Data Caches in Multitasking Hard Real-Time Systems

Xavier Vera*†, Björn Lisper
Institutionen för Datavetenskap
Mälardalens Högskola
Västerås, 721 23, Sweden
{xavier.vera,bjorn.lisper}@mdh.se

Jingling Xue †
School of Computer Science and Engineering
University of New South Wales
Sydney, NSW 2052, Australia
jxue@cse.unsw.edu.au

Abstract

Data caches are essential in modern processors, bridging the widening gap between main memory and processor speeds. However, they yield very complex performance models, which makes it hard to bound execution times tightly.

This paper contributes a new technique to obtain predictability in preemptive multitasking systems in the presence of data caches. We explore the use of cache partitioning, dynamic cache locking and static cache analysis to provide worst-case performance estimates in a safe and tight way. Cache partitioning divides the cache among tasks to eliminate inter-task cache interferences. We combine static cache analysis and cache locking mechanisms to ensure that all intra-task conflicts, and consequently, memory access times, are exactly predictable. To minimize the performance degradation due to cache partitioning and locking, two strategies are employed. First, the cache is loaded with data likely to be accessed so that their cache utilization is maximized. Second, compiler optimizations such as tiling and padding are applied in order to reduce cache replacement misses.

Experimental results show that this scheme is fully predictable, without compromising the performance of the transformed programs. Our method outperforms static cache locking for all analyzed task sets under various cache architectures, with a CPU utilization reduction ranging between 3.8 and 20.0 times for a high performance system.

1. Introduction

The speed of processors increases faster than the speed of memories. Cache memories are used to bridge this widening gap, and have become the dominant constraint to achieve high processor utilization.

Schedulability analysis of hard real-time systems relies on the assumption that tasks' worst-case execution times (WCETs) are known. Current WCET platforms are applied to rather simple architectures (they usually do not consider caches) and make simplifying assumptions such that the tasks are not preempted. In order to consider the costs of task preemption, some studies incorporate into the schedulability analysis the costs of manipulating queues, processing interrupts and performing task switching [5, 6, 19, 21].

In order to get an accurate WCET, a tight worst-case memory performance (`WCMP`) is needed. However, caches introduce behaviors that are hard to predict. This leads to an unpredictable `WCMP`, and thus an unpredictable WCET as well, which jeopardizes the safety of the controlled system. For that reason, many safety-critical systems either do not use caches or disable them. Nevertheless, a system with disabled caches will waste a lot of resources; the CPU will be underutilized, and also the power consumption will be larger since memory accesses that fall into the cache consume less power than accesses to main memory. Thus, bounding memory performance tightly in hard real-time systems with caches is important to use the system resources well.

The computation of WCET in the presence of instruction caches has progressed in such a way that it is now possible to obtain an accurate estimate of the WCET for non-preemptive systems [1, 2, 17]. These results can be generalized to preemptive systems [3, 7, 8, 9, 23, 25, 34, 35]. However, there has not been much progress with the presence of data caches. Instructions such as loads and stores may access multiple memory locations (such as those that implement array or pointer accesses), which makes the attempt to classify memory accesses as hits/misses very hard.

We summarize below the approaches that can be used for analyzing WCET in the presence of data caches for multitasking hard real-time systems.

1. **Static Cache Analyses**. They attempt to classify stat-

---

* Supported by VR grant no. 2001–2575.

† Supported in part by Australian Research Council Grant A10007149.

ically the different memory accesses as hits or misses. However, the best static cache analyses do not consider preemptive systems and are limited to codes free of data-dependent constructs. In addition, only results for direct-mapped caches have been reported [22, 27, 29, 45].

2. **Cache-Preemption Delays**. When a task resumes its execution, it may spend a long time reloading the cache with previously loaded cache blocks. Some studies have addressed the issue of incorporating cache preemption costs into the schedulability analysis [3, 7, 25]. However, preemption changes the cache contents in an unpredictable manner. Thus, a cache-sensitive analysis of a task assumed to run in isolation might be invalid in a context where the task is preempted: the worst-case execution path may not be the same anymore since hits may be turned into misses and vice versa. Adding a penalty by assuming the cache is cold-started 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 [31]. Moreover, this method resorts to a static cache analysis to obtain the WCET.
3. **Cache Locking**. The ability to lock cache contents is available on several commercial processors (PowerPC 604e [33], 405 and 440 families [18], Intel-960, some Intel x86, Motorola MPC7400 and others). Each processor implements cache locking in several ways, allowing in all cases *static locking* (the cache is loaded and locked at system start) and *dynamic locking* (the state of the cache is allowed to change during the system execution). Provided that the cache contents are known, the time required for a memory access is predictable. Cache locking can be applied to each task in isolation or at system startup [35].
4. **Cache Partitioning**. These techniques [8, 23, 28, 34] give reserved portions of the cache to certain tasks to guarantee that data will be in cache despite preemptions, thus eliminating inter-task conflicts. The reduction of the cache size that each task uses may, however, translate to a loss of performance.

1.1. An Overview

If a memory access is not classified definitely as a hit or miss, a subsequent pipeline analysis in a WCET analysis would have to consider both situations to detect the worst-case path. Thus, we want to guarantee an exact prediction of hits or misses for all memory accesses.

This paper combines cache partitioning, dynamic cache locking and static cache analysis for preemptive multitasking real-time systems with data caches. Cache partitioning allows us to eliminate inter-task conflicts, thus we can analyze each task in isolation. There are typically parts of the code which are statically analyzable and where each access can correctly be categorized as a cache hit or miss. For the other parts, with data-dependent accesses (such as indirection arrays) or where multiple paths can be taken, we lock the cache by inserting lock/unlock instructions in the code. Partitioning the cache reduces the cache size each task now uses. Hence, we apply compiler optimizations such as tiling [10, 24, 46] and padding [36, 37] to reduce the number of misses. Moreover, before locking the cache we also load it with data likely to be accessed, thus minimizing the possible loss of performance due to locking. Finally, we run the static analysis to estimate the `WCMP` of each task assuming it has a portion of the cache, and the schedulability test is performed.

This work makes several significant contributions. Our previous work describes how applying cache locking dynamically can enhance predictability [42], but it was only applied to non-preemptive systems. The new framework presented in this paper applies to preemptive multitasking systems, thereby bounding `WCMP` for data caches in such systems. We have also introduced a set of transformations to optimize the placement of lock/unlock instructions. In addition, we have written a new algorithm that applies compiler optimizations in concert with cache partitioning and cache locking. Our approach yields safe and exact `WCMP`s; it avoids overestimations from assumed cold-starts of the data cache and eliminates possible safety issues, while having a low CPU utilization. We have implemented our system in the SUIF2 [32] compiler. Each task is compiled independently, and only knowledge of the allocated cache partition is required. We do not rely on any specific hardware component, thus its applicability is not limited to a particular architecture.

The rest of the paper is organized as follows. Section 2 introduces the program model, cache architecture and task model used in our approach. We review the static analysis in Section 3. Section 4 discusses the transformations to have a statically predictable data cache for multitasking systems. Section 5 presents our experimental framework, and Section 6 discusses our results. Section 7 reviews some related works. Finally, we conclude and give a road map to future extensions in Section 8.

2. Terminology

2.1. Program Model

We consider programs consisting of subroutines, calls, arbitrarily nested but well-structured loops, and assignments possibly guided by IF conditionals. Our method can be extended to unstructured code, but for simplicity we stay with this program model.

In this paper, all programs are written in `C`. Thus, all arrays are assumed to be in row major order. The following restrictions define the scope of programs where we can apply our static analyzer [42] without any transformation, that is, those programs where only one path is analyzed:

- Calls are non-recursive.
- Bounds of all loops are known and affine.
- The IF conditionals are analyzable at compile time.

In order to ensure that the static analysis [15, 41, 42] can be done in the polyhedral model [12], we add the following constraint:

- The subscript expressions of array references are affine.

We rely on the compiler to identify compile-time and run-time constants. We use standard compile-time techniques such as constant propagation to detect more constants, which allows more expressions to be analyzed statically. To address the symbolic loop bound problem, we use interprocedural constant propagation to eliminate as many symbolic loop bounds as possible. This may also be helpful to know statically which recursive calls are made, thus enlarging the scope where the static analysis is applied.

Otherwise, when multiple paths are possible, we apply path merging in order to reduce the number of paths being analyzed. We assume that the maximum number of iterations of a loop and the maximum number of recursive calls are known. This can be done by either manual annotations [11] or automatic approaches [16].

2.2. Cache Model

We consider a uniprocessor with a two-level memory hierarchy consisting of a virtually-indexed K-way set-associative data cache using LRU replacement policy followed by main memory. Note that this cache architecture is assumed in all the existing analytical methods reviewed in Section 7.

In a K-way set-associative cache, each cache set contains K cache lines. Let $C_s(L_s)$ be the cache (line) size in bytes. The total number of cache sets is thus $C_s/(L_s \times K)$. A cache is called direct-mapped when K=1, and fully-associative when K=C_s/L_s.

In order to use cache locking, we assume that there exists a locking mechanism that allows a cache line to be locked. Such mechanism is available in several modern processors such as PowerPC 604e [33], 405 and 440 families [18], Intel-960, some Intel x86 and Motorola MPC7400. Besides, we assume that the processors offer the ability to load and invalidate cache lines selectively. Otherwise, both of them can be "simulated" on software at a cost of some performance loss. Our approach assumes that cache partitioning is implemented either by a hardware or software means. The partition unit is a cache set.

2.3. Task Model and Schedulability Analysis

We consider a set of N periodic tasks T_i, $1 \leq i \leq N$. We denote the period and worst-case execution time of task T_i by P_i and C_i, respectively.

We consider two schedulability analyses for periodic tasks, UA (utilization-based analysis) and RTA (response time analysis). For dynamic priority preemptively scheduled systems (e.g., *earliest deadline first*), the utilization condition $U \leq 1$ is necessary and sufficient, where U is defined as follows:

$$U = \sum_{i=1}^{N} \frac{C_i}{P_i} \tag{1}$$

For static priority preemptively scheduled systems such as *rate monotonic*, we use response time analyses [20, 38] to obtain a necessary and sufficient condition. For a task T_i, the idea is to consider all preemptions produced by higher priority tasks on an increasing window time. The fixed point of the following recurrence gives the response time R_i of task T_i:

$$\begin{aligned} R_i^0 &= C_i \\ &\vdots \\ R_i^{n+1} &= C_i + \sum_{T_j \in HP(T_i)} \left\lceil \frac{R_i^n}{P_j} \right\rceil \times C_j \end{aligned} \tag{2}$$

where $HP(T_i)$ is the set of tasks with higher priority than T_i. In order to check the schedulability of task T_i, one only has to compare the response time R_i with its period P_i. Task T_i is schedulable if and only if $R_i \leq P_i$.

Our approach eliminates cache penalties due to cold-starting the cache after a context switch. Thus, classical non-cache sensitive schedulability analyses should be used rather than their cache-sensitive versions, CUA [3] and CRTA [7].

3. CMEs Overview

CMEs [15] are mathematical formulas that provide a precise characterization of the cache behavior for perfectly nested loops consisting of straight-line assignments. In order to describe data reuse, we use an extension [44, 48] of the well-known concept of reuse vectors [46], which describes the most recent previous access (MRPA) among arbitrary loop nests.

Based on the description of reuse given by reuse vectors, some equations are set up that describe those iteration points where the reuse is not realized. Solving them gives information about the number of misses and where they occur. In a previous work [44], we further extended them in order to make whole program analysis feasible, by handling call statements, IF statements and arbitrarily nested loops.

Write Policy. Given a memory reference, the equations are to investigate whether the reuse described by its reuse vectors is realized or not. We now briefly discuss how we extend our analysis to caches with write-no-allocate features.

The idea consists of treating in a different manner the reuse vectors corresponding to the reuses of data previously accessed by write references [14]. For a direct-mapped cache, the solution is as simple as ignoring all these reuses. For set-associative caches, we modify the solver so a write access that misses does not modify the LRU algorithm.

D-TLB. TLB is a hardware table of frequently used page translations. It is usually implemented as a set-associative cache, which is indexed with a subset of the bits that form the virtual address. Thus, in order to simulate its behavior we only have to compute the MRPAs [48] for each memory access in terms of pages instead of memory lines.

Multilevel Caches. For these architectures, we have to analyze differently memory references depending on the cache level they are accessing [40]. For that purpose, a set of equations is set up for each of the cache levels. When analyzing potential cache set contentions, only memory accesses that miss in lower cache levels are considered. Thus, we can see the equations for each level as filters, where only those memory accesses that miss are analyzed in higher levels.

While the results are accurate, safety is not guaranteed for systems with unified caches or where the upper levels of cache are physically indexed. An extension where the effects of instructions on unified caches and the use of physical addresses are considered is left as future work.

4. Obtaining a Predictable System

When considering cache memories, schedulability analyses should consider the cost of reloading the cache lines that may have been evicted from cache. When a preempted task resumes its execution, it may spend a lot of time reloading those cache lines that have been displaced from cache. Recent studies incorporate some *cache-related* preemption costs into the schedulability analysis [3, 7, 25]. They basically consider that the preempted task will incur a miss for each cache line when resuming execution. However, this approach cannot be used when dynamic cache locking is used, since the cost of preempting a task that is accessing a locked region may be much larger than a cache miss for every cache line. A preempting task may unlock the cache and load it with its own data; when the preempted task resumes its execution, it will not reload the cache since the cache is locked. Thus, there may be more extra misses than one per cache line throughout the locked region.

Our goal is to have a method that allows obtaining an exact (we want to guarantee an exact classification of memory accesses as cache hits or misses) and safe `WCMPs` of tasks for multitasking systems with data caches, so that current schedulability analyses can be applied without modifications.

PredictMultiTask given in Figure 1 takes as input a set of tasks and a cache architecture, and generates a set of cache partitions and a set of analyzable tasks that have the same semantics as the original tasks. Then, we run our static analyzer [42] which calculates an *exact* `WCMP` for each transformed task. In this section, we explain in detail the different parts of the algorithm. We first discuss the implications of using the cache partitioning technique. Then, we review our solution to the problem of predictability for data caches. Finally, we discuss the application of different techniques to optimize the cache behavior of tasks, so that the performance is not jeopardized.

4.1. Cache Partitioning

Inter-task interference occurs when cache lines from different tasks conflict in cache, which causes unpredictability. Cache partitioning [23] divides the cache into disjoint partitions, which are assigned to tasks in such a way that inter-conflicts are removed.

Let $\{T_1, \dots, T_n\}$ be a set of tasks. Usually, cache partitioning creates $n + 1$ 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. Note that tasks that have the same priority (thus, they are non-preemptively related to each other) can share the same partition, since they are only preempted by tasks that have higher priority, and thus the predictability of cache behavior is not affected. Therefore, it is enough to divide the cache in p partitions, where p is the number of different priorities.

Cache partitioning can be implemented either in software [34, 47] or hardware [23]. Both techniques impose the partition size to be a power of two, so that the pointer transformation to access data structures can be performed in a fast way.[1] The software approach requires compiler and linker support [34], which are responsible for relocating data to provide exclusive mappings on the cache for each task.

---

[1] This restriction does not apply to instruction caches.

```
INPUT
  $S$ = a set of tasks
  $C$ = a cache architecture

OUTPUT
  PredictMultiTask($S$, $C$) = <set of tasks, set of partitions>

ALGORITHM
  $CP$:= CreatePartitions($S$, $C$);                          // set of partitions
  $S\_aux := \emptyset$;                                       // set of modified tasks
  for each task $T_i \in S$
      $CP_i$ is $T_i$'s cache partition
      $P\_aux$:= LockUnpredictableRegions($T_i$);             // modified task after locking
      $P\_aux$:= OptimizeLock($P\_aux$);
      $P\_aux$:= LoadData($P\_aux$);
      $P\_aux$:= CacheOptimize($P\_aux$, $CP_i$);
      $S\_aux$.insert($P\_aux$);

  PredictMultiTask($S$, $C$) := $< S\_aux, CP >$
```

Figure 1. An algorithm for obtaining a predictable set of tasks on a multitasking system.

When a cache is partitioned, each task will access a smaller fraction of the cache, which may cause capacity misses to increase. Thus, the size of the partitions has an impact on the overall performance. In order to obtain the best data cache partitioning, the decision should be taken based on the priorities and the reuse patterns of tasks. For instance, a task that has a workload of 8KB but only accesses each cache line once only needs one cache line, whereas a task with a workload of 1KB that reuses each cache line one million times would suffer a performance loss with a partition smaller than 1KB.

Our approach works with both hardware and software mechanisms, and it does not depend on the size of the partitions created. From now on we assume that the cache is divided in n equally-sized partitions, one for each task. Yet simple, we show how it is good enough for scheduling real sets of tasks and better utilizing the CPU than other approaches. An algorithm to obtain even better performance, by choosing different cache partition sizes for different tasks, is left as future work.

4.2. Obtaining a Predictable Program

A practical limitation for WCET estimation is that the number of paths to be analyzed can easily be prohibitive, especially when studying loop constructs with multiple paths inside. Thus, we use path merging to reduce the path explosion by merging paths in those cases where a path enumeration is needed [13, 17, 30]. This includes data-dependent conditionals, loops with multiple paths inside and loops with unknown loop bounds.

On the other hand, there are data-dependent memory accesses. This includes 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. We also include nonlinear array references that are not handled by our static analyzer (e.g., a[i*j]) and library and operating system calls.

Both situations lead to unpredictability. Merging paths causes an unknown state of the cache, since a new state of the cache is created based on the state of the cache at the end of each path [1, 30]. Data-dependent memory accesses cannot be classified definitely as hits/misses and also cause an unknown state of the cache.

We avoid this source of unpredictability by locking those parts of the code where paths are merged or have data-dependent memory accesses [42]. *LockUnpredictableRegions* makes use of the control-flow graph (CFG) of the program, and inserts lock/unlock instructions (lock/unlock nodes in the control-flow graph) when necessary. In our approach, we always apply lock/unlock instructions to the whole cache. We use the code in Figure 2(a) to illustrate how *LockUnpredictableRegions* works. It traverses the control-flow graph of the program and detects the two constructs, in a well-structured CFG, that might give rise to multiple paths and thus needs path merging (data-dependent conditionals, and loop nests with unknown number of iterations). Besides, it detects data-dependent memory accesses which are non-analyzable. Figure 2(b) shows the code after the lock/unlock instructions have been inserted. Region 1 is created due to the non-analyzable memory access. Regions 2 and 2.1 are created because of the path merging needed there.

4.2.1. Optimizing Placement of Lock/Unlock Instructions Placement of lock/unlock instructions may affect performance; (i) the execution of lock/unlock instructions in-

```
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 accesses:
b[a[i]]

Merging constructs:
for (i=0;i<N;i++)
if (c[i]>15)

(a) 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();
```

(b) 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();
```

(c) Final version

Figure 2. Detailed steps of our algorithm to obtain a predictable program.

curs a run-time overhead which may be significant for the instructions placed within loops, and (ii) locking the cache when not necessary usually degrades performance.

OptimizeLock goes through the CFG looking for redundant lock/unlock instructions. It is an algorithm that keeps iterating while some progress is done. We have currently implemented the following optimizations, expressed in a simple, self-explanatory language for well-structured CFGs.

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
```

We define two extra rules to optimize the placement of lock/unlock instructions:

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
```

Whereas Rules 1–3 do not modify cache behavior, the last two rules may not always be beneficial. If data accessed in the newly locked statements are already in cache then these transformations do not hurt performance. However, they may create opportunities for other optimizations. Figure 2(c) shows the resulting code with the lock/unlock instructions for code in Figure 2(b) after running *OptimizeLock*. We can observe how Region 2.1 has been eliminated, and that Region 1 now is the whole loop nest. Figure 2(c) shows the final code after *LoadData* has been applied. For a detailed description of the algorithm, we refer the interested reader to our previous work [42].

4.3. Optimizing Cache Behavior

The combination of cache partitioning and cache locking may cause a poor cache behavior, due to increased number of capacity and conflict misses. In order to enhance locality, we consider two different transformations: loop tiling and padding.

Loop tiling is used to reduce capacity misses by reordering accesses in such a way that reuse distance is shortened. It combines strip-mining with loop interchange for increasing the effectiveness of memory hierarchy. Loop tiling basically consists of two steps [46]. The first one consists of restructuring the code to enable tiling those loops that carry reuse. The second one is to select the tile sizes that maximize locality. It is the latter step that is sensitive to the characteristics of the cache memory considered. Due to hardware constraints, caches have limited associativity, which may cause cache lines to be flushed out of the cache before they are reused despite sufficient capacity in the overall cache.

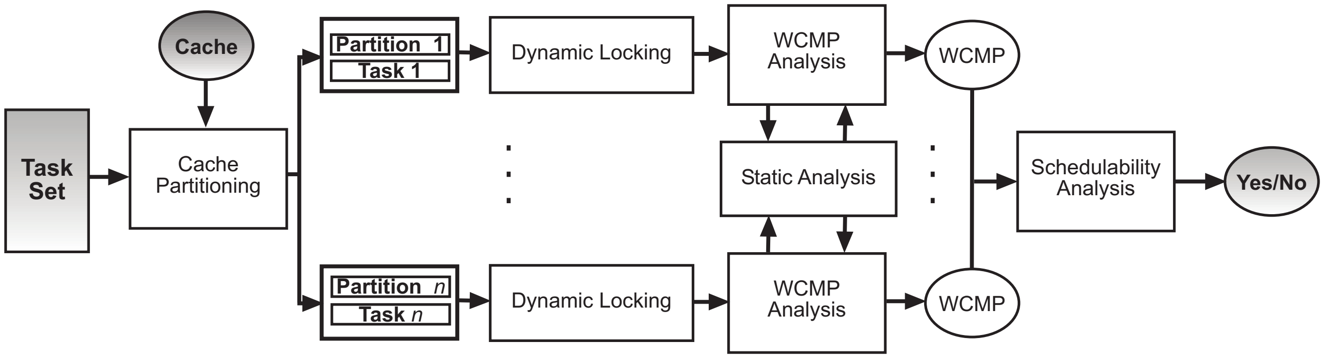


Figure 3. A framework for worst-case performance computation.

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.

Unlike loop tiling, padding [36] modifies the data layout to eliminate conflict misses. Some arrays may interfere severely for pathological alignments, which translates to a severe performance degradation. Padding changes the data layout in two different ways. Inter-padding modifies the base addresses of the arrays, whereas intra-padding changes the size of array dimensions. We use Vera *et al*'s [40] algorithm to select tile and pad sizes in concert.

5. Experimental Framework

We have conducted experiments for data caches commonly used in real-time systems. We have chosen 16KB and 32KB caches with 32B lines. For each cache, we have considered a direct-mapped cache, 2-way and 4-way set associative caches.[2] The timing model considered is very simple: we only consider memory and lock/unlock instructions. We chose the hit and miss access times after the PowerPC 604e [33], where each hit takes 1 cycle and each miss 38 cycles. Lock and unlock instructions take 1 cycle each. Each instruction to load the cache is treated as a normal memory access. Writes and reads are modeled identically. Thus, we present results in terms of `WCMP`.

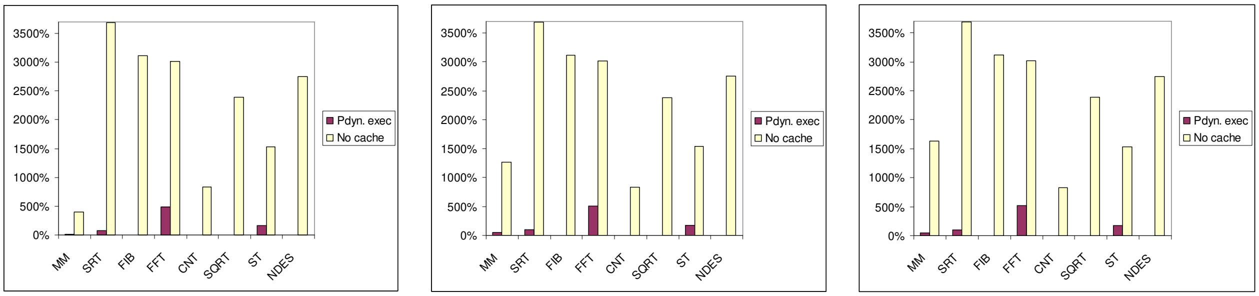
Figure 3 depicts the framework used to compute the worst-case performance and study the schedulability of a task set. The compiler passes (issuing lock/unlock instructions, inserting loads and applying tiling and padding) are written using the SUIF2 internal representation, which can be generated from different front-ends. We use SUIF2 to collect all information about memory accesses and control flow (it basically applies abstract inlining [44] and detects loops and IF statements). The paths that are used to obtain the path corresponding to the worst-case scenario are currently manually fed to our system.

The central component is the static analyzer. We have implemented the CMEs [15] following the techniques outlined in the literature [4, 41, 43, 44], and extended them to deal with locked regions [42].

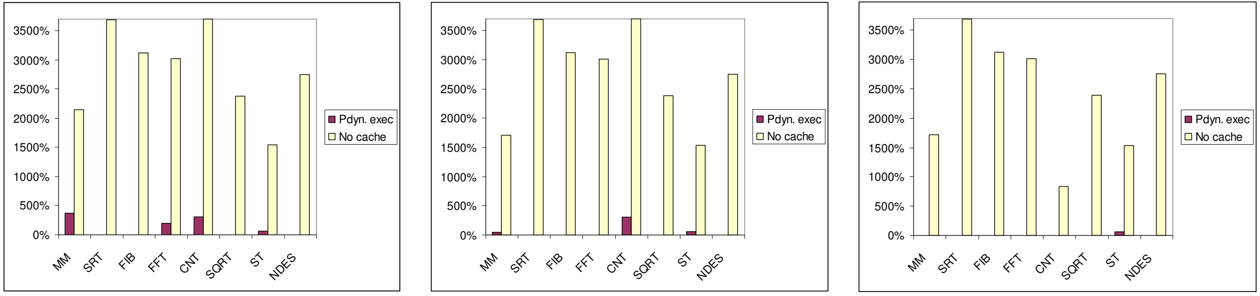
We present the performance of our approach for two real task sets. We set up a **large** task set in order to evaluate the efficiency of cache partitioning and compiler optimizations. The **medium** task set is used to show that even for smaller workloads, our approach performs better than static cache

---

[2] Caches with larger associativity usually use random or FIFO replacement policies.



16KB 1-way　16KB 2-way　16KB 4-way



32KB 1-way　32KB 2-way　32KB 4-way

Figure 4. Cache partitioning impact: comparison of performance degradation for a system with a partitioned cache and a system without a cache.

locking. An overview of the eight benchmark programs can be seen in Table 1. They are all written in `C`, drawn from different papers that analyze data cache behavior [1, 22, 45]. For each task, we present its name, its description, and the `WCMP` when the data cache is disabled. We give two possible periods. The *normal* periods of tasks have been selected so that the relation between CPU utilization[3] and amount of data is the same for each task set. We chose a CPU utilization of 2.03 for the **large** task set, and 4.69 for the **medium**. For the *HP* (high performance) periods, we chose them so that tasks have higher throughput. The resulting CPU utilization is 4.5 and 10.0 for the **large** and **medium** task set respectively. Thus, the task sets are not feasible if a data cache is not used for any of the period configurations.

Simulation results are obtained with a locally written memory hierarchy simulator [39]. We have modified it to handle locking caches and traces from different tasks. We have validated the new features with micro-benchmark simulation, running small kernels and comparing the results with the expected ones. Traces only contain load/store of data and lock/unlock instructions, and are tagged with a task ID. All results are in terms of memory cost.

6. Experimental Results

We now present results from our studies. We first discuss the impact of partitioning the cache on the system's throughput. Then, we show the worst-case performance when our method is applied, and compare it with static data cache locking [35].

6.1. Performance of Cache Partitioning

The goal of using cache partitioning is to eliminate unpredictability due to inter-task conflicts for multitasking systems that have data caches. However, they trade predictability for performance, which may cause some performance degradation. In order to evaluate the effectiveness of applying cache partitioning, we have compared the following three situations, where cache locking is not used:

- **Fully dynamic execution.** Each task uses the whole cache.
- **Partitioned dynamic execution.** We create equally-sized partitions. Each task runs on its own partition.
- **Cache disabled.** We consider the system without cache.

Figure 4 shows the results of this experiment. We present results in terms of slowdowns when compared to the memory cost of each task when fully dynamic execution is allowed.

---

[3] In terms of our simple timing model.

	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.

	Large Task Set						**Medium Task Set**					
	32KB			16KB			32KB			16KB		
Ways	1	2	4	1	2	4	1	2	4	1	2	4
Lock	3.55	3.55	3.55	3.85	3.85	3.85	2.97	3.44	3.44	5.11	4.37	4.37
Ours	0.40	0.21	0.17	0.99	0.85	0.81	0.79	0.79	0.79	0.92	0.93	0.93

Table 3. Performance of static cache locking and our cache analysis for a high-performance system.

We can observe that the average memory cost increases by $(79\%, 2470\%)$[4] for partitioned dynamic execution and a system without cache. This demonstrates that cache partitioning degrades performance compared to a system where each task uses the whole cache, but it is much better than not having a cache at all. Thus, we are trading performance for predictability.

6.2. Worst-Case Performance

In order to see the effectiveness of our approach, we have compared our method (all optimizations are on) to have a predictable multitasking system with data caches when static cache locking [9, 35] is applied.[5] For that purpose, we have loaded the cache with the most frequently accessed memory lines[6] for each task set, and locked it for the whole execution (it is the same as *Lock-MU* in [35]). This is the best worst-case performance that can be obtained with a shared cache using static cache locking; it gives better results than applying static locking for each task independently once the cache is partitioned since tasks that use the cache intensively use more cache lines.

The worst-case system performance of both task sets is given in Table 2. Each cell contains the CPU utilization (if it is smaller than 1, it is schedulable for dynamic priority preemptive schedules by (1)). A bold number indicates that the task set is *not* schedulable according to fixed priority schedules by (2). We can see that our dynamic cache locking performs better than static cache locking for all cases. Even though our approach only uses a fourth of the whole cache for each task, the combination of dynamic locking and static analysis makes better use of the cache, thus reducing the `WCMP`. Static cache locking is only able to schedule (both dynamic and fixed priority systems) the **large** task set for all 32KB cache configurations. However, our approach schedules all task sets for all cache architectures. Furthermore, the CPU utilization is between 3.2 and 9.8 times smaller for the 32KB architecture, and between 1.5 and 3.8 times smaller for the 16KB cache.

Optimizing Performance. The use of cache partitioning increases predictability by removing inter-task cache conflicts. However, it may increase intra-task cache conflicts since each task uses a smaller cache. This can be critical for direct-mapped caches, whereas set-associative caches can handle conflicts in a better way. In order to reduce intra-task conflicts, we apply well-known compiler cache optimizations in concert with dynamic cache locking. For the **large** task set, the application of tiling has translated to a 5.6% (1%) `WCMP` reduction for MM on the 16KB direct-mapped (2-way) partitioned cache, and padding has reduced the `WCMP` for FFT by 99.9% on the 32KB direct-mapped partitioned cache. The average memory cost compared to the partitioned dynamic execution scheme drops to $(189.12\%, 7.23\%)$ for the 16KB cache and 32KB cache respectively. If the optimizations had not been applied, we would not have been able to schedule successfully the **large** data set on the 16KB direct-mapped cache. In addition, it has allowed reducing the CPU utilization on the other cases.

6.3. High-Performance Systems

Finally, we show results for a high-performance multitasking system in Table 3, where throughput is higher and thus the CPU utilization increases. For that purpose, we have chosen the *HP* periods in the last column of Table 1. Since the magnitude of the periods is very different among tasks, fixed priority systems do not perform well, and thus

---

[4] Since a cache hit is 1 cycle and a cache miss 38 cycles, the worst possible slowdown is $\frac{38-1}{1} \times 100\% = 3700\%$.

[5] For a comparison of performance between dynamic and static locking, see our previous work [42].

[6] We assume the worst-case path for each task is known.

we only compute the CPU utilization. We can observe that our approach works better under tight deadlines, and it is able to schedule all task sets. However, static cache locking fails to schedule any of the task sets. In this case, the CPU utilization of our method is between 3.8 and 20.0 times smaller for a 32KB cache and between 3.8 and 5.5 for the 16KB architecture. This indicates that our method scales better than static cache locking for systems that need high throughput.

6.4. Summary

Overall, we have demonstrated the effectiveness of our approach. First, we have evaluated the impact of applying cache partitioning on a multitasking system. We have seen that even though the performance degrades, partitioning the cache is much better than not having a cache at all. We have also pointed out how the application of compiler cache optimizations can be useful to reduce the performance degradation caused by the use of a small fraction of the cache. Finally, we have compared our approach with static cache locking in which all the tasks share the whole cache. We have shown that our method performs much better, and is capable of scheduling tasks that need a high throughput.

7. Related Work

In the past few years several strategies have been presented for analyzing cache memory behavior analytically. Ghosh *et al* [15] presented the CMEs framework targeted at isolated perfect loop nests consisting of straight-line assignments. Vera and Xue [44] examine the problem of analyzing whole programs. This model is able to predict misses for large codes consisting of data-independent constructs (including calls and IF statements).

Meanwhile, the real-time community has intensified the research in the area of predicting WCET of programs in presence of caches. Calculation of a tight WCET bound of a program involves difficulties that come from the very characteristics of data caching. Even though some progress has been done when studying processors with instruction caches [2, 17, 26], few steps have been done towards analyzing data caches.

Alt *et al* [1, 13] provide an estimation of WCET by means of abstract interpretation. As well as the usual drawbacks from abstract analysis (i.e., time consuming and lack of accuracy), they only analyze memory references which are scalar variables. When providing experimental results, they only deal with instruction caches. Lim *et al* [29] present a method that computes the WCET taking into account data caching. However, they only analyze static memory references (i.e., scalars), failing to study real codes with dynamic references (i.e., arrays and pointers). Kim *et al* [22] propose a method that improves the previous method extending the analysis that classifies references as either static or dynamic. However, they deal with neither arrays nor pointers (i.e., only detecting temporal locality). Further, it is limited to basic blocks, without taking into account possible reuse among different subroutines or loop nests. Li *et al.* [27] describes a method which does not merge the cache state but tries to calculate possible cache contents along with the timing of the program. The whole CPU is modeled by a linear integer programming problem, and a new constraint is added for each element of a calculated reference. This requires a very large computation time, and has problems of scalability with large arrays. Besides, they do not report results for WCET in the presence of data caches.

White *et al* [45] propose a method for direct-mapped caches based on static simulation. They categorize static memory accesses into (i) first miss, (ii) first hit, (iii) always miss and (iv) always hit. Array accesses whose addresses can be computed at compile-time are analyzed, but they fail to describe conflicts which are always classified as misses. For instance, they overestimate the memory cost by 10% and 17% for MM and ST respectively (we estimate the `WCMP` exactly without issuing lock instructions).

Lundqvist and Stenström [30] propose an approach where variables that have non-analyzable references are mapped onto a non-cacheable memory space. They show that the majority of data structures in their benchmarks are predictable, but they have not presented the overhead of the transformed program. Neither have they reported results for WCET or `WCMP` using their approach.

Campoy *et al* [9] introduce the use of locking instruction caches for multitasking systems. They use static locking, and present a genetic algorithm in an attempt to reduce the solution space when selecting the best contents for the cache. They represent each memory block by means of one bit, which flips between 0/1 (in-cache/out-cache). On one hand, we have shown that static locking is not a good solution for data caches. On the other hand, while this approach may work for small programs, it is not easy to see how it can be extended to data caches:(i) each possible solution would occupy a lot of memory (data is typically much larger than programs), and (ii) we would need a static analysis to evaluate each potential solution. Puaut and Decotigny [35] extend it by introducing two polynomial algorithms to select the instructions to lock in cache.

8. Conclusions

This paper combines cache partitioning and dynamic data cache locking with static cache analysis to estimate the worst-case memory performance of a multitasking system in a safe, exact and fast way. The static analysis uses a precise characterization of reuse, and results in a set of equations that describes whether the reuse translates to locality.

Our method partitions the cache in equally-sized partitions, which are assigned to tasks. Cache partitioning allows us to eliminate unpredictability due to inter-task conflicts. In order to overcome the problem of data-dependent constructs, we combine it with dynamic cache locking. Finally, we run a static analysis. This results in a tool that predicts the worst-case memory performance in an *exact* and *safe* way, with an acceptable loss of tasks' performance. Combined with a timing analysis platform, we may estimate a tight worst-case performance.

Overall, this paper contributes a new technique that provides a considerable step toward a useful worst-case execution time prediction of actual architectures. To the best of our knowledge, this is the first approach that presents a method to estimate worst-case performance for multitasking systems in the presence of set-associative data caches. It is written as a compiler pass, which partitions the cache, issues lock/unlock instructions and computes the worst-case memory performance in the presence of set-associative data caches. Moreover, our framework guides the compiler in order to generate code that exploits the cache memory by inserting load instructions and restructuring the data layout (padding) and the loops (tiling). A better use of the cache is very useful in order to reduce power consumption and better utilize the CPU, which allows running more real-time tasks simultaneously.

There are still some issues that can be investigated further. A better pointer analysis could be beneficial to lock fewer regions, and would help us to classify their accesses as hits or misses. It may also be interesting to take into account the overall performance when selecting the size of the cache partitions. We plan to investigate these research directions in order to have increased predictability and better performance.

Acknowledgements

The authors thank Nerina Bermudo for her valuable help implementing the CMEs solver. We would like to thank the anonymous referees for providing insightful comments.

References

[1] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behaviour prediction by abstract interpretation. In *Proceedings of Static Analysis Symposium (SAS'96)*, Lecture Notes in Computer Science (LNCS) 1145, pages 52–66. Springer-Verlag, Sep. 1996.

[2] R. Arnold, F. Müeller, D. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *Proceedings of 15th Real-Time Systems Symposium (RTSS'94)*, pages 172–181, 1994.

[3] S. Basumallick and K. Nielsen. Cache issues in real-time systems. In *Proceedings ACM Workshop on Languages, Compilers and Tools for Real-Time Systems (LCTES'94)*, Jun. 1994.

[4] N. Bermudo, X. Vera, A. González, and J. Llosa. An efficient solver for cache miss equations. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'00)*, 2000.

[5] A. Burns, K. Tindell, and A. Wellings. Effective analysis for engineering real-time fixed priority schedulers. *IEEE Transactions on Software Engineering*, 21:475–480, 1995.

[6] A. Burns and A. Wellings. The impact of an Ada run-time system's performance characteristics on scheduling models. In *Proceedings of 12th Ada-Europe International Conference*, pages 240–248, Jun. 1993.

[7] J. V. Busquets-Mataix, J. J. Serrano, R. .Ors, P. Gil, and A. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proceedings of 2nd Real-Time Technology and Applications Symposium (RTAS'96)*, Jun. 1996.

[8] J. V. Busquets-Mataix, J. J. Serrano, and A. Wellings. Hybrid instruction cache partitioning for preemptive real-time systems. In *Proceedings of 9th Euromicro Workshop on Real-Time Systems (EUROMICRO-RTS'97)*, Jun. 1997.

[9] M. Campoy, A. P. Ivars, and J. V. Busquets-Mataix. Static use of locking caches in multitask preemptive real-time systems. In *Proceedings of IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium)*, 2001.

[10] S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Proceedings of Supercomputing (SC'92)*, pages 114–124, Nov. 1992.

[11] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *Proceedings of Euro-Par (EUROPAR'97)*, pages 1298–1307, Aug. 1997.

[12] P. Feautrier. Automatic parallelization in the polytope model. In G. R. Perrin and A. Darte, editors, *The Data Parallel Programming Model*, Lecture Notes in Computer Science 1132, pages 79–103. Springer Verlag, 1996.

[13] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17:131–181, 1999.

[14] S. Ghosh. *Compiler Analysis Framework for tuning memory behavior*. PhD thesis, Princeton University, Nov. 1999.

[15] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4):703–746, 1999.

[16] J. Gustafsson. *Analyzing Execution Time of Object-Oriented Programs Using Abstract Interpretation*. PhD thesis, Uppsala University, May 2000.

[17] C. A. Healey, D. Whalley, and M. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *Proceedings of 16th Real-Time Systems Symposium (RTSS'95)*, pages 288–297, 1995.

[18] IBM Microelectronics Division. *The PowerPC 440 core*, 1999.

[19] K. Jeffay and D. L. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *Proceedings of 14th Real-Time Systems Symposium (RTSS'93)*, pages 212–221, Dec. 1993.

[20] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[21] A. I. Katcher, H. Arakawa, and J. K. Strosnider. Engineering and analysis of fixed priority schedulers. *IEEE Transactions on Software Engineering*, 19:920–934, 1993.

[22] S. K. Kim, S. L. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *Proceedings of IEEE Real-Time Technology and Applications Symposium (RTAS'96)*, 1996.

[23] D. B. Kirk. SMART (strategic memory allocation for real-time) cache design. In *Proceedings of 10th Real-Time Systems Symposium (RTSS'89)*, Dec. 1989.

[24] M. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance of blocked algorithms. In *Proceedings of IV International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'91)*, Apr. 1991.

[25] C. G. Lee, J. Hahn, Y. M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transaction on Computers*, 47, 1998.

[26] Y. T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of 16th Real-Time Systems Symposium (RTSS'95)*, pages 298–307, 1995.

[27] Y. T. S. Li, S. Malik, and A. Wolfe. Cache modeling and path analysis for real-time software. In *Proceedings of 17th Real-Time Systems Symposium (RTSS'96)*, 1996.

[28] J. Liedtke, H. Härtig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *Proceedings of 3rd IEEE Real-Time Technology and Applications Symposium (RTAS'97)*, 1997.

[29] S. S. Lim, Y. H. Bae, G. T. Jang, B. D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Kim. An accurate worst case timing analysis technique for RISC processors. In *Proceedings of 15th Real-Time Systems Symposium (RTSS'94)*, pages 97–108, 1994.

[30] T. Lundqvist and P. Stenström. A method to improve the estimated worst-case performance of data caching. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, pages 255–262, Dec. 1999.

[31] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of 20th Real-Time Systems Symposium (RTSS'99)*, Dec. 1999.

[32] The SUIF Compiler Group. SUIF: An infrastructure for research on parallelizing and optimizing compilers. http://suif.stanford.edu.

[33] Motorola Inc. *PowerPC 604e RISC Microprocessor Technical Summary*, 1996.

[34] F. Müeller. Compiler support for software-based cache partitioning. In *Proceedings ACM Workshop on Languages, Compilers and Tools for Real-Time Systems (LCTES'95)*, Jun. 1995.

[35] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proceedings of 23th Real-Time Systems Symposium (RTSS'02)*, Dec. 2002.

[36] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, pages 38–49, 1998.

[37] O. Temam, E. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for accessing when data copying should be used to eliminate cache conflicts. In *Proceedings of Supercomputing (SC'93)*, pages 410–419, 1993.

[38] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *Real-Time Systems*, 6(1):133–151, 1994.

[39] X. Vera. Coyote project: The simulator. Technical Report MRTC Report 95/2003, Mälardalens Högskola, Apr. 2003.

[40] X. Vera, J. Abella, A. González, and J. Llosa. Optimizing program locality through CMEs and GAs. In *Proceedings of 12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)*, New Orleans, Sept. 2003.

[41] X. Vera, N. Bermudo, J. Llosa, and A. González. A fast and accurate framework to analyze and optimize cache memory behavior. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, To Appear.

[42] X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. In *Proceedings of International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'03)*, pages 272–282, Jun. 2003.

[43] X. Vera, J. Llosa, A. González, and N. Bermudo. A fast and accurate approach to analyze cache memory behavior. In *Proceedings of European Conference on Parallel Computing (Europar'00)*, 2000.

[44] X. Vera and J. Xue. Let's study whole program cache behaviour analytically. In *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA 8)*, Cambridge, Feb. 2002.

[45] R. T. White, F. Müeller, C. Healy, D. Whalley, and M. Harmon. Timing analysis for data caches and set-associative caches. In *Proceedings of Third IEEE Real-Time Technology and Applications Symposium (RTAS'97)*, pages 192–202, 1997.

[46] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'91)*, pages 30–44, Jun. 1991.

[47] A. Wolfe. Software-based cache partitioning for real-time applications. In *Proceedings of the 3rd International Workshop on Responsive Computer Systems*, 1993.

[48] J. Xue and X. Vera. Efficient and accurate analytical modeling of whole-program data cache behavior. *IEEE Transactions on Computers*, To Appear.