Caches in Real-Time Systems

[Xavier Vera, Bjorn Lisper, Jingling Xue, "Data Caches in Multitasking Hard Real-Time Systems", RTSS 2003.]

- Ignoring cache leads to significant resource under-utilization.
- Q: How to appropriately account for cache?

Instruction Cache vs. Data Cache

- Computation of WCET with instruction cache for non-preemptive systems (e.g. Static Cache Simulation)
- Extension: Computation of WCET with instruction cache in preemptive systems.

- Analysis of Data Cache harder
  - Single instruction can refer to multiple memory locations.
  - Locality of reference harder to capture for data access.
WCET Analysis in the Presence of Data Caches (I)

- Static Analysis
  - Attempts to classify statically the different memory accesses as hits or misses.
  - Typically does not consider preemptive systems
  - Limited to codes free of data-dependent constructs

- Cache Preemption Delay
  - Incorporate cache preemption cost as context switch overhead into schedulability analysis.
  - Cold-started cache after preemption?
    - Might be unsafe on processors with out-of-order instruction scheduling, where a cache hit under some circumstances may be more expensive than a miss.

WCET Analysis in the Presence of Data Caches (I)

- Cache Locking
  - Available on many current microprocessors (e.g. PowerPC 604e, 405 and 440 families, Intel-960, some Intel x86, Motorola MPC7400)
  - Static Locking
    - cache is loaded and locked at system start
  - Dynamic Locking
    - state of the cache is allowed to change during the system execution
- Cache Partitioning
  - Eliminate inter-task conflicts by giving reserved portions of cache to certain tasks
  - May give raise to fragmentation, and translate to a loss of performance
Program Model

- Programs consist of subroutines, calls, arbitrarily nested but well-structured loops, and assignments possibly guided by IF conditionals.
- Extensions possible to unstructured code.
- In this paper, all programs are in C. Thus, all arrays are assumed to be in row major.
- Static analysis possible with additional constraints
  - Calls are non-recursive.
  - Bounds of all loops are known and affine.
  - The IF conditionals are analyzable at compile time.

Cache Model

- Uniprocessor with two-level memory hierarchy
  - virtually-indexed K-way set-associative data cache using LRU replacement
  - main memory.
- K-way set-associative cache
  - Cache set contains K cache lines.
  - Let $C(L)$ be the cache (line) size in bytes. The total number of cache sets is thus $C/(L\times K)$.
  - A cache is called direct-mapped when $K=1$
  - A cache is called fully-associative when $K=C/L$.
- Cache locking
  - Cache locking mechanism allows a single cache line to be locked.
- Pre-fetch / Invalidate
  - Processors can load and invalidate cache lines selectively. (This can be emulated in software.)
- Cache partitioning
  - Implemented either in hardware or software.
  - Partition unit is a cache set.
Approach

- Summary: Need method that allows obtaining an exact and safe WCMPs of tasks for multitasking systems with data caches, so that current schedulability analyses can be applied without modifications.

- Cache partitioning to eliminate inter-tasks conflicts.
  - This allows us to compute the WCMP of each task in isolation.

- Compensate performance loss through use of compiler cache optimizations (such as tiling and padding).

- Use Static Analysis to compute WCMP of a task.
  - transform the program issuing lock/unlock instructions to ensure a tight WCMP estimate at static time.
  - Cache pre-fetching added when necessary to improve performance

Cache Partitioning

- Inter-task interference occurs when cache lines from different tasks conflict in cache, which causes unpredictability.

- Partitioning:
  - Divide the cache into disjoint partitions, which are assigned to tasks in such a way that inter-conflicts are removed.
  - Create $n + 1$ partitions, one for each real-time task and another one which is shared among non-real-time tasks.
  - Each task is only allowed to access its own partition, thus removing inter-task conflicts.

- Tasks with same priority can share the same partition
  - Only preempted by tasks with higher priority, and thus the predictability of cache behavior is not affected. (Therefore, $p$ partitions are sufficient, where $p$ is the number of different priorities).

- Partition-size:
  - Size of the partitions impacts performance.
  - Optimal partitioning depends on the priorities and the reuse patterns of tasks. Equally-sized partitions give significant improvement.
Predictable Cache Behavior

- Unpredictability caused by path merging and data dependent memory access.
- Path Merging:
  - Reduce overhead of analyzing loop constructs with multiple paths inside (data-dependent conditionals, loops with unknown loop bounds).
  - Cache state at the end of the merged path is unknown.
- Data Dependent Memory Access:
  - Indirection arrays (e.g., a[b[i]], where b[i] is not statically known)
  - Variables allocated dynamically (e.g., mallocs) and pointer accesses that cannot be determined statically.
  - Nonlinear array references that are not handled by static analyzer (e.g., a[i*j])
  - Library and operating system calls.
- Solution: Cache locking during unpredictable regions of code.

Cache Locking: Example

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Lock/Unlock Placement</th>
</tr>
</thead>
<tbody>
<tr>
<td>int a[100], b[100]; int c[100], k=0; for (i=0;i&lt;100;i++) a[i]=random(i); for (i=0;i&lt;100;i++) c[i]=b[a[i]]+c[i]; N=random(i)*100; for (i=0;i&lt;N;i++) { if (c[i]&gt;15) k++; c[i]=0; }</td>
<td>int a[100], b[100]; int c[100], k=0; for (i=0;i&lt;100;i++) a[i]=random(i); for (i=0;i&lt;100;i++) { lock(); /<em>Region 1</em>/ c[i]=b[a[i]]+c[i]; unlock(); } N=random(i)*100; lock(); /<em>Region 2</em>/ for (i=0;i&lt;N;i++) { register int temp=(c[i]&gt;15); lock(); /<em>Region 2.1</em>/ if (temp) k++; unlock(); c[i]=0;unlock(); }</td>
</tr>
</tbody>
</table>
Optimizing Lock Placement

- **Rule 1.** Lock/unlock instructions that lock the whole loop body (including the test) are placed outside the loop.
  
  \[
  \text{loop; lock; } S; \text{unlock; endloop} \Rightarrow \text{lock; loop; } S; \text{endloop; unlock}
  \]

- **Rule 2.** Remove nested lock regions.
  
  \[
  \text{lock; lock; } S; \text{unlock; unlock} \Rightarrow \text{lock; } S; \text{unlock}
  \]

- **Rule 3.** Fuse two consecutive locked regions.
  
  \[
  \text{lock; } S_1; \text{unlock; lock; } S_2; \text{unlock} \Rightarrow \text{lock; } S_1; S_2; \text{unlock}
  \]

- **Rule 4**. Move a statement past a lock instruction.
  
  \[
  S_1; \text{lock; } S_2; \text{unlock} \Rightarrow \text{lock; } S_1; S_2; \text{unlock}
  \]

- **Rule 5**. Move an unlock instruction past a statement.
  
  \[
  \text{lock; } S_1; \text{unlock; } S_2 \Rightarrow \text{lock; } S_1; S_2; \text{unlock}
  \]

(*) May affect cache behavior.

Selecting Data to Lock into Cache
Optimizing Lock Placement: Example

```c
int a[100], b[100];
int c[100], k=0;
for (i=0;i<100;i++)
a[i]=random(i);
for (i=0;i<100;i++) {
    lock(); /*Region 1*/
c[i]=b[a[i]]+c[i];
    unlock();
}
N=random(i)*100;
lock(); /*Region 2*/
for (i=0;i<N;i++){
    register int temp=(c[i]>15);
    lock(); /*Region 2.1*/
    if (temp)
        k++;
    unlock();
    c[i]=0;
}
unlock();
```

Lock/Unlock Placement

```c
int a[100], b[100];
int c[100], k=0;
for (i=0;i<100;i++)
a[i]=random(i);
IssueLoads(c);
IssueLoads(b);
lock(); /*Region 1*/
for (i=0;i<100;i++)
    a[i]=random(i);
unlock();
N=random(i)*100;
lock(); /*Region 2*/
for (i=0;i<N;i++){
    register int temp=(c[i]>15);
    lock(); /*Region 2.1*/
    if (temp)
        k++;
    unlock();
    c[i]=0;
}
unlock();
```

Final Version

Overview of System

Figure 4: A framework for worst-case performance computation.
Performance: Effect of Partitioned Cache

Figure 5: Cache partitioning impact: comparison of performance degradation for a system with a partitioned cache and a system without a cache.

Performance: Static vs. Dynamic Locking

Figure 6: Comparison of performance between dynamic locking and static locking when applied to partitioned caches.