Coyote Project: Documentation

www.mrtc.mdh.se/projects/wcet/Coyote

Nerina Bermudo, Xavier Vera*
Institutionen för Datateknik
Mälardalens Högskola
Västerås, Sweden

1 Introduction

Data caches are a key component of current microprocessors in order to reduce the memory latency. The effectiveness of a cache memory depends not only on the hardware structure, but also on the code generated by the compiler. Optimizing compilers can significantly improve the performance of the memory system by means of several program transformations/optimizations such as prefetching, blocking, padding and data layout. However, this type of techniques require some knowledge of the behavior of the program.

The process of obtaining information of the locality characteristics of a given program is known as *data locality analysis*. This analysis has been performed traditionally either at compile-time using heuristics or at run-time by means of simulators. The problem is that simulators are very slow whereas heuristics can be very imprecise. Analytical methods model the cache behavior in such a way that information about the causes of the misses can be obtained. Unfortunately, they can have huge computation times, lack of accuracy and only suit for perfectly nested loops without conditionals.

Cache Miss Equations (CMEs) [3] is another method that accurately describes the cache behavior by means of a set of equations. These diophantine equations constitute an analytical model of the cache memory that allows us to use mathematical techniques to gather information about the cache behavior. Unfortunately, a direct solution of the CMEs is computationally intractable. Even though the computation cost of generating CMEs is a linear function of the number of references, to solve them is a NP-Hard problem and thus trying to study a whole program may be unfeasible.

Like the CMEs, the analytical method described in this paper describes the cache behavior by means of a set of equations. These equations describe accurately the relationship among loop indices, array sizes, base addresses and the cache parameters for a loop nest. Some statistics-based methods has been reported to reduce the execution time of solving the equations. We have followed the ideas presented by Vera et al. [2, 6]. Their method is based on two facts. Firstly, the equations describe convex bounded polyhedra. The integer points inside of them represent the iteration points where a potential miss occurs. By exploiting

* *emails:{nerina.bermudo, xavier.vera}@mdh.se*

some intrinsic properties of the particular types of polyhedra generated by the equations, they reduce the complexity of the algorithm. On the other hand, one of the advantages of using equations towards simulators is that they allow studying each reference in a particular iteration point independently of all other memory references. They estimate the miss ratio by means of sampling techniques.

1.1 Goals and Philosophy

We present an implementation of analytical method to analyze the performance of the memory hierarchy. It is based on the Cache Miss Equations (CMEs) [3]. Like them, we set up a collection of equations that describe the cache behavior. As we mentioned before, we apply some statistics-based and polyhedra techniques to reduce the execution time needed to obtain the miss ratio [2, 6].

Unfortunately, the current method is limited to special forms of perfectly nested loops. For example, they must be free of IF statements and they can not contain parallel loop nests. Since one of our future research aim is to overcome this problem, our goal was to make an implementation in such a way so that it could be easily extended and could analyze both C and FORTRAN codes. Besides, it needed to be as efficient and accurate as possible. We need sometimes to obtain safe approximations of the cache miss ratio, so the whole iteration space is likely to be analyzed (efficiency) and we are interested in obtaining the most accurate approximation (accuracy).

We have chosen C++ as the language to implement it. It gives us the required data-abstraction mechanism and we can use all the low level tricks available from C, so we can obtain an efficient implementation.

2 Generating the Equations

The analysis of complete iteration spaces showed that the locality analysis that was provided by the reuse vectors introduced by Lam [7] is not sufficient in some cases and in others the equations did not take profit of all the information. Specific reuse vectors to the shape of the iteration space and the cache parameters (the very information ignored by Wolf and Lam's framework) should be provided.

Section 2.1 describes how we handle the standard reuse vectors [7] in order to use them afterwards to generate the equations. Section 2.2 presents the different reuse extensions that have been developed in order to obtain tighter results. Finally, section 2.3 gives some details about the equations that are generated.

2.1 Basic Reuse Vectors

For each reference, we compute the reuse vectors that describe the potential data reuse applying Wolf and Lam's framework reuse. This reuse vectors determine the distance and direction of reuse between pairs of uniformly generated references. Non-affine references which yields little reuse [7] are ignored.

Given a reference R_A, we compute four different types of general reuse vectors which correspond to the kinds of reuse that may exist: *self spatial, self temporal, group spatial* and *group temporal*. In the case of group reuse vectors, we call R_A the *trailing*, and the reference from which it reuses R_B is called the *leader*.

Once the reuse vectors are computed, they should be modified in order to be used to generate the equations. The following rules are applied:

- If we obtain two identical reuse vectors that only differ in its type (temporal or spatial), we always keep the temporal reuse vector and remove the spatial one, since the temporal reuse vector is the more restrictive one.
- All self reuse vectors are normalized, since the reuse distance is not relevant, and this helps to remove redundant vectors.
- Since negative reuse vectors model forward reuse and a reference can not reuse from an iteration that has not been executed yet, whenever a negative reuse vector is computed, we change its orientation and interchange leader and trailing references, since the former trailing will be the one that will actually reuse data from the former leader.
- If a group reuse vector is zero and the leader is executed after the trailing in each iteration, we change this vector by $(0, \ldots, 0, 1)$ (assuming that innermost component carries the reuse). The fact that the reuse vector is zero means that the trailing reference reuses from the leader in the same iteration, but if the leader reference has not been executed yet, this reuse cannot exist. At most, that reuse will be realized in the following iteration of the loop nest that carries the reuse, which corresponds to the reuse vector $(0, \ldots, 0, 1)$.

In order to obtain the number of misses, we study the reuse vectors in lexicographical ascendent order [4]. In case we have more than one reuse vector with the same distance, the following order applies: group temporal, group spatial, self temporal, self spatial.

2.2 Reuse Extensions

One of the flaws of Wolf and Lam's framework is that they do not take in account the shape of the iteration space and the configuration of the cache. In order to achieve a more accurate description of the reuse in the loop and get exact information of the cache behavior, several extensions of those reuse vectors have been developed.

Let *dim* be the dimension of the iteration space of the loop, which is also the dimension of the reuse vectors. Let ub_i and lb_i be the upper and lower bounds of the $i-th$ dimension of the iteration space.

The reuse extensions included in this implementation are the following:

1. If we have a reuse vector of the form $(0, \ldots, 0, n)$, no matter what type of vector it is, we add the following new spatial reuse vectors:

$$(0, \ldots, 1, n - ub_{dim})$$
$$(0, \ldots, 0, 1, 1 - ub_k, \ldots, 1 - ub_{dim-1}, n - ub_{dim}), \quad k = dim - 1, \ldots, 1$$

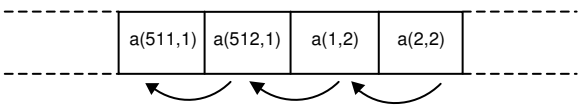

Figure 1: Reuse between different columns.

This reuse vectors do not describe a new reuse that was not given by the general ones, but they describe it for some special cases in a different way in order to be able to deal with CMEs.

Let us consider a loop of the form

$$\begin{array}{ll} do & j = 1, 511 \\ & do \quad i = 1, 511 \\ & \quad\quad a(i,j) \\ & \quad\quad \vdots \\ & \quad\quad a(i+1,j) \\ & enddo \\ enddo & \end{array}$$

In this case, we would obtain a group temporal reuse vector $(0, 1)$ which shows that reference $R_1 = a(i, j)$ reuses data from the reference $R_2 = a(i + 1, j)$ one iteration of the inner loop later. For example, reference R_1 in iteration $(1, 2)$ accesses the same data that reference R_2 has accessed in the iteration $(1, 1)$.

The original equations do not get the reuse that is realized when changing the iteration of the outer loop, although reference R_1 reuses from the "previous" iteration of the whole loop from reference R_2 (see Figure 1). Our extended reuse vectors are built in order to express this reuse in such a way that the equations can detect it.

2. Let us assume we have a reuse vector of the form $(0, \ldots, 0, j_1, \ldots, j_k)$

   Then, we create a new reuse vector that has the form

   $$(0, \ldots, 0, 0, (j_1) * colsize - Cl/sizeofelem + 1 + j_2, j_3, \ldots, j_k)$$

   where *colsize* is the size of the column of the variable, Cl is the size of the cache line and *sizeofelem* is the size of the elements of the array.

3. Let us consider an original reuse vector as above. If this reuse vector is temporal, then we create several new reuse vectors in the following way:

   $$(0, \ldots, 0, j_1, j_2 - s, \ldots, j_k), s = 1, \ldots, nelems$$

   In order to explain the kind of reuse that express these reuse vectors, we consider the following loop:

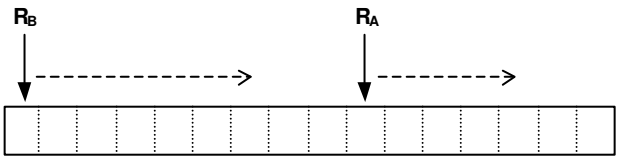

Figure 2: Reuse between two different references that are not evicted from the cache.

```
do   j
     do   i
          h(i, j)
          ⋮
          h(i, j + 1)
     enddo
enddo
```

Let be $R_A = h(i, j+1)$ and $R_B = h(i, j)$ be the two references in the loop. Let us assume that the columns $h(\cdot, j)$ and $h(\cdot, j+1)$ are stored in the same memory line. Figure 2 illustrates the situation that occurs. Reference R_A starts accessing the memory at a position that reference R_B will access in the following iteration of the outer loop. The new reuse vectors are added in order to describe the reuse that exists due to the fact that the columns of the matrix are in the same memory line.

2.3 Equations

There are two types of situations where a miss arises: *cold misses* and *replacement misses*. Given a reference R and an iteration point $\vec{i}$, R suffers a cold miss at $\vec{i}$ when the memory line accessed is touched for the first time. A replacement miss occurs if the line was accessed before and flushed out later, so that it is not in the cache anymore.

When computing the *cold miss equations* that describe the reuse that is not realized when a new cache line is accessed, the original CMEs framework assumes that all the columns are aligned. In our model, we use the exact position of all the columns of the arrays for the sake of accuracy. Furthermore, we can generate the equations in two different ways:

- *Polyhedral way*: we build a polyhedron that describes all the iteration points where the reuse is not realized
- *Mapping function*: we keep the function that allows us to compute the cache line where the references are mapped. This way we can add some extra reuse vectors between non-uniformly generated references.

3 Coyote Environment

Currently our tool works fine on the following platforms:

- Windows environment (NT, 2000 and 98), compiler: Visual Studio 6.0
- Solaris, compiler: gcc 2.8.1 and gcc 2.95
- OSF (on alpha machines): gcc 2.8.1 and gcc 2.95
- Linux: egcs 2.90 and gcc 2.95
- Cygwin (it is a Unix environment under NT): gcc 2.95

We have evaluated our tool through the SPECfp95 programs for a 32KB cache and different cache architectures: direct-mapped, 2, 4 and 8-way set associative caches. In all cases, we obtain exactly the same number of misses as the simulator.

In order to test our analyzer, we have written a program that generates all the compile time information needed to generate our equations. This program has been written for Fortran codes by means of the Polaris Compiler [5]. We have used the Ictineo library [1] to obtain the relative access order of the references from a RIS low-level optimized code. Information such as the base addresses of the variables are obtained from the native compiler. The same information has been fed to the simulator.

3.1 Performance Evaluation

We report our experimental results for a range of programs from SPECfp95. We present the estimation of the cache misses for three different architectures: direct-mapped cache, 2-way and 4-way set associative caches.

The problem sizes used for all the examples are those of the reference input files. We assume a cache of 32KB and 32B per cache line.

All results have been obtained using a PentiumIII, 950MHz with 512MB memory running SUNOs 5.6. All the simulation results are obtained using a trace-driven simulator.

Basic Reuse Vectors Table 3.1 shows the total number of misses for different loops of Swim, Tomcatv and Hydro2D. First two columns identify the loop nest, giving the name of the program and the function where it is. Column 4 gives the number of misses obtained running the simulator, whereas column 5 shows the number of misses estimated by means of *Coyote*. In the last column we present the relative error. As it can be seen, in all the cases but one the reuse vectors are not enough, and therefore we overestimate the number of misses. In some case this overestimation is rather large (fct12, artdif8 or main5).

Extensions We have analyzed exactly the same codes using the extensions explained in section 2.2. In all the case we obtain the same number of misses for all the cache configurations. Table 3.1 shows the time needed to analyze the whole iteration space. Although the

Program	Function	Cache	#Cache Misses		E.
			Sim.	*Coyote*	
Swim	Initial4	direct	98562	99776	1.2
		2-way	98562	99776	1.2
		4-way	98562	99776	1.2
	Initial8	direct	197382	200070	1.3
		2-way	197382	200070	1.3
		4-way	197382	200070	1.3
	Calc1	direct	262789	266050	1.2
		2-way	230021	232899	1.2
		4-way	230021	232899	1.2
	Calc2	direct	361355	365827	1.2
		2-way	328586	332804	1.2
		4-way	328586	333187	1.4
	Calc3z	direct	296073	300105	1.3
		2-way	296073	300105	1.3
		4-way	328969	332937	1.2
	Calc3	direct	296073	298944	0.9
		2-way	328327	331648	1.0
		4-way	361159	364352	0.8
Tomcatv	Main2	direct	2622730	2632945	0.3
		2-way	2622730	2632945	0.3
		4-way	2622730	2634220	0.4
	Main3	direct	664282	666837	0.3
		2-way	655364	656640	0.2
		4-way	655364	656640	0.2
	Main5	direct	1637760	1640950	0.2
		2-way	1637760	1640950	0.2
		4-way	1637760	1967350	20
	Main7	direct	1311985	1311985	0.0
		2-way	1311985	1311985	0.0
		4-way	1311985	1311985	0.0
	Main8	direct	1310720	1313265	0.2
		2-way	1310720	1313265	0.2
		4-way	1310720	1313265	0.2

Program	Function	Cache	#Cache Misses		E.
			Sim.	*Coyote*	
Hydro2D	artdif4	direct	50174	55963	11
		2-way	48529	51409	5.9
		4-way	48502	48682	0.3
	artdif6	direct	48322	48480	0.3
		2-way	50202	50360	0.3
		4-way	48322	48480	0.3
	fct2	direct	80522	80960	0.5
		2-way	80522	80960	0.5
		4-way	80522	80960	0.5
	fct4	direct	95981	96455	0.4
		2-way	95981	96455	0.5
		4-way	80102	80497	0.5
	fct10	direct	60211	67009	11
		2-way	48689	49169	1
		4-way	48485	48864	0.8
	fct12	direct	69306	82023	18
		2-way	64544	70367	9.0
		4-way	64503	65104	0.9
	filter2	direct	32161	32320	0.5
		2-way	32161	32320	0.5
		4-way	32161	32320	0.5
	filter7	direct	32161	32319	0.5
		2-way	32161	32319	0.5
		4-way	32161	32319	0.5
	filter10	direct	64322	64720	0.6
		2-way	64322	64720	0.6
		4-way	64322	64720	0.6
	filter17	direct	63918	64236	0.5
		2-way	63918	64236	0.5
		4-way	63918	64236	0.5

Table 1: Cache miss ratios for caches with $\mathcal{C}$=32KB and $\mathcal{L}$=32B.

Program	Function	Cache	Time
Swim		direct	1.2
	Initial4	2-way	1.2
		4-way	1.2
	Initial8	direct	1.3
		2-way	1.3
		4-way	1.3
	Calc1	direct	1.2
		2-way	1.2
		4-way	1.2
	Calc2	direct	1.2
		2-way	1.2
		4-way	1.4
	Calc3z	direct	1.3
		2-way	1.3
		4-way	1.2
	Calc3	direct	0.9
		2-way	1.0
		4-way	0.8
Tomcatv	Main2	direct	0.3
		2-way	0.3
		4-way	0.4
	Main3	direct	0.3
		2-way	0.2
		4-way	0.2
	Main5	direct	0.2
		2-way	0.2
		4-way	20
	Main7	direct	0.0
		2-way	0.0
		4-way	0.0
	Main8	direct	0.2
		2-way	0.2
		4-way	0.2

Program	Function	Cache	Time
Hydro2D	artdif4	direct	11
		2-way	5.9
		4-way	0.3
	artdif6	direct	0.3
		2-way	0.3
		4-way	0.3
	fct2	direct	0.5
		2-way	0.5
		4-way	0.5
	fct4	direct	0.4
		2-way	0.5
		4-way	0.5
	fct10	direct	11
		2-way	1
		4-way	0.8
	fct12	direct	18
		2-way	9.0
		4-way	0.9
	filter2	direct	0.5
		2-way	0.5
		4-way	0.5
	filter7	direct	0.5
		2-way	0.5
		4-way	0.5
	filter10	direct	0.6
		2-way	0.6
		4-way	0.6
	filter17	direct	0.5
		2-way	0.5
		4-way	0.5

Table 2: Execution time (in seconds).

execution time needed is not very large, it is unfeasible for large problems, and the use of sampling techniques is recommended.

4 Coyote Internal Representation

The internal representation is the main basis of our tool. All future software based in this implementation up to the obtention of the number of misses is related to this representation.

The current internal representation is made upon a set of classes, which describe the program we want to analyze, as well as its locality and cache behavior. In order to do that, several modules have been developed:

Base Classes. This module contains all the classes that any piece of this implementation can be expected to handle. We can find classes that describes a program and the different classes that help us to create the equations.

Generator Module. This module contains a set of functions that allows us to compute the locality of a program (i.e the reuse vectors) and generate the set of equations. Those functions belong to different base classes.

Solver Module. We can find in it a set of routines that solves the equations. Besides, it contains some classes where we can store detailed information about the cache memory behavior of the program.

Statistic Module. It contains all the routines that deal with the statistical part of this implementation.

The different modules but the statistical one are described in the following sections. The next subsection gives an overview of support classes.

4.1 Support Classes

In order to provide a full support to the internal representation, we have created an infrastructure of classes, types and functions that are used very often by the other classes.

The infrastructure includes a **List** class hierarchy (which contains **ListElem** objects). Actually, most of the classes used in our representation derives from the **ListElem** class and they are stored in different lists. An example of the use of this lists can be seen in the representation of a function of the program we are analyzing. A function contains a list of all the variables that are declared within it and another list of all the loops we plan to analyze.

We can insert in different places of the **List**, access a concrete element of it or we can iterate the **List** using the in-built iterator. **List** has also a method that allows us to sort it using the compare function from the **ListElem** object.

Serialization All lists, as well as all the **ListElem** and the other objects that are within the internal representation can be serialized from and to a stream. All the objects have an

overloaded operator << and >>. These operators can be used to keep track of the different sessions that have been executed. If we want a pretty output that helps us in the debugging stuff, all the objects have a **print()** method which gives much more readable information to the user.

4.2 Writing a pass

Now, we explain how we can create a main program that takes a description of the program we want to analyze and gives us its cache behavior for a given cache architecture.

The generator of the CMEs needs certain information about the program we want to analyze as well as the architecture that will be considered. Currently, this information has to be provided via an input stream, using the serialization methods described above. The stream should be like this:

Information on the cache

cache line, cache size, associativity of the cache

Information on the variables

number of variables

(For each variable)

length of the name of the variable, name of the variable

base address, size of the element

dimensions of the array

bounds of every dimension

Information on the loops

(For each loop)

Number of references in the loop

(For each reference)

variable

dimensions of the matrix H

matrix H (in rows) and vector C

number of reuse vectors (*This field has to be set to zero.*)

About the iteration space

dimension

bounds of each dimension

```
program test
parameter (n=1000)
real A(n,n), B(n,n), C(n,n)
do i = 1,n
  do j = 1,n
    do k = 1,n
      A(i,j)=A(i,j) + B(i,k) * C(k,j)
    enddo
  enddo
enddo
end
```

Figure 3: Matrix Multiply Algorithm

Figure 4 shows an example of input stream for the matrix multiply program (see Figure 3).

In this example we have one loop, three 2-dimensional variables (A,B and C) and four references. Note that the variables are labelled from 0 to 2 and are referred like that when expressing to which variable each reference is associated.

Not that since this format corresponds to the internal structure of the equations, the number of reuse vectors has to be entered.

In order to use the functions to generate and solve the equations, a main program is needed that fulfils following properties.

Files to be included:

- CFunction.h
- CVariable.h
- CSample.h
- CMLoop.h
- FSolver.h

Figure 5 shows an example of how to analyze a function. In this example, we assume a direct mapped cache and the sample of the iteration space is generated in order to obtain an accuracy of 95% confidence and a 0.05 interval length. For every loop in the function (in this case we have only one), the reuse vectors of the current loop are computed in the instruction

function.loops().current().Compute_Locality(function.Cl(), function.variables());.

The computed reuse vectors are directly added to the internal structure that contains all the information about the program. Then the equations are generated calling

function.loops().current().Generate_Equations(function.variables(), function.Cl(), function.Cs());.

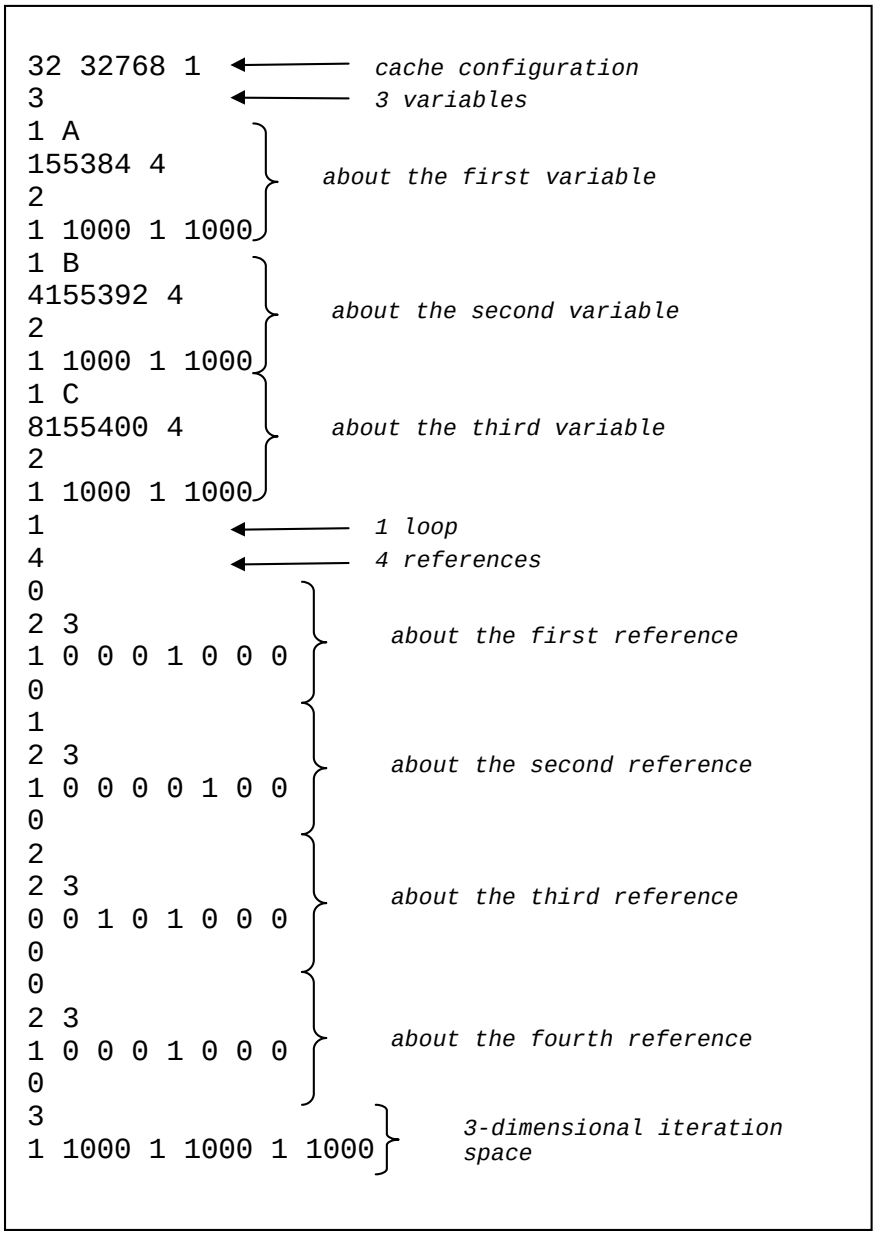

Figure 4: Example of input file (matrix multiply).

```
// Base includes
#include "CFunction.h"
#include "CVariable.h"

// Solving includes
#include "CSample.h"
#include "CMLoop.h"
#include "FSolver.h"

// Standard includes
#include "iostream.h"
#include "fstream.h"

int main (void)
{
      char name[80];
      cin>>name;
      ifstream file (name);
      ofstream output("output.sol");

      CFunction function;
      file>>function;

      function.loops().reset();

      CSample sample(0.95, 0.05, function.loops().current().ispace());

      int nloops=function.loops().entries();

      for(int i=0; i<nloops; i++)
      {
            function.loops().current().Compute_Locality(function.Cl(),
                  function.variables());

            function.loops().current().Generate_Equations(function.variables(),
                  function.Cl(), function.Cs());

            CMLoop *sol=Number_Misses_CM(function.loops().current(), sample);
            sol->Compute_Interval(0.05);
            sol->print(output);
            delete sol;
      }
      return 0;
}
```

Figure 5: Example of main program.

```
Loop misses
===========

Total number of misses: 1159
Total number of accesses: 4328
Miss Ratio: 0.267791
      Compulsory misses: 0
      Replacement misses: 1159
      Spatial Repl. misses: 9
      Temporal Repl. misses: 1150
      --------------------------------

            Reference 1
            Total number of misses: 0
            Total number of accesses: 1082
            Miss Ratio: 0
            [ 0, 0.0687786 ]
            Compulsory misses: 0
            Replacement misses: 0
                  Spatial : 0
                  Temporal: 0

            Reference 2
            Total number of misses: 1082
            Total number of accesses: 1082
            Miss Ratio: 100
            [ 99.9312, 100 ]
            Compulsory misses: 0
            Replacement misses: 1082
                  Spatial : 2
                  Temporal: 1080

            Reference 3
            Total number of misses: 72
            Total number of accesses: 1082
            Miss Ratio: 6.65434
            [ 6.57425, 6.7617 ]
            Compulsory misses: 0
            Replacement misses: 72
                  Spatial : 7
                  Temporal: 65

            Reference 4
            Total number of misses: 5
            Total number of accesses: 1082
            Miss Ratio: 0.462107
            [ 0.419259, 0.537693 ]
            Compulsory misses: 0
            Replacement misses: 5
                  Spatial : 0
                  Temporal: 5
```

Figure 6: Output

There are different functions for solving the equations, depending on the cache configuration and the way we want to obtain the information (see section 5.2.2).

In the example we are considering (see Figure 5), we use the last one of these functions. The CMLoop *sol* is returned. We dump the information contained in it to an output file called **output.sol** (see Figure 6). Note that we get the total number of misses and misses of each type as well as the miss ratio and the confidence interval for each reference in the loop.

Note that different samples can be generated for different references in order to obtain more independent results.

5 Classes

In this section we introduce the different classes that build the CMEs library.

5.1 Base Classes

1. **Class CFunction**

   CFunction represents the function we want to analyze. A function is represented by a list of variables that appear in it and a list of loops. Besides, this class contains the information on the cache configuration.

   Constructors:

   - **CFunction** (**void**)
   - **CFunction** (**unsigned int** CS, **unsigned int** CL, **unsigned int** K=1) where
     - CS stands for the size of the cache.
     - CL stands for the size of the cache line.
     - K represents the associativity of the cache.

   Clone:

   - **virtual CListElem *clone** (**void**)

   Destructors:

   - **CFunction** (**void**)

   Member access:

   - **unsigned int Cs** (**void**) **const**
     Returns the cache size.
   - **unsigned int Cs** (**unsigned int** CS)
     Sets the cache size to the value CS.
   - **unsigned int Cl** (**void**) **const**
     Returns the size of the cache line.
   - **unsigned int k_way** (**void**) **const**
     Returns the associativity of the cache.

- **unsigned int k_way** (**unsigned int** K)
  Sets the associativity of the cache to the value K.
- **CLlista<CVariable> & variables** (**void**) **const**
  Returns the symbol table of the function.
- **CLlista<CLoop> & loops** (**void**) **const**
  Returns the list of loops.

  CVariable & operator() (**unsigned int** i) **const**;
  Returns the i-th variable in the function.

2. **Class CVariable**

   This class contains all the information about the variables in the function. For each variable we store its base address in the memory as well as the size of the variable and the number of elements. We are also interested in its dimension (note that a scalar is said to have dimension 0 and a vector has dimension 1) and the bounds of each dimension.

   Constructors:

   - **CVariable** (**void**)
   - **CVariable** (**char** *NAME, **unsigned int** DIMS, **int** *VDIMS, **unsigned long** ADDRESS, **unsigned int** SIZE)
     where
     - NAME is the name of the variable.
     - DIMS is the dimension of the array. (Note that scalars are considered to have dimension 0).
     - VDIMS contains the lower and upper bounds of each of the dimensions of the variable.
     - ADDRESS is the base address of the variable in the main memory.
     - SIZE gives the size of the elements of the array.

   Clone:

   - **virtual CListElem *clone** (**void**)

   Destructors:

   - **CVariable** (**void**)

   Member access:

   - **char *name** (**void**) **const**
     Returns the name of the variable.
   - **unsigned int dimensions** (**void**) **const**
     Returns the dimension of the array.
   - **int &dim_lbound** (**unsigned int** i) **const**
     Returns the lower bound of the i-th dimension of the array.
   - **int &dim_lbound** (**unsigned int** i)
     Assigns the desired value to the lower bound of the i-th dimension of the variable i.
   - **int &dim_ubound** (**unsigned int** i) **const**
     Returns the upper bound of the i-th dimension of the array.

- **int &dim_ubound** (**unsigned int** i)
  Assigns the desired value to the lower bound of the i-th dimension of the variable i.
- **unsigned long address** (**void**) **const**
  Returns the base address of the variable.
- **unsigned long address** (**unsigned long** NEW)
  Sets the base address of the variable to NEW.
- **unsigned int size** (**void**) **const**
  Returns the size of the elements of the array.
- **unsigned int elements** (**void**) **const**
  Returns the number of elements of the variable.

Functions:

- **void Compute_Size** (**void**)
  Computes the number of elements in the array.

3. **Class CLoop**

This class contains all the information about the loop. A loop consists on the iteration space and the references that appear in the loop.

Constructors:

- **CLoop** (**void**)
- **CLoop** (**unsigned int** REFERENCES)
  where
  - REFERENCES is the number of references in the loop.
- **CLoop** (**unsigned int** references, **CISpace ***ISPACE)
  where
  - REFERENCES is the number of references in the loop.
  - ISPACE is a pointer to the iteration space of the loop.

Destructors:

- **CLoop** (**void**)

Clone:

- **virtual CListElem *clone** (**void**)

Members access:

- **unsigned int references** (**void**) **const**
  Returns the number of references in the loop.
- **CRef & operator ()** (**unsigned int** i) **const**
  Accesses the i-th reference in the loop.
- **CRef &add** (**unsigned int** i, **CRef ***NEW)
  Sets the reference NEW as the i-th reference in the loop.
- **CISpace &ispace** (**void**) **const**
  Returns the iteration space of the loop.

Functions:

- **void Compute_Locality** (**unsigned int** L, **CLlista<CVariable>** **&**VARS)
  where
  - L is the size of the cache line.
  - VARS is the list of variables in the function that we are analyzing.

  This function computes the reuse vectors for all the references in the loop. The reuse vectors are directly stored in the corresponding reference.
- **int **EqualHs** (**void**) **const**
  This function returns a matrix of dimensions nxn, where n is the number of references in the loop defined by:

  $H[i][i] = -1$

  $H[i][j] = 1$*, if the i-th reference accesses the same variable as the j-th reference and both have the same access matrix.*

  $H[i][j] = 0$*, otherwise*
- **void Generate_Equations** (**CLlista<CVariable>&** VARS, **unsigned int** CL, **unsigned int** CS, **unsigned int** K=1, **unsigned int** _delete=1)
  where
  - VARS is the list of variables in the function.
  - CL is the size of the cache line.
  - CS is the cache size.
  - K is the associativity of the cache.
  - _delete is set to 0 if the equations have to be removed from the list in case they are found to be empty.
- **void ReGenerate_Equations** (**CLlista<CVariable>** **&**VARS)
  This function generates the CMEs for different base addresses of the variables.

4. **Class ISpace**

   Given a loop, this class represents its iteration space, which is defined by the bounds of the induction variables.

   Constructors:

   - **CISpace** (**void**);
   - **CISpace** (**unsigned int** LOOP);
   - **CISpace** (**unsigned int** LOOP, **int** ***LB, **int** ***UB);
     where
     - LOOP is the depth of the loopnest.
     - LB is a vector that contains the lower bounds of the induction variables in the loop.
     - UB is a vector that contains the upper bounds of the induction variables in the loop.

   Destructor:

   - **CISpace** (**void**);

   Member access:

   - **int &**lower (**unsigned int** i) **const**
     Returns the lower bound of the i-th induction variable.

- **int &**upper (**unsigned int** i) **const**
  Returns the upper bound of the i-th induction variable.
- **int &**lower (**unsigned int** i)
  Assigns the desired value to the lower bound of the i-th induction variable.
- **int &**upper (**unsigned int** i)
  Assigns the desired value to the upper bound of the i-th induction variable.
- **unsigned int** loopnest (**void**) **const**
  Returns the depth of the loopnest.
- **unsigned int** iteration_points (**void**) **const**
  Returns the number of points in the iteration space.

Clone:

- virtual CListElem * clone (void)

5. **Class CRef**

It represents a single reference in the loop. A reference is given by the variable which it accesses and the access function. In our case this access function is a linear function of the induction variables, which is given by a matrix H and a vector of constants C.

Constructor:

- **CRef** (**void**)

Destructor:

- **CRef** (**void**)

Clone:

- **CRef *clone** (**void**) **const**

Member access:

- **CLlista<CReuseVector> &reusevectors** (**void**) **const**
  Returns a list of the reuse vectors, from which the actual reference is the trailing.
- **CLlista<CReuseVector> &reusevectors** (**CLlista<CReuseVector> ***NEW)
  Sets the list of reuse vectors NEW as the list of reuse vectors of the actual reference.
- **unsigned int variable** (**void**) **const**
  Returns the position of the variable in the list of variables.
- **unsigned int variable** (**unsigned int** NEW)
  Associates the actual reference to the variable that has the position NEW in the list of variables in the loop.
- **unsigned int rows** (**void**) **const**
  Returns the dimension of the reference.
- **unsigned int loopnest** (**void**) **const**
  Returns the number of induction variables of the loop to which the reference belongs.
- **int & H** (**unsigned int** i, **unsigned int** j) **const**
  Returns H(i,j).

- **int & H** (**unsigned int** i, **unsigned int** j)
  Accesses to H(i,j) to set it to the desired value.
- **int & C** (**unsigned int** i) **const**
  Returns C(i).
- **int & C** (**unsigned int** i)
  Accesses to C(i) to set it to the desired value.
- **void new_matrix** (**unsigned int** i, **unsigned int** j)
  Creates a new matrix H.
- **void copy_matrix** (**int** *M)
  Copies the matrix H to the matrix M.

Functions:

- **void Compute_Reuse_Vectors** (**unsigned int** L, **CRef** **REFS, **int** ACTUAL, **int** NREFS, **int** **EqualHs, **CISpace** &ISPACE, **ARRAY_LAYOUT** LAYOUT)
  where
  - L is the size of the cache line.
  - REFS is the list of references in the loop.
  - ACTUAL is the position of the actual reference in the list.
  - NREFS is the number of references.
  - EqualHs is the H matrix of the actual reference.
  - ISPACE is the iteration space of the loop.
  - LAYOUT is the considered array layout.

  This function computes the general reuse vectors for the reference.
- **void Extend_Reuse_Vectors** (**CISpace** &ISPACE, **unsigned int** CL, **CLlista**<**CVariable**> &VARS)
  where
  - ISPACE is the iteration space of the loop.
  - CL is the size of the cache line.
  - VARS is the list of variables.

  This function computes the additional reuse vectors for the actual reference.
- **void Compute_Equations** (**CISpace** &ISPACE, **CRef** **REFS, **CLlista**<**CVariable**> &VARS, **unsigned int** CL, **unsigned int** CS, **unsigned int** K, **unsigned int** idRef, **unsigned int** to_delete)
  where
  - ISPACE is the iteration space of the loop.
  - REFS is the list of references in the loop.
  - VARS is the list of variables.
  - CL is the size of the cache line.
  - CS is the cache size.
  - K is the associativity of the cache.
  - idRef is the position of the actual reference in the list of references.
  - to_delete is set to 0 if the equations have to be removed from the list in case they are found to be empty. Otherwise, to_delete=1, which is its default value.

Generation of the Cache Miss Equations.

- **void ReGenerate_Equations** (**CISpace &**ISPACE, **CRef ****REFS, **CLlista<CVariabl**
  &PARAMETERS)
  where
  - ISPACE is the iteration space of the loop.
  - REFS is the list of references in the loop.
  - PARAMETERS

  This function regenerates the Cache Miss Equations for the given parameters. Note that it does not create a new structure, it just replaces the existing one.

6. **Class ReuseVector**

Apart from the vector components there are several other parts that characterize a reuse vector: its reuse and locality type and the references involved in the reuse. Besides, the Cache Miss Equations are associated to the reuse vectors, so that each reuse vector has a list of Cold, Cold Miss Bounds and Replacement Equations.

Constructor:

- **CReuseVector** (**void**)
- **CReuseVector** (**unsigned int** NREFS)
  where
  - NREFS is the number of references in the loop.
- **CReuseVector** (**int ***VEC, **unsigned int** DIM, **REUSE_TYPE** TIPUS, **LOCALITY_TYPE** LOCAL, **unsigned int** NREFS, **int** LEADER, **unsigned int** TRAILING)
  where
  - VEC contains the reuse vector.
  - DIM is the dimension of the loopnest.
  - TIPUS determines the type of reuse (temporal or spatial).
  - LOCAL determines whether the reuse vector is group or self.
  - NREFS is the number of references in the loop.
  - LEADER indicates which is the reference that is reusing.
  - TRAILING indicates the reference from which the LEADER reuses.

Clone:

- **virtual CListElem *clone** (**void**)

Destructor:

- **CReuseVector** (**void**)

Member access:

- **unsigned int references** (**void**) **const**
  Returns the number of references in the loop.
- **unsigned int dim** (**void**) **const**
  Returns the dimension of the loopnest.
- **int &operator()** (**unsigned int** i) **const**
  Returns the i-th position of the reuse vector.

- **int &operator()** (**unsigned int** i)
  Accesses the i-th position of the reuse vector, which can be set to the desired value.
- **REUSE_TYPE type** (**void**) **const**
  Returns the reuse type of the reuse vector.
- **LOCALITY_TYPE local** (**void**) **const**
  Returns the locality type of the reuse vector.
- **LOCALITY_TYPE local** (**LOCALITY_TYPE** NEW)
  Sets the locality type to NEW.
- **int leader** (**void**) **const**
  Returns the leader reference.
- **int leader** (**int** NEW)
  Sets the leader reference to NEW.
- **unsigned int trailing** (**void**) **const**
  Returns the trailing reference.
- **unsigned int trailing** (**unsigned int** NEW)
  Sets the trailing reference to NEW.
- **CLlista<CEquation_CMB> &compulsory_g** (**void**) **const**
  Returns the list of Cold Miss Bounds equations.
- **CLlista<CEquation_CMB> & compulsory_g** (**CLlista<CEquation_CMB>** ***NEW**)
  Sets the list NEW as the list of Cold Miss Bounds equations associated to the reuse vector.
- **CLlista<CEquation_C> & compulsory** (**void**) **const**
  Returns the list of Cold Miss equations.
- **CLlista<CEquation_C> & compulsory** (**CLlista<CEquation_C>** ***NEW**)
  Sets the list NEW as the list of Cold Miss equations associated to the reuse vector.
- **CLlista<CEquation_R> & replacement** (**unsigned int** i)
  Returns the list of Replacement equations.
- **CLlista<CEquation_R> & replacement** (**unsigned int** i, **CLlista<CEquation_R>** ***NEW**)
  Sets the list NEW as the list of Replacement equations associated to the reuse vector.
- **bool & c_generated** (**void**)
  Returns whether the compulsory equations have already been generated for the actual reuse vector.
- **bool & r_generated** (**void**)
  Returns whether the replacement equations have already been generated for the actual reuse vector.

Functions:

- **unsigned int no_null** (**void**)
  Returns which is the first non-null component of the vector.
- **bool null** (**void**)
  Checks whether the vector is null. If it is, it returns true, otherwise, it returns false.

- **int compare** (**CReuseVector &** OTHER)
  Checks whether the actual reuse vector is equal to the reuse vector OTHER.
- **void normalize** (**void**)
  This function normalizes the reuse vector.

Generating CMEs:

- **void Generate_C** (**CISpace &** ISPACE)
  where
  - ISPACE is the iteration space of the loop.

  This function generates the Cold Miss Equations.
- **void Generate_CMB** (**CRef &**REF, **CVariable &**VAR, **CISpace &**ISPACE, **unsigned int** CL, **unsigned int** _delete=1)
  where
  - REF is the actual reference.
  - VAR is the variable associated to the reference.
  - ISPACE is the iteration space of the loop.
  - CL is the size of the cache line.
  - _delete is set to 0 if the equations have to be removed from the list in case they are found to be empty.

  This function generates the Cold Miss Bounds equations when considering a self spatial reuse vector.
- **void Generate_CMB** (**CRef &**REF, **CVariable &**VAR, **CISpace &**ISPACE, **CRef &**OTHER, **unsigned int** CL, **unsigned int** _delete=1) where
  - REF is the actual reference.
  - VAR is the variable associated to the reference.
  - ISPACE is the iteration space of the loop.
  - OTHER
  - CL is the size of the cache line.
  - _delete is set to 0 if the equations have to be removed from the list in case they are found to be empty.

  This function generates the Cold Miss Bounds equations when considering a group spatial reuse vector.
- **void Generate_R** (**CRef &**REF, **CVariable &**VAR, **CRef ****REFS, **CLlista<CVariab &**VARS, **CISpace &**ISPACE, **unsigned int** CL, **unsigned int** CS, **unsigned int** K, **unsigned int** idRA, **unsigned int** _delete)
  where
  - REF is the reference for which the Replacement Equations will be generated.
  - VAR is the variable associated to the reference.
  - VARS is the list of variables in the loop.
  - REFS is the list of references in the loop.
  - ISPACE is the iteration space.
  - CL is the size of the cache line.
  - CS is the cache size.

- K is the associativity of the cache.
- idRA is the position of the reference REF in the list of references REFS.
- _delete is set to 0 if the equations have to be removed from the list in case they are found to be empty. Otherwise _delete=1, which is its default value.

This function generates the Replacement Equations.

Regenerating CMEs:

- **void ReGenerate_CMB** (**CRef &**REF, **CVariable &**PARAMETER, **CISpace &**ISPACE) where
  - REF is the reference for which the Equations will be generated.
  - PARAMETER contains the new information of the variable.
  - ISPACE is the iteration space.

  This function generates the Cold Miss Bounds Equations for a given variable when considering a self spatial reuse vector.
- **void ReGenerate_CMB** (**CRef &**TRAILING, **CVariable &**VAR_TRAILING, **CISpace &**ISPACE, **CRef &**LEADER) where
  - TRAILING is the reference for which the Equations will be generated.
  - VAR_TRAILING contains the new information of the variable.
  - ISPACE is the iteration space.
  - LEADER is the reference that reuses from the TRAILING along the given reuse vector.

  This function generates the Cold Miss Bounds Equations for a given variable when considering a group spatial reuse vector.
- **void ReGenerate_R** (**CRef &**REF, **CVariable &**VAR, **CRef ****REFS, **CLlista<CVariable> &**VARS) where
  - REF is the reference for which the Replacement Equations will be generated.
  - VAR is the variable associated to the reference.
  - REFS is the list of references in the loop.
  - VARS is the list of variables in the loop.

  This function generates the Replacement Equations for the new configuration of the variable.

7. **Class CEquation_C**

   The Cold Miss Equations.

   Constructors:

   - **CEquation_C** (**void**)**:CListElem()**
   - **CEquation_C** (**unsigned int** i, **int** coef_i, **int** rhs)
     where
     - i is the induction variable restricted by the equation.
     - coef_i is the sign of the variable in the equation.
     - rhs is the right hand side of the equation.

Member Access:

- **unsigned int i** (**void**) **const**
  Returns the induction variable that's involved in the equation.
- **unsigned int i** (**unsigned int** NEW)
  The new variable restricted by the equation is the variable NEW.
- **int coef_i** (**void**) **const**
  Returns the coefficient of the induction variable in the equation.
- **int coef_i** (**int** NEW)
  Sets the coefficient of the induction variable in the equation to the value NEW.
- **int rhs** (**void**) **const**
  Returns the right hand side of the equation.
- **int rhs** (**int** NEW)
  Sets the right hand side of the equation to the value NEW.

Clone:

- **virtual CListElem *clone** (**void**)

Functions:

- **bool Emptiness** (**CIPoint &**IPOINT) **const** where
  - IPOINT is the iteration point that is being studied.

  This function checks whether the iteration point IPOINT is not a solution to the equation.

8. **Class CEquation_CMB**

The Cold Miss Bounds Equations.

Constructors:

- **CEquation_CMB** (**void**)
- **CEquation_CMB** (**unsigned int** NEST)
  where
  - NEST is the depth of the loop nest.

Clone:

- **virtual CListElem *clone** (**void**)

Destructor:

- **CEquation_CMB** (**void**)

Member access:

- **Value lb** (**void**) **const**
  Returns the lower bound of the expression in the equation.
- **Value lb** (**Value** NEW)
  Sets the lower bound of the expression to the value NEW.
- **Value ub** (**void**) **const**
  Returns the upper bound of the expression in the equation.
- **Value ub** (**Value** NEW)
  Sets the upper bound of the expression to the value NEW.

- **unsigned int nest** (**void**) **const**
  Returns the depth of the loop nest.
- **Value & z** (**void**) **const**
  Returns the coefficient of the variable z.
- **Value & z** (**void**)
  Allows to change the value of the coefficient of the variable z.
- **Value & i** (**unsigned int** k) **const**
  Returns the coefficient of the k-th induction variable.
- **Value & i** (**unsigned int** k)
  Allows to change the value of the coefficient of the k-th induction variable.
- **unsigned int empty** (**void**) **const**
  Returns whether the polyhedron that defines the equation is empty.
- **unsigned int empty** (**unsigned int** NEW)
  This function allows to set change the parameter of the equation which says whether the equation has integer solutions or not.

Empty Criteria:

- **bool CMB_Empty** (**CISpace &**ISPACE)
  Checks whether polyhedron defined by the equations is empty.
- **bool Emptiness** (**CIPoint &**IPOINT) **const**
  Checks the emptiness of the convex region.

9. **Class CEquation_R**

The Replacement Equations.

Constructors:

- **CEquation_R** (**void**)
- **CEquation_R** (**unsigned int** NEST, **unsigned int** NJ, **CISpace &**ISPACE) where
  - NEST is the depth of the loop nest.
  - NJ is the number of j variables involved in the equations.
  - ISPACE is the iteration space.

Clone:

- **virtual CListElem *clone** (**void**)

Destructor:

- **CEquation_R** (**void**)

Member access:

- **unsigned int nest** (**void**) **const**
  Returns the depth of the loop nest.
- **unsigned int nest** (**unsigned int** NEW)
  Sets the depth of the loop nest to the value NEW.
- **unsigned int nj** (**void**) **const**
  Returns the number of variables j in the equations.

- **unsigned int nj** (**unsigned int** NEW)
  Sets the number of variables j involved in the equations to NEW.
- **SIGN sig_n** (**void**) **const**;
  Returns the valid sign of the variable n.
- **SIGN sig_n** (**SIGN** NEW)
  Sets the sign of the variable n to SIGN NEW
- **unsigned int empty** (**void**) **const**
  Returns whether the polyhedron that defines the equation is empty.
- **unsigned int empty** (**unsigned int** NEW)
  This function allows to set change the parameter of the equation which says whether the equation has integer solutions or not.
- **Value history** (**unsigned int** k) **const**
  When in the convex regions we find the case $j = i - r$, the variable j is fixed ([2]). This function returns the value r corresponding to the k-th variable j.
- **CLlista<CPseudoConvexRegion> &convex_regions** (**CLlista<CPseudoConvexRe**
  ***NEW)**
  This function sets the convex regions of the equation to a given one (NEW).
- **CLlista<CPseudoConvexRegion> &convex_regions** (**void**) **const**
  Returns the convex regions of the Replacement Equation.

Member access:

- **Value &i** (**unsigned int** eq, **unsigned int** k) **const** where
  - eq gives the equation form the Replacement we refer to.
  - k gives the induction variable.

  This function returns the coefficient of the k-th induction variable in the equation eq from the Replacement Equations.
- **Value &i** (**unsigned int** eq, **unsigned int** k)
  This function allows to change the coefficient of the k-th induction variable in the eq-th equation from the Replacement Equations.
- **Value &j** (**unsigned int** k) **const**
  Returns the coefficient of the k-th variable j in the equations.
- **Value &j** (**unsigned int** k)
  This function allows to change the coefficient of the k-th variable j in the equations.
- **Value &ub** (**unsigned int** eq) **const**
  Returns the upper bound of the eq-th equation from the Replacement.
- **Value &ub** (**unsigned int** eq)
  Allows to change the upper bound of the eq-th equation from the Replacement.
- **Value &lb** (**unsigned int** eq) **const**
  Returns the lower bound of the eq-th equation from the Replacement.
- **Value &lb** (**unsigned int** eq)
  Allows to change the lower bound of the eq-th equation from the Replacement.
- **Value &z** (**void**) **const**
  Returns the coefficient from the variable z.
- **Value &n** (**void**) **const**
  Returns the coefficient from the variable z.

- **Value &n** (**void**)
  Function for changing the coefficient of the variable n.
- **CISpace &ispace** (**void**) **const**
  Returns the iteration space.

Computation of the convex regions:

- **CEquation_R* Fix_j** (**unsigned int** k, **int** POSITION, **CReuseVector** &REUSE_VECTOR, **int** extra)
  This function creates another equation from a replacement equation, where all the possible variables j are fixed. This is done as follows: Variables $j_1, \ldots, j_{k-1}$ are fixed using the identity $j_s = i_s$. Variables $j_k, \ldots, j_{POSITION}$ are fixed using the identity $j_s = i_s - r_s$, where r is the reuse vector given by REUSE_VECTOR. Finally, the parameter extra is used in order to create open or closed intervals for the domains of the different variables j.
- **void Compute_Convex_Regions** (**CReuseVector** &REUSE_VECTOR, **unsigned int** Ref, **INTERVAL_TYPE** left, **INTERVAL_TYPE** right, **unsigned int** _delete) where
  - REUSE_VECTOR is the reuse vector we are considering when building the Replacement Equation.
  - Ref is the position of the actual Replacement in the list of Replacement Equations from the reuse vector.
  - left tells us whether the interval is open or closed on the left side.
  - right tells us whether the interval is open or closed on the right side.
  - _delete is set to 0 if the replacement equations that correspond to the actual convex regions have to be removed from the list in case they are found to be empty. Otherwise _delete=1, which is its default value.

  This function computes the convex regions of a Replacement Equation.

Empty Criteria:

- **bool CR_Empty** (**unsigned int** k, **unsigned int** UI, **unsigned int** LI, **int** LB, **int** UB, **CISpace** &ISPACE)
  where
  - k gives the variable j for which the convex region is being computed.
  - UI is the coefficient of the variable i_k in the equation of the convex region that gives the upper bound of the expression $j_k - i_k$.
  - LI is the coefficient of the variable i_k in the equation of the convex region that gives the lower bound of the expression $j_k - i_k$.
  - LB is the lower bound of $j_k - i_k$
  - UB is the upper bound of $j_k - i_k$

  This function checks whether the convex regions are empty.
- Functions for detecting Replacement Equations with no integer solutions.
  - **bool Set_n_Sign** (**CVariable** &A, **CVariable** &B)
    where A and B are the variables of the references whose interferences are modelled by the actual Replacement Equation.
    This function restricts the variable n to the possible sign it can have, so that the equation is not empty.

- **bool P_Empty** (**void**)
- **bool N_Empty** (**void**)

5.2 Solver Module

In this module we find different classes and functions needed for solving the CMEs. We have divided them in three parts: the implementation of the sampling techniques, the functions for actually solving the CMEs and finally different classes to store the information obtained when calling the previous functions.

5.2.1 Sampling

1. **Class CIPoint**

   The iteration points that are studied to solve the CMEs.

   Constructors:

   - **CIPoint** (**void**)
   - **CIPoint** (**unsigned int** DIM, **int** *VECTOR) where
     - DIM is the dimension of the point, which is the same as the depth of the loop nest.
     - VECTOR gives the components of the point in the iteration space.

   Clone:

   - **virtual CListElem *clone** (**void**)

   Destructors:

   - **CIPoint** (**void**)

   Member access:

   - **unsigned int dim** (**void**) **const**
     Returns the dimension of the point.
   - **int &operator ()** (**unsigned int** k) **const**
     Returns the k-th component of the point.
   - **int &operator ()** (**unsigned int** k)
     This function allows to change the k-th component of the iteration point.

   Functions:

   - **int compare** (**CIPoint &**OTHER)
     Compares the actual iteration point with the iteration point OTHER.

2. **Class CSample**

   The sample of iteration points that is created in order to avoid studying all the iteration space when solving the equations.

   Constructors:

   - **CSample** (**void**)

- **CSample** (**double** CONFIDENCE, **double** I_WIDTH, **CISpace &**ISPACE) where ISPACE is the iteration space.
  This function generates a sample of iteration points for the chosen confidence (CONFIDENCE) and interval width (I_WIDTH).
- **CSample** (**CISpace &**ISPACE)
  This function is used when the whole iteration space must be studied.

Destructors:

- **CSample** (**void**)

Members Access:

- **unsigned int elements** (**void**) **const**
  Return the number of points in the sample.
- **unsigned int elements** (**unsigned int** NEW)
  Sets the number of points in the sample to the value NEW.
- **unsigned int entries** (**void**) **const**
  Returns the number of iteration points that are studied for the current reuse vector.
- **double confidence** (**void**) **const**
  Returns the confidence that has been chosen.
- **unsigned int to_do** (**void**) **const**
  Returns the number od iteration points that will be studied when considering the next reuse vector.
- **CIPoint &operator ()** (**unsigned int** k)
  Returns the k-th point of the sample.
- **void add_current** (**void**)
  Adds the current point to the sample.
- **void delete_current** (**void**)
  Removes the current point from the sample.
- **void next_vector** (**void**)
  Changes the set of points to be studied to the set of points that have not been determined by the previous reuse vector.
- **void reset_sample** (**void**)
  Resets the sample to its original form.

5.2.2 Solver

We have developed different functions for solving CMEs depending on the information we want to obtain as well as the performance requirements of the user.

- **unsigned int Number_Misses (CLoop &loop, CSample &sample);**

  This function returns the total number of misses for the given loop when considering a direct mapped cache.

- **unsigned int Number_Misses (CLoop &loop, CSample &sample, unsigned int k_way);**

This is the associative version of the previous function. The motivation for having two different functions for the different types of cache configuration is that the way of solving the equations is different in both cases. We separate them for efficiency reasons.

- **unsigned int Number_Misses (CLoop &loop, CSample &sample, ostream &o, unsigned int k_way);**

  This function writes in the given output stream all kind of information about the number of equations per reference and reuse vector, as well as the misses per reference and per loop. It works for all kinds of cache configurations. In this case, as well as in the following one, we have not distinguished between direct and associative caches, since the efficiency is already seriously decreased by all the serializing work.

- **CMLoop* Number_Misses_CM (CLoop &loop, CSample &sample, unsigned int k_way);**

  This function creates a CMLoop structure that contains all the information about the misses in the loop.

5.2.3 Misses storage

1. **Class CMRef**

   This class contains information on the misses per reference: the number of compulsory misses, replacement misses, spatial and temporal misses as well as the total number of misses and accesses. Besides, it contains the confidence interval.

   Constructor:

   - **CMRef** (**void**)

   Destructor:

   - **CMRef** (**void**)

   Member access:

   - **unsigned int &compulsory** (**void**)
     Returns the number of compulsory misses.
   - **unsigned int &replacement** (**void**)
     Returns the number of replacement misses.
   - **unsigned int &spatial** (**void**)
     Returns the number of spatial misses.
   - **unsigned int &temporal** (**void**)
     Returns the number of spatial misses.
   - **unsigned int &total** (**void**)
     Returns the total number of misses for the current reference.
   - **unsigned int &accesses** (**void**)
     Returns the number of accesses of the reference, which is needed in order to compute the miss ratio.

   Functions:

- **void compute_interval** (**double** ALPHA)
  where ALPHA is 1-CONFIDENCE, being CONFIDENCE the desired confidence.
  This function computes the confidence interval.

2. **Class CMLoop**

   This class contains the same information about the misses as CMRef, but when considering the whole loop.

   Constructors:

   - **CMLoop** (**void**)
   - **CMLoop** (**unsigned int** REFERENCES)
     where REFERENCES is the number of references in the loop.

   Destructor:

   - **CMLoop** (**void**)

   Member access:

   - **unsigned int &compulsory** (**void**)
     Returns the number of compulsory misses.
   - **unsigned int &replacement** (**void**)
     Returns the number of replacement misses.
   - **unsigned int &spatial** (**void**)
     Returns the number of spatial misses.
   - **unsigned int &temporal** (**void**)
     Returns the number of temporal misses.
   - **unsigned int &total** (**void**)
     Returns the total number of misses for the current loop.
   - **unsigned int &accesses** (**void**)
     Returns the total number of accesses of the references in the loop, which is needed in order to compute the miss ratio.
   - **unsigned int references** (**void**) **const**
     Returns the number of references in the loop.
   - **CMRef &operator()** (**unsigned int** k) **const**
     Access to the k-th reference in the loop.

   Functions:

   - **void Compute_Interval** (**double** ALPHA)
     where $ALPHA = 1.0 - CONFIDENCE$, being CONFIDENCE the desired confidence.

References

[1] Eduard Ayguadé et al. *A uniform internal representation for high-level and instruction-level transformations.* UPC, 1995.

[2] Nerina Bermudo, Xavier Vera, Antonio González, and Josep Llosa. An efficient solver for cache miss equations. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'00)*, 2000.

[3] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: an analytical representation of cache misses. In *International Conference on Supercomputing (ICS'97)*, 1997.

[4] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*, 1998.

[5] David Padua et al. *Polaris developer's document*, 1994.

[6] Xavier Vera, Josep Llosa, Antonio González, and Nerina Bermudo. A fast and accurate approach to analyze cache memory behavior. In *European Conference on Parallel Computing (Europar'00)*, 2000.

[7] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN91*, 1991.