PROXIMA

Mitigating Software Instrumentation Cache Effects in Measurement-Based Timing Analysis

1 

2 

3 IK4 IKERLAN
Research Alliance

Enrique Díaz[1,2] , Jaume Abella[2], Enrico Mezzetti[2],
Irune Agirre[3], Mikel Azkarate-Askasua[3],
Tullio Vardanega[4], Francisco J. Cazorla[2,5]

4

5

16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)
Toulouse, France, 5th July 2016

This project and the research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7 / 2007-2013] under grant agreement 611085

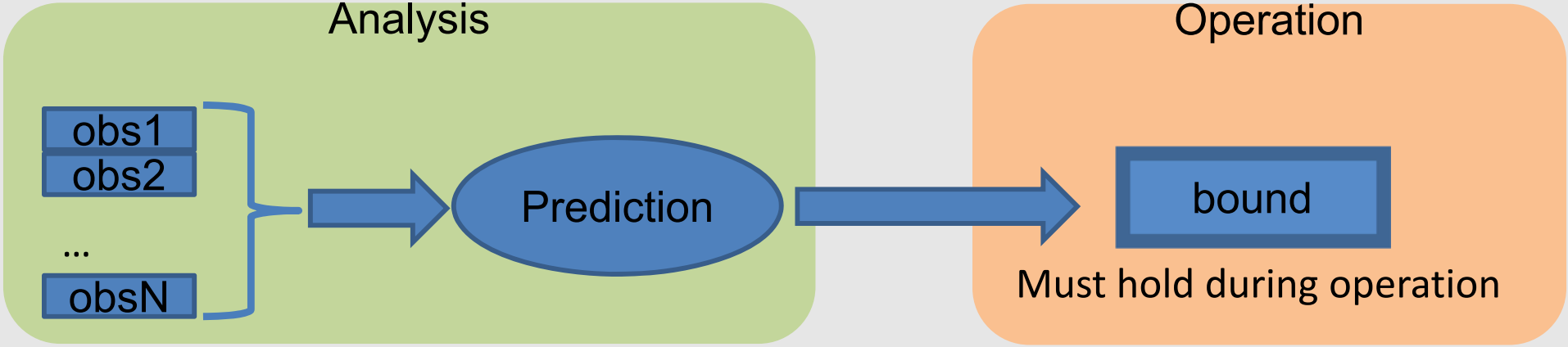
www.proxima-project.eu

Agenda

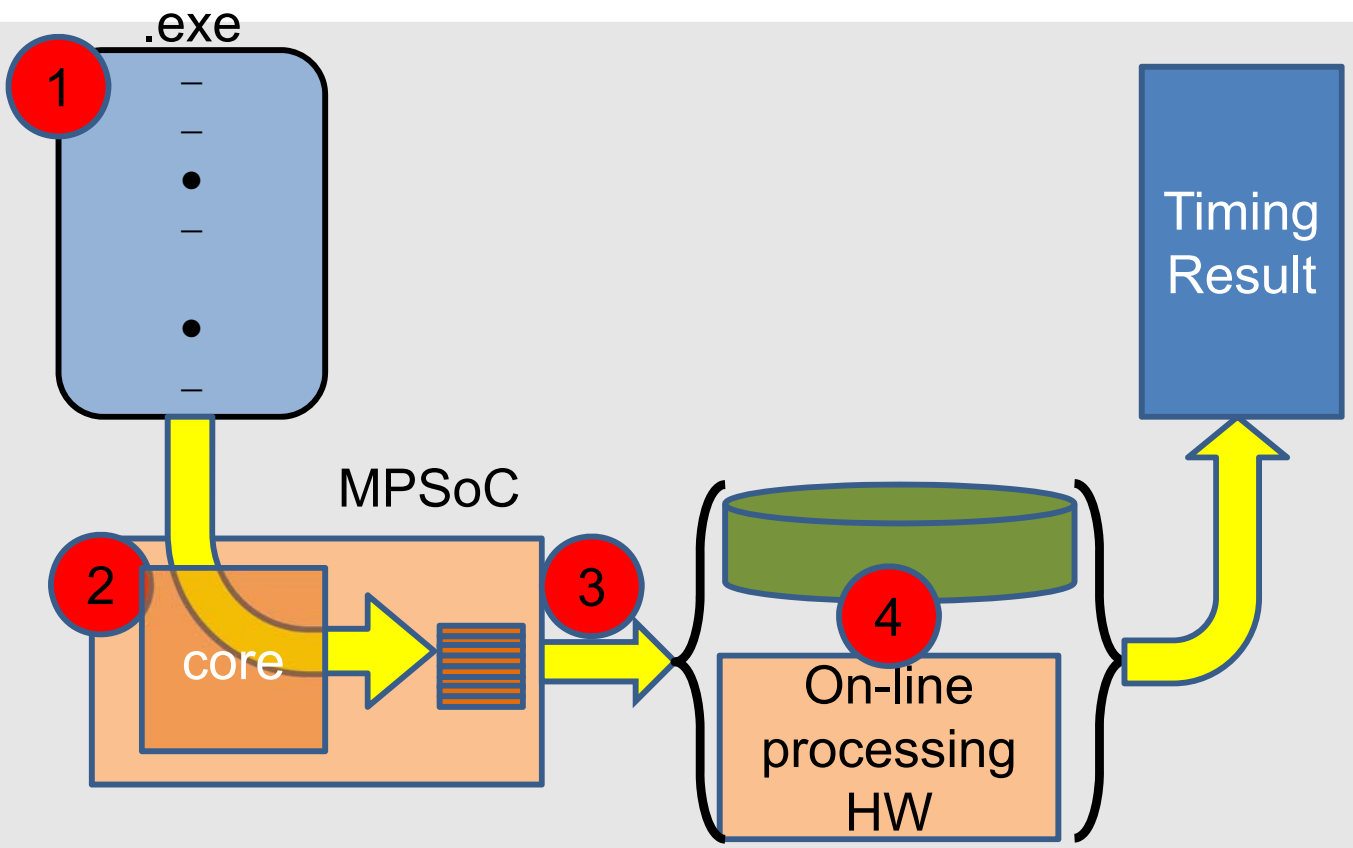
- Measurement-Based Timing Analysis (MBTA)
  - Introduction
  - General application process
    - Allocation of ipoints
    - Trace generation
      - Hardware and Software
    - Trace collection and
    - Trace processing
- Software trace generation
  - Need and problems in the presence of caches
- Solution Proposal
- Evaluation: Setup and Results
- Conclusions

Introduction to MBTA

- MBTA
  - Widely used in industry space, automotive, railway, aerospace, …
- Phases:
  - Analysis phase
    - Collect measurements to derive a WCET estimate that holds valid during system operation
  - Operation phase
    - Actual use of the system (under assumption is stays within its performance profile)



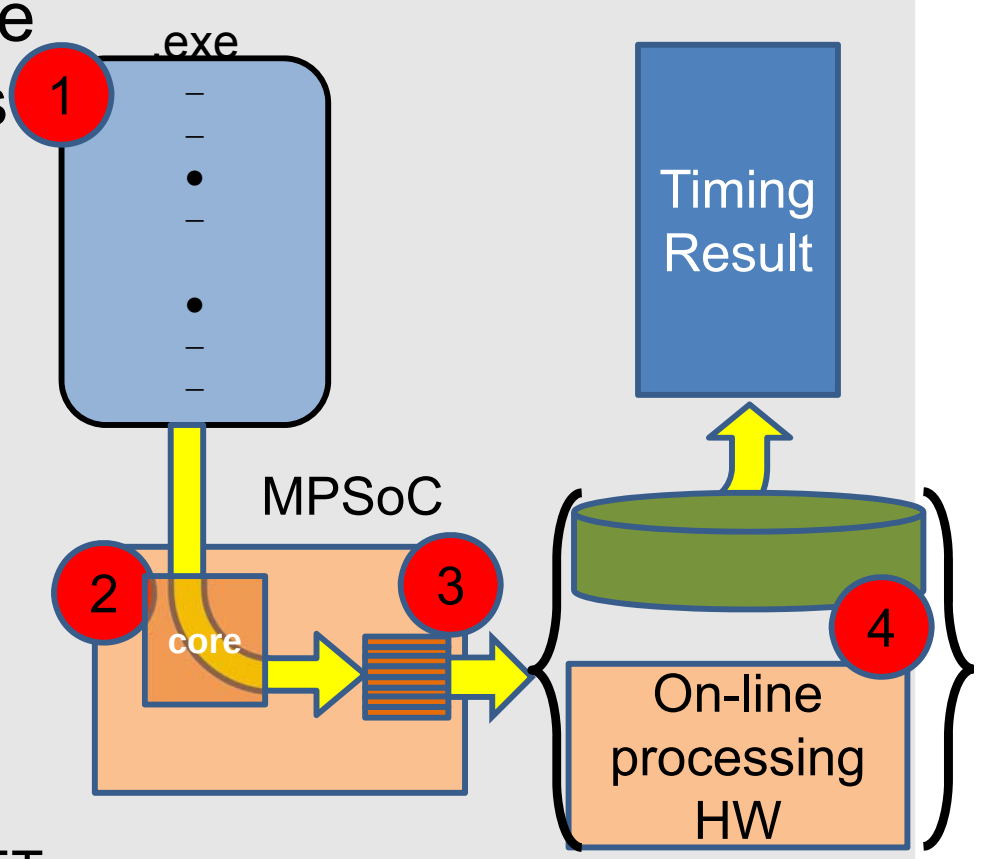
MBTA: General Processs



- Generates a time trace that logs the time at which ipoints are hit
  1) Ipoint (●) placement
  2) Trace generation: 'Read time when hitting an ipoint'
  3) Trace collection: 'Get the reading outside the board'
  4) Trace processing: 'Make sense of the readings'

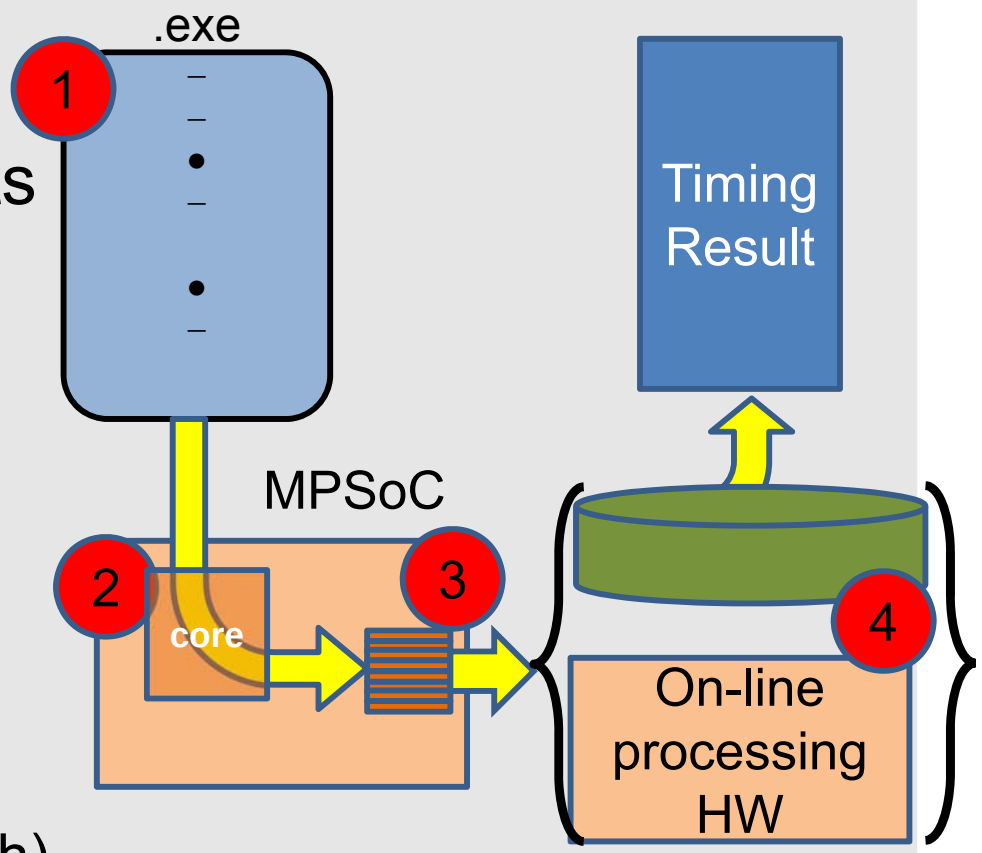
1. Ipoint location

- The number and location of the ipoints depend on the analysis
- Extremes of the spectrum
  - Unit of Analysis (e.g. function)
  - Basic block boundary
- In general:
  - Identify small program parts/segments (extracted from an analysis of the CFG) [6][1]
  - Segments chosen to
    - facilitate the derivation of a WCET by composing the WCET of each segment [19][1] or
    - to reduce the number of ipoints



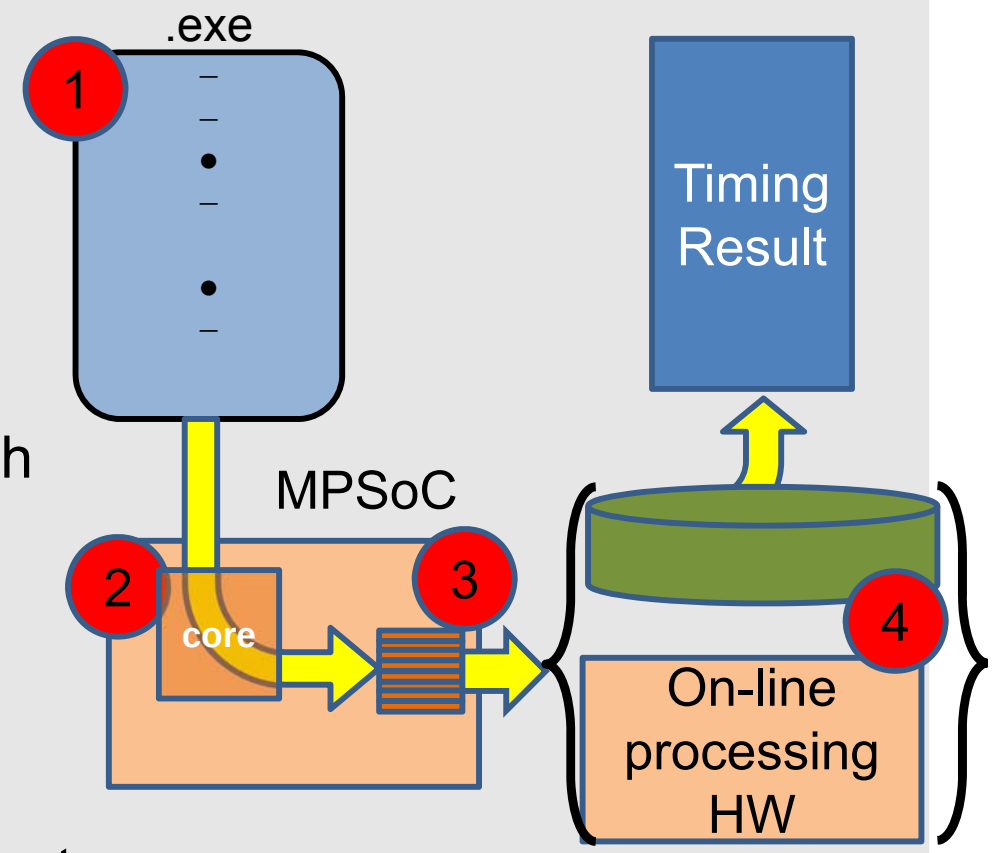
3. Trace Collection and 4. Processing

- Instrumented program execution on the target results in a set of timestamps and events
- Collection
  - Out-of-band support exists so trace collection does not impact program execution
- Processing
  - Either on-line via specialized hardware (can be costly)
  - Or off-line (trace files can be high)
  - Balance ipoint frequency
- Their impact assumed null
  - Otherwise, its additive nature will allow to easily factor them in



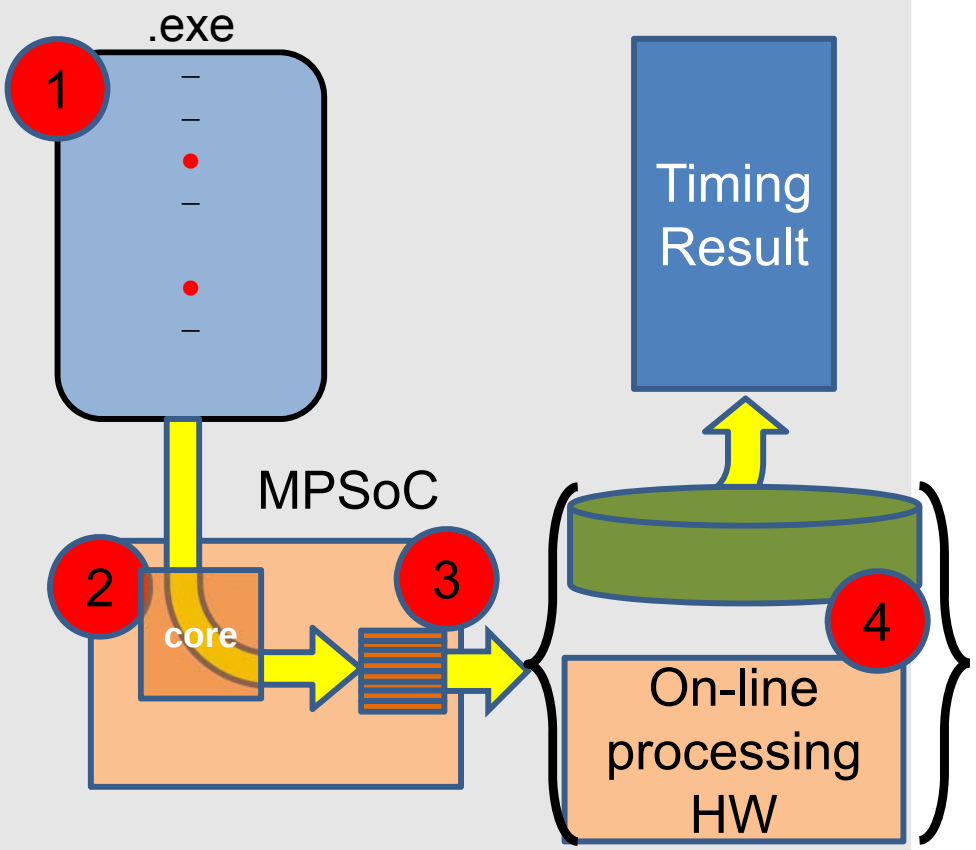
2.a. Hardware Trace Generation

- Advance debug hardware trigger specific actions when certain opcodes are executed
- Interfaces exist to program:
  - The type of instruction to trace
  - The action to perform when such an instruction is hit
  - E.g. Nexus or GRMON for the LEON processor family
- In general
  - Debug hardware of that kind is not present in all processors used in real-time systems
  - In many systems software instru-mentation support is needed



2.b. Software Trace Generation

- Instrumentation instructions/code (icode) are inserted
  - E.g icode that reads the time-base register and output its contents to a specific I/O address
  - Instrumentation instructions: move time to a special purpose register / memory position
- Added by the instrumenter



2.b. Software Trace Generation: overheads

- Direct: execution of executing instrumentation code
  - Core: $\Delta_{icode}^{core-exec}$
  - MPSoC (chip): $\Delta_{icode}^{chip-exec}$
- Indirect: change in the layout of program code in memory.
  - Ipoints shift the memory position of following instructions → address shift → different cache set layout → different program!
  - Evidence that the execution-time the instrumented binary (iprog) is larger or smaller than those obtained with oprog?

$$ETiprog = ETorprog + \left(\Delta_{icode}^{core-exec} + \Delta_{icode}^{chip-exec} + \Delta_{icode}^{collect}\right) + \Delta_{icode}^{malign}$$

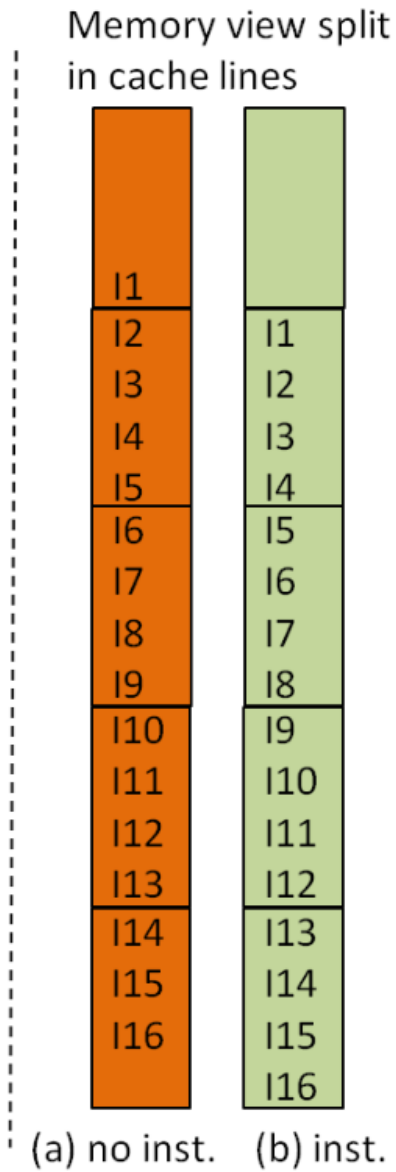
  - $\Delta_{icode}^{malign} > 0$ or $\Delta_{icode}^{malign} < 0$
    - **With as low as a single instrumentation instruction**

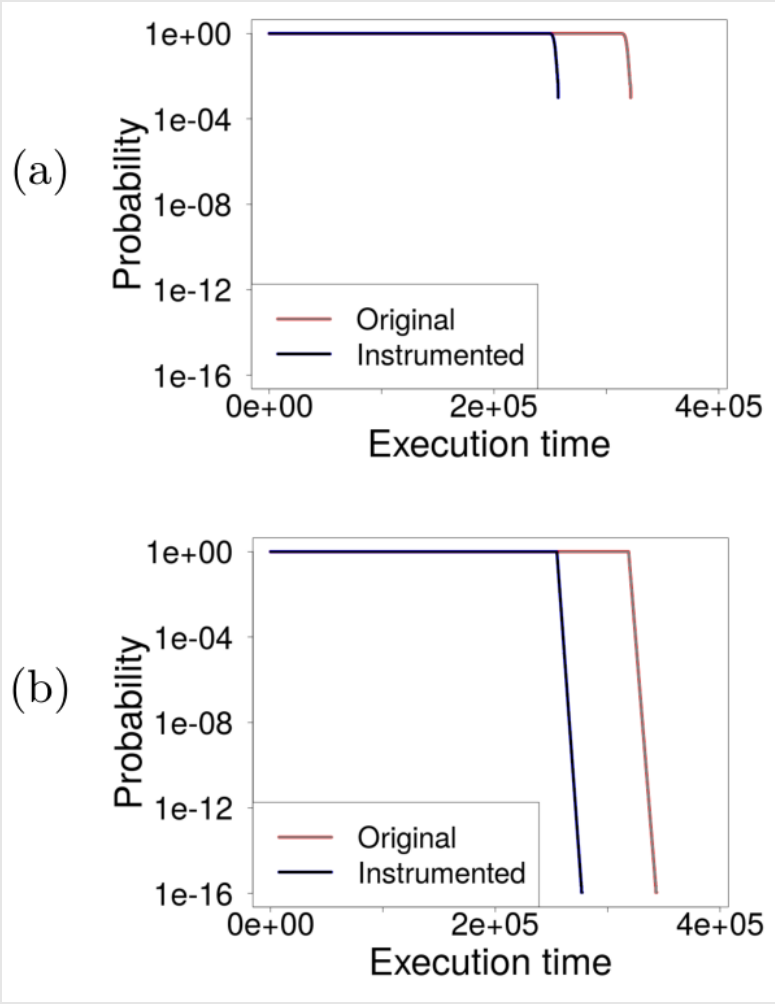
To leave or not to leave (the icode)

- Removing icode (from the final executable)
  - How the execution-time observations taken with the *iprog* correlate with the timing behaviour of the *oprog*
  - Functional and timing verification conducted on different software
    - Strong additional argument must be provided for the analysis result to hold
- Leaving icode
  - Cost and complexity to demonstrate equivalent functionality
    - Certification and qualification practices may simply not accept the presence of this instrumenter-added code
  - Likely to worsen memory footprint and average performance
  - Some memory-mapped I/O space – where execution-time readings might be kept – may be unnecessarily wasted

Removing the code: example

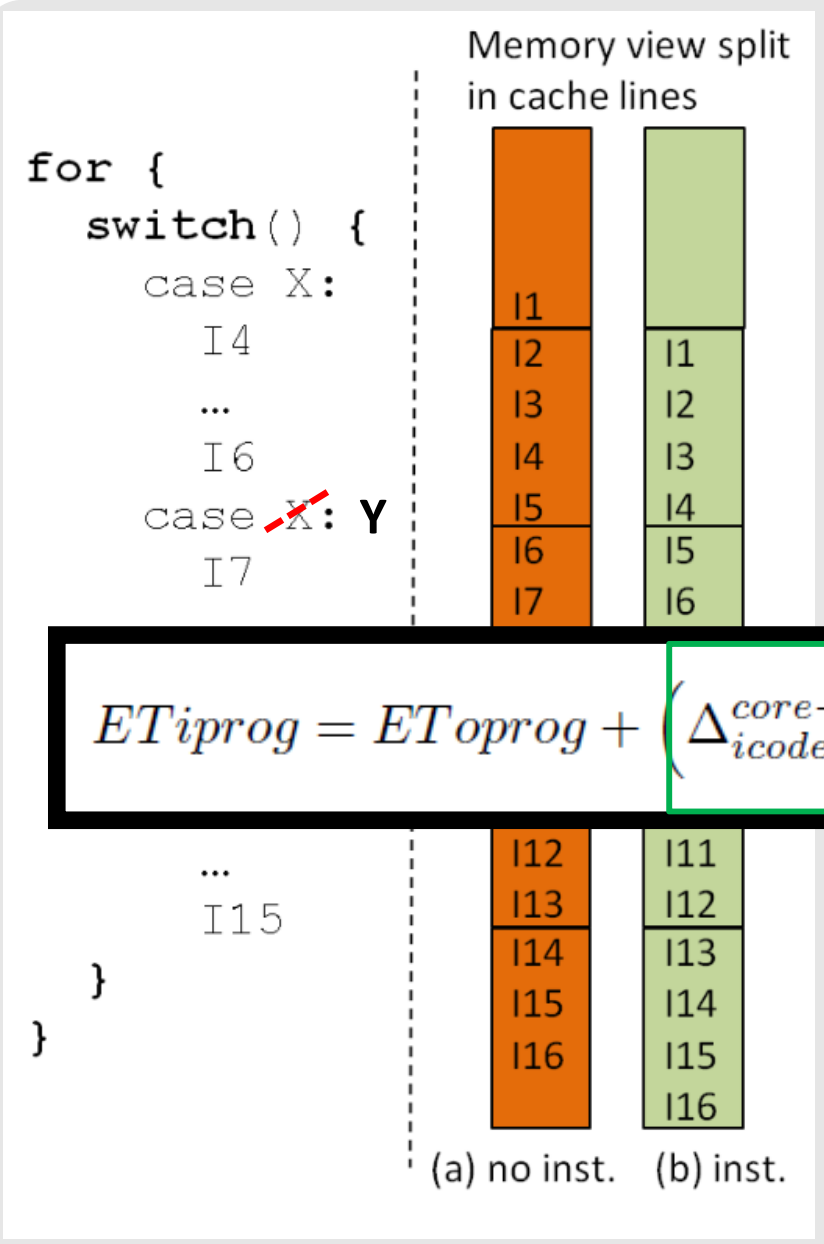
```
for {
   switch() {
      case X:
         I4
         …
         I6
      case X: Y
         I7
         …
         I10
      default:
         I11
         …
         I15
   }
}
```



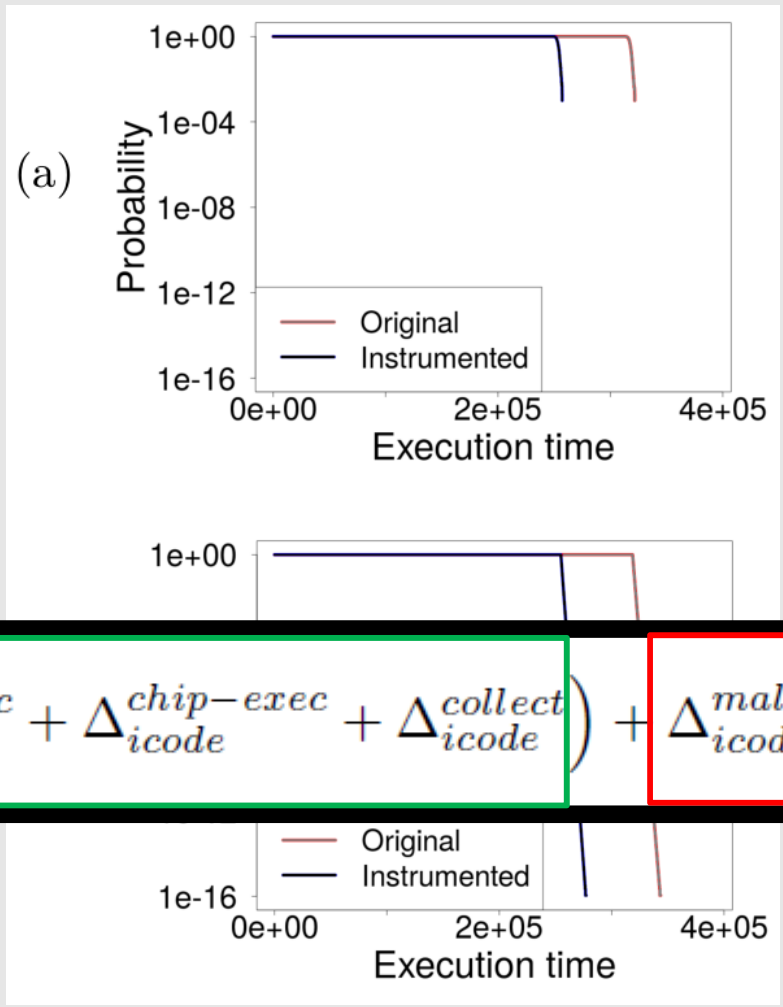


- 2 set – 2 way cache
- Time *iprog* < Time *oprog*

Removing the code: example



$$ETiprog = ETorprog + \left(\Delta_{icode}^{core-exec} + \Delta_{icode}^{chip-exec} + \Delta_{icode}^{collect}\right) + \Delta_{icode}^{malign}$$



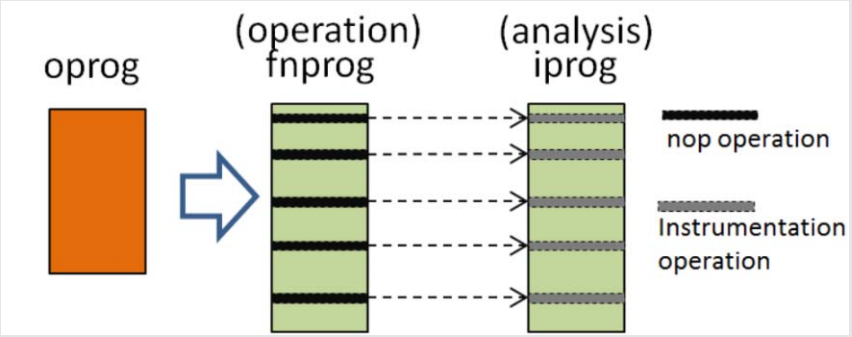
- 2 set – 2 way cache
- Time *iprog* < Time *oprog*

Our approach: goals

- G1:
  - Execution time (version of the program for WCET analysis) > execution time (version of the program used during operation)
    - Reliability
- G2 (secondary):
  - Reduce overhead of the program used at operation in
    - memory size and
    - average execution time

Proposal

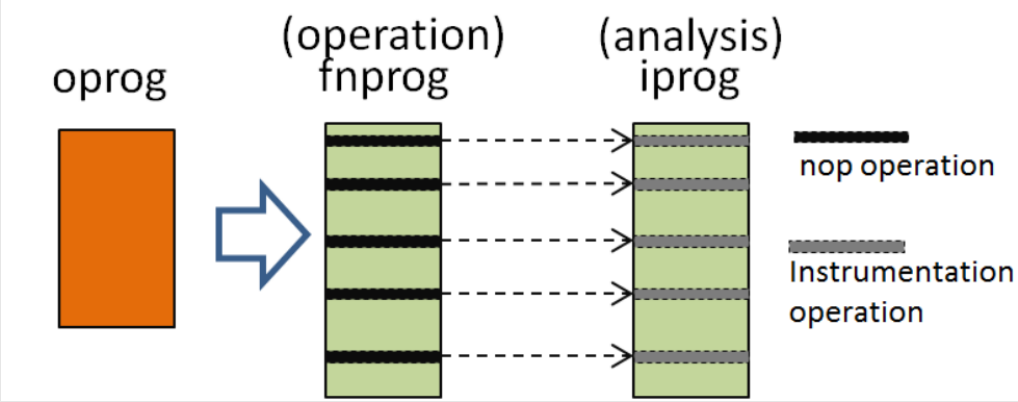
- Three versions of the program:
  - Original (*oprog*)
  - Functionally neutral (*fnprog*)
  - Instrumented (*iprog*)



- *fnprog* (operation):
  - Generated from oprog by inserting nop instructions at desired instrumentation points
- *iprog* (analysis):
  - For timing analysis, nops are replaced by actual instr. Operations

Number of nops inserted per ipoint in *fnprog* so that cache alignment of code in *fnprog* and *iprog* stays unchanged

Arguments to be made



- A1: *fnprog* provides the same functional output as *oprog*
- A2: execution time (*iprog*) > execution time (*fnprog*)
  - *iprog* → analysis
  - *fnprog* → operation
- Reduce overhead of *fnprog*

A1: fnprog = oprog functionally speaking

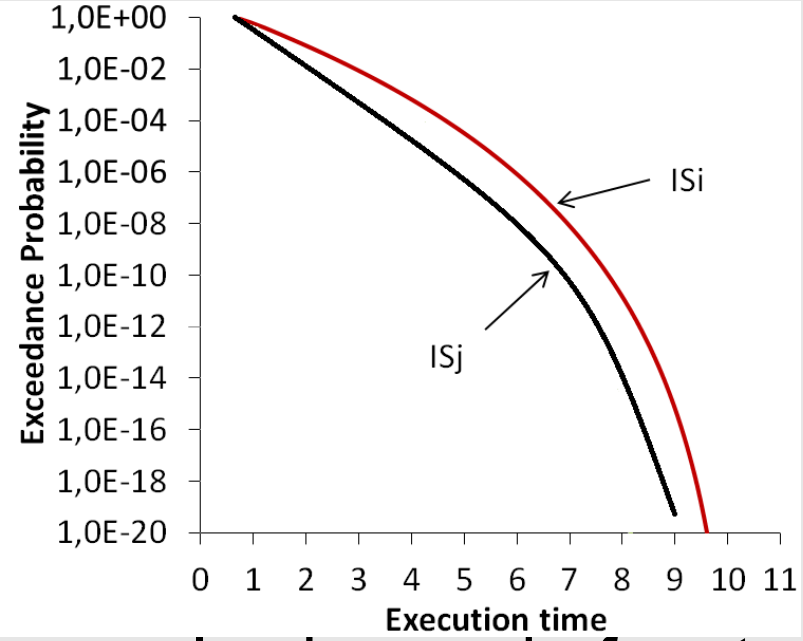
- ‘fnprog = oprog + nops’

- A nop operation:
  1) by definition performs no operation
  2) its does not change status flags or any other control registers
  3) generates neither interrupts nor exceptions
  4) uses no architectural (programmer accessible) register
     - Allows inserting nops anywhere in the code
  5) has no input and no output (register) dependences

- From all these properties it follows that *fnprog* cannot change the functional behaviour of *oprog*

A2: et(iprog) > et(fnprog)

- Measurement-Based Probabilistic Timing Analysis MB**P**TA[5]:
  - ISi = instruction sequence
  - pET(ISi) = its probabilistic execution time (pET)
  - ISi = ISj + {instruction} → pET(ISi) ≥ pET (ISj)
    - For any cut-off probability the exec. time of ISi ≥ exec. time of ISj .



- This argument can also be made for standard MBTA

Average performance

- Nops:
  - usually take a few cycles to execute
  - The processor may even strip them out from the pipeline before they reach the execution stage.

- Instrumentation instructions:
  - Usually need to access off-core (or off-chip) resources such as I/O ports or trace buffers, thus incurring longer execution times.

Setup

- Cycle-accurate simulator
- Cache:
  - 4KB L1 instruction- and data-caches
  - 128 sets and 2 ways each
  - Random placement and replacement
- Latencies:
  - The access latency to the L1 caches is 1 cycle
  - The access latency to main memory is 28 cycles.
- Instrumentation overhead:
  - For the instrumentation instructions, we assume they have the cost of 2 cycles.

Benchmarks

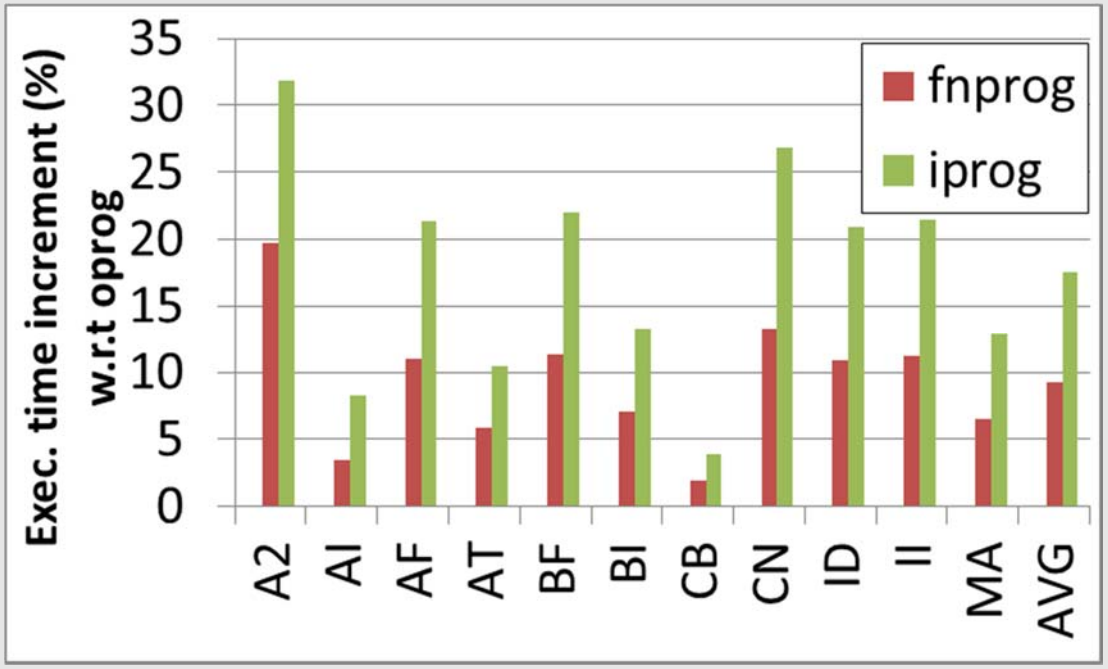
- EEMBC automotive benchmarks:
  - a2time(A2), aifftr(AI), aifirf(AF), aiifft(AT), bitmnp(BI), cacheb(CB), canrdr(CN), idctrn(ID), iirflt(II), matrix(MA)
- Railway case-study application
  - Part of the European Railway Traffic Mgmt. System (ERTMS)
  - On-board unit of the ERTMS, called European Train Control System (ETCS).
  - We consider 10 different input sets (S0 to S9)

Results: EEMBC. Code & time overhead

- Code size and exec. time increase (bb instrumentation)
  - fnprog and iprog w.r.t oprog

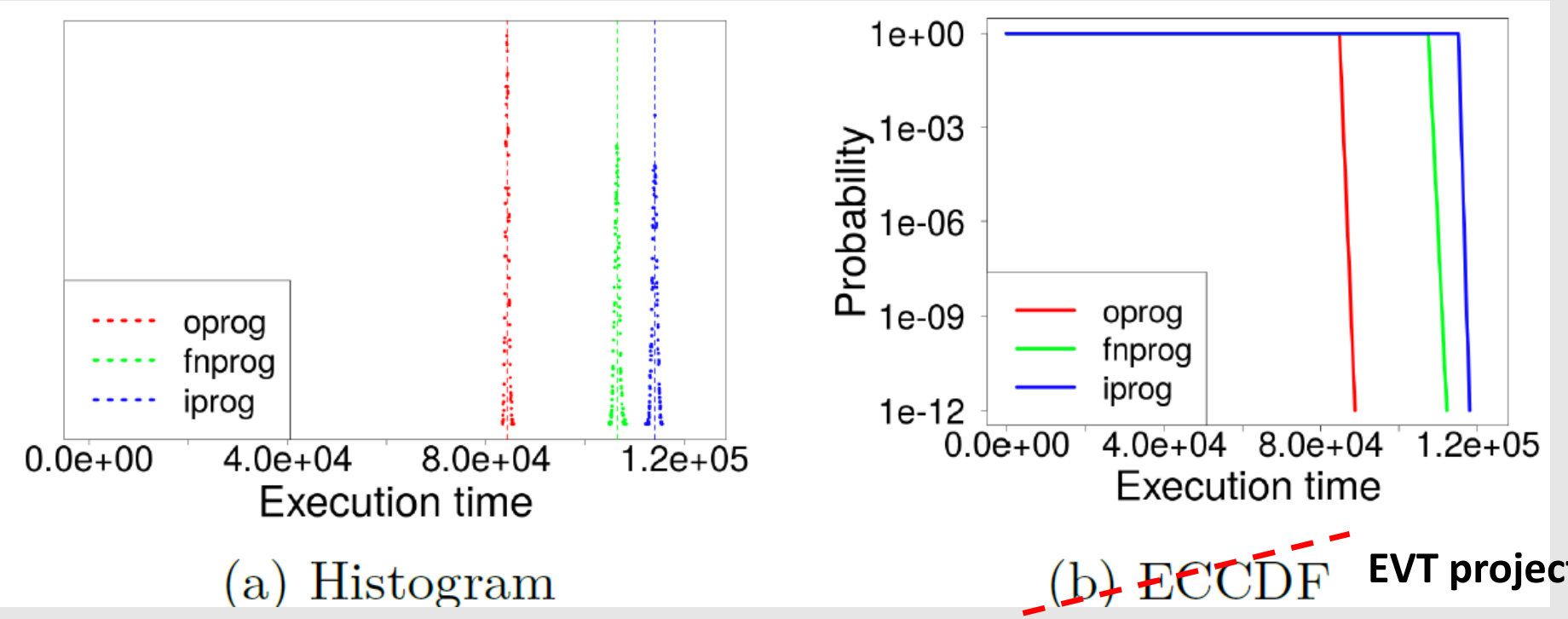
no. instruct.	Code Size	execution time fnprog	execution time iprog
1 inst./2 inst.	6.87%/13.74%	4.35%/9.28%	8.33%/17.54%

  - Execution Time overhead (breakdown per task)



Results: EEMBCs. pWCET results

- Example for a2time



- Results all benchmarks @ cutoff probability of 10e-12

prog.	A2	AI	AF	AT	BF	BI	CB	CN	ID	II	MA
fnprog	26.4%	3.8%	11.6%	7.1%	11.5%	11.9%	2.1%	14.1%	12.3%	11.9%	6.4%
iprog	33.0%	9.3%	22.1%	12.4%	22.0%	16.8%	4.0%	27.1%	23.3%	21.6%	12.7%

Results: Railway case study

- 2 instrumentation instructions per ipoint
- Code and execution time overhead results
  - Tighter on average than those for EEMBC
  - Average pWCET estimate increase estimates across Sx
    - 8.7% (*fnprog*)
    - 11.9% (*iprog*)

prog.	S0	S1	S2	S3	S4	S5	S6	S7	S8	S9
fnprog	8.4%	7.6%	9.5%	9.1%	8.1%	9.5%	9.4%	8.3%	8.6%	8.9%
iprog	11.5%	10.4%	12.1%	12.3%	12.2%	13.3%	12.4%	11.9%	12.2%	11.1%

Table 3 pWCET estimates for 10^{-12} for *fnprog* and *iprog* w.r.t to that for *oprog*

- Code size increase
  - 12%
  - less than the average incurred with the EEMBC benchmarks

Conclusions

- We presented an approach to
  - mitigate the impact of instrumentation code to prevent cache misalignments from occurring between the *iprog* and *oprog*
  - while incurring low overhead in terms of execution time
- We build upon the use of *functionally-neutral* operations such as nops
  - Easy to show that the program version to be deployed that is functionally equivalent to the original program
  - Has a provable lower execution time than the instrumented version
- Future work:
  - Evaluate the fnprog approach in a real hardware platform and a commercial timing analysis tool

PROXIMA

Mitigating Software Instrumentation Cache Effects in Measurement-Based Timing Analysis

1 

2 

3 IK4 IKERLAN
Research Alliance

Enrique Díaz[1,2] , Jaume Abella[2], Enrico Mezzetti[2],
Irune Agirre[3], Mikel Azkarate-Askasua[3],
Tullio Vardanega[4], Francisco J. Cazorla[2,5]

4

5

16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)
Toulouse, France, 5th July 2016

This project and the research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7 / 2007-2013] under grant agreement 611085

www.proxima-project.eu