TECHNISCHE
UNIVERSITÄT
WIEN
Vienna University of Technology

WCET Driven Design Space Exploration of an Object Cache

Benedikt Huber, Wolfgang Puffitsch, Martin Schoeberl

JTRES’10

Computer Architecture Design for Embedded Hard-RT Systems

- Application Area
  - Resource-constrained hard real-time systems
  - Timing needs to be predictable!
  - Target Platform: Java Optimized Processor
- Choosing the right components
  - Wide variety of performance-enhancing techniques
  - Example here: Data caches to bridge CPU/memory gap
  - Which choices are favorable for hard RT?

Cache Design for JOP

- Small processor for Safety-Critical Java
- Designed to allow precise worst-case execution time (WCET) estimations
- Non-trivial dynamic memory behavior
  - Garbage Collector
  - Objects shared between threads
- Data cache promises significant speedup (especially for multiprocessor version)

Conventional Cache Evaluation

- Create different implementations (Simulator/FPGA)
- Measure runtime on set of representative benchmarks
- Rank designs based on
  - Measurement results
  - Implementation cost
- Problem: No quantitative metric for timing predictability

Our Approach

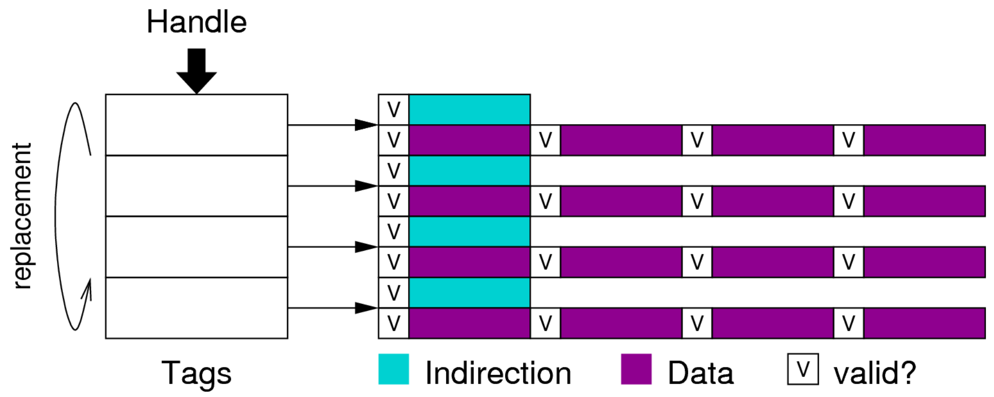
- Use both simulation and static analysis results
- Based on static WCET analysis techniques
  - Program analysis (Dataflow analysis)
  - Worst-case calculation (ILP based)
- Avoid architecture designs without precise timing models
  - Waste of resources for hard RT systems
  - Usually more complex (error prone) static analysis

→ WCET-guided architecture design

Split Cache Architecture

- Distinguish data accesses based on address predictability and coherence issues
  i. Address known statically, immutable data
  ii. Same, but mutable data (cache coherence)
  iii. Heap allocated data (address statically unknown)
- Split data cache for predictability!
  - Direct-mapped / set-associative cache for static data
  - No interference with unknown addresses → precise timing estimation possible
  - Object cache for heap allocated objects

The Object Cache



- Fully-associative cache
  - Keeps track of 16-64 “active” objects
  - Handles (indirections not mutated by GC) as tag
- Object Cache Entries
  - One (longer) cache line per object
  - Word Fill: one valid bit for each field
  - Burst Fill: fill cache line (or parts of it) at once

Data Cache Predictability

- Is it possible to effectively limit the number of cache misses in a program fragment?
- Addresses of heap-allocated objects?
  - Dynamic memory allocation
  - Garbage Collector (changes address)
  - Allocated in a different thread
- Heap-allocated objects + Direct Mapped Cache
  - If the address of accessed datum is unknown, it might evict any other datum from a direct-mapped cache

Object Cache Predictability

- First approximation: Number of possible conflicting accesses in program fragments
- Object Cache with Associativity *N* (FIFO & LRU): No conflict if at most N distinct objects are accessed in a program fragment
- Compare to set-associative cache
  - Assuming address of handle is unknown
  - Worst-case scenario: All objects map to the same cache line in set-associative cache

Object Cache: Static Analysis (1)

- Cache Hit/Miss Classification
  - Standard technique for instruction caches
  - Does not work (well) if addresses are unknown
- Local persistence analysis
  - Restrict number of cache misses in program fragment
  - Requires architecture with composable timing
- Integration into WCET calculation
  - We use Implicit Path Enumeration Technique (IPET)
  - Cache analysis adds inequalities restricting cache cost

Object Cache: Static Analysis (2)

- Persistence Analysis Implementation
  - Run on selected program fragments (bottom-up search)

  i. *Dataflow Analysis*
     Compute symbolic name of accessed objects (relative to scope entry)

  ii. *Max-Cost Network Flow Analysis*
     Compute maximal number K of distinct objects used in the scope

  iii. *IPET Integration*
     If K <= Associativity: IPET inequalities to restrict number of cache misses

WCET-driven Object Cache Evaluation

- Uses our WCET Analysis framework
- Compute cache miss cycles for set of embedded Java Benchmarks
- Assume cold cache, no interference with other components
- Different configurations
  - Different Associativity, Line Size
  - Burst mode (load full line at once)
  - SRAM and SDRAM (latency for first word)

Evaluation: Object Cache Configuration

- Line Size
  - Object sizes vary depending on benchmark
  - 16 words sufficient for all benchmarks
- Associativity
  - Few „active objects“ (2-8) relevant
  - *Realistic*? Benchmarks are all we have
- Burst Mode
  - Line fill (avoids valid bit) does not work well enough
  - Coincides with average case observation
  - Small benefits from 4-word burst (SDRAM)

Evaluation: Hitrate

- Results close to measured average case performance
  - 43%-91% hit rate
  - For some benchmarks, not a lot of locality
  - Analysis also needs improvements
- Revealed a few weaknesses in the analysis
  - Does not take positive effect of aliasing into account
  - Does not use known loop bounds when counting number of distinct objects

Conclusion

- Designers need to take predictability into account
  - Need WCET to verify temporal behavior
  - Unpredictable architecture: gross overestimations → waste of resources
- WCET Analysis techniques for quantitative estimate of „worst-case performance predictability“
- Implementation and analysis for the split cache architecture to be finished