Foundations of Cyber-Physical Systems

WCET Analysis: High-Level Overview

Rupak Majumdar & Björn Brandenburg


© 2014 B. Brandenburg (MPI-SWS)
Worst-Case Execution Time Analysis (WCET)

How to determine $e_i$?

WCET analysis: given a hardware platform and the implementation of a task, for at most how long will a single job execute (in isolation)?

Schedulability analysis: given multiple tasks and the WCET for each task, is it possible to host them on the same hardware platform?

Ambiguous terminology: “timing analysis” can refer to either or both types of analyses.
Execution Time Histogram\textsuperscript{Wil08}

Fig. 1. Basic notions concerning timing analysis of systems. The lower curve represents a subset of measured executions. Its minimum and maximum are the minimal observed execution times and maximal observed execution times, resp. The darker curve, an envelope of the former, represents the times of all executions. Its minimum and maximum are the best-case and worst-case execution times, resp., abbreviate BCET and WCET.

The literature on timing analysis has created a confusion by not always making a distinction between worst-case execution times and estimates for them. We will avoid this misnomer in this survey. We will use the term timing analysis for the process of deriving execution-time bounds or estimates. A tool that derives bounds or estimates for the execution times of application tasks is called a timing-analysis tool. We will concentrate on the determination of upper bounds or estimates of the WCET unless otherwise stated. All tools described in Section 6 with the exception of SymTA/Solver timing analysis of tasks in uninterrupted execution. Here, a task may be a unit of scheduling by an operating system, a subroutine, or some other software unit. This unit is mostly available as a fully-linked executable. Some tools, however, assume the availability of source code and of a compiler supporting subsequence timing analysis.

Organization of the article
Section 2 introduces the problem and its subproblems and describes methods being used to solve it. Sections 3 and 4 present two categories of approaches, static and measurement-based. Section 6 consists of detailed tool descriptions. Section 7 resumes the state of the art and the deployment and use in industry. Section 8 lists limitations of the described tools. Section 9 gives a condensed overview of the tools in a tabulated form. Section 10 explains, how timing analysis is or should be integrated in the development process. Section 11 concludes the paper by presenting open problems and the perspectives of the domain mainly determined by architectural trends.

\textsuperscript{Wil08} R. Wilhelm et al. (2008). The Worst-Case Execution Time Problem — Overview of Methods and Survey of Tools.

© 2014 B. Brandenburg (MPI-SWS)
Terminology

WCET = maximum ever observed (on target platform)

BCET = minimum ever observed

ACET = average, dependent on input, BCET \leq ACET \leq WCET

It’s important to distinguish between *bounds* (or *estimates*) and the *actual* WCET/BCET.

→ **Safety**: bound \(\geq\) actual.

→ **Tightness**: bound close close to actual value.
WCET Analysis Challenges

Two issues must be considered:
→ **software behavior** (control flow)
→ **hardware timing** (basic block bounds).

**Processor caches, out-of-order** processor pipelines, **speculative execution**
→ move the ACET closer to the BCET (and may even reduce the BCET)
→ typically make the WCET worse
→ increase the span between ACET and WCET.

*Caches and speculation make the precise timing more dependent on the execution history, which is difficult to predict precisely.*
Typical Software Restrictions

• no recursion
• no unbounded loops
• no function pointers / virtual method dispatch
• no/restricted pointer aliasing
• no dynamic linking
• no dynamic memory management
Memory and Pipeline Pitfalls

Even without caches, **memory access times** are also highly variable.
- memory bus contention and DMA
- contention at the memory controller
- DRAM controllers are stateful!
- row buffer misses/hits can have a significant effect.

**Long term effects** in processor pipelines
- non-adjacent instructions can interact (= change timing)
- In principle, such long term effects can occur **across arbitrary distances**.
- Can be “hidden” with more pessimistic WCET bounds (i.e., safe margin).
Prefetching and Branch Prediction

- **Memory prefetching** can evict useful cache contents.

- **Branch prediction** is another source of headaches.
  → **Dynamic** branch prediction: based on the *execution history*.

- **Instruction prefetching**: keep the processor pipeline filled.

  “There are examples of cases where executing more iterations of an inner loop takes less time than iterating fewer iterations, due to branch prediction effects in a loop which is visited several times.” EE08

---

Timing Anomalies

• Common in modern processors.

• Exist if a **local worst-case event** (e.g., a cache miss, a branch mis-prediction, a row-buffer miss) does not result in **global worst case**.

• Example: cache miss in a FIFO cache that results in a more favorable cache state for the subsequent code.

→ *Timing anomalies prevent local (“greedy”) analysis.*
Execution Time Analysis Approaches

1. Whole-program measurements:
   - “just” measure execution for “representative” input data after setting up “worst-case initial hardware state”

2. Measurement of basic blocks + control-flow analysis

3. Static WCET analysis

© 2014 B. Brandenburg (MPI-SWS)
Measurement-based Approaches

- difficult to find the worst case
- virtually impossible to guarantee that it has been found without static analysis
- as timing variability increases with processor complexity, coverage becomes more difficult
- even if worst case path / scenario is known, recreating it can be difficult (branch predictor state, cache state, etc.)
How to measure?

• **Probe effects** make measurements with cycle counters / other forms of instrumentation *potentially inaccurate.*
  → Inline probes perturb cache, branch prediction, code layout, ...

• **Cycle-accurate** system simulators or hardware traces are better, but *much slower* to use and/or difficult to set up.

• In practice, few simulators are guaranteed to be accurate. *(Validating simulators is difficult.)*
Static WCET Analysis
Analysis Phases

1. Flow analysis phase

2. Low-level analysis phase
   → Basic block timing analysis
   → Address analysis
   → Cache analysis

3. Calculation phase
   → arrive at overall bound
   → Infeasible path removal

© 2014 B. Brandenburg (MPI-SWS)
(1) Flow Analysis Phase

- Extract **control flow graph** (CFG) from binary (or source code).
- Discover path constraints and input dependencies.
- Techniques: symbolic execution & abstract interpretation.
- Discover loop bounds.
  → Often provided as annotations by developer.
- Compiler optimizations vs. control-flow extraction...
(2) Low-Level Analysis

• How long does each basic block take to execute?
  → *The hardest part of WCET analysis.*

• Need an *accurate* model of the processor and memory hierarchy.
  → cycles per instruction
  → processor pipeline, #execution units, reordering, ...
  → cache sizes, replacement policies, etc.
  → ISA peculiarities: delay slots, serializing instructions, etc.

• Even the processor manufacturer may not have all required information. (divide & conquer engineering → *emergent behavior*)
Low-Level Analysis must be Context-Aware

• Need to consider code in different *contexts*.
  → First iteration of a loop has different cache behavior.
  → Last iteration has different branch prediction behavior.

• **Virtual unrolling** of loops: pull out special cases.
  → When and how much are “tunable parameters”.

• Function behavior also depends on **call site**.
  → Consider function in contexts specific to call sites.
  → “Virtual inlining”...
(3) Final Calculation Phase

**Goal:** combine basic block costs to arrive at overall bound.

Three common methods:

1. Tree-based: annotate timing in the *abstract syntax tree* (AST).
3. IPET: *implicit path enumeration*  
   → used by commercial tools (e.g., aiT)
Tree- and Path-Based Analysis

• The **tree-based** approach works on the AST of the program.
  → For each node, a WCET is computed based on its children.
  → Example: $\max()$ for conditionals (IF statements).
  → AST leaves = basic blocks, for which the WCET is known.

• In a **path-based** analysis, (feasible) execution paths are explicitly represented.
  → Found via symbolic execution or abstract interpretation.
  → Challenge: there can be an exponential number of feasible paths.
Implicit Path Enumeration (IPET)

- **IPET**: each basic block and each edge between basic blocks is represented by an integer variable that represents the number of times that the control flow traverses the vertex.

- **Linear integer constraints** encode the structure of the CFG.
  - basic block entered = basic block exited
  - if edge taken = basic if block entered
  - many other conservation of flow invariants...

- **Objective function**: maximize sum of basic block costs
  - coefficients = basic block WCETs
  - solve with off-the-shelf ILP or CSP solvers

- With IPET, the worst-case path is no longer explicitly represented in the analysis, but it can be reconstructed from the solution of the ILP.
Caches
Review: Basic Cache Parameters

- **Block size**: operate on cache lines, not individual addresses

- **Direct-mapped vs. fully associative**
  - direct-mapped: each memory location maps to a specific cache location
  - fully associative: each memory location can be stored anywhere in the cache
  - $n$-way associative: each memory location can be stored at a $n$ cache locations

- From WCET analysis point of view, an $n$-way associative cache operates like $n$ independent fully associative caches.

- **Cache Size**: bigger cache sizes better for predicting cache hits, but worse for predicting precise cache contents.
Review: Common Replacement Policies

- **Least-recently used** (LRU): best for real-time systems.

- **First in, first out** (FIFO): cheap, but difficult to analyze.
  → Also referred to as *round-robin* (RR).

- **Most-recently used** (MRU): does not evict block most recently accessed.
  → Status bit set to 1 on access; reset all when last bit is set.
  → Evict lowest-indexed cache block with zero bit.

- **Pseudo LRU** (PLRU): cheaper approximation of LRU...
In cache of size $k$, $k - 1$ bits indicate eviction path: 0 — left, 1 — right. On access, point all bits away from accessed location.

---

Memory Analyses

**Address analysis:** which addresses does an instruction reference?

**Cache analysis:** is the accessed address going to be cached?

- Address analysis and cache analysis must be carried out on the **final compiled binary**. Memory layout must be known.

- Somewhat simpler for instruction fetches. Hard for data accesses and especially for input-dependent accesses.
Must vs. May Analysis

- Address analysis over-approximates set of referenced addresses → **uncertainty**.

- **Must** and **may** information: must $P$ is an invariant ($P$ always holds), whereas as may $P$ means that “not $P$” cannot be proven.

- **Must** cache abstraction: addresses known to be cached at a given program point.
  → Used to predict **cache hits**.

- **May** cache abstraction: addressed that cannot be shown to not be cached.
  → Complement used to predict **cache misses**.
Must and May Analyses

- Can be formalized as **abstract interpretation**.

- Intuitively, at control-flow join points:
  - → **intersection** of *must* states
  - → **union** of *may* states.

- Larger *must* cache: more accurate cache hit prediction, lower WCET estimate.
  - → longer element survival across branches

- Smaller *may* cache: more accurate cache miss prediction, more precise BCET estimate.
  - → fewer elements introduced at join
Observation: Must initially assume that cache contents are unknown.
→ Assuming empty caches is not always conservative.

Question: how quickly does a cache converge to known state?
→ How many references need to be observed?

Two metrics:

• **fill**: how many accesses until cache state completely known?

• **evict**: how many accesses does it take until we can be sure that a particular item has been replaced?

---

LRU Optimality

• For an LRU cache of size $k$:

$$fill = evict = k$$

• This is optimal: no replacement policy or analysis can learn information about cache states with fewer references.

• Reineke provides a detailed characterization of the other policies.\(^\text{Re08}\)

Making Caches More Predictable

• In software: **cache coloring** (exploit set-associativity)
  → allocate memory to avoid or limit conflict misses

• In hardware: **cache locking**
  → prevent (configurable) cache lines from being evicted

• Software-controlled caches: **scratchpad memories**
  → Fully software-controlled cache.
  → Non-trivial management task for the OS...
Downsides of Software-Controlled Caches

- **Cache locking effectively decreases cache size** for all code that doesn’t access the locked cache lines.

- It can be quite difficult to determine which lines to lock.

- Scratchpads pose significant management overheads. → Paging contents in and out is not free.

- **Caches** are widespread and work well precisely they are **not** programmer-controlled...
WCET Analysis — Pragmatic Conclusions

• Static timing analysis is hard.

• Proper measurements are hard.

• \textit{Perfect} analysis only possible in ivory-tower lab conditions.

• Pragmatism: schedulability analysis with estimated/measured parameters is better than none at all.

\rightarrow \textit{Don't let the (impossibility of the) perfect be the enemy of the good.}