# Data Cache Locking for Tight Timing Calculations

XAVIER VERA
Mälardalens Högskola
and
BJÖRN LISPER
Mälardalens Högskola
and
JINGLING XUE
University of New South Wales

Caches have become increasingly important with the widening gap between main memory and processor speeds. Small and fast cache memories are designed to bridge this discrepancy. However, they are only effective when programs exhibit sufficient data locality. In addition, caches are a source of unpredictability, resulting in programs sometimes behaving in a different way than expected.

Detailed information about the number of cache misses and their causes allows us to predict cache behavior and to detect bottlenecks. Small modifications in the source code may change memory patterns, thereby altering the cache behavior. Code transformations which take the cache behavior into account might result in a high cache performance improvement. However, cache memory behavior is very hard to predict, thus making the task of optimizing and timing cache behavior very difficult.

This article proposes and evaluates a new compiler framework that times cache behavior for multitasking systems. Our method explores the use of *cache partitioning* and *dynamic cache locking* to provide worst-case performance estimates in a safe and tight way for multitasking systems. We use cache partitioning, which 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.

The results of our experiments demonstrate the capability of our framework to describe cache behavior at compile time. We compare our timing approach with a system equipped with a non-partitioned but statically locked data cache. Our method outperforms static cache locking for all analyzed task sets under various cache architectures, demonstrating that our fully predictable scheme does not compromise the performance of the transformed programs.

Categories and Subject Descriptors: B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids ; C.4 [**Performance of Systems**]: Performance Attributes

General Terms: Measurement, Performance

Additional Key Words and Phrases: Worst-Case Execution Time, Data Cache Analysis, Embedded Systems, Safety Critical Systems.

This work has been supported by VR grant no. 2001–2575.
Xavier Vera and Jingling Xue have been supported in part by Australian Research Council Grants A10007149 and DP0452623.
Authors' addresses: X. Vera, Institutionen för Datateknik, Mälardalens Högskola, P.O. BOX 883, Västerås 721 23, Sweden; email: xavier.vera@mdh.se; B. Lisper, Institutionen för Datateknik, Mälardalens Högskola, P.O. BOX 883, Västerås 721 23, Sweden; email: bjorn.lisper@mdh.se; J. Xue, Programming Languages and Compilers Group, School of Computer Science and Engineering, University of New South Wales, NSW 2052, Sydney, Australia; email: jxue@cse.unsw.edu.au.

## 1. INTRODUCTION

With ever-increasing clock rates and the use of new architectural features, the speed of processors increases dramatically every year. However, memory speeds have lagged behind, thus increasing memory latency [Hennessy and Patterson 1996]. This widening performance gap affects all computer systems, and is a key obstacle to achieving high processor utilization due to memory stalls. The basic solution that almost all systems rely on is the use of cache memories.

Memory is organized hierarchically in such a way that the lower levels are smaller and faster. In order to fully exploit the memory hierarchy, one has to ensure that most of the memory references are handled by lowest levels of cache. Programmers spend a significant amount of time improving locality, which is tedious and error prone. Compilers apply useful loop transformations and data layout transformations to take better advantage of the memory hierarchy. In all cases, a fast and accurate assessment of a program's cache behavior is needed at compile time to make an appropriate choice of transformation parameters.

Unfortunately, cache memory behavior is hard to predict. That makes it very difficult to statically analyze the interaction between a program's reference pattern and the memory subsystem. This work describes a static approach to characterizing whole programs' cache memory behavior, which is used to accurately time memory performance. Our method explores the use of *cache partitioning* and *dynamic cache locking* to provide worst-case performance estimates in a safe and tight way for multitasking systems.

### 1.1 Motivation

Hard real-time systems are those where a failure to meet a deadline can be fatal. To guarantee their behavior, the worst-case behavior has to be analyzed. It will be used to ensure timely responses from tasks and as input to scheduling algorithms.

When using caches in hard real-time systems there is an unacceptable possibility that a high cache miss penalty combined with a high miss ratio might cause a missed deadline, jeopardizing the safety of the controlled system. Besides, caches also increase the variation in execution time, causing jitter. Thus, many safety-critical systems either do not have 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.

That implies that it is necessary to know the execution times for the tasks in a real-time system. The analysis from a high-level point of view is concerned with all the possible paths through the program. Low-level analysis determines the effect on the program timing of machine-dependent factors, such as caches. While control flow can be modeled precisely, hardware (caches, branch predictors, etc.) can give rise to actual unpredictability. Therefore, real-time systems have to be analyzed as a whole, where both software and hardware play their respective roles.

In order to obtain an accurate worst-case execution time (WCET), a tight worst-case memory performance (`WCMP`) is needed. Current WCET platforms are applied

to rather simple architectures (they usually do not consider data 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 [Burns et al. 1995; Burns and Wellings 1993; Jeffay and Stone 1993; Katcher et al. 1993].

However, cache behavior is very hard to predict, which leads to an overestimation of the WCMP, and thus for the WCET as well. Our goal is to guarantee an exact prediction of hits or misses for all memory accesses. In modern processors, it may happen that a cache hit is more expensive than a cache miss [Lundqvist and Stenström 1999b]. If a memory access is not classified definitely as a hit or miss, a subsequent calculation pass in a WCET analysis would have to consider both situations to detect the worst-case path, which would increase the complexity of the problem, and thus, the time needed to calculate the WCET.

### 1.2 Problem Statement

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 [Alt et al. 1996; Arnold et al. 1994; Healey et al. 1995]. These results can be generalized to preemptive systems [Basumallick and Nielsen 1994; Busquets-Mataix et al. 1996; Busquets-Mataix et al. 1997; Campoy et al. 2001; Kirk 1989; Lee et al. 1998; Müeller 1995; Puaut and Decotigny 2002]. 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 or misses very hard.

Current approaches [Alt et al. 1996; Ferdinand and Wilhelm 1999; Kim et al. 1996; Lim et al. 1994] provide an estimation of WCET by considering data caching where only memory references which are scalar variables are considered. Thus, they do not study real codes with dynamic references (i.e., arrays and pointers). White *et al* [White et al. 1997] 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 do not describe conflicts which are always classified as misses.

Hence, there is a need for a tool that computes the WCET in the presence of set-associative data caches for real programs.

### 1.3 Overview

This paper presents a compiler algorithm to estimate the worst-case memory performance (WCMP) for multitasking systems in the presence of data caches [Vera et al. 2003b]. We use cache partitioning to eliminate inter-task conflicts, thus we can analyze each task in isolation. Tasks now use a smaller cache. Hence, we apply compiler cache optimizations such as tiling [Lam et al. 1991; Xue and Huang 1998] and padding [Rivera and Tseng 1998] to reduce the number of misses.

An accurate information of the cache memory behavior is essential to both optimize and time memory behavior. We use an analytical method presented in our

previous work [Vera et al. 2004; Vera and Xue 2002] which describes the cache misses of a given path with regular computations. This analysis is the first step required by any compiler transformation or WCET analysis.

However, when computing WCET we may have to consider different paths, which may cause cache contents to be different than expected. Moreover, the state of the cache can only be *determined* if *all* memory addresses are known. The state of the cache is *unknown* from the point in the code where an unknown cache line is accessed. Thus, even if we know the memory addresses of the future memory accesses, the cache behavior cannot be predicted exactly; it may be that the unknown memory access has trashed the cache line we planned to reuse; it may be that it has actually brought the data we are going to reuse.

We have developed a compile-time algorithm that identifies those regions of code where we cannot exactly determine the memory accesses. In those situations (i.e., merging of two different paths or non-predictable memory accesses), the cache is locked so we do not jeopardize the cache analysis [Vera et al. 2003a]. Precisely, the cache is locked just before such a code region is executed and unlocked just after the region has been executed. We shall see shortly that such a dynamic locking scheme is preferred since static locking can cause significant performance degradations. In order to obtain the most benefit from the cache, we use a locality analysis based on Wolf and Lam's reuse vectors [Wolf and Lam 1991] to select data to be loaded and locked in. Since the state of the cache is known when leaving the region, we can apply our static analyzer [Vera and Xue 2002] for the next regions of code, thus having both predictability and good performance. To the best of our knowledge, this is the first framework that obtains an exact `WCMP` for multitasking systems.

We have implemented our system in the SUIF2 compiler. It includes many of the standard compiler optimizations, which allows us to obtain a code competitive to production compilers. Using SUIF2, we identify high-level information (such as array accesses and loop constructs) that can be further passed down to the low-level passes as annotations. We plan to integrate our `WCMP` calculation with an existing WCET tool [Engblom and Ermedahl 2000] that already analyzes pipelines and instruction caches.

In order to show to what extent our method can estimate the `WCMP`, we present results for a collection of task sets consisting of programs drawn from several related papers [Alt et al. 1996; Kim et al. 1996; White et al. 1997]. This collection includes kernels operating on both arrays and scalars, such as SQRT or FIBONACCI. We have also used FFT to show the feasibility of our approach for typical DSP codes. For the sake of concreteness, we have chosen the cache architectures of a set of modern processors widely used in the real-time area: microSPARC-IIep [Sun Microelectronics 1997], PowerPC 604e [Motorola Inc. 1996], IDT 79RC64574 [Integrated Device Technologies 2001] and MIPS R4000 [MIPS Technologies 2001].

### 1.4 Static Cache Locking

In this work, we propose a dynamic cache locking scheme by which the cache is locked dynamically for the code regions whose memory accesses are not exactly known at compile time (and unlocked for the other parts of the program). Another approach, known as static cache locking, works by pre-loading the most frequently accessed data in the program into the cache and then locking the cache for the

| Program | Analysis | Miss Ratios | | | | Cycles |
|---|---|---|---|---|---|---|
| | | MIN | MAX | AVG | Increase (%) | Degradation (%) |
| MM | Normal | 1.88 | 33.53 | 10.01 | 721.77 | 599.55 |
| | Static Locking | 59.14 | 99.36 | 82.27 | | |
| CNT | Normal | 5.67 | 8.33 | 7.94 | 710.92 | 565.67 |
| | Static Locking | 18.08 | 98.72 | 64.45 | | |
| ST | Normal | 3.57 | 14.29 | 7.66 | 389.87 | 307.87 |
| | Static Locking | 3.57 | 96.80 | 35.87 | | |
| SQRT | Normal | 1.43 | 1.43 | 1.43 | 0.00 | 0.00 |
| | Static Locking | 1.43 | 1.43 | 1.43 | | |
| FIB | Normal | 0.49 | 0.49 | 0.49 | 0.00 | 0.00 |
| | Static Locking | 0.49 | 0.49 | 0.49 | | |
| SRT | Normal | 8.37 | 16.74 | 10.93 | 69.16 | 58.28 |
| | Static Locking | 8.37 | 93.73 | 18.49 | | |
| NDES | Normal | 0.90 | 1.74 | 0.96 | 38.40 | 12.30 |
| | Static Locking | 0.90 | 6.56 | 1.33 | | |
| FFT | Normal | 0.59 | 56.10 | 9.18 | 119.37 | 97.66 |
| | Static Locking | 0.59 | 93.64 | 20.15 | | |

Table I. Overhead of static cache locking. For "Normal", the cache behaves in the normal manner. For "Static Locking", the cache is pre-loaded with the most frequently accessed lines and then locked during the entire execution of the program. *Increase* (*Degradation*) represents the average increase in miss ratios (cycles) of "Static Locking" over "Normal" across all cache architectures.

entire execution of the program. We show that static cache locking is impractical since it leads to poor cache utilization when data do not fit the cache. In order to confirm this, we have run a number of different benchmark programs comparing static locking against the same configuration in which the cache is used but not locked. Table IV contains a detailed description of these benchmarks.

In order to obtain the most frequently accessed data, we run each program once and collect statistics for each memory line accessed. Then we load each cache set with the most frequently accessed memory lines that are mapped to the set.

We have analyzed four cache architectures (4KB, 8KB, 16KB and 32KB (32B per line)) for three different associativities (direct-maped, 2-way and 4-way). We have also simulated the microSPARC I cache architecture (direct-maped, 512 bytes, 32B per line). We present the results accounting only for load/store instructions. We assume conservatively that a cache hit takes 1 cycle and a cache miss 50 cycles. Table I shows the results of our experiments. For each program, we report the minimum, maximum and average miss ratios across all cache architectures for both "Normal" and "Static Locking" scenarios. The last two columns show to what extent static locking has degraded performance in terms of the two metrics, miss ratios and simulated cycles. We can see that performance drops significantly for most programs, and in some cases, the performance degradation can be more than 500% in cycles. Static locking performs well only in those cases where all data fit the cache such as SQRT (which only accesses a few floating point values) and SRT (when the cache is large enough to hold the array being sorted).

### 1.5 Paper Organization

The rest of this paper is organized as follows. Section 2 gives an overview of the flow and cache analysis used in our `WCMP` tool. Section 3 describes our scheme in

detail. We first present our solution to have predictable programs on a unitask environment and then extend it to multitasking systems. Section 4 presents extensive evaluations of our method. We start by evaluating the accuracy and contributions of all different components independently, followed by performance measurements for a multitasking system. We discuss some related work in Section 5. Finally, we conclude and give a road map to some future extensions in Section 6.

## 2. A WCET TOOL OVERVIEW

A real-time system is a computer-based system where the *timing* of a computation is as important as the actual *value*. In most cases, steady and predictable behavior is the desired property; sometimes *too fast* is as bad as *too slow*.

Hard real-time systems are those where a failure to meet a deadline can be fatal. To guarantee their behavior, the worst-case behavior has to be analyzed. It will be used to ensure timely responses from tasks as well as input to scheduling algorithms.

That implies that it is necessary to know the execution times for the tasks in a real-time system. However, a task does not have a unique execution time. There are two sources of execution time variation: (i) the task may have different work loads depending on the input, and (ii) the initial state of the hardware where the task is executed may change for different runs. Since execution time varies, the WCET (i.e., the longest execution time for a program for all possible input data) is used as a safe upper limit.

A naïve approach to computing the WCET of a task would be to run the program for each possible input. However, this is not possible in practice due to measurement time. Running the program with the input data that cause the WCET would be a solution, but it is usually hard to know such data. A third option is running the code with an estimated very *bad* input data. Then a safety margin is added. However, the WCET that is obtained in each of these approaches is not safe, since it cannot be proved that it is the actual WCET.

We divide static WCET into three different phases:

—*Flow analysis*, which determines the possible paths through the program

—*Low-level analysis*, which determines the effect on program timing of hardware factors like cache memories

—*Calculation*, which calculates the actual WCET based on the information provided by the other components.

In this section we first review the task model. Then, we outline the flow and cache analyses, and how they impact WCET computation.

### 2.1 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 [Joseph and Pandya 1986; Tindell et al. 1994] 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 [Basumallick and Nielsen 1994] and CRTA [Busquets-Mataix et al. 1996].

### 2.2 Flow Analysis

In order to estimate the WCET statically, we must analyze all possible paths. For each path generated, the execution time will be estimated for a particular architecture.

Program flow analysis determines the possible paths through a program.

*Definition 1* **(Path)**. A path from $u$ to $v$ in the control flow graph of a program is a sequence of directed control flow edges, $n_0, n_1, \ldots, n_k$, such that $n_0 = u$, $n_k = v$ and $(n_i, n_{i+1})$ is an edge in the graph.

Flow analysis yields information about which functions are called, the number of iterations of a loop, etc. Unfortunately, it is infeasible to analyze all possible paths in a program.

Approximations during computation must be selected so that path explosion is reduced: a simple loop with an if-then-else statement that iterates a hundred times generates $2^{100}$ possible paths. Whenever it is possible, infeasible paths are removed [Ermedahl and Gustafsson 1997; Lundqvist and Stenström 1998], reducing the number of paths to be analyzed.

The following restrictions define the scope of programs where the execution path can be determined statically, and thus, only one path is considered when computing the WCET:

—Calls are non-recursive.

—All loop bounds and if conditionals must be either known or in terms of the loop indices of the enclosing loops.

```
if (!a[i])
   b[i]++;
else
   c[i]--;
```

(a) if Construct

```
for (i=0;i<4;i++)
   if (a[i])
      break;
```

(b) Loop Construct

```
for (i=0;i<2;i++){
   if (a[i]){
      a[i]--;
      break;
   }
   else
      a[2*i]++;
}
```

(c) Loop with if

Fig. 1. Codes where more than one path has to be analyzed.

We rely on the compiler to identify compile-time and runtime constants. Standard compile-time techniques can be very helpful in gathering missing information. To address the symbolic loop bound problem, we use interprocedural constant propagation to eliminate as many symbolic loop bounds as possible. Besides, we assume that the maximum number of iterations of a loop and the maximum number of recursive calls are known. This can be done either by manual annotations [Ermedahl and Gustafsson 1997] or by automatic approaches [Gustafsson 2000].

Otherwise, when there is a lack of information at compile time that prevents from analyzing only one path, we will apply a common technique known as *path merging* in order to reduce the number of paths being analyzed and make the analysis more efficient. This basically consists of reducing the path explosion by merging paths in those cases where a path enumeration is needed [Ferdinand and Wilhelm 1999; Healey et al. 1995; Lundqvist and Stenström 1999a]. This includes data-dependent conditionals, loops with multiple paths inside and loops with unknown loop bounds. Figure 1 introduces three examples that will be used through this article to illustrate path merging.

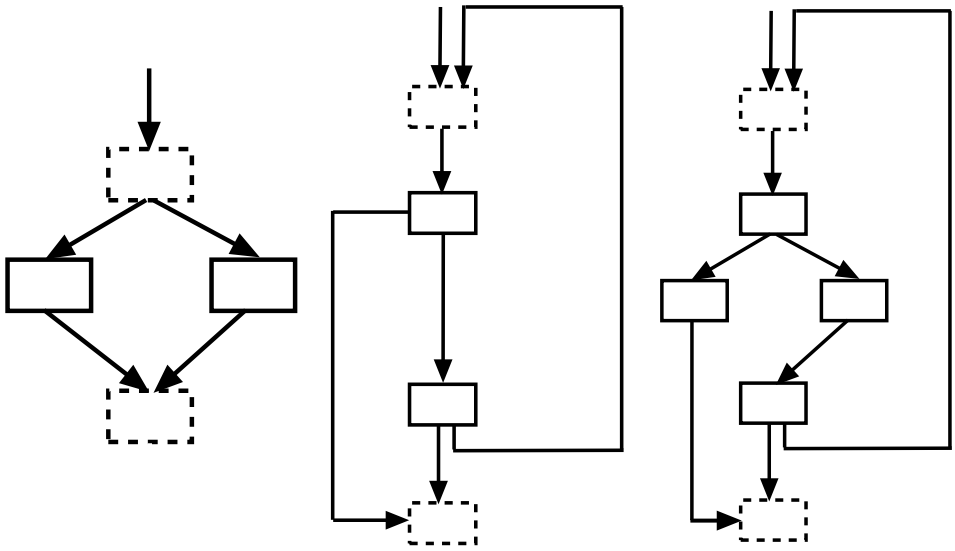
However, this approximation trades performance for accuracy. At every merge point, the most pessimistic assumptions are made in order to have a safe estimate. In the presence of caches, this generally translates to an unknown state of the cache, since the final state of the cache for each path is also merged.

Next, we explain in detail where we insert the merging operator. Later, we will formally define the merging operator and its impact on cache analysis.

### 2.3 Merging Operator Placement

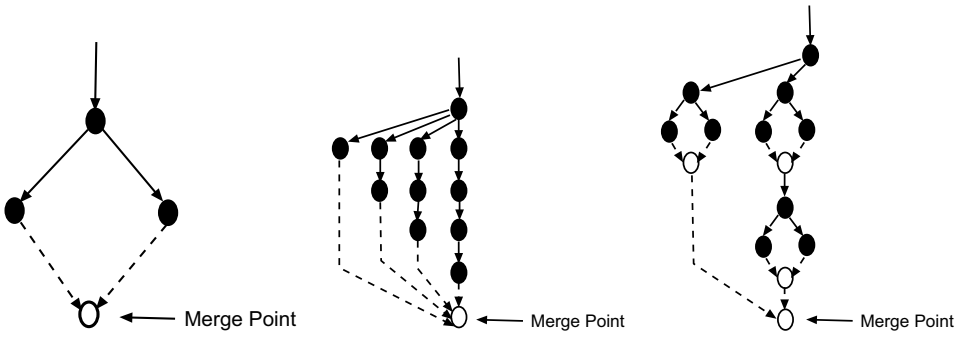
Merge points can be chosen arbitrarily depending on accuracy and execution time desired. We use a strategy where the number of paths to be explored are reduced from $n^k$ to $kn$ for a loop that contains $n$ possible paths and iterates $k$ times. In each iteration, all $n$ paths are analyzed, but at the start of the next iteration all these paths are merged into one. Note that we only consider natural loops, which may have multiple exits. In the other cases, path merging is not applied and thus all paths are analyzed.

Conditionals are also treated in a special way. When a conditional branch whose condition is unknown is found, both paths have to be analyzed. In order to keep the number of paths that are analyzed reasonable, both paths are merged. We



(a) if Construct (b) Loop Construct (c) Loop with if

Fig. 2. Control-flow graph for examples in Figure 1. Dashed boxes represent entry/exit nodes.




(a) if Construct (b) Loop Construct (c) Loop with if

Fig. 3. Basic merge situations.

enumerate the situations where the merge operator is applied below. We will use the code snippets in Figure 1 as concrete examples. Their associated control-flow graphs are shown in Figure 2, where the dashed nodes represent the entry/exit nodes. The unfolded versions are shown in Figure 3.

*Data-dependent conditionals.* This includes all constructs like the C-language *switch* and *if-then-else* conditionals that cannot be determined at compile time.[1] Figure 3(a) shows an example of such a case. At compile time, it is impossible to

[1]From now on, we will refer to this case as IF conditionals.

figure out which branch is going to be executed. The merge point is set in such a way that it merges the outcomes from both branches.

*Unknown number of iterations of a loop.* This situation arises when either the loop bounds are unknown or there is a jump out of the loop. Either way, a path is created for each possible number of iterations, and all of them merged later when they exit the loop (see Figure 3(b)).

Notice that these two situations can be combined. When analyzing a loop with a data-dependent conditional, we may want to merge the branches of each iteration and later, all the iterations (see Figure 3(c)).

### 2.4 Merging Operator

We now discuss how path merging impacts cache analysis.

*Definition 2* **(System State)**. For each path p, we denote its *system state* as the result of its (partial) execution, and it may include information about the program counter (which instruction is executed), memory locations, registers contents, etc.

Regarding our analysis, we are only concerned about the program counter (PC) and the cache state (CC).

*Definition 3* **(Set of Cache Lines)**. $L$ is the set of cache lines.

*Definition 4* **(Memory)**. $S$ is the set of all memory lines. To represent the absence of any block in a cache line, we introduce a new element $I \notin S$, $S' = S \cup \{I\}$.

*Definition 5* **(CC)**. The *state of the cache* is a mapping

$$CC : L \to S'$$

When considering set-associative caches, we assume that cache lines are sorted within sets according to the replacement policy.[2]

For each merge operation, one must union the *system states* of all merging paths. The merging operator $\sqcup$ that merges paths is outlined in Figure 4. The union operation for cache states is defined as usual [Alt et al. 1996; Lundqvist and Stenström 1998]:

$$a \sqcup_{CC} b = CC : L \to S' \quad l \mapsto a(l) \sqcup_L b(l)$$

$$x \sqcup_L y = \begin{cases} I \text{ if } x \neq y \\ x \text{ otherwise} \end{cases}$$

Note that the program counter of two merging paths at the merge point is the same by definition. However, the cache states do not have to be the same, thus some approximations when merging are taken. Our approach will eliminate this situation: all merging paths have the same cache state at the merging point.

[2]The current implementation only supports LRU replacement policy, which is the one adopted by all studied real-time processors.

INPUT

$\mathsf{p}_a$ = a path

$\mathsf{p}_b$ = a path

pre-condition:

$\mathsf{p}_a.state.PC = \mathsf{p}_b.state.PC$

OUTPUT

$\mathsf{p}_a \sqcup \mathsf{p}_b$ = a path

ALGORITHM

$\mathsf{p}_c$ is a path

$\mathsf{p}_c.state.PC := \mathsf{p}_a.state.PC$;

$\mathsf{p}_c.state.CC := \mathsf{p}_a.state.CC \sqcup_{CC} \mathsf{p}_b.state.CC$ ;

$\mathsf{p}_a \sqcup \mathsf{p}_b := \mathsf{p}_c$

Fig. 4. Merging operator for paths.

### 2.5 Cache Analysis Review

Given an execution path, we use *FindMisses* [Vera and Xue 2002], an analytical method which builds on the top of the Cache Miss Equations [Ghosh et al. 1999]. In order to get the best performance from the cache, we should try to lock it as few times as possible. Besides, each locked region should be as small as possible. Thus, the more constructs we can analyze statically, the better. We have extended Cache Miss Equations to make whole-program analysis feasible. *FindMisses* sets up a set of equalities and inequalities that describe the cache behavior of whole programs with regular computations, which may consist of subroutines, call statements, if statements and arbitrarily nested loops. This provides a framework that can be integrated into any static tools, like compiler optimizations or WCET analysis tools.

*Abstract Call Inlining* [Vera and Xue 2002] allows us to quantify the impact of subroutine calls precisely. It works in two steps:

(1) It models the accesses to the runtime stack, so we can handle conflicts between stack locations and accesses to arrays and other major data structures. The runtime stack is modeled as a 1-D array, and then treated just like an ordinary array.

(2) It transforms the references to dummies so that the information of the matching actuals is incorporated into the new references.

Notice that library and operating system calls whose source code is not available cannot be analyzed. Thus, in order to have a predictable code, we will have to lock the cache for that particular call.

The following restrictions define the memory references that are analyzable statically:

—The base addresses of all non-register variables including actual parameters (scalars or arrays) must be known at compile time.

—The sizes of an array in all but the first[3] dimension must be known statically.

[3]This applies to C codes. For FORTRAN77 codes it would be the last dimension.

In order to ensure that our analysis can be done in the polyhedral model [Feautrier 1996; Xue 2000], we add the following constraint:

—The subscript expressions of array references and loop bounds are affine.

If any of the previous conditions does not hold, we lock the cache (and load it with data likely to be accessed) as we do when we place merging operator, thus ensuring that the cache contents are always known. We discuss it in detail in Section 3.2.2.

**Impact of replacement policies.** *FindMisses* assumes a $k$-way set-associative cache with LRU replacement policy. However, some processors have caches with less predictable replacement policies, such as pseudo-random or pseudo-LRU. If one has the description of the replacement policy (i.e., its behavior can be reproduced in *software*), it can be incorporated into *FindMisses*. Otherwise, the analysis will not guarantee hits or misses for all memory accesses. Moreover, we will not be able to guarantee a safe WCET.

### 2.6 Integration in a WCET tool

Some WCET tools integrate the cache analysis into the calculation phase. For instance, Li *et al* [Li et al. 1996] describe cache behavior as a set of constraints, which are solved as part of the calculation phase. A different approach is to perform a separate cache analysis phase to determine the cache behavior, and then use this information in the calculation phase.

Since our cache analysis [Vera and Xue 2002] describes cache behavior by means of linear constraints and details cache behavior for every memory access, it can be used in both situations. One may decide to add the linear constraints to the description of the WCET and solve them as part of the calculation phase, or may decide to solve them beforehand, and use that information in the calculation phase. We plan to integrate our approach to an IPET-based WCET tool such as [Engblom and Ermedahl 2000], where equations will be solved for any path that the WCET tools requests.

## 3. PREDICTABLE CACHE BEHAVIOR

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 [Basumallick and Nielsen 1994; Busquets-Mataix et al. 1996; Lee et al. 1998]. They basically assume 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 schedula-

```
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 := ∅;                                   // set of modified tasks
  for each  task T_i ∈ S
     CP_i is T_i's cache partition
     P_aux:= LockAndLoad(T_i);                  // modified task after locking
     P_aux:= CacheOptimize(P_aux, CP_i);
     S_aux.insert(P_aux);

  PredictMultiTask(S, C) := < S_aux, CP >
```

Fig. 5. An algorithm for obtaining a predictable set of tasks on a multitasking system.

bility analyses can be applied without modifications. We propose the use of cache partitioning for avoiding inter-tasks conflicts. This allows us to compute the WCMP of each task in isolation. We combine it with some compiler cache optimizations (such as tiling and padding) to reduce the loss of performance due to the use of a smaller cache. When calculating the WCMP of a task, we use *FindMisses* combined with dynamic cache locking. We first transform the program issuing lock/unlock instructions to ensure a tight WCMP estimate at static time. In order to keep a high performance, load instructions are added when necessary.

*PredictMultiTask* given in Figure 5 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 *FindMisses* for all possible execution paths, obtaining an *exact* WCMP for each transformed task. Below we explain in detail the different parts of the algorithm in separate sections. We first discuss the implications of using the cache partitioning technique. Then, we outline how we solve the problem of predictability for data caches. In order to optimize the cache behavior of tasks, we have implemented padding and tiling as described in [Vera et al. 2003]. Thus, the performance of the tasks is not jeopardized.

### 3.1 Cache Partitioning (CreatePartitions)

Inter-task interference occurs when memory accesses from different tasks conflict in cache (i.e., different tasks use the same cache lines and thus, a task may evict data that have been brought by another task), which causes unpredictability. Cache partitioning [Kirk 1989] divides the cache into disjoint partitions, which are assigned to tasks in such a way that inter-conflicts are removed.

Let $\{T_1, \ldots, T_n\}$ 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,

INPUT
$P$ = a program

OUTPUT
$LockAndLoad(P)$ = a predictable program

ALGORITHM
$P\_aux$ is a program

$P\_aux := LockMergingPoints(P);$
$P\_aux := LockDataDependent(P\_aux);$
$P\_aux := OptimizeLock(P\_aux);$
$P\_aux := LoadData(P\_aux);$

$LockAndLoad(P) := P\_aux$

Fig. 6. An algorithm for obtaining a predictable program.

where $p$ is the number of different priorities.

Cache partitioning can be implemented by either software [Müeller 1995; Wolfe 1993] or hardware [Kirk 1989]. Both techniques restrict 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.[4] The software approach requires compiler and linker support [Müeller 1995], which are responsible for relocating data to provide exclusive mappings on the cache for each task.

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.

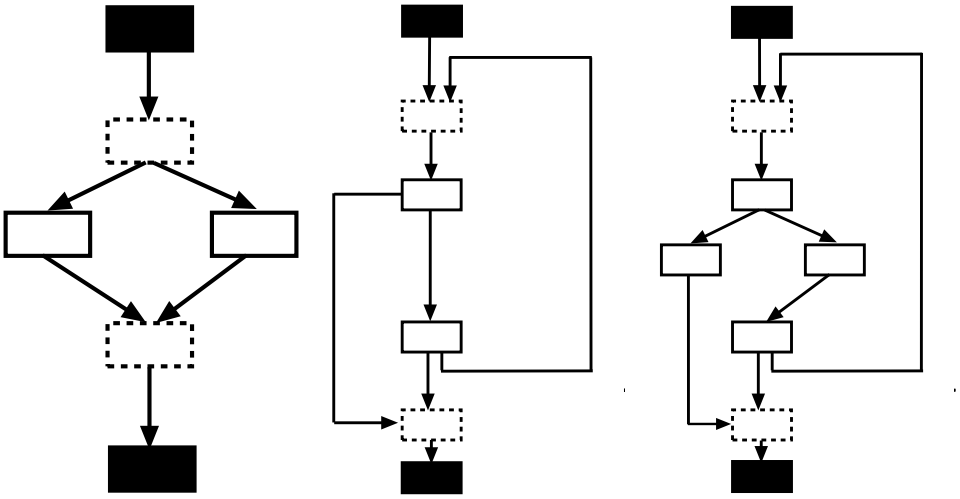
Our approach works with both hardware and software mechanisms, and it does not depend on the size of the partitions created. From now on we assume that the cache is divided in $n$ equally-sized partitions, one for each task. Such a simple approach is good enough for scheduling real sets of tasks and 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.

## 3.2 Dynamic Cache Locking (LockAndLoad)

*LockAndLoad* given in Figure 6 takes as input a general program, and generates an analyzable program with the same semantics. In this section, we explain in detail the different parts of the algorithm to have a predictable program.

3.2.1 *Path Merging (LockMergingPoints).* 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. We have shown in Section 2.2 how we manage to bound the number of analyzed paths to just a few

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



(a) if Construct (b) Loop Construct (c) Loop with if

Fig. 7. Control-flow graphs with lock/unlock nodes for examples in Figure 2. Black boxes represent the lock/unlock nodes with the lock/unlock instructions.

by means of path merging. However, merging paths leads to 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 [Alt et al. 1996; Lundqvist and Stenström 1999a].

When a memory access is analyzed on the pipeline and it cannot be classified as a hit or a miss, both situations should be analyzed to identify the longest path [Engblom and Ermedahl 1999]. This translates to larger analysis times. In order to avoid an unknown state of the cache due to merging, we lock the regions where paths are merged.

*LockMergingPoints* makes use of the control-flow graph of the program. It inserts lock/unlock instructions (lock/unlock nodes in the control-flow graph) exactly for the situations described in Section 2.3:

—The lock instruction is placed at the top of the entry node of the IF or loop construct.

—The unlock instruction is placed after the exit node of the if or loop construct. For loops with multiple exits, an unlock instruction is placed for each possible exit. Without loss of generality, from now on we only consider loops with one exit.

Figure 7 shows the resulting control-flow graphs after the lock/unlock instructions have been inserted for the control-flow graphs shown in Figure 2. Figure 8 shows the codes with the lock instructions for the corresponding codes in Figure 1.

3.2.2 *Data Cache Locking (LockDataDependent).* We have discussed in Section 2.5 the situations where *FindMisses* cannot accurately predict the cache behavior for an execution path. This includes indirection arrays (e.g., a[b[i]]), variables allocated dynamically (e.g., mallocs), pointer accesses that cannot be determined

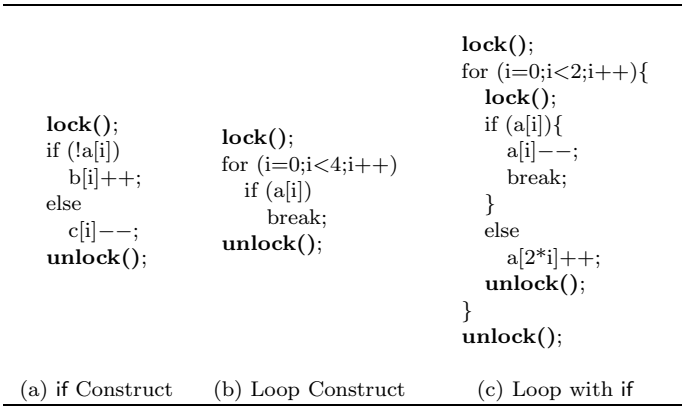



Fig. 8. Non-analyzable codes with lock instructions.

statically, and non-linear array references (e.g., a[i*j]). We also include library and operating system calls whose source code is unavailable. We obtain a tight prediction of the WCMP by automatically locking (and loading if necessary) the cache.

When a non-analyzable memory reference is found, we place a lock instruction at the top of the control-flow node where the memory access is realized. The corresponding unlock instruction is placed at the bottom. When memory references are found within a loop, we lock the cache for the loop nest, thus minimizing the overhead of executing lock/unlock instructions.

In order to obtain an accurate WCMP of a task with library calls, we would need to analyze the source code of the library to generate annotations that would help our analysis. Otherwise, to just ensure that those calls do not interfere with our analysis, we lock the cache before each call statement and unlock it afterwards. A lock instruction is placed at the top of the control-flow node where the call statement is placed. An unlock instruction is placed at the bottom. Unfortunately, the memory accesses within the library call will not be guaranteed as hits or misses, thus both situations will be analyzed in the pipeline analysis of the WCET calculation.

3.2.3 *Optimizing Placement of Lock/Unlock Instructions (OptimizeLock).* Automatic placement of lock/unlock instructions may cause performance degradation for a program. On one hand, the execution of lock/unlock instructions incurs a run-time overhead. This can be particularly so for instructions placed within loops, since they will execute several times. On the other hand, locking the cache when it is not necessary usually worsens performance.

Let us consider the code in Figure 8(c). At first sight, the lock/unlock instructions within the loop nest are unnecessary, due to the lock/unlock instructions placed at the entry/exit of the loop respectively. If we assume that each lock/unlock executes in 1 cycle, removing the unnecessary instructions will reduce the number of cycles needed to run the code by $2*k$, where $k$ is the number of iterations. For our example, we would save 4 cycles.

*OptimizeLock* goes through the control-flow graph looking for redundant lock/unlock instructions. It is an algorithm that keeps iterating while some progress is done.

```
lock();
lock();
for (i=0;i<2;i++){
  if (a[i]){
    a[i]−−;
    break;
  }
  else
    a[2*i]++;
}
unlock();
unlock();
```

(a) After 1st iteration

```
lock();
for (i=0;i<2;i++){
  if (a[i]){
    a[i]−−;
    break;
  }
  else
    a[2*i]++;
}
unlock();
```

(b) After 2nd iteration

```
lock();
for (i=0;i<2;i++){
  if (a[i]){
    a[i]−−;
    break;
  }
  else
    a[2*i]++;
}
unlock();
```

(c) Final code

Fig. 9. Application of *OptimizeLock* on the code in Figure 8(c).

We have currently implemented the following optimizations, where "`;`" represents arcs between nodes in the control-flow graph.

*Rule 1.* Lock/unlock instructions that lock the whole loop body 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.* Inspecting the memory accesses for the different outcome branches of an IF statement may allow us to detect that the memory accesses are actually the same, thus we do not have to distinguish among them.

*if* `S1`.memory_accesses=`S2`.memory_accesses *then*
`lock;if;then S1;else S2;unlock` ⇛ `if;then S1;else S2`

We show in Figure 9 the results of running the code in Figure 8(c) through *OptimizeLock*. The first iteration applies Rule 1, whereas in the second iteration it uses Rule 2 to remove the innermost locked region. Finally, it stops at the third iteration since no further changes are done. The final code is shown in Figure 9(c).

Whereas Rules 1–4 are beneficial, overhead in deeply nested loops may be large if we cannot lift lock/unlock instructions to higher levels. Thus, we define two extra rules that may allow to lock whole loop bodies by moving lock instructions:

*Rule 5.* Move a statement past a lock instruction.

`S1;lock;S2;unlock` ⇛ `lock;S1;S2;unlock`

*Rule 6.* Move an unlock instruction past a statement.

`lock;S1;unlock;S2` ⇛ `lock;S1;S2;unlock`

```
INPUT
  P    =    a program [with locked regions]

OUTPUT
  LoadData(P)    =    program with selective
                      load instructions

ALGORITHM
1   P_aux := P is a program
2   Lock(P) := locked regions of P;
3   for each locked region L ∈ Lock(P)
4       NAV(L) := non-analyzable variables ∈ L;
5       AV(L) := analyzable variables ∈ L;
6       P_aux := IssueInstrForAV(P_aux, AV(L));
7       P_aux := IssueInstrForNAV (P_aux, NAV(L));
8   LoadData(P) := P_aux
```

Fig. 10. Algorithm for selective loading.

Unlike Rules 1–4 that do not modify cache behavior, these last two rules may not always be beneficial. If data accessed in the newly locked statements is already in cache, then these transformations do not hurt performance. However, they may create later opportunities for other optimizations. The automatic placement of lock/unlock instructions to achieve the best performance remains as future work.[5]

3.2.4 *Selecting Data to Lock in the Cache (LoadData).* The benefit of cache locking is clear from the predictability point of view. Locking the cache allows us to analyze data-dependent constructs while not jeopardizing the analysis of the forthcoming code. However, locking disables the normal behavior of the cache, and hence programs may not hide the memory latency if the data they access is not in cache.

In order to overcome this problem, we can load the cache with data likely to be accessed. Nevertheless, determining accurately which data in the cache gives best performance is too expensive; it would be the same as knowing, before running the program, the most frequently accessed memory lines for each cache set. However, we can use a simple analysis based on reuse analysis [Sánchez et al. 1997; Wolf and Lam 1991] to determine which data to load, if any, for those variables that are statically analyzable.

Figure 10 outlines *LoadData*, the algorithm we use to load the cache selectively. The for in line 3 analyzes all locked regions sequentially. For each locked region, it collects all variables that are accessed within it, classifying the variables depending on whether they have data-dependent (non-analyzable) accesses or not (analyzable). For a thorough understanding of reuse vectors, we refer the interested reader to previous work [Wolf and Lam 1991; Vera et al. 2004].

3.2.5 *Loading Analyzable Variables.* We first study those variables whose memory accesses are statically analyzable. Since we want to maximize the locality,

[5]This is equivalent to optimize the conversion of data between two formats which is known to be NP-hard.

```
INPUT
  P   =   a program
  S   =   set of variables
OUTPUT
  IssueInstrForAv(P,V)   =   program with selective
                             load instructions
ALGORITHM
1   P_aux := P is a program
2   for each variable V ∈ S (in desc. order)
3       UGR(V) := classify references ∈ V in UGR classes;
4       for each reference class R ∈ UGR(V) (in desc. order)
5           if (R has no locality)
6               P_aux := AddLoads(P_aux,R);
7   IssueInstrForAv(P,V) := P_aux
```

Fig. 11. Algorithm for issuing load instructions for analyzable variables.

*IssueInstrForAV* in Figure 11 analyzes variables in descending order; the variable that has more memory references is going to be allocated first. In order to decide which memory lines to load, we compute, for each variable, the range of addresses that it accesses. When analyzing array variables, we use the concept of *uniformly generated references (UGR)* [Gannon et al. 1988] to decide which part of the array is accessed within the region. Two references are called *uniformly generated* when their array subscripts are affine and differ at most in their constant terms [Gannon et al. 1988].[6] At line 3 we classify all memory references to the studied variable $V$ into uniformly generated classes.

We estimate the amount of data that can be reused from outside the locked region using the reuse vectors. The for in line 4 studies all UGR classes, again in descending order, giving priority to those that have more references. Our algorithm is a simple volume analysis based on reuse vectors (line 5). It is a modified version of those proposed previously [Sánchez et al. 1997; Wolf and Lam 1991] in order to handle locked regions.

If we detect that some elements will not be in cache when we lock the cache, *AddLoads* includes the necessary load instructions to place them in cache. For large data sets, we may try to load a memory line that maps to a cache set that is already full. In those cases, we do not reload it since it has been loaded by a variable with higher locality.

3.2.6 *Loading Non-Analyzable Variables.* A precise approach to computing WCET uses global information (such as cache behavior) as input to the local low-level analysis, simulating the result of the cache miss/hit on the actual execution of instructions in the processor pipeline. However, when the result of a cache access is unknown, both possible results have to be simulated.

Our *IssueInstrForNAV* analyzes those variables that have non-analyzable accesses. Since it is not possible to determine at compile time which part of the array

[6]For instance, reference $a[i][j]$ is uniformly generated with respect to references $a[i][j+1]$ and $a[i-1][j]$, but not with respect to reference $a[j][i]$.

```
INPUT
  P   =   a program
  S   =   set of variables
OUTPUT
  IssueInstrForNAV(P,V)   =   program with selective
                              load instructions
ALGORITHM
1   P_aux := P is a program
2   for each variable V ∈ S (in desc. order)
3       if (V does not fit in cache)
4           P_aux := InvalidateArray(P_aux,V);
5   for each variable V ∈ S (in desc. order)
6       if (V fits in cache)
7           P_aux := LoadArray(P_aux,V);
8   IssueInstrForNAV(P,V) := P_aux
```

Fig. 12. Algorithm for issuing load instructions for non-analyzable variables.

is accessed, we assume that the whole array is accessed. Besides, since we want to avoid those situations where a cache access cannot be classified, we load the whole array if there is space in the cache. Otherwise we remove all elements present in cache, thus ensuring cache misses for all accesses to that array. This algorithm is illustrated in Figure 12. It first makes room in the cache invalidating all data from variables that do not fit in cache (lines 2–4). Then, it tries to load into cache those variables that do fit in cache.

3.2.7 *Architectural Support.* Several commercial processors (such as (PowerPC 604e [Motorola Inc. 1996], 405 and 440 families [IBM Microelectronics Division 1999], Intel-960, some Intel x86, Motorola MPC7400 and others) offer the ability to load and invalidate cache lines selectively, with *cache fill* and *invalidate* instructions respectively. Thus, no special hardware support is necessary to implement our *LoadData* algorithm.

However, both of them could be "simulated" in software if necessary, even though at the cost of some performance loss.

3.2.8 *Remarks.* In the discussions so far, we have ignored the effects of possible conflicts with memory accesses coming after the locked region. It may happen that due to the added load instructions, a memory line that otherwise would have been reused later is flushed out from cache, thus worsening cache behavior for that particular access. This would cause, in the worst case, one miss per each cache line. However, keeping those lines could cause a poor performance for the locked region. Achieving the best overall performance (i.e., deciding which memory lines to load taking into account the whole program) is a challenging problem that we plan to address in the future.

### 3.3 Putting It All Together

In this subsection, we will use the code in Figure 13(a) to illustrate how *Lock-DataDependent* and *LoadData* work. We assume, for this example, a 4KB direct-mapped cache with 16B per line.

```
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];
for (i=0;i<100;i++)
   if (c[i]>15)
      k++;
   c[i]=0;
```

Non-analyzable constructs:
*b[a[i]]*
*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);
lock(); /*region 1*/
for (i=0;i<100;i++)
   c[i]=b[a[i]]+c[i];
unlock();
for (i=0;i<100;i++){
   register int temp=(c[i]>15);
   lock();/*region 2*/
   if (temp)
      k++;
   unlock();
   c[i]=0;
}
```

(b) Code with lock/unlock

Fig. 13. A code before and after applying *LockDataDependent* and *OptimizeLock*.

| Anal. Vars | Refs | Locality | Load |
|---|---|---|---|
| c: | *c[i], c[i]* | N/A | YES |
| a: | *a[i]* | YES | NO |

⇓ UGR classes for $c$ ⇓
*c[i]*

⇓ $c$ has not locality ⇓
Issue loads for *c[i]*, $i = 0 \ldots 99$

⇓ UGR classes for $a$ ⇓
*a[i]*

⇓ $a$ has locality ⇓
Do not issue loads

| Non-Anal. Vars | Refs | Fits |
|---|---|---|
| b: | *b[...]* | YES |

⇓ $b$ fits the cache ⇓
Issue loads for $b$, $i = 0 \ldots 99$

Fig. 14. Detailed steps for the *LoadData* execution for region 1.

We start running *LockDataDependent*, which detects those constructs that are not analyzable at compile time and places lock/unlock instructions. After having applied *OptimizeLock* to avoid unnecessary locks/unlocks at every iteration, we obtain the code shown In Figure 13(b).

The next step consists in running *LoadData* to decide which data to load. We summarize the steps applied when analyzing the first locked region in Figure 14.

We start studying the analyzable variables for each region. For the first region, it identifies two analyzable variables, $c$ and $a$. It first analyzes $c$ since it has two references, and then $a$, determining that $c$ is not in cache yet. Thus, it issues the corresponding load instructions. Using the reuse vectors, we detect temporal locality between the two occurrences of *a[i]*, and the volume analysis says that

```
int a[100], b[100];
int c[100], k=0;

for (i=0;i<100;i++)                lock(); /*region 1*/
   a[i]=99-i;                      for (i=0;i<100;i++)
load(c[0]);                           c[i]=b[a[i]]+c[i];
load(c[4]);                        unlock();
⋮                                  for (i=0;i<100;i++){
load(c[92]);                          register int temp=(c[i]>15);
load(c[96]);                          lock();/*region 2*/
load(b[0]);                           if (temp)
load(b[4]);                              k++;
⋮                                     unlock();
load(b[92]);                          c[i]=0;
load(b[96]);                       }
```

Fig. 15. Transformed code with lock/unlock and load instructions.

neither access will flush the datum accessed out from the cache. A similar analysis is performed for the second region, determining that $k$ is already in cache due to the initialization.

Then, the non-analyzable variables for each locked region are analyzed. Our approach identifies variable $b$ in the first locked region, which is accessed for the first time. Assuming that there are no interferences, there is enough space in the cache to load $b$.

Eventually, the worst-case memory performance will be computed. Figure 15 shows the final transformed code. With the information of when a memory access is to be a miss/hit, we compute that the longest path is the one where *c[i]>15* holds in all instances. It results in 26 misses due to first accesses to $k$ and $a$, 50 misses due to the loading of $b$ and $c$ and 775 hits. In case that array $b$ did not fit the cache, we would estimate all its accesses as a miss, since we would not know the memory lines being accessed (besides, we would have invalidated array $b$ since *FindMisses* would not take advantage of it).

## 4. EXPERIMENTAL RESULTS

We now present results from our simulation studies. We first introduce the cache architectures simulated. Then, we present simulation results from our approach.

We start analyzing dynamic cache locking. We first evaluate the accuracy of our static data cache analysis when adding the locking features. Then, we analyze the efficiency of our loading algorithm for reducing the performance degradation due to the lock/unlock instructions. Next, we present our estimated `WCMP` for the set of benchmarks for different architectures.

As a second step, we discuss the impact of partitioning the cache on the system's throughput. Later, we compare the performance of different methods that ensure predictability when applied to partitioned caches. Finally, we show the worst-case

| Name | Freq. | L1(C,L,k) | H/M |
|---|---|---|---|
| microSPARC II-ep | 100MHz | (8,16,1) | 1/10 |
| PowerPC 604e | 300MHz | (16,32,4) | 1/38 |
| MIPS R4000 | 250MHz | (16,16,1) | 1/40 |
| IDT 79RC64574 | 200MHz | (32,32,2) | 1/16 |

Table II. Microprocessors and microcontrollers used for the experimentation. `C` stands for cache size in KB, `L` stands for cache line size in bytes, and `k` stands for the degree of associativity. H/M is the hit/miss cycles for each cache level.

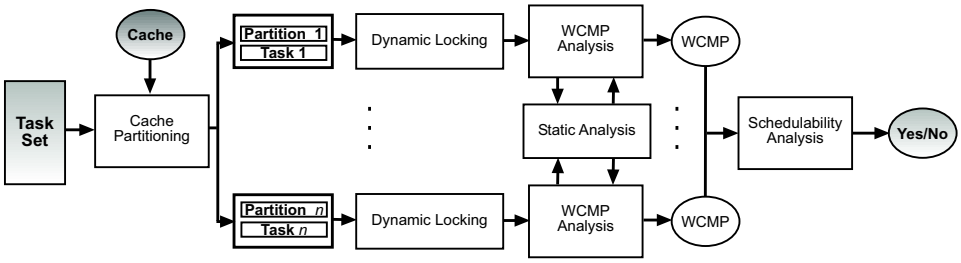



Fig. 16. A framework for worst-case performance computation.

performance when our method is applied, and compare it with static data cache locking [Puaut and Decotigny 2002].

### 4.1 Experimental Framework

Embedded systems are commonly considered very specific systems based on a microprocessor with memory integrated on the chip. Nowadays, emerging areas like car computers demand real-time capabilities but at a relative low cost. Yet, some of these new applications require a high throughput.

Microprocessor companies are moving from 4-, 8- and 16-bit processors to the current 32-bit architectures. This includes the new ARM processors, some downscaled x86, and simplified PowerPC and MIPS processors. Caches are mainly used on many high-end embedded 32-bit processors. We have conducted experiments for data caches commonly used in real-time systems.

The cache configurations and access times are the ones specified in Table II. We name the cache configurations after the processors that include them, and latencies have been obtained from vendors' specifications. Otherwise, we make clear which cache architecture is used. Each instruction to load the cache is treated as a normal memory access. We present results in terms of `WCMP`. Integrating our approach into a WCET tool remains as future work.

Figure 16 depicts the framework used for computing the worst-case performance and studying the schedulability of a task set. 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 used our *FindMisses* algorithm (see Section 2.5) to obtain precisely which memory accesses result in a miss. We present the performance of our approach for two real task sets. We

| Name | Workload (bytes) | WCMP (no cache) | Period (Normal) | Period (HP) |
|---|---|---|---|---|
| **Large Task Set** | | | | |
| MM | 120000 | 153140000 | 117800000 | 102093333 |
| SRT | 8000 | 113925998 | 159496397 | 227851996 |
| FIB | 16 | 7790 | 155800000 | 3895 |
| FFT | 8192 | 1655808 | 152334336 | 3311616 |
| **Medium Task Set** | | | | |
| CNT | 40000 | 1140000 | 570000 | 285000 |
| SQRT | 16 | 5360 | 241200 | 2680 |
| ST | 16000 | 532000 | 266000 | 266000 |
| NDES | 960 | 220938 | 331407 | 110469 |

Table III. Task Sets used.

| Name | Description | Workload |
|---|---|---|
| MM | Multiply two 100x100 Int matrices | 120000 |
| CNT | Count and sum values in a 100x100 Int matrix | 40000 |
| ST | Calc Sum, Mean, Var (2 arrays of 1000 doubles) | 16000 |
| SQRT | Computes square root of 1384 | 16 |
| FIB | Computes the first 30 Fibonacci numbers | 16 |
| SRT | Bubblesort of 1000 double array | 8000 |
| NDES | Encrypts and decrypts 64 bits | 960 |
| FFT | Fast Fourier transformation of 512 complex numbers | 8192 |

Table IV. Real-time benchmarks used. Workload is expressed in bytes

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 locking. An overview of the two task sets can be seen in Table III. The programs are introduced in Table IV. They are all written in C, drawn from different real-time papers that analyze data cache behavior [Alt et al. 1996; Kim et al. 1996; White et al. 1997]. 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[7] and amount of data is the same for each task set. We have chosen a CPU utilization of 2.03 for the **large** task set and 4.69 for the **medium**. For the *HP* (high performance) periods, the CPU utilizations for the **large** and **medium** task sets are 4.5 and 10.0, respectively, so that tasks will have higher throughput. As a result, the task sets will not be feasible if a data cache is not used for any of the period configurations.

## 4.2 Dynamic Cache Locking

The goal of using data cache locking is to eliminate unpredictability by locking those regions in the code where a static analyzer cannot be applied. However, cache locking may cause degradation in performance, which we try to avoid by means of loading the cache with data likely to be accessed. In order to isolate the results from

[7]In terms of our simple timing model.

| Name | C. | Unlock | Lock | Lock & Load | $\Delta_U$(%) | $\Delta_L$(%) | #Loads |
|---|---|---|---|---|---|---|---|
| SQRT | S | 158 | 330 | 158 | 108.8 | 0.0 | 1 |
| | P | 214 | 881 | 214 | 311.6 | 0.0 | 1 |
| | M | 185 | 456 | 185 | 146.4 | 0.0 | 1 |
| | I | 218 | 960 | 218 | 340.3 | 0.0 | 1 |
| SRT | S | 7507 | 7507 | 7507 | 0.0 | 0.0 | 0 |
| | P | 12285 | 12285 | 12285 | 0.0 | 0.0 | 0 |
| | M | 10513 | 10513 | 10513 | 0.0 | 0.0 | 0 |
| | I | 12787 | 12787 | 12787 | 0.0 | 0.0 | 0 |
| NDES | S | 6299 | 6992 | 6992 | 11.0 | 11.0 | 0 |
| | P | 6970 | 6970 | 6970 | 0.0 | 0.0 | 0 |
| | M | 6641 | 7361 | 7361 | 10.8 | 10.8 | 0 |
| | I | 7034 | 7034 | 7034 | 0.0 | 0.0 | 0 |
| FFT | S | 88696 | 231296 | 118544 | 160.7 | 33.6 | 256 |
| | P | 52736 | 805888 | 52736 | 1428.1 | 0.0 | 128 |
| | M | 50944 | 356480 | 50944 | 599.7 | 0.0 | 256 |
| | I | 53248 | 847104 | 53248 | 1490.8 | 0.0 | 128 |

Table V. Memory cost in cycles for the lock & load algorithm. *S* stands for microSPARC-IIep, *P* for PowerPC 604e, *M* for MIPS R4000, and *I* for IDT 79RC64574. $\Delta_U$=loss of performance with locks/unlocks but without loads and $\Delta_L$=loss of performance with locks/unlocks and loads. In both cases, the baseline is the same system with the same cache when neither locks/unlocks nor loads are used. $\Delta_U$ and $\Delta_L$ are computed with respect to the execution of the program without lock and load instructions for the cache.

those of the `WCMP` computation, we consider as a particular case the actual path that is executed (i.e., we do not use the longest path). We only analyze programs where lock/unlock and load instructions were issued. For the other programs, our flow analysis managed to identify only one path, and *FindMisses* could analyze all memory references. Thus, locking was not necessary. Compared to previous work, White *et al* [White et al. 1997] overestimate the memory cost by 10% and 17% for MM and ST respectively.

4.2.1 *Impact of Loads.* To evaluate the effectiveness of loading the cache, we first compare the memory cost of the resulting code with lock/unlock instructions against the same code extended with selective load instructions. The baseline is the execution of the program on the same system with the same cache except that neither lock/unlock instructions nor load instructions are used. In order to accurately evaluate to what extent selectively loading the cache can improve performance, we do not consider the additional cycles due to the extra loads and locks/unlocks used. The results of this experiment are shown in Table V. We can see that in the general case, locking the cache without loading it leads to a significant performance degradation, in one case as large as over 1000%. SRT exhibits almost the same performance regardless whether the load instructions are used or not. This is because when the lock regions are executed, all data accessed are already in the cache. The same happens to NDES. However, in the case of microSPARC-IIep, the small cache cannot hold all data and thus the performance drops. In the case of MIPS R4000, the performance drops since it uses a direct-mapped cache.

When loading the cache, performance degradation is usually eliminated. In those cases where there are conflicts among data accessed in the locked regions, loading

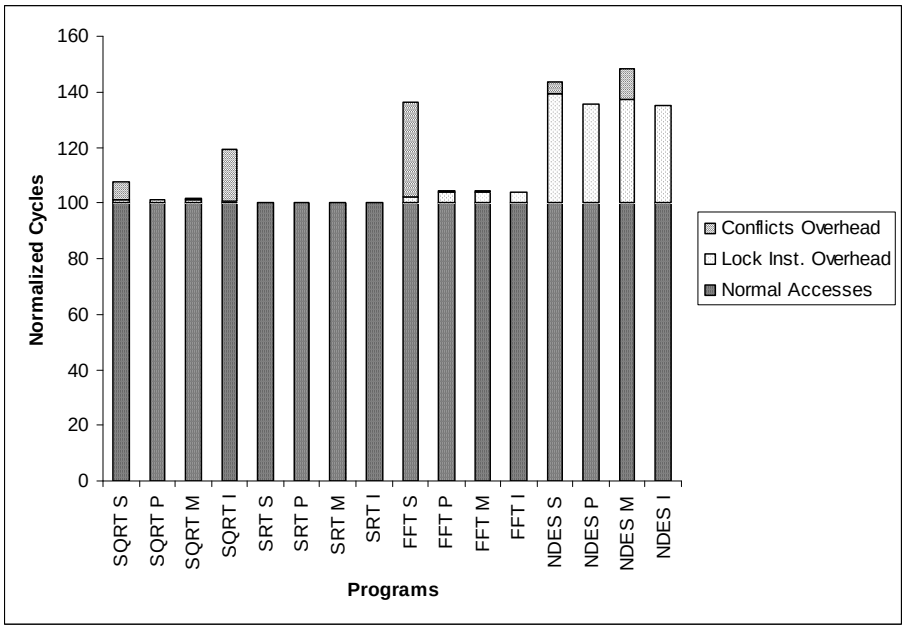



Fig. 17. Overall overhead of the cache locking. *S* stands for microSPARC-IIep, *P* for PowerPC 604e, *M* for MIPS R4000, and *I* for IDT 79RC64574.

the cache reduces the performance degradation, but it cannot eliminate it completely. Finally, the last column indicates that the reduction of memory cost can be achieved with a few selected loads issued to load the cache.

4.2.2 *Performance Evaluation.* We have evaluated the overall overhead of the resulting code in more detail. Figure 17 contains the results where cycles due to locks/unlocks and extra loads are considered. The memory cost is normalized to the memory cost of the actual execution of the program running on the same system with the same cache when neither lock/unlock nor load instructions are used. We can see that the slowdown ranges from 0% to 43%, mainly because the cache is not big enough to contain all data accessed in the locked regions. For instance, FFT has an overhead of 43% for the microSPARC-IIep architecture. When the cache size is increased, the conflicts disappear and the overhead is minimal.

NDES deserves special comments. The overhead is basically due to the lock/unlock instructions. This happens because a majority of these instructions are nested in loops, and consequently, cannot be removed by applying Rules 1–4 given in Section 3.2.3 alone. Thus, we have decided to apply Rules 5–6 to optimize the placement of lock/unlock instructions in these benchmarks. Figure 18 shows the reduced overhead for NDES when Rules 5–6 are used. The transformations as defined in these two rules have worked as *enablers*, i.e., they have enabled Rules 1–4 to achieve a better instruction placement. As a result, our algorithm has now issued only 12 load instructions each for microSPARC-IIep and MIPS R4000, and 22 each for PowerPC 604e and IDT 79RC64574. Table VI details the contributions

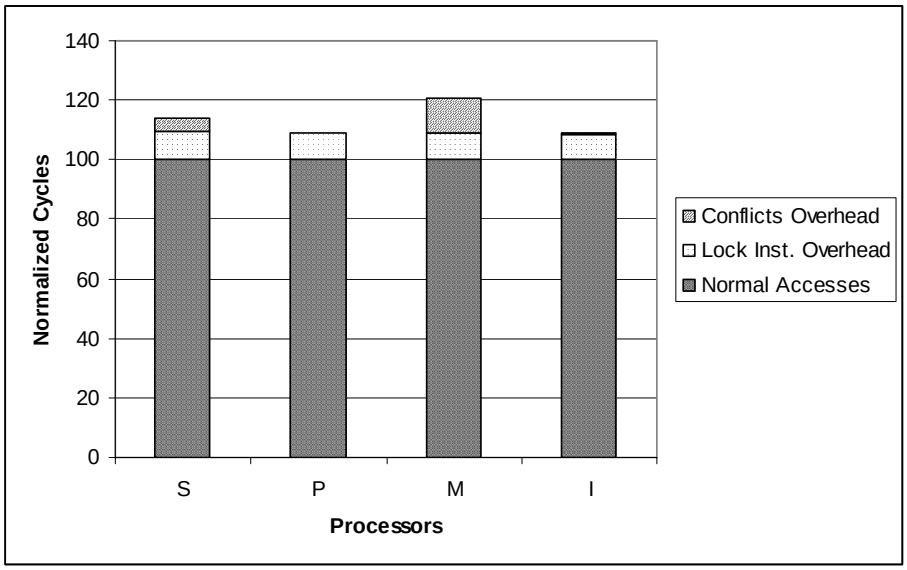



Fig. 18. Overhead of cache locking for the NDES program after the placement of lock/unlock instructions is further optimized by Rules 5–6. *S* stands for microSPARC-IIep, *P* for PowerPC 604e, *M* for MIPS R4000, and *I* for IDT 79RC64574.

| Overhead | *S* | | *P* | | *M* | | *I* | |
|---|---|---|---|---|---|---|---|---|
| | B | A | B | A | B | A | B | A |
| Locking instructions | 39.5% | 4.7% | 35.5% | 8.8% | 37.3% | 9.2% | 35.2% | 8.7% |
| Conflicts | 4.1% | 4.4% | 0.0% | 0.1% | 10.8% | 11.1% | 0.0% | 0.1% |

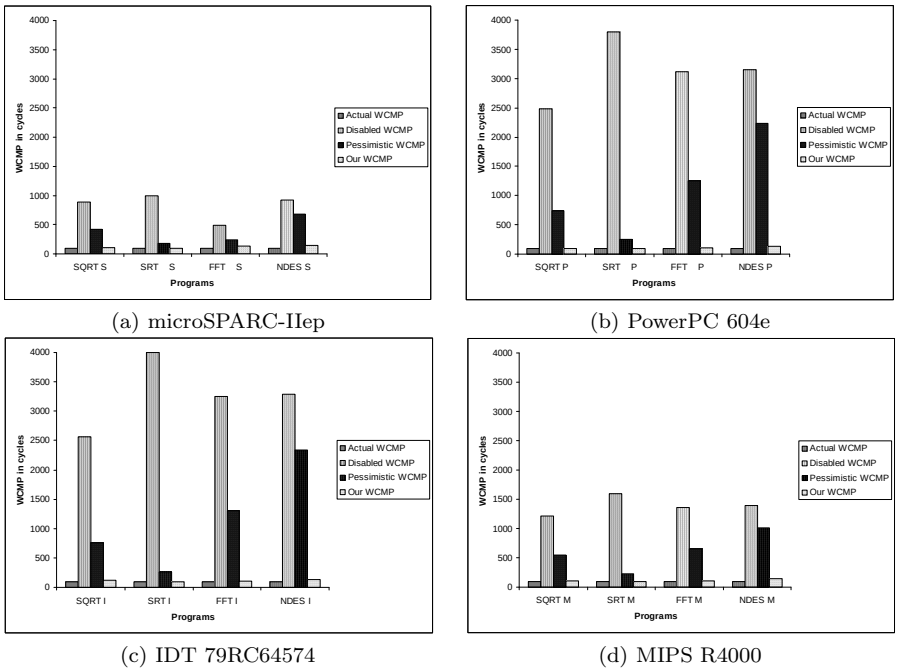
Table VI. Detailed overhead before (B) and after (A) applying Rules 5–6 for NDES. *S* stands for microSPARC-IIep, *P* for PowerPC 604e, *M* for MIPS R4000, and *I* for IDT 79RC64574.

of locking and cache conflicts to the total overhead. When Rules 5–6 are used, the overhead due to lock/unlock instructions drops, whereas the number of cache conflicts increases slightly due to the extra loads introduced.

In the following section, we show how this small performance degradation leads to a fully predictable program. Thus, we can compute the `WCMP` for a program in a tighter way than it has to be estimated pessimistically. Even though the actual execution time of the task may increase, the `WCMP` will be smaller. As a result, we will be able to make better use of resources available.

### 4.3 WCMP

Locking may sometimes increase the actual execution time, and thus the WCET of a program. However, without locking the WCET may have to be more pessimistically estimated, so that the WCET estimate without locking may be actually greater than the WCET estimate obtained with locking and loading. This section presents experimental results validating this point (at least for the benchmarks used). WCET's are important for the design and validation of hard real-time sys-




(a) microSPARC-IIep (b) PowerPC 604e

(c) IDT 79RC64574 (d) MIPS R4000

Fig. 19. Estimates of the WCMP.

tems, but since the WCET estimate is what is available it is the size of the estimate that is important in practice.

Our locking algorithm will be successful if the presence of locked regions allows us to compute a smaller WCMP than before, i.e., if

$$\texttt{WCMP}(task + locks/unlocks + loads) < \texttt{WCMP}(task)$$

In order to see the effectiveness of our approach, we have compared our method to compute WCMP with two other methods that are currently used:

—Cache disabled (i.e., cache locked all the time).

—Cache unlocked, making pessimistic assumptions whenever we do not know what happens. If the addresses of the memory accesses are unknown, we consider an empty cache where we would unlock the cache in our approach. Otherwise, we only invalidate cache lines where there is a conflict due to merging.

We use as a reference the actual WCMP of the program without lock instructions, which is obtained running the program with the worst-case input data.

Figure 19 compares the different estimates for each method. When the cache is disabled, all memory accesses are considered as misses, which yields a very large overestimation of the WCMP. The estimated WCMP is between 5 and 38 times larger than the actual one. The pessimistic approach performs better but the estimated WCMP, which is between 2 and 22 times larger than the actual WCMP, is far from tight. Our approach gives an exact WCMP of the transformed program (i.e., the program with lock/unlock and load instructions).

### 4.4 Dynamic Locking: Summary

We have shown the effectiveness of dynamic locking. Whereas some performance may be lost due to the locking mechanism (in the worst case, the program runs 0.4 times slower), we can achieve a perfect estimate of `WCMP` for the benchmarks given. Besides, we have seen that the estimate of `WCMP`(task+locks/unlocks+loads) is much smaller than the best estimate of `WCMP`(task). For those programs where lock/unlock instructions are not issued, our estimate is exact and there is no overhead.

We have presented results that highlight the accuracy of our static approach. Later, we have seen that in all cases, our selective locking technique allows us to accurately predict the cache behavior, which translates to an exact computation of the `WCMP`. We have shown that estimating the `WCMP` without the help of locking the cache is very hard, and it usually yields very large overestimates. Moreover, the knowledge of the memory behavior will allow us to compute tighter WCET.

**Analysis time** Our experiments were done on a Pentium-4 processor at 1.6GHz. The average execution time needed to analyze a cache configuration is 0.6 seconds. In particular, MM takes the longest to analyze, with 3 seconds for each configuration. The problem size is `N=100`, which means that we have to evaluate around 3 million accesses. We believe this time is reasonable for the kind of analysis performed.

### 4.5 Performance of Cache Partitioning

For analyzing the whole system, we have chosen 16KB and 32KB caches with 32B lines (like the ones of PowerPC 604e and IDT 79RC64574). For each cache, we have considered a direct-mapped cache, 2-way and 4-way set associative caches.[8] We chose the hit and miss access times after the PowerPC 604e [Motorola Inc. 1996], where each hit takes 1 cycle and each miss 38 cycles. Lock and unlock instructions take 1 cycle each.

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 20 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. We can observe that the average memory cost increases by 79% and 2470% for partitioned dynamic execution and a system without cache, respectively. 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.

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

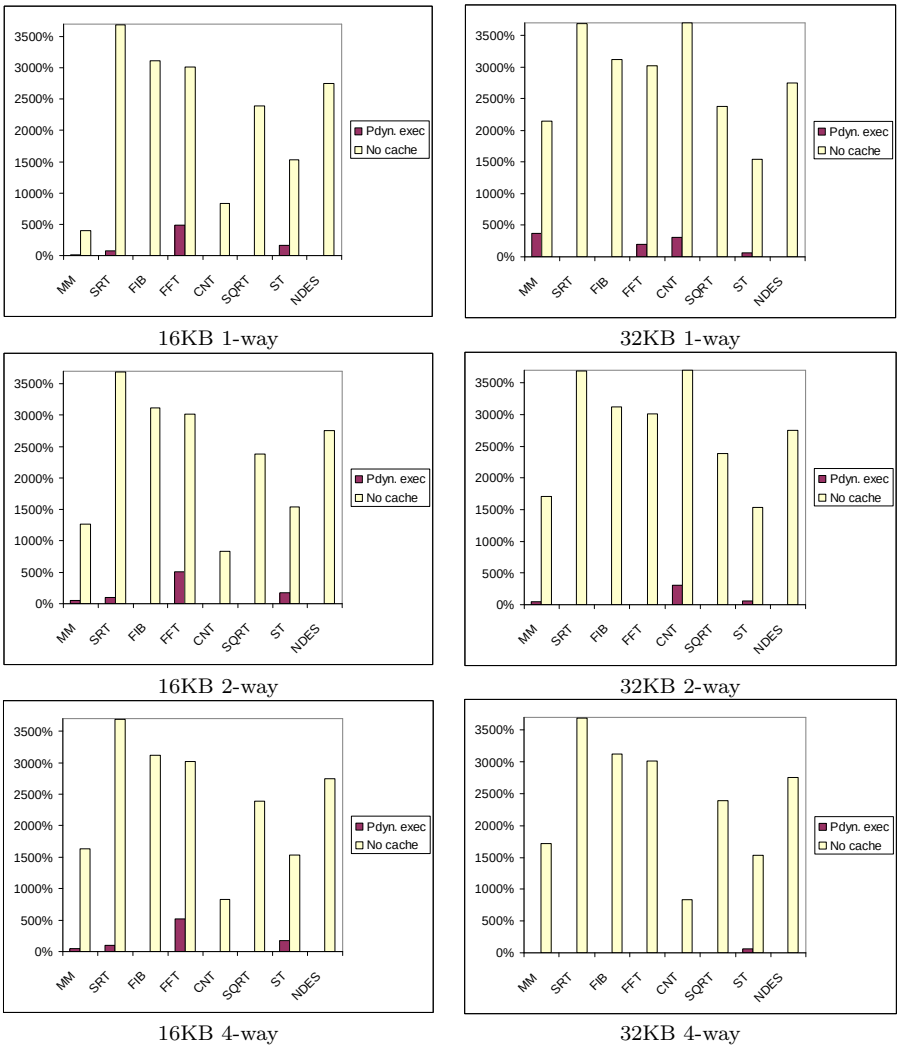



Fig. 20. Cache partitioning impact: comparison of performance degradation for a system with a partitioned cache and a system without a cache.

## 4.6 Optimizations

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 tiling and padding as shown in [Vera et al. 2003] 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

| | **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 VII. Performance of static cache locking and our cache analysis.

dynamic execution scheme drops to 189.12% and 7.23% for the 16KB cache and 32KB cache respectively. This has allowed us to schedule successfully the **large** data set on the 16KB direct-mapped cache, whereas it has reduced the CPU utilization on the other cases.

### 4.7 Worst-Case Performance: Schedulability

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 [Campoy et al. 2001; Puaut and Decotigny 2002] is applied. For that purpose, we have loaded the cache with the most frequently accessed memory lines[9] for each task set, and locked it for the entire execution (it is the same as *Lock-MU* in [Puaut and Decotigny 2002]). 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 VII. 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.

### 4.8 High-Performance Systems

Finally, we show results for a high-performance multitasking system, 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 III. Since the magnitude of the periods is very different among tasks, fixed priority systems do not perform well, and thus 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

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

| | 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 VIII. Performance of static cache locking and our cache analysis for a high-performance system.

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.

### 4.9 Cache Partitioning: Summary

We have demonstrated the effectiveness of our approach. 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. Then, we have evaluated the application of static locking and dynamic locking to ensure predictability once the cache is partitioned. 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.

## 5. RELATED WORK

During the last years, 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 [Arnold et al. 1994; Healey et al. 1995; Li et al. 1995], few steps have been done towards analyzing data caches.

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 statically 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 [Kim et al. 1996; Li et al. 1996; Lim et al. 1994; White et al. 1997].

(2) **Cache-Preemption Delays**. When a task resumes its execution, it may spend a long time reloading the cache with previously loaded cache blocks. This increases the execution time of the task, and may invalidate the results of schedulability analysis. Some studies have addressed the issue of incorporating cache preemption costs into schedulability analysis [Basumallick and Nielsen 1994; Busquets-Mataix et al. 1996; Lee et al. 1998]. 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 [Lundqvist and Stenström 1999b]. 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 [Motorola Inc. 1996], 405 and 440 families [IBM Microelectronics Division 1999], 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 [Puaut and Decotigny 2002].

(4) **Cache Partitioning**. These techniques [Busquets-Mataix et al. 1997; Kirk 1989; Liedtke et al. 1997; Müeller 1995] 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 significant loss of performance.

Now, we describe the most relevant approaches in detail. Alt *et al* [Alt et al. 1996; Ferdinand and Wilhelm 1999] 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* [Lim et al. 1994] present a method for computing 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* [Kim et al. 1996] 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.* [Li et al. 1996] describe 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* [White et al. 1997] consider direct-mapped caches based on static simulation by categorizing 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. As a result, 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 [Lundqvist and Stenström 1999a] 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* [Campoy et al. 2001] 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 [Puaut and Decotigny 2002] extend it by introducing two polynomial algorithms to select the instructions to lock in cache.

## 6. CONCLUSIONS

We have introduced an approach that 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.

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 performance. Combined with a timing analysis platform, we may estimate a tight worst-case performance.

Overall, we contribute a new technique that provides a considerable step toward a useful worst-case execution time prediction for 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.

We believe this approach is highly attractive for hard-real time systems, where the problem sizes are not very big. Moreover, while being not really large, the compilation time can be amortized across the number of products shipped. We also believe that the higher throughput of the systems due to the smaller overestimation of the WCET may make this approach very useful. 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.

While this work represents an important step towards program predictability in presence of data caches, there are still some issues that can be investigated further. Pointer analysis can be used to determine some pointer values [Wilson 1997]. Besides, programmer annotations may be used to tighten the analysis. It may also be interesting to take into account the overall performance when inserting lock/unlock instructions and selecting data to lock in the cache. We plan to investigate these research directions in order to have fully predictability and better performance.

ACKNOWLEDGMENT

The authors thank Ebbe for providing guidance for much of this research. We are obliged to Jakob Engblom for his technical comments. We are grateful to Thomas Höveken from NEC Electronics (Europe) for supplying information about the NEC controllers. Many thanks to Janne and Jan Gustafsson for reading earlier drafts of this work.

We especially appreciate the countless contributions of Nerina Bermudo, without whom this would still be far from plausible.

REFERENCES

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

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

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

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

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

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

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

Campoy, M., Ivars, A. P., and Busquets-Mataix, J. V. 2001. 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)*.

Engblom, J. and Ermedahl, A. 1999. Pipeline timing analysis using a trace-driven simulator. In *Proceedings of 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*.

Engblom, J. and Ermedahl, A. 2000. Modeling complex flows for worst-case execution time analysis. In *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS'00)*.

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

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

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

Gannon, D., Jalby, W., and Gallivan, K. 1988. Strategies for cache and local memory management by global program transformations. *Journal of Parallel and Distributed Computing 5*, 587–616.

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

Gustafsson, J. 2000. Analyzing execution time of object-oriented programs using abstract interpretation. Ph.D. thesis, Uppsala University.

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

Hennessy, J. L. and Patterson, D. A. 1996. *Computer architecture: a quantitative approach*. Morgan Kaufman Publishers.

IBM Microelectronics Division. 1999. *The PowerPC 440 core*.

Integrated Device Technologies. 2001. *79RC64574/RC64575 Data Sheet*.

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

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

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

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

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

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

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

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

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

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

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

Lundqvist, T. and Stenström, P. 1998. Integrating path and timing analysis using instruction-level simulation techniqes. In *Proceedings of ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*. 1–15.

Lundqvist, T. and Stenström, P. 1999a. 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)*. 255–262.

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

MIPS Technologies. 2001. *MIPS32 4Kp- Embedded, MIPS Processor Core*.

Motorola Inc. 1996. *PowerPC 604e RISC Microprocessor Technical Summary*.

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

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

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

Sánchez, F., González, A., and Valero, M. 1997. Static locality analysis for cache management. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT'97)*.

Sun Microelectronics. 1997. *microSPARC-IIep User's Manual*.

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

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

Vera, X., Bermudo, N., Llosa, J., and González, A. 2004. A fast and accurate framework to analyze and optimize cache memory behavior. *ACM Transactions on Programming Languages and Systems (TOPLAS) 26,* 2.

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

Vera, X., Lisper, B., and Xue, J. 2003b. Data caches in multitasking hard real-time systems. In *Proceedings of International Real-Time Systems Symposium (RTSS03)*.

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

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

Wilson, R. P. 1997. Efficient context-sensitive pointer analysis for C programs. Ph.D. thesis, Stanford University.

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

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

Xue, J. 2000. *Loop Tiling for Parallelism*. Kluwer Academic Publishers.

Xue, J. and Huang, C.-H. 1998. Reuse-driven tiling for data locality. *International Journal of Parallel Programming 26,* 6, 671–696.