[0, IMU_SETS_MAX -1]      set   = (addr /  32) % IMU_SETS_MAX;
  int                          tag   =  addr / (32  * IMU_SETS_MAX);

  bool found = false;

  way = 0;
  while (!found && way < IMU_WAYS_MAX)
    if (IMU.ICache.tags[way][set] == -1)
      found = true;
    else ++way;
  if (found) {
    // free slot found
    IMU.ICache.tags[way][set] = tag;
  } else {
    // no free slot found (pseudo round-robin replacement policy)
    way = IMU.ICache.rp_way;
    IMU.ICache.tags[way][set] = tag;
    IMU.ICache.rp_way = (IMU.ICache.rp_way +1) % IMU_WAYS_MAX;
  }
}
```

Validation

Validation of the hardware model is the most difficult part

- Documentation of micro-architecture generally lacks precision
- Micro-benchmarks can be used to fill in the gaps in the documentation but it is a tedious job with no completness guarantee
- Moreover, complexity of models is high enough to raise concerns
    - C-like language is not a 1st class citizen in UPPAAL/Roméo
- Model-checking can be used to verify some properties but we can only find the bugs that we are looking for.

WCET analysis

WCET, execution trace, counters

Both UPPAAL and Roméo provide dedicated algorithms to search for the maximum value of a clock

- UPPAAL: `sup: _clock`
- Roméo[5]: `maxcost(Program::END_INST == 1)`

The result is composed of:

- a value: the WCET bound
- a trace: a run of $\hat{\mathcal{P}}$ on $\hat{\mathcal{A}}$ that yields the WCET bound
- $\hat{\mathcal{A}}$ can be instrumented to also embed performance counters (*e.g.*, cache hit/miss or BTB hit/miss)

If the control flow of the program is independant from unknown input-data, the trace can be replayed on the real system.

[5]In practice, we need to search the mincost of a negative value

About the tightness of the estimation

We did once the effort to validate closely hardware models[6]

- Model of an ARM9TDMI core by micro-benchmarking of the system
    - some hidden architectural features have been discovered using µBenchmarking but some were probably not
- Ad-hoc program slicing
- WCET analysis with UPPAAL
- Then measurements were done on the real hardware
    - for programs with input-dependant control flow, worst-case inputs were used but the measure must be considered as a lower bound of the actual WCET

[6] Cassez F. and Béchennec J.-L., *Timing Analysis of Binary Programs with UPPAAL*, in 13th Int. Conf. on Application of Concurrency to System Design, 2013

About the tightness of the estimation: data

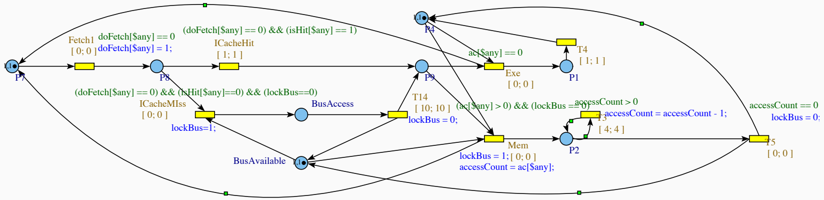
Program	Analysis time	# States	Analysed WCET	Measured WCET	Error
Single-path programs					
fib-O0	2s	74,181	8,098	8,064	0.42%
fib-O1	0.6s	22,333	2,597	2,544	2.0%
fib-O2	**0.3s**	9,711	1,209	1,164	3.8%
jane-complex-O0	1.7s	38,038	4,264	4,164	2.4%
jane-complex-O1	0.5s	14,600	1,715	1,680	2.0%
jane-complex-O2	0.5s	13,004	1,557	1,536	1.3%
fdct-O1	21s	60,534	4,245	4,092	3.7%
fdct-O2	3.2s	55,285	19,231	18,984	1.3%
Single-path programs w/ data dependant instr. durations					
fdct-O0	124s	85,008	11,800	11,448	3.0%
matmult-O0	217s	10,531,262	529,250	528,684	**0.1%**
matmult-O1	25s	1,112,527	156,367	153000	2.2%
matmult-O2	121s	6,780,931	148,299	140,664	5.4%
jfdcint-O0	92s	100,861	12,918	12,588	2.6%
jfdcint-O1	12s	35,419	5,072	4,688	**8.6%**
jfdcint-O2	5.38s	175,661	16,938	16,380	3.4%
Multi-path programs (input data dependant control flow)					
bs-O0	30s	1,421,274	1068	1,056	1.1%
bs-O1	23s	1,214,673	738	720	2.5%
bs-O2	12s	655,870	628	600	4.6%
cnt-O0	4s	77,002	9,027	8,836	2.1%
cnt-O1	1.4s	27,146	4,123	3,996	3.1%
cnt-O2	9s	11,490	3,067	2928	4.6%
insertsort-O0	**598.98s**	24,250,738	3133	3108	0.8%
insertsort-O1	353.80s	11,455,293	1533	1500	2.2%
insertsort-O2	11.68s	387,292	1326	1320	0.4%
ns-O0	60s	3,064,316	30,968	30,732	0.8%
ns-O1	8s	368,720	11,701	11,568	1.1%
ns-O2	55s	1,030,746	7280	7236	0.6%

WCET analysis of multicore systems (WIP)

Architecture modeling

Using the Roméo tool (Time Petri Net)

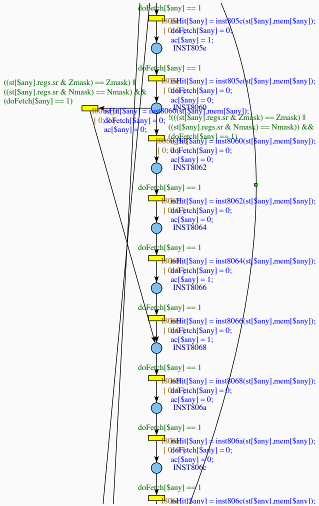
- Simple architecture with 2-stages pipeline (Fetch + Execute)
- Handle the Load/Store multiple words of ARM Cortex
- Small 512 bytes direct mapped instruction cache
- No data cache



Program modeling (1)

Cortex M0+ binary code of bsort compiled with -02 optimization (no slicing)

```
0000804c <BubbleSort>:
 804c: b570 push  {r4, r5, r6, lr}
 804e: 2463 movs  r4, #99 ; 0x63
 8050: 1d06 adds  r6, r0, #4
 8052: 0033 movs  r3, r6
 8054: 2201 movs  r2, #1
 8056: 2501 movs  r5, #1
 8058: e00a b.n   8070 <BubbleSort+0x24>
 805a: 6819 ldr   r1, [r3, #0]
 805c: 6858 ldr   r0, [r3, #4]
 805e: 4281 cmp   r1, r0
 8060: dd02 ble.n 8068 <BubbleSort+0x1c>
 8062: 2500 movs  r5, #0
 8064: 6018 str   r0, [r3, #0]
 8066: 6059 str   r1, [r3, #4]
 8068: 3201 adds  r2, #1
 806a: 3304 adds  r3, #4
 806c: 2a64 cmp   r2, #100 ; 0x64
 806e: d001 beq.n 8074 <BubbleSort+0x28>
 8070: 4294 cmp   r4, r2
 8072: daf2 bge.n 805a <BubbleSort+0xe>
 8074: 2d00 cmp   r5, #0
 8076: d102 bne.n 807e <BubbleSort+0x32>
 8078: 3c01 subs  r4, #1
 807a: 2c00 cmp   r4, #0
 807c: d1e9 bne.n 8052 <BubbleSort+0x6>
 807e: bd70 pop   {r4, r5, r6, pc}
```



doFetch[$any] == 1
isHit[$any] = inst805c(st[$any],mem[$any]);
doFetch[$any] = 0;
ac[$any] = 1;
INST805e
((st[$any].regs.sr & Zmask) == Zmask) ||
((st[$any].regs.sr & Nmask) == Nmask) &&
(doFetch[$any] == 1)
isHit[$any] = inst805e(st[$any],mem[$any]);
INST8060
!(((st[$any].regs.sr & Zmask) == Zmask) ||
((st[$any].regs.sr & Nmask) == Nmask)) &&
(doFetch[$any] == 1)
isHit[$any] = inst8060(st[$any],mem[$any]);
INST8062
isHit[$any] = inst8062(st[$any],mem[$any]);
INST8064
isHit[$any] = inst8064(st[$any],mem[$any]);
INST8066
isHit[$any] = inst8066(st[$any],mem[$any]);
INST8068
isHit[$any] = inst8068(st[$any],mem[$any]);
INST806a
isHit[$any] = inst806a(st[$any],mem[$any]);
INST806c
isHit[$any] = inst806c(st[$any],mem[$any]);

Program modeling (2)

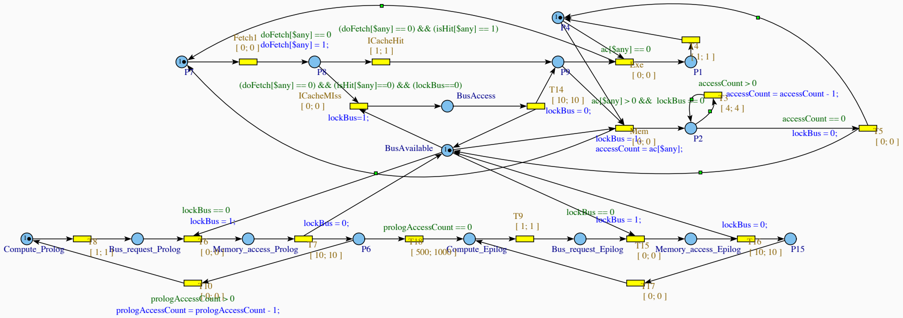
```
int inst8078(core_t &core, mem_t &mem) { // 8078: subs r4, #1
  core.regs.r[4] = core.regs.r[4] — 1;
  updateSR(core.regs, core.regs.r[4]);
  return cacheAccess(core.ICache, 32888);
}

int inst807a(core_t &core, mem_t &mem) { // 807a: cmp r4, #0
  uint32_t val = core.regs.r[4] — 0;
  updateSR(core.regs, val);
  return cacheAccess(core.ICache, 32890);
}

int inst807c(core_t &core, mem_t &mem) { // 807c: bne.n 8052
  return cacheAccess(core.ICache, 32892);
}

int inst807e(core_t &core, mem_t &mem) { // 807e: pop {r4, r5, r6, pc}
  core.regs.r[15] = memRead(mem, core.regs.r[13] + 0);
  core.regs.r[6] = memRead(mem, core.regs.r[13] + 4);
  core.regs.r[5] = memRead(mem, core.regs.r[13] + 8);
  core.regs.r[4] = memRead(mem, core.regs.r[13] + 12);
  core.regs.r[13] = core.regs.r[13] + 16;
  return cacheAccess(core.ICache, 32894);
}
```

Phased opponent modeling



Conclusions and future work

Conclusions

- Of course there are scalability problems. Naive modeling generally leads to an explosion of the state space
- The more uncertainties there are about the behavior of a program, the greater the number of traces to explore (Timed model checkers use symbolic states for time but not for discrete variables)
- Aggressive yet accurate abstraction techniques can enable effective use of these methods

As is often the case, going from a research prototype to industrial-grade tool will require a huge engineering effort.

Future work

- Detection of temporal anomalies using observers added to the model
- Aggressive abstractions for multi-core:
    - slicing to reduce programs to memory accesses
    - taking into account the phases in the behavior of the programs
- Ad-hoc model checking techniques taking into account the specificities of the problem

Thanks for your attention!