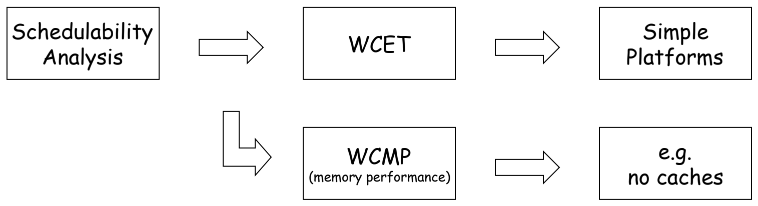
Caches in Real-Time Systems

[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.

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

```
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;
}
```

Data-dependent access

Merging construct

Merging construct

Original Code

```
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

Optimizing Lock Placement

- **Rule 1.** Lock/unlock instructions that lock the whole loop body (including the test) are placed outside the loop.
  `loop;lock;S;unlock;endloop` → `lock;loop;S;endloop;unlock`
- **Rule 2.** Remove nested lock regions.
  `lock;lock;S;unlock;unlock` → `lock;S;unlock`
- **Rule 3.** Fuse two consecutive locked regions.
  `lock;S1;unlock;lock;S2;unlock` → `lock;S1;S2;unlock`
- **Rule 4*.** Move a statement past a lock instruction.
  `S1;lock;S2;unlock` → `lock;S1;S2;unlock`
- **Rule 5*.** Move an unlock instruction past a statement.
  `lock;S1;unlock;S2` → `lock;S1;S2;unlock`

(*) May affect cache behavior.

Optimizing Lock Placement: Example

```
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

```
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++)
    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);
    if (temp)
        k++;
    c[i]=0;
}
unlock();
```

Final Version

Overview of System

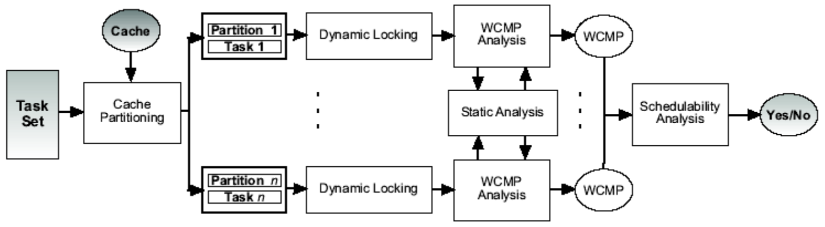


Figure 4: A framework for worst-case performance computation.

Test Workload

Name	Description	Workload (bytes)	WCMP (no cache)	Period (Normal)	Period (HP)
	Large Task Set				
MM	Multiplication of two 100x100 Int matrices	120000	153140000	117800000	102093333
SRT	Bubblesort of 1000 double array	8000	113925998	159496397	227851996
FIB	Computation of the 30 first Fibonacci numbers	16	7790	155800000	3895
FFT	Fast Fourier transformation of 512 complex numbers	8192	1655808	152334336	3311616
	Medium Task Set				
CNT	Counting and sum of values in a 100x100 Int matrix	40000	1140000	570000	285000
SQRT	Computation of the square root of 1384	16	5360	241200	2680
ST	Computation of Sum, Mean, Var (1000 doubles)	16000	532000	266000	266000
NDES	Encryption and decryption of 64 bits	960	220938	331407	110469

Table 1. Benchmarks used.

Performance: Effect of Partitioned Cache

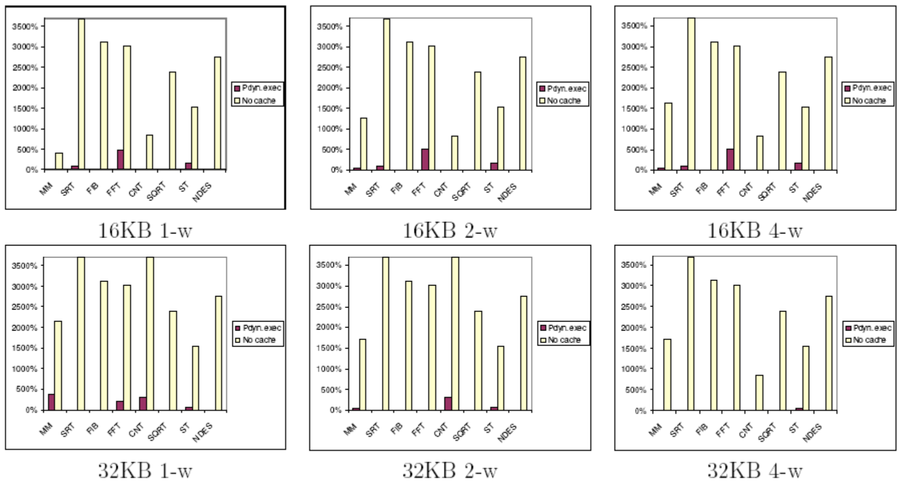


Figure 5: Cache partitioning impact: comparison of performance degradation for a system with a partitioned cache and a system without a cache.

Performance: Static vs. Dynamic Locking

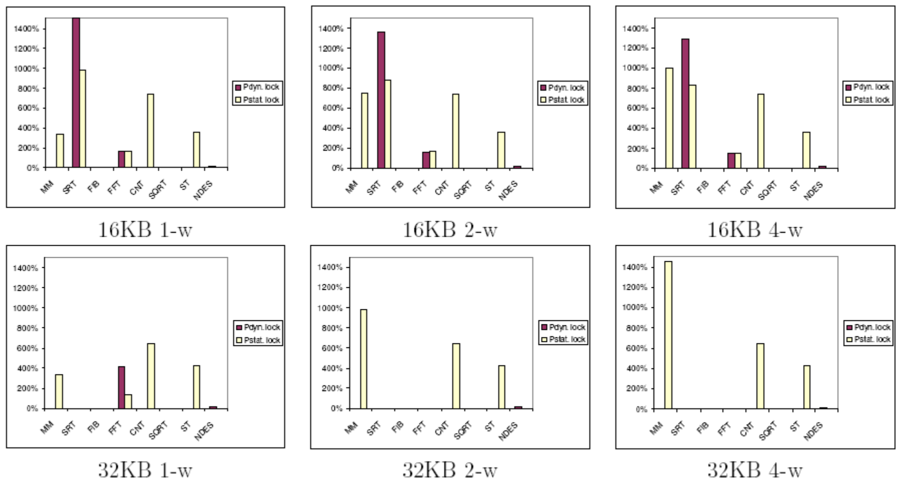
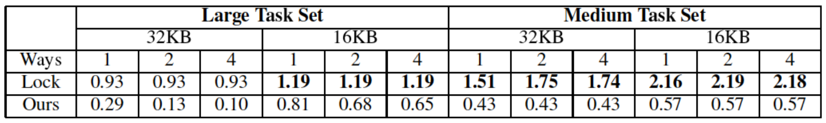


Figure 6: Comparison of performance between dynamic locking and static locking when applied to partitioned caches.

Worst-Case Performance

Compares utilization levels.



	Large Task Set						Medium Task Set					
	32KB			16KB			32KB			16KB		
Ways	1	2	4	1	2	4	1	2	4	1	2	4
Lock	0.93	0.93	0.93	**1.19**	**1.19**	**1.19**	**1.51**	**1.75**	**1.74**	**2.16**	**2.19**	**2.18**
Ours	0.29	0.13	0.10	0.81	0.68	0.65	0.43	0.43	0.43	0.57	0.57	0.57

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