Caches in Real-Time Systems


- 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.

Program Model

- Programs consist of
  - subroutines, calls,
  - arbitrarily nested but well-structured loops,
  - 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.
How can Caches help?

- **Cache Locking**
  - Available on many 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.

---

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 (Overview)

- 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.
- Use **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

```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++)
c[i]=b[a[i]]+c[i];
N=random(i)*100;
for (i=0;i<N;i++)
   if (c[i]>15)
      k++;
   c[i]=0;
}
```

```
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();
```
Optimizing Lock Placement

- **Rule 1.** Lock/unlock instructions that lock the whole loop body (including the test) are placed outside the loop.

  \[
  \text{loop; lock; S; unlock; endloop} \rightarrow \\
  \text{lock; loop; S; endloop; unlock}
  \]

- **Rule 2.** Remove nested lock regions.

  \[
  \text{lock; lock; S; unlock; unlock} \rightarrow \text{lock; S; unlock}
  \]

- **Rule 3.** Fuse two consecutive locked regions.

  \[
  \text{lock; S1; unlock; lock; S2; unlock} \rightarrow \text{lock; S1; S2; unlock}
  \]

- **Rule 4**. Move a statement past a lock instruction.

  \[
  \text{S1; lock; S2; unlock} \rightarrow \text{lock; S1; S2; unlock}
  \]

- **Rule 5**. Move an unlock instruction past a statement.

  \[
  \text{lock; S1; unlock; S2} \rightarrow \text{lock; S1; S2; unlock}
  \]

(*) May affect cache behavior.

---

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

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

Figure 4: A framework for worst-case performance computation.

Test Workload

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Workload (bytes)</th>
<th>WCMP (no cache)</th>
<th>Period (Normal)</th>
<th>Period (HP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM</td>
<td>Multiplication of two 100x100 Int matrices</td>
<td>12000</td>
<td>153140000</td>
<td>117800000</td>
<td>102993333</td>
</tr>
<tr>
<td>SRT</td>
<td>Bubblesort of 1000 double array</td>
<td>8000</td>
<td>113925998</td>
<td>159496397</td>
<td>227851996</td>
</tr>
<tr>
<td>FIB</td>
<td>Computation of the 30 first Fibonacci numbers</td>
<td>16</td>
<td>7790</td>
<td>155800000</td>
<td>3895</td>
</tr>
<tr>
<td>FFT</td>
<td>Fast Fourier transformation of 512 complex numbers</td>
<td>8192</td>
<td>1655808</td>
<td>152334336</td>
<td>3311616</td>
</tr>
<tr>
<td>CNT</td>
<td>Counting and sum of values in a 100x100 Int matrix</td>
<td>40000</td>
<td>1140000</td>
<td>570000</td>
<td>285000</td>
</tr>
<tr>
<td>SQRT</td>
<td>Computation of the square root of 1384</td>
<td>16</td>
<td>5360</td>
<td>241200</td>
<td>2680</td>
</tr>
<tr>
<td>ST</td>
<td>Computation of Sum, Mean, Var (1008 doubles)</td>
<td>16000</td>
<td>532000</td>
<td>266000</td>
<td>266000</td>
</tr>
<tr>
<td>NDES</td>
<td>Encryption and decryption of 64 bits</td>
<td>960</td>
<td>220938</td>
<td>331407</td>
<td>110469</td>
</tr>
</tbody>
</table>

Table 1. Benchmarks used.
**Performance: Effect of Partitioned Cache**

![Graphs showing performance impact of partitioned cache](image1)

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**

![Graphs showing static vs. dynamic locking performance](image2)

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

Compares utilization levels.

<table>
<thead>
<tr>
<th>Ways</th>
<th>Large Task Set</th>
<th>Medium Task Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>22KB 16KB</td>
<td>32KB 16KB</td>
</tr>
<tr>
<td>Lock</td>
<td>0.93 0.93</td>
<td>1.19 1.19</td>
</tr>
<tr>
<td>Ours</td>
<td>0.29 0.13</td>
<td>0.81 0.68</td>
</tr>
</tbody>
</table>

Table 2. Performance of static cache locking and our cache analysis.