Data Cache Locking for Higher Program Predictability

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

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

ABSTRACT

Caches have become increasingly important with the widening gap between main memory and processor speeds. However, they are a source of unpredictability due to their characteristics, resulting in programs behaving in a different way than expected.

Cache locking mechanisms adapt caches to the needs of real-time systems. Locking the cache is a solution that trades performance for predictability: at a cost of generally lower performance, the time of accessing the memory becomes predictable.

This paper combines *compile-time cache analysis* with *data cache locking* to estimate the worst-case memory performance (WCMP) in a safe, tight and fast way. In order to get predictable cache behavior, we first lock the cache for those parts of the code where the static analysis fails. To minimize the performance degradation, our method loads the cache, if necessary, with data likely to be accessed.

Experimental results show that this scheme is fully predictable, without compromising the performance of the transformed program. When compared to an algorithm that assumes compulsory misses when the state of the cache is unknown, our approach eliminates all overestimation for the set of benchmarks, giving an exact WCMP of the transformed program without any significant decrease in performance.

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

Keywords

Worst-Case Execution Time, Data Cache Analysis

SIGMETRICS'03 June 10–14, 2003, San Diego, California, USA.
Copyright 2003 ACM 1-58113-664-1/03/0006 ...$5.00.

1. INTRODUCTION

With ever-increasing clock rates and the use of new architectural features, the speed of processors increases dramatically every year. Unfortunately, increased memory latency affects all computer systems, being a key obstacle to achieve high processor utilization. The basic solution that almost all systems rely on is the cache hierarchy.

While caches are useful, they are effective only when programs exhibit sufficient data locality in their memory accesses. Various hardware and software approaches have been proposed lately to exploit caches efficiently. Software prefetching [26] hides the memory latency by overlapping a memory access with computation and other accesses. Another useful optimization is applying loop transformations such as tiling [4, 7, 19, 34] and data transformations such as padding [5, 16, 27, 30]. In all cases, a fast and accurate assessment of a program's cache behavior at compile time is needed to make an appropriate choice of parameter values.

1.1 Caches in Real-Time Systems

Real-time systems rely on the assumption that tasks' worst-case execution times (WCETs) are known. In order to get an accurate WCET, a tight worst-case memory performance (WCMP) is needed. However, cache behavior is very hard to predict, which leads to an overestimation of the WCMP, and thus for the WCET as well. For this reason, many safety-critical systems such as antilock brake systems do not use caches: it is very hard to prove that the system is reliable under all circumstances. For instance, the ARM966E-S processor does not have a cache in order to have predictable memory timings.

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. A system with disabled caches will however waste a lot of resources; not only will the CPU be underutilized but the power consumption will be higher. Memory accesses that fall into the cache are faster and consume less power than accesses to larger or off-chip memories.

Frameworks of WCET prediction are used to ensure that deadlines of tasks can be met. While the computation of WCET in the presence of instruction caches has progressed in such a way that makes it possible to obtain an accurate estimate of the WCET [1, 2, 13], there has not been much progress with the presence of data caches. The main problem when dealing with data caches is that each load/store instruction may access multiple memory locations (such as those that implement array or pointer accesses).

Cache locking allows some or all of the contents of the cache to be locked in place. Disabling the normal replacement mechanism, provided that the cache contents are known, makes the time required for a memory access predictable. This ability to lock cache contents is available on several commercial processors (PowerPC 604e [25], 405 and 440 families [15], 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).

Whereas loading and locking the cache offers predictability, it does not guarantee good response time of tasks (thus, we are trading performance for predictability). On the other hand, static cache analysis allows us to predict the WCMP and does not affect the performance. However, static analyses only apply to codes free of data-dependent constructs.

We introduce a method that combines static cache analysis and cache locking in order to achieve both predictability and good performance. Furthermore, it allows computing a WCMP estimate of tasks in a fast and tight way. Our approach first transforms the original program issuing lock/unlock instructions to ensure a tight analysis of the WCMP at static time. In order to keep a high performance, load instructions are added when necessary. Later, the actual computation of the WCMP estimate is performed. We present results for a collection of programs drawn from several related papers in the real-time area [1, 17, 33]. 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 present results for a direct-mapped and a set-associative cache with different cache line sizes. We have chosen the memory hierarchies of two modern processors widely used in the real-time area: microSPARC-IIep [29] and PowerPC 604e [25].

1.2 An Overview

This paper addresses the bounding of WCMP in the presence of data caches. Moreover, we want an accurate analysis that, combined with a low-level analysis (i.e., pipeline timing analysis), allows us to obtain a tight WCET. In particular, we use a static data cache analysis. We first start modifying the program, issuing lock/unlock instructions when necessary. There are typically parts of the code which are analyzable and where each access can correctly be categorized as a cache hit or miss. For some other parts in which data-dependent situations arise (such as indirection arrays) or where multiple paths can be taken, we lock the cache and load into it, if necessary, data likely to be accessed.

There are a few important concepts useful for developing a static cache analysis. 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. In order to simplify WCET computation when studying pipelined processors, we want to guarantee hits or misses for each memory access. Thus, even if we know the memory addresses of the further 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 access.

We have developed a compile-time algorithm that identifies those regions of code where we cannot exactly determine the memory accesses, and locks the cache. It uses a locality analysis based on Wolf and Lam's reuse vectors [34] to select the data to be loaded. Since the state of the cache is known when leaving the region, we can apply a static analyzer for the next regions of code, thus having both predictability and good performance.

Once the program is transformed, the static analyzer determines the worst-case memory performance. It analyzes scalars and array accesses whose subscripts are affine functions of the loop indices. We have implemented Ghosh *et al*'s Cache Miss Equations (CMEs) [12], extending its applicability following our previous work [32]. This allows us to analyze very large codes consisting of subroutines, call statements, IF statements and arbitrarily nested loops free of data-dependent constructs. We have extended this analysis in such a way that it takes into account memory accesses in locked regions, as well as the state of the cache at the entry and exit points of the locked regions.

We have implemented our system in the SUIF2 compiler. It includes many of the standard optimizations, which allows us to obtain a code competitive to product 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 to an existing WCET tool [8] that already analyzes pipelines and instruction caches. The WCET tool generates possible paths which are analyzed by the WCMP method. Finally, the cache behavior is fed back and used to compute the WCET (i.e., the longest path) of the task.

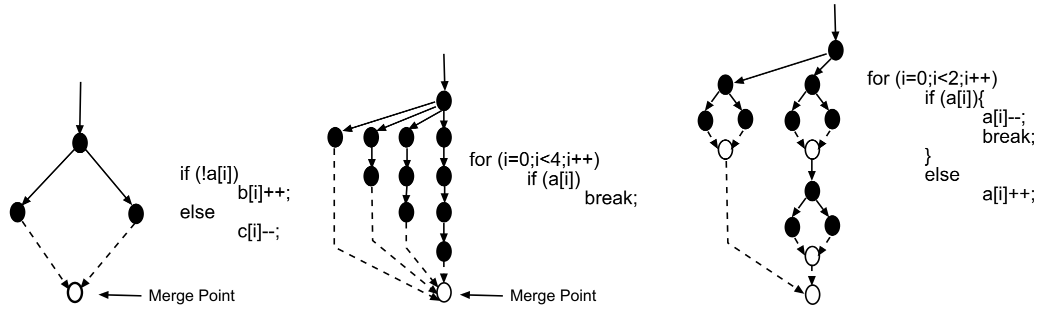
The rest of the paper is organized as follows. Section 2 reviews the flow analysis used in our approach. Section 3 describes an algorithm for having a predictable and high performance data cache. Section 4 presents our experimental framework, and Section 5 discusses our results. Section 6 contains some related works in the area that aim at analyzing the cache behavior statically and computing the WCET in the presence of data caches. Finally, we conclude and give a road map to some future extensions in Section 7.

2. MERGING OF PATHS

Real-time requirements on a system are passed on as requirements on all system parts. That implies that is necessary to know the execution time for the tasks in a real-time system. Since execution time varies, the WCET (i.e., the longest execution time for a program for all possible inputs) is used as a safe upper limit.

The analysis from a high-level point of view is concerned with the possible paths through the program. The temporal behavior of the processor is the basis which all other calculations rely on. This means that caches have to be considered. A naive approach to compute 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 causes the WCET would be a solution, but it is usually hard to know such data for regular programs. Besides, caches may give different results for two identical runs due to the previous state of the cache. Therefore, static analysis is needed.

Unfortunately, it is infeasible to analyze all possible paths. Approximations during computation must be selected so



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

Figure 1: Basic merge operations.

that path explosion is reduced: a simple loop with an IF-THEN-ELSE statement that iterates a hundred times generates 2^{100} possible paths. We use a common technique known as *merging* to 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 [9, 13, 23].

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.

Merge points can be chosen arbitrarily depending on accuracy and execution time desired. We use the following merge points when the actual control flow is unknown:

Data-dependent conditionals. Figure 1(a) shows an example of such a case. At compile time, it is impossible to 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 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 1(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 1(c)).

Some of the problems can be partially solved at compile time. To address the symbolic loop bound problem, we use interprocedural constant propagation to eliminate as many symbolic loop bounds as possible. 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.

When all else fails, we generate the control flow graph and analyze all different paths for those sections that are not statically analyzable. We lock those regions in order to avoid an unknown state of the cache due to merging: when a memory access cannot be classified as a hit or a miss, both situations should be analyzed later in the pipeline analysis.

The approach presented here merges paths exactly in the situations described above. Figure 2 shows the codes with the lock instructions for the corresponding codes in Figure 1. A later step goes through the graph looking for redundant lock/unlock instructions. Figure 2(d) shows the final code for Figure 2(c) after unnecessary lock/unlock instructions have been removed.

3. PREDICTABLE CACHE BEHAVIOR

In this section, we introduce our method to have a predictable program. We first discuss some important concepts related to data cache analysis and how we solve the problem of predictability. Then, we outline an algorithm to selectively load the cache, so that the performance is not jeopardized. Finally, we present how this approach can be used to compute the WCMP and WCET.

Understanding data reuse is essential to predict cache behavior, since a datum will only be in the cache if its line was referenced some time in the past. *Reuse* happens whenever the same data item is referenced multiple times. This reuse results in *locality* if it is actually realized; reuse will result in a cache hit if no intervening reference flushes out the datum.

Given that, a static data cache analysis can be split into the following steps:

1. *Reuse Analysis* describes the intrinsic data reuse among all different memory references[1].

2. *Data Locality Analysis* describes the subset of reuses that actually results in locality.

In the following, we describe each step and explain how we use data cache locking for those cases where this cache analysis cannot be applied.

3.1 Reuse Vectors

In order to describe data reuse, we use the well-known concept of reuse vectors [34]. They provide a mechanism for summarizing repeated memory accesses which is limited to perfect loop nests. We have extended them in a previous

[1] We use *memory reference* to note a static read or write in the program. A particular execution of that read or write at run-time is a *memory access*.

```
lock();
if (!a[i])
  b[i]++;
else
  c[i]−−;
unlock();
```

(a) IF Construct

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

(b) Loop Construct

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

(c) Loop with IF

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

(d) Loop with IF (final code)

Figure 2: Non-analyzable codes with lock instructions.

work [32, 35] so they can describe the most recent previous access (MRPA) among arbitrary loop nests[2].

Trying to determine all iterations that use the same data is extremely expensive. Thus, we use a concrete mathematical representation that describes the direction as well as the distance of the reuse in a methodical way. The shape of the set of iterations that uses the same data is represented by a *reuse vector space* [34]. Whereas self reuse (both spatial and temporal) and group temporal reuse is computed in an exact way, group spatial reuse is only considered among uniformly generated references (UGRs), this is, references whose array index expressions differ at most in the constant term [11].

Note that reuse vectors provide a pessimistic (safe) approach to describe reuse[3]. In the case that a reuse vector is not present, we assume there is no reuse, thus, there will not be locality and a cache miss will be computed.

3.2 Data Locality

Data locality is the subset of reuse that is realized; i.e., reuse where the subsequent use of data results in a hit. To discover whether a reuse translates to locality we need to know all data brought to the cache between the two accesses (this implies knowledge about loop bounds and memory access addresses) and the particular cache architecture we are analyzing.

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. CMEs [12] are mathematical formulas that provide a precise characterization of the cache behavior for perfectly nested loops consisting of straight-line assignments. Based on the description of reuse given by reuse vectors, some equations are set up that describe those iteration points where the reuse is not realized. Solving them gives information about the number of misses and where they occur. In a previous work [32], we further extended them in order to make whole program analysis feasible, by handling call statements, IF statements and arbitrarily nested loops.

Even though generating the equations is linear in the number of references, solving them can be very time consuming. Since hard real-time systems need a safe WCET bound, all different iteration points have to be analyzed. For soft real-time software with large data sets, we can use probabilistic methods based on sampling [31, 32] to solve the equations in a faster way.

3.2.1 CMEs for Cache Locking

Given a memory reference, the equations are to investigate whether the reuse described by its reuse vectors is realized or not. We now briefly discuss how we extend our analysis to caches with locking features.

For these caches, we have to treat in a different manner references within a locked region compared to those within an unlocked region. Regarding the reuse vectors, it is enough to ignore reuse vectors whose tail reference[4] is in a locked region. As those accesses within a locked region do not bring data to the cache, they cannot affect the result of future accesses. Furthermore, they cannot affect the decision of the LRU replacement policy since they do not create a recent use of a memory line.

When analyzing potential cache set contentions, references within a locked region should be ignored. Again, since they do not either bring data or modify the LRU state, they will not generate any set contention. The effect of extra load instructions is implicit in the analysis, since they are treated as usual memory accesses.

3.3 Data Cache Locking

In the discussion so far, we have ignored the effects of data-dependent memory accesses. Whereas the inability of expressing reuse leads to an overestimation of the miss ratio, the presence of data-dependent accesses makes the estimate of whether the reuse is realized or not infeasible. Furthermore, a safe approach that considered all further accesses as misses would have a large overestimation.

One approach to overcome this problem in real-time systems is to lock the cache for the whole execution of the program. Unfortunately, this translates to very poor cache utilization, especially when data does not fit the cache. In order to confirm this intuition, we have run all different programs (for a detailed description of the benchmarks, see Section 4) in two different ways: (i) unlocked cache (i.e., enabled

[2] An isolated statement is considered to be inside a loop that iterates only once.

[3] In presence of timing anomalies, some modifications have to be done in the pipeline analysis [24].

[4] Reference that brings data to cache.

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

Table 1: Comparison of performance between an unlocked cache and a locked cache loaded with the most accessed lines for programs in Table 2. *Increase* **represents the average increase in miss ratio across all architectures.** *Degradation* **stands for the the average increase in cycles across all architectures.**

cache), and (ii) loading the cache with most frequently accessed memory lines[5] and locking it. We have analyzed the following caches: 4KB, 8KB, 16KB and 32KB (32B per line) for three different associativities (direct-mapped, 2-way and 4-way). We have also simulated the microSPARC I cache architecture (direct-mapped, 512 bytes, 32B per line). We present results accounting only for load/store instructions: we assume a conservative architecture where a cache hit takes 1 cycle and a cache miss 50 cycles. Table 1 shows that the loss in performance for all different programs is significant (in some cases, it degrades more than 500% in cycles). Only in those cases where all data fits the cache such as SQRT (it only accesses a few floating point values) or SRT (when the cache is large enough to store the vector being sorted) cache locking performs well.

Initially it may appear that obtaining a reasonable bound on the WCMP when the data accessed is unknown is far from being feasible. This includes indirection arrays (e.g., a[b[i]]), variables allocated dynamically (e.g., mallocs) and pointer accesses that cannot be determined statically. However, a tight prediction of the WCMP can be achieved by automatically locking and loading the cache in those regions where we find those accesses.

Real-time codes are usually free from dynamic memory allocation. Otherwise, as long as the actual calls to the *malloc* routine and the size of the memory allocated are known at static time, it is possible to figure out where the *mallocs* go, just by keeping information about the allocated and deallocated memory in the program. If everything else fails, the only option is to lock the cache when accessing data allocated dynamically.

Pointer analysis is used to determine some pointer values, and programmer annotations can be used to tighten the analysis. When analyzing indirection arrays in a loop, we lock the cache for the loop nest. If the array being accessed fits the cache, we load it. Otherwise, we make sure that it is not in the cache by invalidating those lines that contain parts of it[6]. This allows us to (i) predict the result of the memory access, and (ii) reduce the variation of the execution time, since we cannot have a hit when we have predicted a miss (and vice versa).

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, just to ensure that those calls do not interfere with our analysis, we lock the cache before each call statement and unlock it afterwards. The memory accesses within the library call will not be guaranteed as hit/miss, thus both situations will be analyzed in the pipeline analysis.

3.4 Selecting Data to Lock in the Cache

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. Unfortunately, it may happen that the program does not benefit from locality.

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 accessed memory lines for each cache set. However, we can use a simple analysis based on the reuse vectors to determine which data to load, if any.

Figure 3 gives an outline of the algorithm we use to load the cache selectively. We begin our analysis collecting all analyzable variables that are accessed in the locked region (*l.2*). For each variable, we try to compute its range (if it is an array, it is the part of it accessed within the region) and classify all its references in uniformly generated classes (*l.6*). We estimate the amount of data that can be reused from outside the locked region using the reuse vectors. Our algorithm is a simple volume analysis based on reuse vectors (*l.9*). It is a modified version of those proposed previously [28, 34] in order to handle locked regions.

Since we want to maximize the locality, we start allocating those variables that are going to be accessed most. Itera-

[5] We collect this information running the program once and collecting statistics for each memory line accessed.

[6] We can obtain this information from our static analyzer.

```
1  for each locked region
2    R:=vector < pair <variable, memory_references>>;     // analyzable variables accessed within the region
3    RS:= sort (R,≻);                                      // the variable with more references is the first
4    for i:=0 to (RS.size()-1) do                          // it iterates over the variables
5      // UGR is a vector <uniform_generated_reference_class>
6      Compute_UGR (UGR, RS[i].memory_references());       // classes are computed
7      UGRS:= sort (UGR, ≻);                               // the class with more elements is the first
8      for j:=0 to (UGRS.size()-1) do
9        if (!Has_Locality(UGRS[j]) {
10         // Range of addresses touched is computed
11         Range:=Compute_Range_of_Variable(RS[i].variable(), UGRS[j]);
12         Load (UGRS [j], Range);                         // code is generated to load the cache
13       }
14   DD=vector <variable>;                                 // variables with data-dependent accesses within the region
15   DDS:= sort (DD,≻);                                    // the most accessed variable is the first
16   for i:=0 to (DDS.size()-1) do                         // it iterates over the variables
17     if (Fits_in_Cache(DDS[i]))
18       Load (DDS[i]);
19     else
20       Invalidate(DDS[i]);
```

Figure 3: Algorithm for selective loading. A cache architecture where individual lines within a set can be invalidated is assumed.

tively in descending order (*l.8*), we analyze the uniformly generated classes, computing the range of memory lines to be loaded (*l.11*). If the data set is larger than the cache, 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. Then, we analyze variables that have non-analyzable accesses, assuming that the whole array is accessed. If there is space in the cache (*l.17*), we load it, otherwise we remove all elements present in cache (*l.20*).

In the discussion so far, we have ignored the effects of possible conflicts with memory accesses coming after the locked region. It may happen that we flush out a memory line that otherwise would have been accessed later on. 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.

3.5 Putting it All Together

In this subsection, we will use the code in Figure 4(a) to illustrate our algorithm. We assume, for this example, a 4KB direct-mapped cache, with 16B per line. We run our compiler, which detects those constructs that are not analyzable at compile time (Figure 4(b)). In Figure 4(c) we show the code after deciding the regions that should be locked. When locking the whole loop body, the compiler decides to lock the whole loop to avoid unnecessary locks/unlocks at every iteration.

The next step consists in deciding which data to load. Figure 4(d) summarizes the outcome of the locality analysis. For the first region, it identifies three variables and the ranges for two of them; for b it assumes the whole domain. Eventually, it checks the locality of the references. a is already in the cache, but c and b are accessed for the first time, thus we would like to load them. First, it loads c since it is more accessed than b. In this example, there is enough space in the cache to load b too, but if there were not space enough in cache, we would prefer to load c rather than b. Using the reuse vectors, we detect temporal locality between the two occurrences of *a[i]*, and the volume analysis says that 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.

Eventually, the worst-case memory performance will be computed. 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 misses, since we would not know the memory lines being accessed (besides, we would have invalidated array b since our analyzer would not take advantage of it).

4. EXPERIMENTAL FRAMEWORK

Figure 5 depicts the framework used in our experiments. We try to implement the analysis as general as possible, so we do not tie ourselves to any specific language. Instead, our compiler is written using the SUIF2 internal representation, which can be generated from different front-ends. We use SUIF2 to collect all information about memory accesses and control flow (it basically applies abstract inlining [32] and detects loops and IF statements). The paths that are used to eventually obtain the longest path (i.e., the one corresponding to the worst-case scenario) are currently manually fed to our system.

The core block is the one that computes the equations and solves them, which describes the cache behavior. For that purpose, we have followed the techniques outlined in the literature [12, 31, 32]. We have extended them to deal with locked regions (see Section 3.2.1). Equations are generated in such a way they take into account the extra load and lock/unlock instructions.

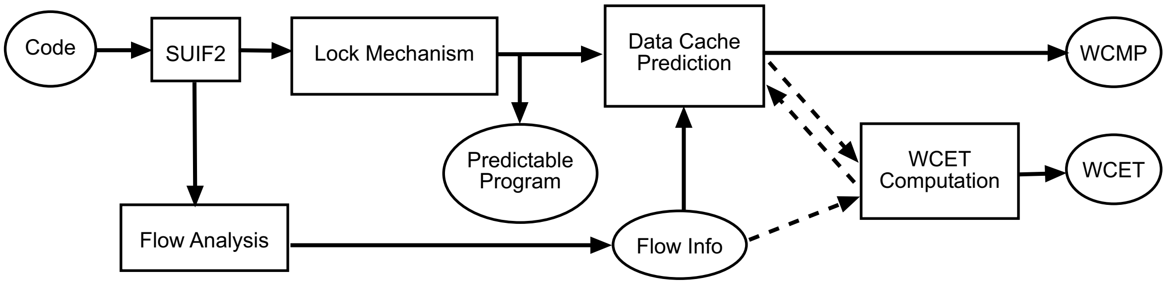


Figure 5: A framework for WCET computation.

(a)

```
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;
```

(b)

Non-analyzable constructs:
b[a[i]];
c[i]>15

(c)

```
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;
}
```

(d)

Region	Variables	Locality	Load
1	c:0. . .99	N/A	YES
	b: N/A	N/A	YES
	a:0. . .99	a[i]	NO
2	k: 0	k	NO

Figure 4: Example of how our algorithm works.

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

Table 2: Benchmarks used

An overview of the eight benchmark programs can be seen in Table 2. They are all written in *C*, drawn from different real-time papers that analyze data cache behavior.

To check the accuracy of our method, we compare our results against a memory simulator[7] modified to handle locking caches. We present results in terms of memory cost. Thus, we account only for load/store and lock/unlock instructions. We analyze two modern architectures to estimate the miss penalties. For the microSPARC II-ep [29] (direct-mapped, 8KB bytes, 16B per line), each hit takes 1 cycle and each miss 10 cycles. For the PowerPC 604e [25] (4-way, 16KB cache, 32B per line), each hit is 1 cycle and each miss accounts for 38 cycles. Lock and unlock instructions take 1 cycle to execute in both processors. Instructions to load the cache are treated as normal memory accesses. Writes and reads are modeled identically.

5. EXPERIMENTAL RESULTS

We now present results from our simulation studies. We first confirm the accuracy of our static data cache analysis comparing it against a simulator. We show that the reuse vectors and the modified equations accurately model the actual cache behavior. Then, we analyze the efficiency of our loading algorithm for reducing the performance degradation due to the lock/unlock instructions. Finally, we present our estimated WCMP for the set of benchmarks.

5.1 Accuracy

The results of our first set of experiments are shown in Table 3. Table 3(a) shows the accuracy of our method for those codes where locking was not necessary. All the programs consist of a set of subroutines, some of them containing IF statements. In all the cases, we predict exactly

[7] A locally written simulator. It has been validated over the years against the well-known DineroIII trace-driven simulator [14].

Name	C.	Simulated Cost	Estimated Cost	Est/Sim Ratio
MM	S	7108684	7108684	1.00
	P	8836226	8836226	1.00
CNT	S	75000	75000	1.00
	P	122500	122500	1.00
FIB	S	223	223	1.00
	P	279	279	1.00
ST	S	31784	31784	1.00
	P	32500	32500	1.00

(a) Codes where cache is not locked.

Name	C.	Simulated Cost	Estimated Cost	Est/Sim Ratio
SQRT	S	332	332	1.00
	P	883	883	1.00
SRT	S	7509	7509	1.00
	P	12287	12287	1.00
NDES	S	9040	9040	1.00
	P	9450	9450	1.00
FFT	S	233344	233344	1.00
	P	807936	807936	1.00

(b) Codes with lock/unlock instructions.

Table 3: Dynamic results for data caching. *S* stands for microSPARC-IIep, *P* for PowerPC 604e.

the same results as yielded by the simulator. Moreover, we predict for each memory access exactly the actual behavior.

Table 3(b) presents the results for those codes where our method issued lock/unlock instructions. In order to show our capability to statically analyze the cache behavior in this situation, we analyze the same path that is actually executed in the simulator. Thus, we can isolate the results of our analyzer from those of the WCMP computation. Our method obtains the same results as the simulator in all cases.

Compile-time Overhead. The average execution time needed to analyze each configuration was 0.6 seconds. MM was the program that took most time, with 3 seconds for each cache configuration, since we have to evaluate more than 4 million accesses.

5.2 Performance of Data Locking

The goal of using data 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. To evaluate the effectiveness of this approach, we compare the memory cost of the resulting code with lock/unlock instructions against the same code extended with selective load instructions. For the sake of comparison, we do not consider the additional cycles due to extra loads and locks/unlocks. In order to isolate the results from those of the WCMP computation, we consider the actual path that is executed.

The results of this experiment are shown in Table 4. We analyze programs where lock/unlock and load instructions were issued. 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%. When loading the cache, performance degradation is usually eliminated. In those cases where there are conflicts among data accessed in the locked regions, loading the cache reduces the performance degradation, but it cannot eliminate it completely. Finally, last column presents the number of extra loads issued to load the cache. It shows that the reduction of memory cost can be achieved with few selected loads.

We have evaluated the overall overhead of the resulting code in more detail. Figure 6(a) 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 without lock instructions. 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-II architecture. When the cache size is increased, the conflicts disappear and the overhead is minimal.

In the following section, we show how this small degradation in performance allows having a fully predictable program. Thus, we can compute the WCMP in a much tighter way than previous approaches. Even though the actual execution time of the task may increase, the WCMP will be smaller, thus we will be able to make better use of resources.

5.3 WCMP

Our locking algorithm will be successful if the presence of locked regions allows us to compute a smaller WCMP than before. This is, if WCMP(task+lock+load)<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. This can be seen as considering an empty cache where we would unlock the cache in our approach.

We use as a reference the actual WCMP of the program without lock instructions.

Figure 6(b) shows the different estimates for each method. When we consider the cache disabled, all memory accesses are considered as misses, producing a very large overestimation of the WCMP. The values show that the estimated WCMP is between 5 and 38 times larger than the actual one.

The pessimistic approach performs better than considering the cache disabled, but it is still far from a tight WCMP. The estimated WCMP is between 2 and 22 times larger than the actual WCMP. Our approach gives an exact WCMP of the transformed program (i.e., the program with lock instructions).

5.4 Summary

Overall, we have shown the effectiveness of our approach. 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 the WCMP for

Name	C.	Unlock	Lock	Lock & Load	Δ_U(%)	Δ_L(%)	#Loads
SQRT	S	158	330	158	108.8	0.0	1
	P	214	881	214	311.6	0.0	1
SRT	S	7507	7507	7507	0.0	0.0	0
	P	12285	12285	12285	0.0	0.0	0
NDES	S	6299	6992	6992	11.0	11.0	0
	P	6970	6970	6970	0.0	0.0	0
FFT	S	88696	231296	118544	160.7	33.6	256
	P	52736	805888	52736	1428.1	0.0	128

Table 4: Memory cost in cycles for the lock & load algorithm. S stands for microSPARC-IIep, and P for PowerPC 604e (Δ_U=loss of performance without loading the cache, Δ_L=loss of performance when loading the cache).

the benchmarks given. Besides, we have seen that the estimate of the WCMP(task+lock+load) is much smaller than the best estimate of the WCMP(task). For those programs where lock instructions are not issued, our estimate is exact and there is no overhead.

We first 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 fully 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 a tighter WCET.

6. RELATED WORK

In the past few years several strategies have been presented for analyzing cache memory behavior analytically.

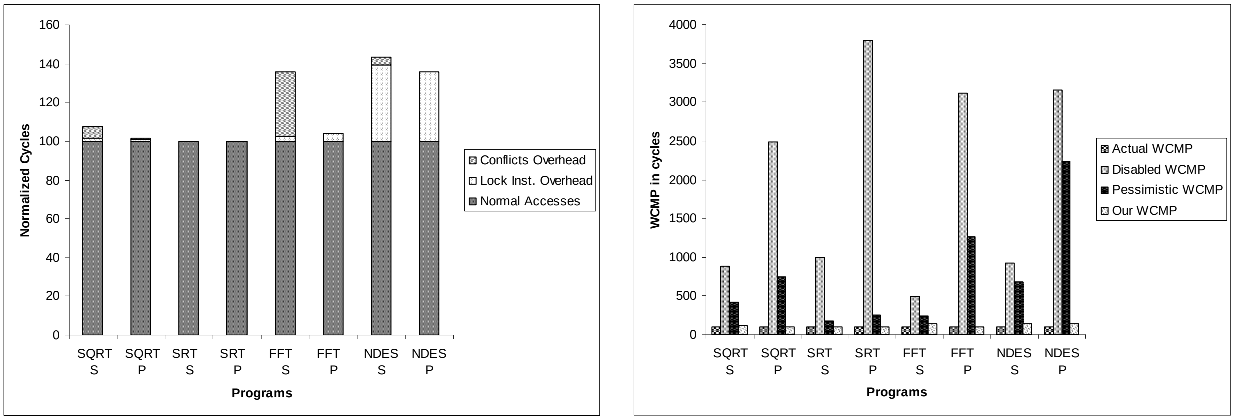
Predicting cache behavior is a key issue for cache optimizers. Ghosh *et al* [12] presented the CMEs framework targeted at isolated perfect loop nests consisting of straight-line assignments. They show that the CMEs can be helpful in reducing the number of cache misses for scientific codes. Fraguela *et al* [10] use a probabilistic method to provide a fast estimate of cache misses, describing reuse only among references in the same nest. Recently, Chatterjee *et al* [6] presented an ambitious method for exactly predicting the cache behavior of loop nests by means of Presburguer formulas. Because of the complexity of their algorithm, they have only evaluated it on very small kernels. Finally, Vera and Xue [32] examine the problem of analyzing whole programs. This model is able to predict misses for large codes consisting of data-independent constructs (including calls and IF statements).

Meanwhile, the real-time community has intensified the research in the area of predicting WCET of programs in presence of caches. Calculation of a tight WCET bound of a program involves difficulties that come from the very characteristics of data caching. Even though some progress has been done when studying processors with instruction caches [2, 13, 20], few steps have been done towards analyzing data caches.

Alt *et al* [1, 9] 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* [22] present a method that computes the WCET taking into account data caching. However, they only analyze static memory references (i.e., scalars), failing to study real codes with dynamic references (i.e., arrays and pointers). Kim *et al* [17] propose a method that extends and improves the previous method extending the analysis that classifies references as either static or dynamic. However, they deal neither with arrays nor with pointers (i.e., only detecting temporal locality). Further, it is limited to basic blocks, without taking in account possible reuse among different subroutines or loop nests. Li *et al.* [21] describes a method which does not merge the cache state but tries to calculate possible cache contents along with the timing of the program. The whole CPU is modeled by a linear integer programming problem, and a new constraint is added for each element of a calculated reference. This requires a very large computation time, and has problems of scalability with large arrays. Besides, they do not report results for WCET in presence of data caches.

White *et al* [33] propose a method for direct-mapped caches based on static simulation. They categorize static memory accesses into (i) first miss, (ii) first hit, (iii) always miss and (iv) always hit. Array accesses whose addresses can be computed at compile time are analyzed, but they fail to describe conflicts which are always classified as misses. For instance, they overestimate the memory cost by 10% and 17% for MM and ST respectively (we estimate the WCMP exactly without issuing lock instructions).

Lundqvist and Stenström [23] 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 do not present the overhead of the transformed program. Neither do they report results for WCET or WCMP using their approach. Finally, Campoy *et al* [3] introduce the use of locking instruction caches. They use static locking, presenting 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.



(a) Overall overhead of the cache locking.

(b) Accuracy of our WCMP estimate.

Figure 6: Statistics of our approach. S **stands for microSPARC-IIep,** P **for PowerPC 604e.**

7. CONCLUSIONS

This paper combines static cache analysis with data cache locking to estimate the worst-case memory performance in a safe, tight and fast way. First, an approach to statically analyze the cache behavior is reviewed. It uses an extended version of the reuse vectors as a safe measure of reuse, resulting in a set of equations that describes where this reuse translates to locality.

Second, we give a novel approach to overcome the problem of data-dependent constructs. We describe how locking caches can be used to avoid interferences and unpredictability. Later, we discuss how to load the cache in order to achieve good performance.

Finally, we combine both methods which results in a tool that predicts the worst-case memory performance in a tight and safe way, with an acceptable loss of performance. Combined with a timing analysis platform, we may obtain a much tighter WCET estimate than previous approaches. Furthermore, it can be used for computing WCET estimates in multi-task systems when combined with the cache partitioning technique [18].

Overall, this paper contributes with a unique technique that provides a considerable step toward a useful worst-case execution time prediction of actual architectures. Written as a compiler pass, it both issues lock/unlock/load instructions and computes the worst-case memory performance in presence of k-way set-associative data caches. Moreover, our framework can be used to guide the compiler in order to generate code that exploits the cache memory and computes the WCMP at the same time. Even though performance is not typically a key issue in real-time systems, a better use of the cache is very useful in order to reduce power consumption.

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. A better pointer analysis could be beneficial to lock fewer regions, and would help us to classify their accesses as misses or hits. It may also be interesting to take into account the overall performance when selecting data to lock in the cache. We plan to investigate these research directions in order to have full predictability and better performance.

Acknowledgements

This work has been supported by VR grant no. 2001–2575. J. Xue was supported in part by Australian Research Council Grant A10007149.

The authors thank Ebbe for his infinite patience answering all our questions about how to compute WCET. We also thank Janne and Jan Gustafsson for reviewing previous drafts of this paper.

Credits go to Nerina Bermudo, without whom the CMEs implementation would not exist.

8. REFERENCES

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

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

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

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

[5] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layout for hierarchical memory systems. In *Proceedings of ACM International Conference on Supercomputing (ICS'99)*, pages 444–453, Rhodes, Greece, Jun. 1999.

[6] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. In *ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI'01)*, pages 286–297, 2001.

[7] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In

Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95), pages 279–290, Jun. 1995.

[8] J. Engblom and A. Ermedhal. Modeling complex flows for worst-case execution time analysis. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS 2000)*, Dec. 2000.

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

[10] B. B. Fraguela, R. Doallo, and E. L. Zapata. Automatic analytical modeling for the estimation of cache misses. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, 1999.

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

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

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

[14] M. Hill. *DineroIII: a uniprocessor cache simulator (http://www.cs.wisc.edu/~larus/warts.html).*

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

[16] M. Kandemir, A. Choudhary, P. Banerjee, and J. Ramanujam. A linear algebra framework for automatic determination of optimal data layouts. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):115–135, Feb. 1999.

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

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

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

[20] Y. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of 16th Real-Time Systems Symposium (RTSS'95)*, pages 298–307, 1995.

[21] Y. Li, S. Malik, and A. Wolfe. Cache modeling and path analysis for real-time software. In *Proceedings of 17th Real-Time Systems Symposium (RTSS'96)*, 1996.

[22] S. Lim, Y. Bae, G. Jang, B. Rhee, S. Min, C. Park, H. Shin, K. Park, and C. Kim. An accurate worst case timing analysis technique for RISC processors. In *Proceedings of 15th Real-Time Systems Symposium (RTSS'94)*, pages 97–108, 1994.

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

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

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

[26] T. Mowry, M. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of V Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'92)*, pages 62–73, Oct. 1992.

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

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

[29] Sun Microelectronics. *microSPARC-IIep User's Manual*, 1997.

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

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

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

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

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

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