Performance enablers

- Micro-architecture and ISA
  - Pipelining, ILP, caching, superscalar out-of-order execution, VLIW, Multicores, GPUs etc. (ADA - A. Seznec)
- Compiler
  - Loop unrolling, splitting, fusion, inlining, etc.
- Optimize frequent, average-case scenarios
Performance predictability

- General purpose systems
  - The “average” user wants **stable** performance:
    - Cannot tolerate variations of performance by orders of magnitude when varying simple parameters
- Safety-critical/real-time systems: need for **guaranteed** performance
  - Worst-case performance (WCET - Worst-Case Execution Times)

Outline

- Performance predictability
  - What is it? For which applications?
- WCET estimation methods
  - Static WCET estimation methods
  - Measurement-based and hybrid WCET estimation methods
  - Beyond WCET estimation: schedulability analysis
  - WCET estimation for parallel codes
- Programming and compiling for worst-case performance
Performance predictability

Influencing elements

- Sequencing of actions (execution paths)
  - Input data dependent
  - A priori unknown input data
- Duration of every action on a given processor
  - Hardware dependent
  - Depends on past execution history
Performance

Distribution of execution times

\[ P(\text{et}=t) \]

min \quad mean \quad worst

Execution time t

Performance

Application needs

- Desktop applications:
  - As-fast, and as stable as possible
  - Typical input data (average-case)
  - Performance evaluation: execution/simulation with typical input data

- Real-time applications:
  - Before a deadline (real-time is not real-fast)
    - Control applications: stability of physical process
    - Delay until system failure
  - For all input data (including the worst-case)
  - Performance evaluation: the subject of this lecture
Classes of real-time systems

- **Hard real-time**
  - Missing a deadline can cause catastrophic consequences in the systems: need for a priori guarantees in the worst-case scenario
  - Ex: control in transportation systems, nuclear applications, etc.

- **Soft real-time**
  - Missing a deadline decrease the quality of service of the system but does not jeopardize its safety
  - Ex: multimedia applications (VoD, etc.)

Validation of real-time systems

What is needed to demonstrate that deadlines are met?

- **System-level validation**
  - Worst-case knowledge of system load (task arrivals, interrupt arrivals)
  - Ex: synchronous arrival of periodic tasks

- **Task-level validation**
  - Worst-case execution time of individual tasks
  - WCET (Worst-Case Execution Time)
    - Upper bound for executing an isolated piece of code
    - Code considered in isolation
    - WCET ≠ response time
Different uses of WCET

- Temporal validation
  - Schedulability analysis
  - Schedulability guarantees (worst-case)
- System dimensioning
  - Hardware selection
- Optimization of application code
  - Early in application design lifecycle

WCET: Definition

- Challenges in WCET estimation
  - Safety (WCET estimate ≥ any possible execution time): confidence in schedulability analysis methods
  - Tightness
    - Overestimation ⇒ schedulability test may fail, or too much resources might be used
- WCET ≠ WCET estimate
Static timing estimation methods

- Overview
- Flow analysis
- WCET computation
- Hardware-level analysis

Static WCET estimation methods

- Principle
  - Analysis of program structure (no execution)
  - Computation of WCET from program structure
- Components
  - Flow analysis
    - Determines possible flows in program
  - Low-level (hardware-level) analysis
    - Determines the execution time of a sequence of instructions (basic block) on a given hardware
  - Computation
    - Computation from results of other components
    - All paths need to be considered: safety
Overview of components

- Source code
- Compiler
- Object code
- Flow analysis
- (Annotations)
- Flow representation
- Low-level analysis
- Computation
- WCET

WCET computation

Assumptions

- Simple architecture
  - Execution time of an instruction only depends on instruction type and operand
  - No overlap between instructions, no memory hierarchy
Tree-based computation

- Data structures
  - Syntax tree (control structures)
  - Basic block

- Principle
  - Determination of execution time of basic block (low-level analysis)
  - Computation based on a bottom-up traversal of the syntax tree (timing schema)

Timing schema

\[
\begin{align*}
\text{WCET(SEQ)} & = W(1) + \ldots + W(n) + WCET(\text{Loop}) + WCET(\text{Seq}2) \\
\text{WCET(IF)} & = \text{if(test) then else WCET(test) + max(WCET(then), WCET(else))} \\
\text{WCET(LOOP)} & = \text{for(;tst;inc) (body) maxiter * (WCET(tst)+WCET(body+inc)) + WCET(test)}
\end{align*}
\]
Tree-based computation

- Advantages
  - Low computational effort
  - Good scalability with respect to program size
  - Good user feedback (source-code level)

- Drawbacks
  - Not compatible with aggressive compiler optimizations
  - Expression of complex flow facts difficult (inter-control-structure flow facts)

IPET (Implicit Path Enumeration Technique)

- Integer linear programming
  - Objective function: max: \( \sum_{i=1}^{n} f_i t_i \)
  - Structural constraints
    \[
    \forall v: f_i = \sum_{a_i \in \text{In}(v)} a_i = \sum_{a_i \in \text{Out}(v)} a_i \\
    f_i = 1
    \]
  - Extra flow information
    - \( f_i \leq k \) (loop bound)
    - \( f_i + f_j \leq 1 \) (mutually exclusive paths)
    - \( f_i \leq 2 f_j \) (relations between execution freqs.)
IPET

- Advantages
  - Supports all unstructured flows (gotos, etc.)
  - Supports all compiler optimizations, including the most aggressive ones

- Drawbacks
  - More time-consuming than tree-based methods
  - Low-level user feedbacks (works at binary level)
  - Annotations are hard to provide (need to know compiler optimizations)

- Mostly used in commercial/academic tools

Low-level analysis

Introduction

- Simple architecture
  - Execution time of an instruction only depends on instruction type and operand
  - No overlap between instructions, no memory hierarchy

- Complex architecture
  - Local timing effects
    - Overlap between instructions (pipelines)
  - Global timing effects
    - Caches (data, instructions), branch predictors
    - Requires a knowledge of the entire code
Low-level analysis

Introduction

- Outline
  - Pipelines, caches, branch predictors,… considered in isolation
  - Interferences between analyses
- Restrictions
  - Simple in-order pipelines first

Pipelining

- Principle: parallelism between instructions
  - Intra basic-block
  - Inter basic-block
Low-level analysis

Pipelining

- Intra basic block: "Simulation" of the flow on instructions into the pipeline stages
  - Takes in consideration data hazards (RAR, RAW, WAW)
  - Bypass mechanism (if any) has to be modeled
  - Obtained by WCET analysis tool or external tool (simulator, measures on the processor)

Fetch
Decode
Execute
Memory
Write Back

\[ t_i = 7 \]

\[ t_j = 7 \]

\[ \delta_{i,j} = -4 \]

\[ \text{max: } \sum_{i} f_{i} t_{i} + \sum_{i,j} a_{i,j} \delta_{i,j} \]
Low-level analysis
Out-of-order execution [Li&Mitra, 06]

Superscalar processor model

- Instr. cache
- fetch buffer
- ROB
- data cache
- register files

In-order fetch and dispatch
Out-of-order execution
In-order commit

Plain arrows: dependencies
- Pipeline stages of same instructions, buffer size, data dependencies among instructions, in-order fetch and ID

Dashed arrows: contentions (shared resources)
Low-level analysis

Out-of-order execution: timing analysis

- Assumptions (simplified version)
  - Fixed-latency functional units

- Principle
  - Applied at every basic block
  - Scheduling problem: find the longest path in an acyclic graph with precedence and exclusion constraints
  - Algorithm (simplified)
    - Traverse the graph in topological order (finish time of all predecessors known)
    - Conservative estimation of contention time
    - See paper for details!

Low-level analysis

Instruction caches

- Cache
  - Takes benefit of temporal and spatial locality of references
  - **Speculative**: future behaviour depends on past behaviour
  - Good average-case performance, but predictability issues
Low-level analysis
Instruction caches

How to obtain safe and tight estimates?
- Simulation along one path not an option
  - Explores only one path, may not be the longest one
- Simple solution (all miss): overly pessimistic
  - And in some cases may be wrong (timing anomalies - see end of chapter)

Objective
- Predict if an instruction will (certainly) cause a hit or might (conservatively) cause miss.
Low-level analysis
Instruction caches

- Based on abstract interpretation [Ferdinand, 2004]
- Computation of Abstract Cache States (ACS)
  - Contains all possible cache contents considering “all” possible execution paths
  - Fixpoint computation
- Instruction categorization from ACS
  - Always hit (AH) - guaranteed to be a hit
  - Always miss (AM) - guaranteed to be a miss
  - First miss (FM) - guaranteed to be a hit after first referenced
  - Not-classified (NC) - don’t know

Low-level analysis
Instruction caches

- Analyses
  - Must: ACS contain all program lines guaranteed to be in the cache at a given point
  - May: ACS contain all program lines that may be in the cache at a given point
  - Persistence: ACS contains all program lines not evicted after loaded
- Modification of ACS
  - Update: at every reference
  - Join: at every path merging point
- Following example: for LRU caches
Low-level analysis
Instruction caches - Must analysis

ACS contain all program lines guaranteed to be in the cache at a given point

Join
Intersection + max age

Update
Apply replacement policy

Low-level analysis
Instruction caches - May analysis

ACS contain all program lines that may be in the cache at a given point

Join
Union + min age

Update
Apply replacement policy
Low-level analysis

**Instruction caches**

- From ACS to classification
  - If in ACS_must: Always Hit
  - Else, if not in ACS_May: Always Miss
  - Else, if ACS_Persistence: First Miss
  - Else Not Classified

- From classification to WCET (IPET)
  - For every BB: t_first, t_next
  - Objective function: \( \max \sum (f_{first_i} + f_{next_i}) \)
  - New constraints:
    - \( f_i = f_{first_i} + f_{next_i} \)
    - Constraints based on cache classification

---

Low-level analysis

**Data caches**

- Extra issue: determination of addresses of data
- Means: abstract interpretation / DF analysis
  - Value analysis
  - Pointer analysis

- Results:
  - Superset of referenced addresses per instruction

- Example:
  - for (int i=0;i<N;i++) tab[i+j/z] = x;
  - Any address inside tab may be referenced per loop iteration (but only one is actually referenced)
  - Hard-to-predict reference (input-dependent)
Low-level analysis
Data caches

- Solutions (with some limitations)
  - Compiler-controlled: don’t cache input-dependent data structures
  - Assume every potentially referenced address is actually referenced (2*N misses in example)
  - Cache Miss Equations (CME): for affine data-independent loop indices

- Tighter estimation method (pigeon hole principle) – re-work this one
  - Unpredictable vs unpredictable accesses (obtained using static analysis)
  - Replacement of other blocks
    - Cache miss counter Cm
    - Incremented only for blocks used after the current instruction (info. obtained using static analysis)
  - Cache miss if element not in the cache
    - Comparison of execution count and array size
    - Persistence analysis of arrays
    - For persistent arrays fitting the cache, number of misses $\leq$ array_range
Low-level analysis
Data caches

- Cache miss equations
  - Unpredictable vs unpredictable accesses (obtained using static analysis)
  - Replacement of other blocks
    - Cache miss counter Cm
    - Incremented only for blocks used after the current instruction (info. obtained using static analysis)
  - Cache miss if element not in the cache
    - Comparison of execution count and array size
    - Persistence analysis of arrays
    - For persistent arrays fitting the cache, number of misses $\leq$ array_range

Low-level analysis
Branch prediction

- Branch predictors
  - Local predictors [Colin, 00]
  - Global predictors [Mitra], [Rochange]
  - Most complex predictors out of reach!
The bad news …
Timing anomalies (out-of-order execution)

<table>
<thead>
<tr>
<th>Disp. Cycle</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>LD r4,0(r3)</td>
</tr>
<tr>
<td>B</td>
<td>ADD r5, r4, r4</td>
</tr>
<tr>
<td>C</td>
<td>ADD r11, r10, r10</td>
</tr>
<tr>
<td>D</td>
<td>MUL r11, r11, r11</td>
</tr>
<tr>
<td>E</td>
<td>MUL r13, r12, r12</td>
</tr>
</tbody>
</table>

The bad news …
Timing anomalies

- Cache miss is not necessarily the worst case
- Pipeline analysis and cache analysis are no longer independent :-(
  - When a miss/hit cannot be predicted, exploration of both outcomes: explosion of pipeline analysis time
  - [Li & Mitra 06]: more complex scheduling problem
The bad news …

Timing anomalies

- Empty cache is not necessarily the worst-case
- Example: PLRU (4-ways)
  - 0 (resp. 1): left (resp. right) subtree is to be replaced next
  - Update/hit: bits on the path flipped

The bad news …

Timing anomalies (credits: Absint)

Empty cache
Non-empty cache

Sequence: c, d, f, c, d, h

This sequence is then repeated ad infinitum
- only cache hits
  - two misses each time
The bad news …

Timing anomalies

- Empty cache phenomenon and replacement policies: LRU and FIFO OK
- Latest development: predictability metrics [Reineke, 2008]
  - Evict: lower bound on eviction time (after evict distinct accesses, guaranteed to be evicted)
    - Used in May analysis
  - Fill: after fill distinct accesses, all possible cache contents are known
    - Used in Must analysis

Flow analysis

- Structurally feasible paths (infinite)
- Basic finiteness (bounded loops)
- Actually feasible (infeasible paths, mutually exclusive paths)
Flow analysis

- Infeasible paths
  
  int baz (int x) {
    if (x<5) // A
      x = x+1; // B
    else x=x*2; // C
    if (x>10) // D
      x = sqrt(x); // E
    return x; // F
  }

- Path ABDEF is infeasible
- Identification of infeasible paths: improves tightness

Flow analysis

- Maximum number of iterations of loops

  \[
  \text{for } i := 1 \text{ to } N \text{ do} \\
  \text{for } j := 1 \text{ to } i \text{ do} \\
  \text{begin} \\
  \quad \text{if } c1 \text{ then } A\.long \\
  \quad \text{else } B\.short \\
  \quad \text{if } c2 \text{ then } C\.short \\
  \quad \text{else } D\.long \\
  \text{end}
  \]

  Loop bound: \(N\)
  Loop bound: \(N\)
  \(\frac{(N+1)N}{2}\) executions

- Tight estimation of loop bounds: improves tightness
- On this specific example, polyhedral techniques help (see later)
Flow analysis

- Determination of flow facts
  - Automatic (static analysis): infeasible in general (equivalent to the halting problem)
  - Manual: annotations
    - Loop bounds: constants, or symbolic annotations
    - Minimum required path information
    - Annotations for infeasible / mutually exclusive paths, relations between execution counts

Flow analysis

- Annotations in aiT
  - Simple example
    - `asm("label:`); (in source code)
    - `ais2 { flow sum: point("label") == cnt; }
    - `ais2 { flow sum: point("label") <= cnt; }
  - More generally, language with documented syntax for
    - Bounds on recursion
    - Loop bound specification (min, max, exact)
    - Linear constraints
    - Maximum execution times for loops and code snippets, address ranges, etc …
  - All this at binary level
Flow analysis

Flow analysis using abstract interpretation

- Abstract domain: intervals of values
- Scopes: « repeated » blocks (loops/functions)
- Fixpoint calculation:
  - Update:
    - At every instruction (source/intermediate/binary)
    - Conditional constructs: two paths may be followed
  - Join:
    - Safe merging of intervals to avoid state explosion
    - Here: end of scope iteration

Example:

```plaintext
i=INPUT; // i=[1..4]  
while (i<10) {  
    // point p  
    ...  
    i=i+2;  
    ...  
}  
// point q
```

End of iter3: [5..8] $\rightarrow$ $i=i+2$ [7..10] $\rightarrow$ 2 abstract states
Flow analysis

Flow analysis using abstract interpretation

- Loop bound detection
  - Loop counter:
    - Incremented at every scope iteration
    - Updated at every scope exit (3,4,5) on the example
- Infeasible node: node counter
  - NB: Nodes with >0 counters might be infeasible
- Upper bounds for node executions
- Infeasible paths
  - For the same scope iteration
  - Matrix of pairs
- IPET: new linear constraints

---

### Some numbers

<table>
<thead>
<tr>
<th>Pgm</th>
<th>Time</th>
<th>WCETorig</th>
<th>Time</th>
<th>WCETff</th>
<th>#FF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crc</td>
<td>4.9</td>
<td>834159</td>
<td>6.65</td>
<td>833730</td>
<td>56</td>
</tr>
<tr>
<td>Insort</td>
<td>0.16</td>
<td>31163</td>
<td>0.17</td>
<td>18167</td>
<td>7</td>
</tr>
<tr>
<td>Ns</td>
<td>6.09</td>
<td>130733</td>
<td>6.81</td>
<td>130733</td>
<td>8</td>
</tr>
<tr>
<td>Nsichneu</td>
<td>36.88</td>
<td>119707</td>
<td>435.70</td>
<td>41303</td>
<td>65280</td>
</tr>
</tbody>
</table>
Measurement-based and hybrid methods

- Principles
- Safe measurement-based methods
- Less-safe measurement-based method

WCET estimation methods

Dynamic methods

- Principle
  - Input data
  - Execution (hardware, simulator)
  - Timing measurement
- Generation of input data
  - User-defined: reserved to experts
  - Exhaustive
    - Risk of combinatorial explosion
WCET estimation methods

Safe dynamic method

- **Principles**
  - Structural testing
  - “All-paths” coverage criterion
  - Avoid enumerating all input-data combination

- **Assumptions**
  - A1. One feasible path in source code gives rise to one path in binary code
  - A2. Execution time of a feasible path is the same for all input values which cause the execution of the path
  - A3. Existence of a worst-case machine state

---

Incremental coverage of input domain

- **Diagram**
  - t₁
  - t₂
  - t₃
  - t₁

---
WCET estimation methods
Safe dynamic method

- Advantages
  - No hardware model
  - All paths are covered

- Drawbacks
  - Assumption A2 restricts the architecture to be used
  - Still an explosion of the number of paths to be explored for some programs

- Ongoing work: measuring fewer execution paths
WCET estimation methods

Less safe dynamic methods

- Use of genetic algorithms
- Data structures
  - Population, individuals, genes (sets of input data)
- Operators
  - Cross-over = mix two genes (sets of input data)
  - Mutation = change one gene (input data)
  - Selection = individuals with largest execution time (obtained through measurements)
- Benefits
  - Measurement-based: no need for hardware models
  - No safety guarantee
  - Used to validate static analysis methods, or when safety is not mandatory

Hybrid method

- High-level analysis and WCET combination
  - Static analysis
- Execution times of basic blocks
  - Measurements
  - Distributions of execution time
- WCET computation (tree-based)
  - Operators defined on distributions instead of single values
    - Independent basic blocks: convolution
    - Non-independent basic blocks: biased convolution
  - Result: probabilistic WCET
- Evaluation
  - Benefits: no need for hardware model, path analysis is not probabilistic
  - Safety is not guaranteed regarding hardware effects
WCET estimation methods

Hybrid method: symbolic simulation

- Based on a cycle-accurate simulator
- Values and registers tagged with a “type” (known/unknown)
- Extension of the semantics of instructions
  - Memory transfer from an unknown register content: target address tagged as “unknown”
  - Conditional branch on unknown condition: exploration of both branches
  - For every instruction …
- State merging to avoid exploring all paths
  - State = (memory/registers)
  - Merging = pessimistic union of the two states
  - Merging points: different granularities
- Benefits: flow-analysis “for-free”
- Drawbacks: hardware models, terribly slow

Open issues (low level analysis)

- Complexity of static WCET estimation tools
- **Timing anomalies**, integration of sub-analyses
- Analysis tools may be released a long time after the hardware is available
- Trends:
  - Probabilistic low-level analysis
  - Software control of hardware (cache locking, compiler-directed branch prediction)
  - Multi-core architectures (still a long way to go, …)
Static WCET estimation methods
A method for every usage

- Static WCET estimation
  - Safety 😊
  - Pessimism 😊
  - Need for a hardware model 😊
  - Trade-off between estimation time and tightness (tree-based / IPET) 😊

- Measurement-based methods
  - Safety 😅 Probabilistic methods
  - Pessimism 😊
  - No need for hardware models 😊

WCET estimation tools

- Academic
  - Cinderella [Princeton]
  - Heptane [IRISA, Rennes]
  - Otawa [IRIT Toulouse]
  - SWEET [Sweden]
  - Chronos [Singapour]

- Industrial
  - Bound-T [SSF]
  - AIT [AbsInt, Sarbrücken]
  - Rapitime [Rapita systems, York]
Some pointers

- **Bibliography**
  - Survey paper in TECS, 2008
  - Workshop on worst-case execution time analysis (2001..2013), in conjunction with ECRTS

- **Working group**
  - Timing Analysis on Code-Level (TACLE)
    - http://www.tacle.eu

Beyond predictable performance

Schedulability analysis
Temporal validation of real-time systems

- **Testing**
  - Input data + execution (target architecture, simulator)
  - Necessary but not sufficient (coverage of scenarios)
- **Schedulability analysis (models)**
  - Hard real-time: need for guarantees in all execution scenarios, including the worst-case scenario
  - Task models
  - Schedulability analysis methods (70s → today)

---

Schedulability analysis

**Introduction (1/2)**

- **Definition**
  - Algorithms or mathematical formulas allowing to prove that deadlines will be met
- **Classification**
  - Off-line validation
    - Target = hard real-time systems
  - On-line validation
    - Acceptance tests executed at the arrival of new tasks
    - Some tasks may be rejected → target = soft real-time systems
Schedulability analysis

Introduction (2/2)

- **Input:** system model
  - Task model
    - Arrival: periodic, sporadic, aperiodic
    - Inter-task synchronization: precedence constraints, resource sharing
    - Worst-case execution time (WCET)
  - Architecture
  - Known off-line for hard real-time systems

- **Output**
  - Schedulability verdict

Schedulability analysis

Example (1/2)

- **System model**
  - Periodic tasks \( (P_i) \), deadline \( D_i \leq P_i \)
  - Fixed priorities (the smaller \( D_i \) the higher priority)
  - Worst-case execution time: \( C_i \)

- **Necessary condition**
  \[
  U = \sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1
  \]

- **Sufficient condition**
  \[
  \sum_{i=1}^{n} \frac{C_i}{D_i} \leq n(2^{\frac{1}{n}}-1)
  \]

- **Low complexity**
Schedulability analysis

Example (2/2)

- Same task model as before
- Estimation of response time: limit of series
  \[ w_k^i = C_i \]
  \[ w_k^{i+1} = C_i + \sum_{j=0}^{\infty} \left[ \frac{w_k^i}{P_j} \right] C_j \]
- The series limit is the task response time (when converges)
- The system is schedulable when \( R_i \leq D_i \)
- More complex schedulability test

Schedulability analysis

Networked systems

- Response time analysis on every processor
- Response time analysis of network transmissions (e.g. CAN: fixed-priority network)
- End-to-end response time analysis
  - Holistic approaches
WCET estimation for parallel codes

- Issues
- Preliminary studies

Hardware architectures
Examples of multi-core systems

- Shared memory + bus + local cache/SPM
- Shared memory + bus + shared last-level cache
- Clustered architectures (Kalray) with NoC
Hardware architectures

Issues with multi-core systems

- Shared hardware resources (to reduce costs)
  - Bus
  - NoC
  - Last Level Cache, memory controller
- Introduces dependencies between cores
  - Interference delays
  - Depends on
    - Arbitration policy
    - Contending activity on the other cores
- Remarks: In some safety critical systems, disable all cores but one

Hardware architectures

Software executing on multi-cores

- Independent tasks
  - Have to consider worst-case contention
- Dependent tasks
  - Dependencies between tasks
  - Examples: Directed Acyclic Graphs (DAGs), Message Sequence Charts (MSCs)

```
Synchronization
Control Flow
---
Start/Fork
```

A → B → C → D → E → F → G
W0 → W1 → S0 → S1

79

80
Shared Last Level Cache (LLC)

- Issues
  - A technique to identify worst-case number of misses in LLC
    - Analysis of cache hierarchies
    - Analysis of shared LLC
  - Using synchronizations to tighten
    - Avoid pressure in shared cache
    - Account for synchronizations

Shared Last Level Cache

- Issues arising from cache sharing

![Diagram showing cache sharing and synchronization issues](image)
Analysis of cache hierarchies

- Hierarchical analysis of caches
- Compute accesses occurrences on cache level L
  - Accessing level L not necessarily the worst-case!
- In the following examples:
  - Non inclusive caches
  - LRU replacement policy
  - Set-associative caches

---

Analysis of cache hierarchies

[RTSS 08]

- Cache access classification at level L
  - Never: never occurs at level L
  - Always: always occur at level L
  - Uncertain: unknown
- Modification of cache analysis
  - Update function
    - Never: not considered
    - Always: considered
    - Unknown: local exploration: Join (Never, Always)
Analysis of shared caches

- Estimate conflicts from rival tasks
- Integrate conflicts into cache classification (Hit/Miss)

References

Access
Classification

Cache analysis
Shared Level L

Analysed task

Conflicts count

Conflict Estimation

Cache analysis
Level L

Rival tasks

Analysis of shared caches: conflict estimation

- Any task, any time may alter shared caches
  - Compute all interleavings: too costly in practice
Analysis of shared caches: conflict estimation

- Any task, any time may alter shared caches
  - Compute all interleavings: too costly in practice
- To reduce this cost, we abstract from a task:
  - Access ordering
  - Occurrence count
Analysis of shared caches: conflict estimation

- Any task, any time may alter shared caches
  - Compute all interleavings: too costly in practice
- To reduce this cost, we abstract from a task:
  - Access ordering
  - Occurrence count
- For each cache set
  - Find memory blocks used by the task.
  - Count the total number of blocks (CCN)

---

Analysis of shared caches: conflict integration

- A cache analysis is performed using modified cache states:
  - CCN cache blocks may have been allocated to rival tasks in the shared cache.
  - Only (Cache L Associativity – CCN) ways are for sure available.

Example:

- Set 0
- Set 1
- Base cache configuration
- Available cache space during analysis

**Cache line occupied by a conflicting block**

Set 0

Set 1

CCN₀ = 2

CCN₁ = 7

Set 0

Set 1
Analysis of shared caches

- Conflict Estimation is pessimistic:
  - Many blocks may clutter up the cache.
  - Every block a task may store on a shared cache level is included.
- Little cache space detected as available during shared cache analyses.

Need to control the impact of tasks on shared caches.

- Bypass of shared cache blocks [RTSS 09, IRISA]
- Accounting for synchronization [RTSS 09, NUS]

Analysis of shared caches: bypass

- If an instruction bypasses a cache level, it does not alter this cache level.

Example:
Analysis of shared caches: bypass

- Reduce tasks’ impact on caches.
  - A bypassing access produces no conflict
- Selection of accesses to bypass
  - Static analysis
  - Accesses not statically detected as reused (single-usage)
  - Never worsens tasks WCET 😊
- Extended to data caches in [RTNS 10]

Analysis of shared caches: accounting for synchronization

- Synchronisation points define zones of conflicting accesses

Without synchronisation

With synchronisation
WCET estimation in multi core architectures: shared busses

- Issues
- Estimation of contention delays for round-robin arbitration
  - Pessimistic contention analysis for RR
  - Accounting for synchronizations

Delay to serve request r1
- Depends on pending requests from the other cores
- Depends on bus arbitration policy
WCET estimation in multi core architectures: shared RR bus

- Round-robin arbitration
  - A time units per core, round-robin
  - In case a core has no pending request, serve request for the text core in round-robin ordering

- Worst-case delay:
  - For a request from core $i$: $WCD_{RR} = (N_c-1)*A$, with $N_c$ the total number of cores
  - For $N_i$ requests from core $i$, $WCD_{RR} = (N_c-1)*A*N_i$

Remarks
- Assumes all concurrent cores always have a pending request, which is a safe over-approximation but introduces pessimism
- Let $N_i$ denote the maximum number of memory accesses performed by a concurrent core $C_i$
- For $N_i$ requests from core $i$
  - Assumes cores are ordered by increasing $N_i$
  - $WCD_{RR}(C_i) = \sum_{j=1}^{s_i-1} N_j * A + \sum_{j=s_i}^{s_q} N_i * A$ cycles

Cores with less than $N_i$ accesses each access delays by $A$

Cores with more than $N_i$ accesses, delay of $N_i*A$
Interfering Memory Accesses Delay

- Graphically

\[ WCD_{RR} (C_i) = \sum_{j=1}^{i-1} A \sum_{j=i}^{i+q} N_i \times A \text{ cycles} \]

WCET estimation in multi core architectures: shared RR bus

- Estimating number of concurrent memory accesses
  - Knowledge of concurrent code snippets
  - May Happen in Parallel relation (MHP)
  - Classical graph algorithms (transitive closure) for graphs without cycles
  - More complex for cyclic structures
WCET estimation in multi core architectures: shared RR bus

<table>
<thead>
<tr>
<th>BB</th>
<th>MHP</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>{E,S1}</td>
</tr>
<tr>
<td>B</td>
<td>{E,S1}</td>
</tr>
<tr>
<td>S0</td>
<td>{E,S1}</td>
</tr>
<tr>
<td>W1</td>
<td>{W0,F,G}</td>
</tr>
<tr>
<td>C</td>
<td>{W0,F,G}</td>
</tr>
<tr>
<td>D</td>
<td>{W0,F,G}</td>
</tr>
<tr>
<td>E</td>
<td>{A,B,S0}</td>
</tr>
<tr>
<td>S1</td>
<td>{A,B,S0}</td>
</tr>
<tr>
<td>W0</td>
<td>{W1,C,D}</td>
</tr>
<tr>
<td>G</td>
<td>{W1,C,D}</td>
</tr>
</tbody>
</table>

Start/Fork

Synchronization
Control Flow

Transitive Closure
Synchronization
Control Flow