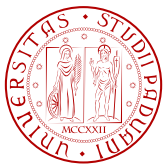
Cache-aware Development of High-Integrity Systems*

E. Mezzetti[†], A. Betts[§], J. Ruiz[‡], T. Vardanega[†]

[†]University of Padua (I), [§]Rapita Systems (UK), [‡]AdaCore (F)



Valencia, June, 15$^{\text{th}}$, 2010

*Work performed with ESA/ESTEC support, under the COLA Project (ESTEC/Contract 22167/09/NL/jk)

Table of contents

High Integrity Real-time Systems

- High requirements on verification and validation (V&V)
  - **time**, space and communication dimensions (above functional)
- Execution of system activities within a least upper bound
  - Sound and early information on the timing behavior
  - Schedulability analysis techniques
    - Preferably on an **architectural model** of the system
    - Worst-Case Execution Time (**WCET**)
- Most conservative domain
  - Especially in **aerospace**
  - Avoid any changes, unless mission-critical
    - w.r.t. both hardware and software technologies
- Driven by ever-increasing user demands
  - Advanced functionalities ▶ more computational power
  - Pushes toward the adoption of more complex processors
    - Advanced features (**caches**, complex pipelines, etc.)

Cache Predictabilty

- Execution-time variation in presence of caches
  - Factors of influence
    - Execution history, memory layout and task interactions
  - Difficult to hit into their worst-case combination
    - Especially in a scenario-based measurement approach
  - Timing behavior depends on **context** (hardware and software)
- Cache-aware timing analysis and industrial-level tools
  - Static WCET analysis (**aiT** form AbsInt)
  - Hybrid WCET analysis (**RapiTime** from Rapita Systems)
    - Combining static analysis with measurements
  - Cache-aware schedulability analysis
    - Accounting for Cache-Related Preemption Delay (CRPD)

Timing Analysis in Industrial Practice

- Still rely on software simulation and testing
  - Early WCET figures drawn from past experiences
  - *Safety margins*
  - WCET bounds consolidated by testing
  - **Unsafe** in the presence of caches
- Existing tools and techniques not acknowledged yet
- Main motivations
  - Overall complexity of analysis (and tools)
  - Sometimes exceeding overestimation (static analysis)
    - May lead to over-dimensioning of a system
  - Late applicability
    - On the final executable ▶ too late in the development process!

Cache-awareness

- The domain seeks guarantees on the timing behavior
  - From design to implementation
    - Even in the presence of caches
  - Development should be **aware** of the cache impact
- Caches accounted for in the **early stages** of development
  - To control cache variability factors
  - To ease system analyzability
  - To be able to predict the system timing behavior earlier
    - Final analysis should only confirm our expectations
- Involved dimensions
  - Improvement of cache predictability at code level
  - Control of the cache behavior
  - Integration in the industrial development process

Cache-aware Coding

- Code Patterns and Coding Styles
  - Affect both cache **performance** and **analysability**
    - Reduce timing variability
    - Avoid sources of overestimation that hamper cache analysis
  - More easily enforced through **automatic** code-generation
  - The role of compilers cannot be disregarded
    - Mapping source code to machine code
    - Several complex optimisation passes
- Software architectures
  - Set the overall structure of the system
    - Memory layout, execution paths, etc.
  - Determine pattern of tasks interleaving and interactions
    - Cache interference between tasks
  - Each SW architecture ▶ differing cache behavior
    - Some architectural choices may reduce cache variability
    - E.g. resource access protocols
    - Cache-awareness as a factor of choice between architectures

Cache-aware Memory Layout

- **Conflict misses** in instruction cache
  - Can be reduced or avoided through code placement
  - Restraining variability from **layout** (and **concurrency**)
  - Advantages include:
    - Better WCET behavior
    - Reduced execution-time variability
    - Guarantees on the worst impact of cache misses
- Linker process is cache-oblivious
  - Places sub-programs in consecutive memory locations according to order found in object files
  - Cache behavior may change because of
    - Sub-programs ordering
    - Increase in sub-program size
    - More sub-programs added
- Compute cache-aware layout and force it on the linker

Effectiveness of a Cache-aware Layout

- Two Implemented Strategies
  - Genetic Algorithm (onerous)
  - Structural-based Algorithm
    - Exploits knowledge of program structure *(call graph, execution frequency)*
- Experimental evaluation
  - Software representative of part of the Attitude and Orbit Control System (AOCS)
  - Instruction cache simulator
    - 32 KB, 32 B lines, LRU, 4-ways set-associative

Layout	Hits	Misses
Worst layout	526,444	55,932
Best layout	582,115	261

  - Worst Layout ▶ sub-procedures mapping to the same cache set
  - Best Layout ▶ structural layout

Run-time support for cache management

- Goals
  - Design of cache-aware application
  - User in control of cache behavior in tasking application
    - Forbid the usage of the cache to some tasks (or parts of it)
    - Activities polluting the cache and not taking advantage of it, e.g.:Memory scrubbing, parity checks, etc.
- Per-task cache control
  - Enable/disable/freeze/flush cache
    - Independent for instruction/data cache
  - Kept in the task control block
    - Stored/restored during context switches
  - Interrupt may automatically freeze cache
    - Handler automatically re-enable cache after frozen-on-interrupt
  - Global operations also possible
    - Changing the cache behavior for all tasks
  - At the cost of few extra assembly instructions in context switches and interrupt handlers

How to handle the cache at run time

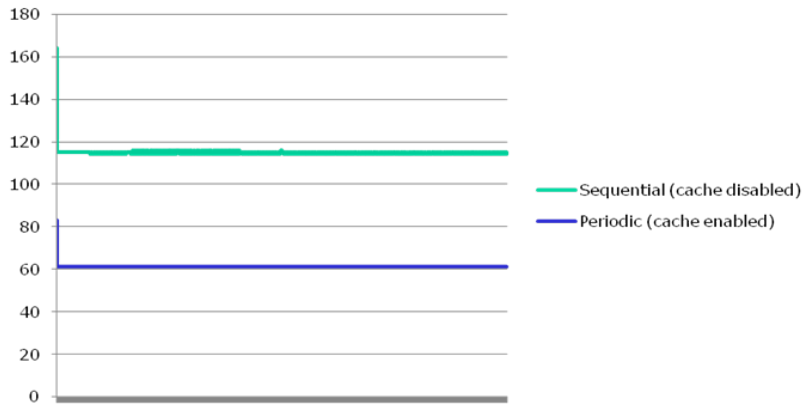
```
task body Use_Cache is
begin
   Set_Cache_State
      (Cache => Instruction, State => Enabled, Partition_Wide => True);
   Set_Cache_State
      (Cache => Data, State => Enabled, Partition_Wide => True);

   Enable_Cache_Freeze_On_Interrupt
      (Cache => Instruction, Partition_Wide => True);
   Enable_Cache_Freeze_On_Interrupt
      (Cache => Data, Partition_Wide => True);
    ...
end Use_Cache;
```

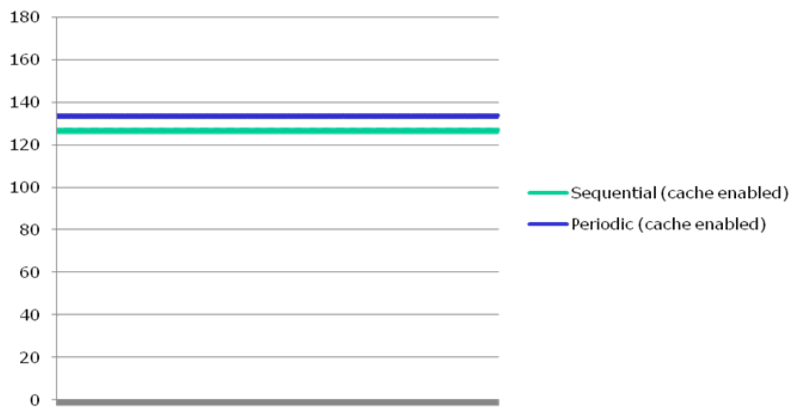
```
task body Optimize_Loops is
begin
   Set_Cache_State (Instruction, Frozen);
    ...
   Set_Cache_State (Instruction, Enabled);
   loop
    ...
   end loop;
   Set_Cache_State (Instruction, Frozen);
    ...
end Optimize_Loops;
```

Effectiveness of cache controlled by the user

Tasking example

- Loop-intensive task benefits from cache
- Sequential task pollutes the cache
  - Long sequential code which does not benefit from cache
- Results
  - Reduced cache interference and hence faster execution





- Cache controlled by the user
  - Better Performance
  - Slightly less variability
- No cache control
  - Decrease cache performance

Integration in the Industrial Development Process

- Timing behavior relevant throughout the whole process
  - At different stages in the development process
    - System and SW design ► system dimensioning, tasks allocation, budgeting, etc.
    - SW Coding ► performance and predictability
    - SW Integration ► final timing behavior
  - At different levels of precision
  - Allow to detect timing problem as early as possible
- Timing behavior determined and analysed at the lowest level
  - Near the machine code
  - Difficult to address it at higher levels of abstraction

WCET Analysis in Early Development Process

- Static Analysis
  - Complex low-level annotations to improve analysis precision
    - No black-box analysis ☢
    - Annotations may need to be changed after a re-compilation ☢
    - Quite onerous and error prone!
  - Does not require the actual hardware ✓
    - A precise and sound model
  - WCET estimates even of uncomplete code
    - Program parts may be temporarily replaced by stubs
- Hybrid Analysis
  - Allows black-box testing ✓
  - Produces not necessarily safe WCET estimates ☢
  - Does require the actual hardware (or simulator)
    - Should not be a problem in the domain
  - All program code must be available ☢

Conclusion

- Cache variability and predictability
  - Hardly accountable for in early development process
- Static analysis ▶ safe but not enough flexible
  - The only solution on the "safe side"
  - Unfit for prototyping and iterative development
- Hybrid analysis ▶ much more agile but unsafe
  - Less demanding approach
  - Can leverage on available test cases
  - WCET estimates must be consolidated
- Partial guarantees on the timing behavior
  - Exclude extremely variable and unpredictable cache behavior
    - Deciding a priori which tasks will benefit from caches
    - Reducing cache variability arising from conflict misses
    - Forcing good code patterns by automatic code generation
- Much work still to be done in this direction...