Worst Case Execution
Time Computation
by Static Analysis

Hugues Cassé <casse@irit.fr>
TRACES team
IRIT – University of Toulouse
Myself

• Assistant Professor
  • IRIT – University of Toulouse
  • team TRACES is Research on Architecture, Compilation and Embedded System

• My work
  • WCET computation by static analysis
  • static analysis of memory hierarchy
  • data flow analysis of machine instructions
  • designer and chief developer of OTAWA – open source tool to compute WCET
Using the Worst Case Execution Time

- context – systems where time matters
- problem 1
  I start my program at time $t$, will it return its result at time $t + \Delta t$?
- problem 2
  - my system is made of tasks $\tau_1$, $\tau_2$, ...
  - each task $\tau_i$ has a dead-line $D_i$ (and a period $T_i$)
  - we need to know if it is always schedulable
    $\Rightarrow$ we need the cost $C_i$ of each task, how?
Execution of a Program (and time)

1. get instruction from memory
2. get the operand
3. perform the computation
4. store the result

wire traversal time
transistor set-up time
capacitor (dis)charging time

machine clock unit = cycle
Yet... not so simple

```
int last = 0;
void process(int input) {
    for(int i = 0; i < input; i++)
        if(last < 0)
            last = -last;
        else
            last -= i;
    if(last > 0)
        do_something();
}
void main(void) {
    while(1)
        process(get_input());
}
```

- Time comes from
  - system input
  - loops / repetitions
  - alternatives in the execution flow
- Does the program halt?
  - easier question in real-time systems
- Effects of time constraint miss?
Different times

number of execution paths

0

BCET

actual WCET

measured WCET

estimated WCET

BCET = Best Case Execution Time
Open Tool for Adaptive WCET Analysis

• developed in University of Toulouse
  ♦ mostly vertical solution
  ♦ based mainly on abstract interpretation
  ♦ freely available
  ♦ several instruction sets and micro-architectures

• alternative project
  ♦ Heptane (Rennes)
  ♦ SWEET (Mälardalen)
  ♦ aiT from AbsInt (industrial)
Outline

- What's the problem with WCET?
- IPET Approach
- Control Flow Problems
- Hardware Support
- Time Production
- Conclusion
Counting instructions

Program in C

```c
s = 0;
for(int i = 0; i < 100; i++)
  s += i;
```

Program in machine language

```
00008b98 <main>:
  8b98: ldr r2, [pc, #40]
  8b9c: mov r3, #0
  8ba0: str r3, [r2]
  8ba4: ldr r1, [r2]
  8ba8: ldr r0, [pc, #24]
  8bac: add r1, r1, r3
  8bb0: add r3, r3, #1
  8bb4: cmpr3, #10
  8bb8: str r1, [r2]
  8bbc: bne 8ba4 <main+0xc>
  8bc0: ldr r0, [r0]
  8bc4: bx lr
  8bc8: .word 0x00089efc
```

- **method**
  - \( C = n_{\text{inst}} \times \text{latency}_{\text{inst}} \)
  - **condition:**
    \( C = C_{\text{cond}} + \max(C_{\text{then}}, C_{\text{else}}) \)
  - **repetition:**
    \( C = C_{\text{cond}} + (C_{\text{cond}} + C_{\text{body}}) \times N \)

- **latency of instruction**
  - variable / instruction
  - variable / hardware state
  - pipeline

- **idea:** work at machine code level
Measuring the time (a)

- not easy
  - what's about sensors / actuators?
  - external measurement hardware (oscilloscope)
  - internal microprocessor counters
- when are the measurements done?
  - which input tests?
  - which coverage? (blocks, edges, paths)
  - how to activate some parts of code?

RapiTime, G. Bernat

N. Williams, B. Marre, P. Mouy, M. Roger
Automatic Generation of Path Tests by Combining Static and Dynamic Analysis, 2005
Measuring the time (b)

- hardware behaviour depends on internal state
- depends on the previously executed code
- is the empty state the worst case? – Not ever!
- example: data cache with write-back
  - empty: 1 miss → 1 memory access
  - not empty: 1 miss → 2 memory accesses (write-back + access itself)
- measurement: how to test all hardware states?
Alternative: Probabilistic WCET

- based on
  - Extreme Value Theory
  - significant set of measurements
- upside – no need of knowledge of
  - hardware
  - software
- downside
  - when do we have enough measures?
Main issues to compute WCET

- execution paths → execution paths blow-up
  - loops bounds
  - infeasible paths
  - other flow facts
- hardware behaviour → hardware state blow-up
  - deterministic
  - predictable
  - timing anomaly – local worst case does not always lead to global worst-case.
- WCET = execution path (WCEP) with the worst execution time
  - overestimation is enough to achieve safety
  - too much overestimation → waste of hardware resources
Outline

- What's the problem with WCET?
- IPET Approach
- Control Flow Problems
- Hardware Support
- Time Production
- Conclusion
Overview

- **program**
- **path analysis**
  - CFG, loop bounds, infeasible paths, etc
- **hardware analysis**
  - qualification and quantification of hardware behaviour
- **time computation**
  - block time computation
  - global time computation
- **WCET**
**CFG**
*(Control Flow Graph)*

- **G = (V, E, ν, ω)**
  - **V = { basic blocks}** – sequence of instruction only accepting branches at end
  - **E ⊆ V × V** – flow of code (sequence, branches)
  - **ν ∈ V** – entry of CFG
  - **ω ∈ V** – unique exit of CFG
- **construction**
  - analysis of binary code
  - extraction of target of branches
- **paths from ν to ω = superset of execution paths of the program**
IPET (Implicit Path Enumeration Technique)

- WCET =
  - maximization of an ILP system (Integer Linear Programming)
  - flow problem

- $C = \max \sum_{v \in V} t_v \times x_v$
  - $t_v$ – execution time of BB $v$
  - $x_v$ – frequency of execution of $v$ on the WCEP

- under constraints
  - path constraints
  - hardware constraints

- smart solution to manage execution path blow-up

C = $\max 4 \times x_1 + 15 \times x_2 + 7 \times x_3$

Path Constraints

- 1 execution of the task
  \[ x_v = x_\omega = 1 \]
- flow enters a node as many times it leaves it
  \[ \forall v \in V, v \neq v \land v \neq \omega \]
  \[ x_v = \sum_{(w,v) \in \text{PRED}(v)} x_{w,v} = \sum_{(v,w) \in \text{SUCC}(v)} x_{w,v} \]
- \( x_{v,w} \) – traversal frequency of edge \((v, w) \in E\) on WCEP
- model the paths of CFG

\[
\begin{align*}
  x_{\text{entry}} &= x_{\text{exit}} = 1 \\
  x_{\text{entry}} &= x_{\text{entry,1}} \\
  x_1 &= x_{\text{entry,1}} = x_{1,2} \\
  x_2 &= x_{1,2} + x_{2,2} = x_{2,2} + x_{2,3} \\
  x_3 &= x_{2,3} = x_{3,\text{exit}} \\
  x_{\text{exit}} &= x_{3,\text{exit}}
\end{align*}
\]
Loop Constraints

- problem
  - there is a loop
  - as is, C tends toward $\infty$
  - bound for $x_{2,2}$ required!

- loop constraint for loop $h$
  \[ \sum_{(v,h) \in \text{BACK}(h)} x_{v,h} \leq N \]
  - $\text{BACK}(h)$ – back edges of the loop headed by $h$
  - $N$ – loop bound

- bound relative to loop head $h$
  \[ \sum_{(v,h) \in \text{BACK}(h)} x_{v,h} \leq N \times \sum_{(v,h) \in \text{ENTRY}(h)} x_{v,h} \]
  - $\text{ENTRY}(h)$ – edges entering the loop headed by $h$
  - nesting loop support

\[ x_{2,2} \leq 10 \]

or

\[ x_{2,2} \leq 10 \times x_{1,2} \]
Solving the ILP System

- assign constants to $t_v$
- use an ILP solver
  - `lp_solve` – open source
  - CPLEX, … – industrial
- result
  - $C$ – WCET
  - $x_v$ – frequency of execution of block $v$ on WCEP
  - $x_{v,w}$ – frequency of traversal of edge $(v, w)$ on WCEP

NOTE 1: $x_v$ and $x_{v,w}$ may represent several WCEP **implicitly**

NOTE 2: $x_v$ are not mandatory as they are represented as a sum of $x_{v,w}$.

\[ C = 82 \]
Outline

● What's the problem with WCET?
● IPET Approach
● Control Flow Problems
● Hardware Support
● Time Production
● Conclusion
Scanning the execution paths

- **program**
  - binary format – ELF (Unix), ECOFF (Windows)
  - big blocks of bytes – possibly qualified executable
  - entry point address → first executed instruction
  - possibly function addresses

- **instructions**
  - computation, memory access instructions
    - next instruction: address + size
  - branch instructions
    - target address encoded in the instruction
    - conditional → branch on target or on next instruction
    - subprogram call → return address stored in the state (register, stack)
    - subprogram return → use of the stored return address
Ambiguity and complexities in the machine instructions (ARM)

- implicit control flow
  - usual subprogram call – **bl label**
    (set PC + 4 in LR used to return)
  - alternative form
    **mov** LR, PC  – set PC + 4 in LR
    **b** label

- obfuscated indirect branch
  - subprogram return – **bx LR** (or **mov** PC, LR)
  - usual indirect branch (from a branch table)
    **ldr** R0, [address]
    **bx** R0
  - maybe optimized form
    **ldr** LR, [address]
    **bx** LR
Identification of loops

- use of dominance
  - $\forall v, w \in V, v \text{ dom } w \iff \forall p$ path from $v$ to $w$, $v \in p$
  - $h \in V$ is header of a loop if $\exists (v, h) \in E \land h \text{ dom } v$
- irreducible loop (“irregular”)
  - with several headers
  - infrequent
  - causes issues with analysis on loops
  - solution:
    - chooses an header
    - duplicate paths from other headers
Execution paths issues

- indirect branches
  - optimized switches → address table
  - function pointer (in C)
  - virtual functions (in C++)
- bounding the iteration number of loops
  - required for WCET
- infeasible paths
  - CFG = superset of executions paths
  - remove semantically infeasible paths

```c
void t1(int (*f)(void)) {
  int i, j, s, k;
  f = 1;
  for(i = 0; i < 100; i++) {
    if(i % 2 == 0) {
      s += f(i);
      for(j = 0; j < i; j++) {
        if(k == 1) {
          g();
          k = 0;
        }
        h(s);
        s <<= 1;
      }
    }
    if(i % 2 == 0) {
      s += f(i);
    }
  }
}
```
A few words about 
Abstract Interpretation [Cousot, 1977]

- concrete domain
  - state $S$: $\text{Var} \rightarrow \text{Int}$
  - initial state: $s_0$
  - execution $\mathcal{E}: \text{Inst} \times S \rightarrow S$

- question Q
  - let $i \in \text{Inst}$, $i$ is infeasible if, for all execution paths of the program, no state exists before $i$.
  - issue: too many executions paths!

- abstract domain
  - state: $S^\#$, sometimes $S^\#: \text{Var} \rightarrow \text{Int}^\#$
  - and execution $\mathcal{E}^\#: \text{Inst} \times S^\# \rightarrow S^\#$ s.t.
  - Q can be answered (yes or no) most of the time
  - no answer for $Q \Rightarrow$ conservative assumption that $i$ is feasible
  - possibly, $|S^\#| \ll |S|$
Example of AI: interval analysis

- concrete domain
  - variable values
  - $S: \text{ID} \rightarrow \mathbb{Z}$
- abstract domain
  - $\text{Int} = (\mathbb{Z} \cup \{-\infty\}) \times (\mathbb{Z} \cup \{+\infty\})$
  - $S#: \text{ID} \rightarrow \text{Int}$
  - $\mathbb{I}#: \text{Inst} \times S# \rightarrow S#$
- example
  - $\mathbb{I}#[x = y + z;] s =$
    let $[l_y, u_y] = s[r_y]$ in
    let $[l_z, u_z] = s[r_z]$ in
    $s[x \rightarrow [l_y + l_y, u_z + u_z]]$
- joining execution paths
  - $J_S: S# \times S# \rightarrow S#$
  - $J_S(s, s') = \{ i \rightarrow J_{\text{int}}(s[i], s'[i]) \}$
  - $J_S([l, u], [l', u']) = [\min(l, l'), \max(u, u')]$

```plaintext
int fact(int n)
{

  n → [0, 100]
  f → [-∞, +∞]
  i → [-∞, +∞]

  n → [0, 100]
  f → [1, 1]
  i → [1, 1]

  while(f <= n)
  {
    f = f * i;
    i = i + 1;
  }

  return f;

  n → [0, 100]
  f → [1, +∞]
  i → [1, 1]

  n → [0, 100]
  f → [1, +∞]
  i → [2, 101]
```
Source Approach

- data flow analysis on the C
  - interval, congruence, polyhedra, etc

- pros
  - source language is richer
  - typing is explicit
  - memory model is explicit
  - programs are smaller

- cons
  - analysis depends on the source language
  - linkage between source information and binary code
  - or specialized compiler
Binary Approach

- several instruction sets in embedded systems (ARM, PowerPC, Sparc, TriCore, etc)
  \[\Rightarrow\] translation to independent language (Alf, OTAWA's semantic instructions)
- loosely typing of machine instructions
  - rebuild types of values
  - adapted Int# abstraction (CLP analysis)
- calculation of addresses (array, linked structures)
  - \( S#: (\text{Reg} \cup \text{Addr}) \rightarrow \text{Int}# \)
  - imprecise address \( \top \rightarrow \text{loss of memory content} \)
  \[\Rightarrow\] separation of memory areas (stack, heap, global, etc)
Outline

● What's the problem with WCET?
● IPET Approach
● Control Flow Problems
● Hardware Support
● Time Production
● Conclusion
Typical hardware

pipeline

cache L1

cache L2

bus

memory

FE
DE
EX
ME
CM

instruction cache L1

data cache L1

unified cache L2

SPM / SRAM

flash / ROM

DRAM

I/O
Supporting variability

- basically, 1 cycle / pipeline stage
- variability $\Rightarrow$ corresponding stage time increase
- questions?
  - how much time (in cycles) increase?
  - how many times it happens?
- two main solutions
  - static analysis $\Rightarrow$ category (mainly used)
  - transition graph
    - vertices = basic block $\times$ hardware state
    - edge = edge $\times$ hardware transitions
    - modelled in ILP as CFG + constraints linking vertices with basic blocks
Example: instruction cache

- variability in FE
  - hit (in the cache) → 1 cycle
  - miss (out of the cache) → memory access time
  - $x_v^{miss}$ – number of misses for instruction in block $v$
    $0 \leq x_v^{miss} \leq x_v$

- categories to qualify behaviour
  - Always Hit (AH) – instruction always in the cache
    $x_v^{miss} = 0$
  - Always Miss (AM) – instruction never in the cache
    $x_v^{miss} = X_v$
  - Persistent relative to loop $h$ (PE($h$)) – instruction in the cache after first access
    $x_v^{miss} \leq X_h$
  - Not Classified (NC) – behaviour too complex to be modelled
    no constraint on $x_v^{miss}$
Cache Model

- memory split in block of size B
- cache split in $S$ sets containing $A$ blocks each (associativity)
- unique mapping between memory blocks and cache sets
  - sets are independent
  - 1 analysis for each set (reduce the complexity of the analysis)
- replacement policy
  - set full → which block to remove?
  - LRU – Least Recently Used
  - other policies: round-robin, MRU, PLRU, Random

```
memory

$B = 16b$
```

```
set 0
set 1
set 2
set 3
```

```
s = 4,
A = 2
```

```
A B
```

```
access C → miss
```

```
A B
```

```
access B → hit
```

```
A B
```

```
access A → hit
```

```
A B
```

```
B wiped out
```

```
A B
```

```
A B
```

```
A B
```

```
A B
```
Abstract Cache State

- ACS – Abstract Cache State
  - Block – set of memory blocks
  - Age – [0, A] (A = out of cache)
  - ACS = Block → Age
- $U_{LRU}$: Block × ACS → ACS – update function
  - $U(b, a) = a'$ s.t. $a'[b] = 0$ and
    - if not $b$ in cache then increase other block ages
    - if $b$ in cache then increase younger block ages
- $J_{LRU}$: ACS × ACS → ACS
  - $J_{LRU+MUST}$ → max of ages (worse age)
    - $a[b] < A$ → $AH$
  - $J_{LRU+MAY}$ → min of ages (best age)
    - $a[b] = A$ → $AM$
    - NC else

[FERDINAND, APPLYING COMPILER TECHNIQUES TO CACHE BEHAVIOR PREDICTION, 1997]
Abstract Cache State (continued)

**Fix point reached!**
Persistence

Solution: extend ACS*
Age* = \{ ⊥ \} ∪ \{0, A\}
⊥ - not already loaded
\( J_{PERS} = J_{MUST} \) extended to ACS*

In the ILP
\[ x_{α^{miss}} \leq x_{v,h} \]

[Ferdinand, *A fast and efficient cache persistence analysis*, 2005]

Multi-level
\[ ACS^+ = ACS^{*[0..n]} \]
\( n \) loop levels

[Ballabriga, *Improving the First-Miss Computation in Set-Associative Instruction Caches*, 2008]
Cache support in static WCET

- **Instruction cache**
  - with LRU 10-15% NC
  - round-robin
  - PLRU
  - Random – hum!

- **Data cache: address analysis**
  - scalar access → 1 address
  - array access → n addresses
    - several possible states
    - categories are not enough ~50% NC
    - alternative → upper bound of miss count

- **Multi-level cache**
  - CAC (Cache Access Classification) from \( L_i \) to \( L_{i+1} \)
  - Never (N), Always (A), Uncertain (U) → join states accessed / not accessed
    [Hardy, *WCET analysis of multi-level non-inclusive set-associative instruction caches*, 2008]

- **Unified cache**
  - mix instruction and data in the same (L1, but more often L2 or L3)
  - imprecision of data addresses impact the instructions
Other effects

- branch prediction
  - category approach – Always D-predicted, First D-predicted, First Unknown, Always Unknown [Colin, 2000]
  - graph approach [Burguière, 2006]
- Category only
  - DRAM buffer re-use [Ballabriga, 2008]
  - MAM flash prefetch [TRACES, WCET Tool Challenge, 2011]
- about DRAM – refresh cycle
  - internal state and work of DRAM is more and more hidden
- bus / interconnection network usage
  - DMA or multicore
  - current trend of research – predicting the access time, sharing the bus
Outline

● What's the problem with WCET?
● IPET Approach
● Control Flow Problems
● Hardware Support
● Time Production
● Conclusion
How to compute the block time?

- work of pipeline
  - instructions enter according to the program order
  - instructions go forward as soon as required resources are available (buffer slot, operand value, stage, functional unit, memory unit, etc.)
- lots of ISA → even much more micro-architecture models
  - generic system to represent instruction execution
  - split in step
    - stage
    - resource requirements (register, memory)
    - time passed in the stage [Herbegue, 2014]
Execution graph approach

I1: ldrhu R1, [R11, #-2]

I2: mov R1, R1, LSL #16

I3: mov R1, R1, ASR #16

I4: cmp R1, #0

I5: bge 0x40001214

[Rochange, A Context-Parameterized Model for Static Analysis of Execution Times, 2009]
Block execution overlapping

- block overlapping
  - \( t_w = D_{WB/I3} - D_{FE/I1} \)
  - \( t_{v,w} = D_{WB/I'4} - D_{WB/I3} \)
  - \( t_w > t_{v,w} \)

- cost for block → cost for edge
  - \( C = \max \sum_{v,w \in E} t_{v,w} x_{v,w} \)
  - reduce overestimation
  - support for branch prediction on edge
  - for \( v \), consider worst case → longer sequences of blocks depending on the size

\[ t_{v,w} = WB/I'4 - WB/I3 \]
Taking into account events

- current WCET formula
  - $C = \max \sum_{(v,w) \in E} t_{v,w} x_{v,w}$
- for an edge $(v, w)$
  - several time variations → time variation on execution graph node or edge (event)
  - $E_{v,w} = \{ e_i \}$ – set of events with variation (+ $n_i$ cycles → $x_i$ over-estimation)
  - $C_{v,w} = \{ c_j \}$ – set of configurations s.t. $e_i$ enabled, disabled → $c_j[e_i] = \{ \text{true, false} \}$ → $2^{|E_{v,w}|}$ configurations
  - $c_j$ applied to exegraph → new execution time $t_{v,w}^j$
- new WCET formula
  - $C = \max \sum_{v,w \in E} \sum_{c_j \in C_{v,w}} t_{v,w}^j x_{v,w}^j$
  - with constraints
    - $\forall (v,w) \in E, \forall e_i \in E_{v,w}, \sum_{c_j \in C_{v,w}} c_j[e_i] = \text{true} \quad x_{v,w}^j \leq x_i$
    - too many variables, constraints!
Events for instruction cache

- category of instruction $I$ for sequence $(v, w)$
  - AH $\rightarrow$ no time variation $\rightarrow$ no event
  - AM $\rightarrow$ FE + memory access time $\rightarrow$ no event
  - NC $\rightarrow$ FE incremented or not $\rightarrow$ event bounded by $x_{v,w}$
  - $\text{PE}(h) \rightarrow$ FE increment or not $\rightarrow$ event bounded by $\sum_{(u,h) \in E} x_{u,h}$

- several events in 1 sequence
  - $B = 16$
  - $v$ from 0x1014 to 0x1028 $\rightarrow$ 2 cache accesses
    (0x1010 – AH, 0x1020 – NC)
  - $w$ from 0x1028 to 0x1044 $\rightarrow$ 3 cache accesses
    (0x1020 – AH, 0x1030 – PE(h), 0x1040 – PE(h))
  - $E_{v,w} = \{ 0x1020 – NC, 0x1030 – PE(h), 0x1040 – PE(h) \}$
  - number of times $= 8$
- ..., data cache access, branch prediction, flash prefetching, ...
Reducing the complexity

- Naive solution – taking the max
  - \( C = \max \sum_{(v,w) \in E} T_{v,w} x_{v,w} \)
  - \( \forall (v, w) \in E, T_{v,w} = \max_{c_j \in c_{v,w}} t_{v,w}^j \)

- Binary approach
  - lots of \( t_{v,w}^j \) have the same value ← pipeline latency smoothing mechanism (buffers), overlap of effects
  - low time (performant hardware work) → frequent
  - high time (cache miss, misprediction, etc) → less frequent → overestimation has little effect

- New ILP formulat
  - \( C_{v,w} = \text{LTS} \cup \text{HTS} \) (Low Time Set – High Time Set)
  - \( C = \max \sum_{(v,w) \in E} t_{v,w}^{\text{LTS}} x_{v,w}^{\text{LTS}} + t_{v,w}^{\text{HTS}} x_{v,w}^{\text{HTS}} \)
  - \( t_{v,w}^{\text{LTS}} = \max_{c_j \in \text{LTS}} t_{v,w}^j, t_{v,w}^{\text{HTS}} = \max_{c_j \in \text{HTS}} t_{v,w}^j \)
  - \( t_{v,w}^{\text{LTS}} \ll t_{v,w}^{\text{HTS}} \)
  - ... \( x_i \) changed according to \( x_{v,w}^{\text{LTS}} \) and \( x_{v,w}^{\text{HTS}} \)

- current research → testing other solutions
Outline

● What's the problem with WCET?
● IPET Approach
● Control Flow Problems
● Hardware Support
● Time Production
● Conclusion
Conclusion

- **IPET approach for WCET computation by static analysis**
  - flexible framework based on ILP
  - path analysis
  - acceleration mechanisms analysis
  - block time analysis
- **Limitations**
  - indirect control flow (branch tables, pointers)
  - analysis of infeasible paths
  - acceleration mechanism analysis → cache: best precision with LRU, may require ad-hoc analysis
  - block time analysis → ILP resolution complexity problem
  - size of the program → size of ILP → resolution time
Opened domains

- support of complex applications
  - parametric WCET
  - adaptive WCET analysis driven by precision
  - closer integration of events in the block time
- support of complex hardware
  - DRAM
  - pseudo-round robin caches
  - improved support for PLRU, Round-Robin, MRU
  - automatic integration of new hardware
- support of multi-execution
  - multi/many-core sharing of bus/interconnection
  - more precise pre-emptive multi-thread/interrupt analysis
- extension
  - architecture – predictable and efficient design
  - compiler – WCET-aware optimizations
  - generator – WCET oriented task generation and mapping
Any question?

http://www.otawa.fr