HAL
open science

Extended Performance Analysis

Abella Jaume, Nerina Bermudo, Carlos Ciuraneta, J.M. Codina, Christine Eisenbeis, Antonio Gonzalez, Josep Llosa, Andry Randrianatoavina, François Thomasset, Sid Touati, et al.

To cite this version:

Abella Jaume, Nerina Bermudo, Carlos Ciuraneta, J.M. Codina, Christine Eisenbeis, et al.. Extended Performance Analysis. [Research Report] 1999, pp.47. hal-00647446

HAL Id: hal-00647446

https://inria.hal.science/hal-00647446v1

Submitted on 2 Dec 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

MHAOTEU: Memory Hierarchy Analysis and Optimization Tools for The End-User

REPORT M2.D1

Extended Performance Analysis

Jaume Abella, Nerina Bermudo, Carlos Ciuraneta, Josep M. Codina,
Christine Eisenbeis, Antonio Gonzalez, Josep Llosa, Andry Randrianatoavina,
François Thomasset, Sid Ahmed Ali Touati, Xavier Vera.
INRIA, UPC

December 13, 1999

Abstract

The gap between processors and main memory performance increases every year. In order to overcome this problem, cache memories are very useful. Compile-time program transformations can significantly improve the performance of the cache. To apply most of these transformations, the compiler requires a precise knowledge of the locality of the different sections of the code, both before and after being transformed. This report describes analysis techniques aiming at providing explanations to performance of a program, as far as the cache is concerned.

General Reference Windows (GRWs) gives a measure of the amount of memory references that are "live" at any point in the execution of the program, that is, references previously loaded into the cache, and to be reused later on. Clearly if the size of the window exceeds the capacity of the cache at some point, then cache misses are due to occur at this point. GRWs can be expressed through sets of linear inequations, which one can view as polyhedra, whose number of integer points have to be computed; for this task we can use the Polylib library. The first part of this reports explains the difficulties encountered in computing the GRWs, which improvements are required from the Polylib, and how we plan to circumvent the problems. We also describe a dynamic computation of GRWs, that may be an alternative complementing the static analysis.

The second part concerns Cache Miss Equations (CME) which allow to obtain an analytical and precise description of the cache memory behavior for loop-oriented codes. Describing the cache behavior by means of diophantine equations allows us to use mathematical techniques to obtain cache misses.

Unfortunately, a direct solution of the CME is computationally intractable due to its NP-hard nature. In this work we present effective techniques that exploit some properties of the particular polyhedra generated by CME. Such techniques reduce the complexity of the algorithm to solve CME from exponential to polynomial, which results in a significant speed-up when compared with traditional methods.

We propose a fast and accurate approach to estimate the solution of the CME, which is based on the use of sampling techniques. Statistical techniques allow us to approximate the absolute miss ratio of each reference by analyzing a small subset of the iteration space. The size of the subset, and therefore the analysis time, is determined by the accuracy selected by the user. The results show that only a few seconds are required to analyze most of the SPECfp benchmarks with an error smaller than 0.01.

Contents

I Generalized Reference Windows **1**

1 Introduction **1**

2 Generalized Reference Window - Definitions and parameterization **1**
2.1 Reference window . . . 2
2.1.1 Domains "before" and "after" . . . 3
2.1.2 Projection . . . 3
2.1.3 Intersection . . . 5
2.1.4 Union . . . 5
2.2 Spatial reference window . . . 5
2.3 Generalized Reference Window . . . 5
2.4 Conclusions for static window computation and parameterization. . . . 6

3 Dynamic computation of GRW **6**
3.1 dgrw . . . 11
3.2 Example of application . . . 11

4 Conclusion **11**

II Cache Miss Equations **16**

5 Introduction **16**

6 Cache Miss Equations Overview **17**
6.1 Compulsory Equations . . . 18
6.1.1 Cold Miss Equations . . . 18
6.1.2 Cold Miss Bounds . . . 18
6.2 Replacement Miss Equations . . . 19
6.3 Finding Cache Misses from CM Equations . . . 20

7 Optimizing CME polyhedra **22**
7.1 Background . . . 22
7.1.1 Definitions . . . 22
7.1.2 Empty Polyhedra . . . 23
7.1.3 Counting integer points . . . 23
7.2 Removing Empty Polyhedra . . . 24
7.2.1 Cold Miss Equations . . . 24
7.2.2 Cold Miss Bounds . . . 24
7.2.3 Replacement Equations . . . 25
7.3 Analyzing Iteration Points . . . 27
7.3.1 Cold Miss Equations . . . 27
7.3.2 Cold Miss Bounds . . . 27
7.3.3 Replacement Equations . . . 27

8 Solving CME using sampling **31**
8.1 Sampling 31
8.2 Statistical overview 31
8.2.1 Discrete random variables 31
8.2.2 Parameters estimation 32
8.3 CME particularization 33
8.4 Generating samples 34

9 Evaluation **34**
9.1 CME polyhedra 35
9.1.1 Empty Polyhedra 35
9.1.2 Analyzing Iteration Points 36
9.2 Solving CME 37
9.2.1 Implementation 37
9.2.2 Example 38
9.2.3 Performance evaluation 38

10 Conclusions **41**

Part I
Generalized Reference Windows

1 Introduction

Generalized Reference Windows theory is an approach intended to evaluate data locality in programs. Unlike Cache Miss Equations, it does not evaluate the number of cache misses in a given program, but is more intended to highlight where bottlenecks may occur.

The approach amounts to establish linear integer equations and inequalities, that are parameterized in terms of time, set number, programs parameters, schedule, array padding. The number of solutions of this system gives a locality criterion that can give hints on how the program should be transformed for improving data locality.

For computing this number of solutions we have started to use the Polylib library that manipulates parameterized integer polyhedra, and specially the computation of the Erhart polynomial for counting number of integer solutions. Unfortunately, very much like Cache Miss Equations, we found that this library is not yet in such a state so that it can be blindly used.

In this report we explain where the difficulties for computing Generalized Reference Windows are lying, and which improvements are required from the Polylib library. Basically, difficulties are twofold. The first problem comes from computation of projections of polyhedra, and more precisely from integer division and modulo operations that describe respectively line number and set number. The second problem is that GRW describes the number of memory lines that comptte for the same cache set and is therefore expected to be small in order of magnitude. This makes the integer system not well posed for Polylib. Since computation is very complex and actually impossible currently, we describe another approach that we have used for studying actual connections between Generalized Reference Windows and program performance. This approach is a dynamic computation of GRW, that gives interesting insights about the data locality properties of the program.

Since our work is still under development, this report should be read as intermediate. We first recall the definition of GRW, and describe more precisely than in the previous report how optimization and program parameters can be taken into account, as well as the difficulties we had to face for implementing automatic computation of Generalized Reference Windows. Then the current implementation and specially connection to the MHAOTEU server is described. Last the dynamic GRW approach is presented with examples of use for program optimization.

2 Generalized Reference Window - Definitions and parameterization

We now recall the basic definitions of GRW — a general presentation can be found in [3]. Although GRW computation can be extended to general programs, as presented in [3], we concentrate in this report on perfectly nested loops. Statements in the loop body are numbered $S_1, S_2, ...S_p$. For convenience it is assumed that only at most one memory reference, denoted as A_S, is performed at any statement S. Iteration domain D is a convex

polyedron possibly linearly parameterized by non numeric program variables — we assume that the loop has been normalized so that all integer points within the domain are iteration points. Any program point is then characterized by some statement S and some iteration point $\overrightarrow{i}$, with corresponding memory reference $Mem_S(\overrightarrow{i})$. We denote as A_S the reference performed at statement S, or A if no confusion is possible. Then $A_S(h_S(\overrightarrow{i}))$ is the memory access performed at statement $S(\overrightarrow{i})$, where $h_S(\overrightarrow{i})$ is an affine function representing the offset of memory access to array A at statement S. We further denote as $Mem_S(\overrightarrow{i})$ the absolute address in memory of array access at operation $S(\overrightarrow{i})$.

Let us denote now C, a, L, and N respectively cache size, cache associativity, line size and set number, in terms of memory words. Then we have $C = a \times L \times N$. The block number $BN(S(\overrightarrow{i}))$ accessed at $S(\overrightarrow{i})$ is defined as

$$BN(S(\overrightarrow{i})) = \left\lfloor \frac{Mem_S(\overrightarrow{i})}{L} \right\rfloor$$

and the cache set number $CN(S(\overrightarrow{i}))$ is

$$CN(S(\overrightarrow{i})) = BN(S(\overrightarrow{i})) \bmod N.$$

For two operations $S(\overrightarrow{i})$ and $T(\overrightarrow{j})$, $S(\overrightarrow{i}) \leq T(\overrightarrow{j})$ means that $S(\overrightarrow{i})$ is executed before $T(\overrightarrow{j})$ in the order specified by the program.

2.1 Reference window

The reference window is defined independently of any cache configuration. At any point in the program it represents the sets of words that are alive, in the sense that they have been referenced in the past relatively to current program point, and will be referenced again the the future. The size of the reference window represents the minimum size of the local memory required for avoiding reloading data during computation.

The reference window relative to $S_0(\overrightarrow{i_0})$ is:

$$\begin{aligned} W_{S_0(\overrightarrow{i_0})} &= \{\ m \in M \ \mid\ \exists S, T,\ \exists \overrightarrow{i}, \overrightarrow{j},\ S(\overrightarrow{i}) \leq S_0(\overrightarrow{i_0}) \leq T(\overrightarrow{j}), \\ &\qquad\qquad m = Mem_S(\overrightarrow{i}) = Mem_T(\overrightarrow{j})\ \} \\ &= \{\ m \in M \ \mid \exists S, \exists \overrightarrow{i},\ S(\overrightarrow{i}) \leq S_0(\overrightarrow{i_0}), m = Mem_S(\overrightarrow{i})\ \} \\ &\qquad \cap \{\ m \in M \ \mid \exists T, \exists \overrightarrow{j},\ S_0(\overrightarrow{i_0}) \leq T(\overrightarrow{j})\ \} \end{aligned}$$

Let us note as $D^-_{S_0(\overrightarrow{i_0})}$ and $D^+_{S_0(\overrightarrow{i_0})}$ the set of statements that are executed respectively before and after $S_0(\overrightarrow{i_0})$. Then $W_{S_0(\overrightarrow{i_0})}$ can be summarized as:

$$W_{S_0(\overrightarrow{i_0})} = (\ \bigcup_S Mem_s(D^-_{S_0(\overrightarrow{i_0})})\ ) \bigcap (\ \bigcup_S Mem_s(D^+_{S_0(\overrightarrow{i_0})})\ )$$

or, equivalently,

$$\bigcup_{S_1,S_2}(\ Mem_{S_1}(D^-_{S_0(\overrightarrow{i_0})}) \bigcap Mem_{S_2}(D^-_{S_0(\overrightarrow{i_0})})\ )$$

The reference window is therefore a union of intersections of projections of polyedra. It is important to detail all the steps in such a computation, with respect to possible parameterization.

2.1.1 Domains "before" and "after"

Domains $D^-_{S_0(\overrightarrow{i_0})}$ and $D^+_{S_0(\overrightarrow{i_0})}$ are polyedra such as the one pictured on figure 4. If there are no program parameters and bounds of polyhedra are purely numerical, then these polyhedra are well defined, and time can be linearized because we know numerical values for bounds. If however there are non numerical parameters, then these domains are no more polyhedra, but unions of polyedra, as already noted in the previous report. There are two ways to overcome this difficulty. One is to actually consider parameterized polyedra of this union one by one, but it adds a level of complexity in the computation. The other possibility is to put artificial (large) bounds for these domains in order to enable time linearization. This is the way we do currently. This allows to keep program parameters as parameters of the computation, and also to keep program point $S_0(\overrightarrow{i_0})$ as a parameter. However, these artificial large bounds introduce large numbers in the system, that were not yet handled correctly by the Polylib library until very recently. There is now a new version using the GNU-MP library, that performs integer computations with infinite precision, and we are planning to use it.

Let us note that if we are able to compute the Erhart polynomial of such a system, parameterized with loops bounds, then we have a closed formula that we can use for choosing "goods" bounds. When the blocking transformation is valid, we can then look for the best blocking factor so that the reference window size is less than the size C of the cache memory. This is a way to avoid capacity misses.

"Before" and "after" are notions that are relative to the current schedule of iterations. We can however think about parameterizing this schedule. This is done through the linearized timing function. Instead of choosing a sequential schedule, we can write the timing function as $T_S(\overrightarrow{i}) = c_S + \overrightarrow{d}^t_S.\overrightarrow{i}$ where c_S is some numerical constant, and $\overrightarrow{d_S}$ some numerical vector. We can not just handle c_S and d_S directly yet as parameters because they create non linear terms. Instead one could think of iteratively searching for values that make the window size smaller than the size of the cache memory, so as to eliminate capacity misses.

2.1.2 Projection

Once we have computed iteration domains, we have to compute their image by h_{S_1}. This is the main difficulty in the computation because basically the projection by an affine function of integer points of a polyedron are not necessarily integer points of a polyedron, or more specifically, they are not integer points of the projection of the convex hull of the initial polyedron. There are here also two difficulties in this problem. The first one comes from the fact that the projection may be a dilatation of the space. For instance $h_S(i) = 2i$. In this case only even points are touched. This difficulty is easily overcomed by changing the basis of the image space, or equivalently realizing that the image of the convex hull is the intersection of a polyedron and an integer lattice. The Polylib library handles this case. The second one is more complex to handle and is illustrated on figure 1. In this case, one part of the polyedron is drawn, as well as its projection over the horizontal axis. In this case also not all integer points are touched, but not for the same reason as before. Unfortunately the Polylib library does not handle this case currently. Hence we must resort to the approximate version (take integer points in the projection of the convex hull). This

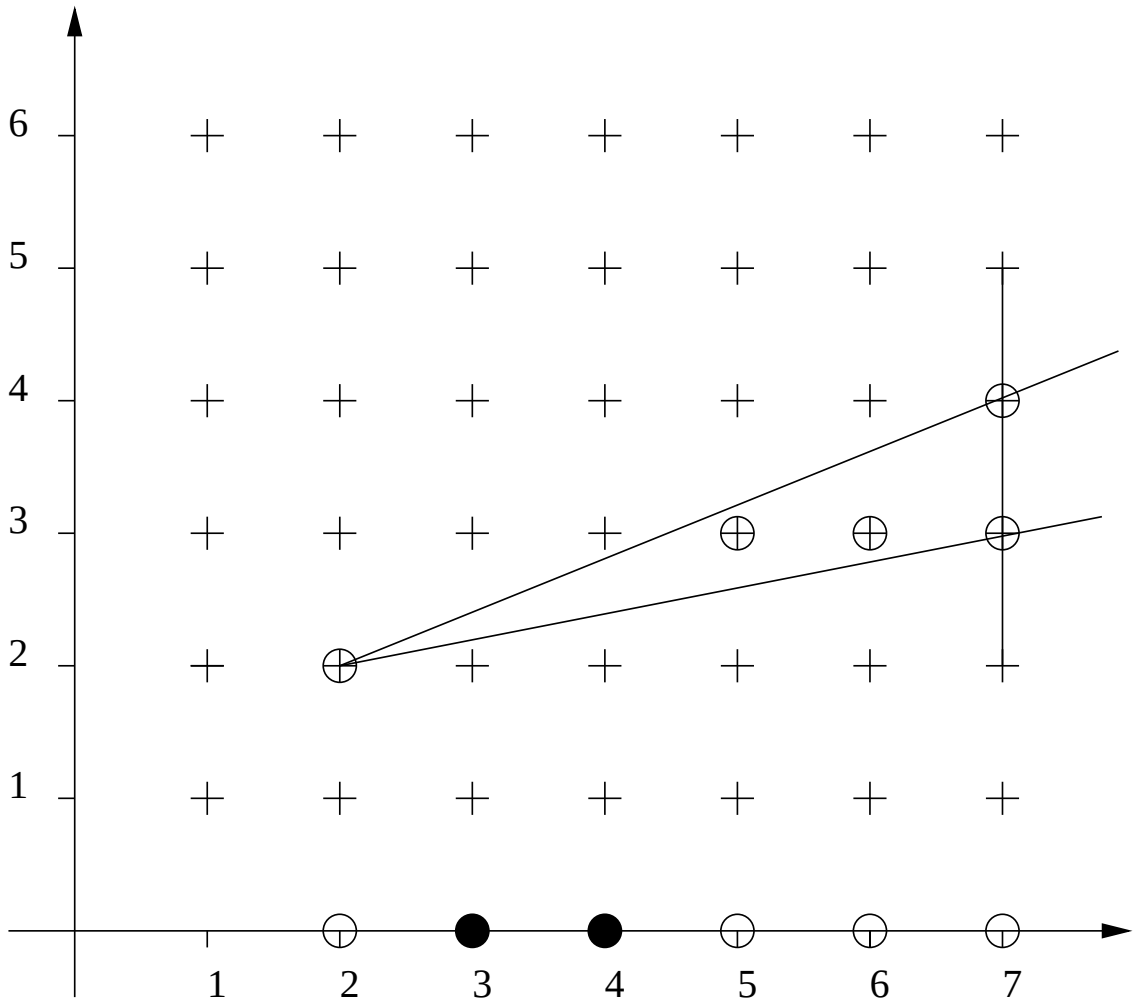


Figure 1: Example of non regular holes in the projection

is quite unsatisfactory, because this case frequently happens, especially when spatiality and associativity are taken into account, see sections 2.2 and 2.3 below. A next version of the Polylib should handle this case, we are also considering taking into account the specificities of the reference window so as to make possible and simplify the computation.

Function h_S is the one specified in the program, where Mem_S gives the scheme of memory allocation of the array accessed at statement S. We can also play with this function and keep it as a parameter as long as possible. Unfortunately, like in the case of parameterizing the scheduling function, this introduces non linear terms in the system. One particular case is padding, which amounts to take the base address of arrays as a parameter. This does not change the complexity of the problem (apart from the fact of adding as many parameters as there are arrays). More generally, we can choose either to keep it as given (i.e. possibly multidimensional), or to linearize it to get the absolute memory address in memory. It is often better not to linearize as long as possible, since this avoids problems in the projection; however linearization is mandatory when one considers mapping memory to cache (section 2.2).

2.1.3 Intersection

When the two projections are computed, one has to intersect them. Intersection is quite direct to handle in the case of integer points within polyedra. When the image is not a polyedron but the intersection of an integer lattice and a polyedron, then one should compute a common basis before projection of both polyedra. To our knowledge, this is not yet done in the Polylib currently.

2.1.4 Union

Computing unions of polyhedra is known to be a complex problem. As far as only the number of points is concerned, this is easier, as already noted in the section about Cache Miss Equations. However this is done at the price of computing a number of intersections, that is as large as 2^q, where q is the number of statements that access a given array. This should not be a difficulty as far as no parameters are taken into account. For the case of parameterized polyhedra, union computation should be available in the next version of the Polylib library.

2.2 Spatial reference window

We now take into account spatial locality, that is to say that instead of counting alive memory words, we count alive memory blocks (or memory lines). Let us denote as $SW_{S_0(\overrightarrow{i_0})}$ spatial reference window at program point $S_0(\overrightarrow{i_0})$. Then it is straightforward to compute the spatial reference window as the set of lines corresponding to reference window at program point $S_0(\overrightarrow{i_0})$.

$$SW_{S_0(\overrightarrow{i_0})} = \{\ l \ \mid\ \exists m \in W_{S_0(\overrightarrow{i_0})},\ l = \left\lfloor \frac{m}{L} \right\rfloor\ \}$$

Therefore the spatial reference window is the image of the reference window by the function: $D_L : m \longmapsto \lfloor \frac{m}{L} \rfloor$. Even though this function is not linear, it is not difficult to circumvent this problem. We just write that $l \times L \leq m \leq l \times (L+1) - 1$, multiply every equation or inequality where m appears (such as $m = Mem_S(\overrightarrow{i})$) by L, and replace m by l.

2.3 Generalized Reference Window

Now, if the cache is not associative, we have to add a new parameter, representing some cache set s, and observe how many memory lines that map to s are alive at some program point $S_0(\overrightarrow{i_0})$. Let M_s denote the set of memory lines that map to s, then the Generalized Reference Window relative to set s and program point $S_0(\overrightarrow{i_0})$ is $GRW_{S_0(\overrightarrow{i_0})} = SW_{S_0(\overrightarrow{i_0})} \cap M_s$. In other words,

$$GRW_{S_0(\overrightarrow{i_0})} \ = \ \{\ l \in SW_{S_0(\overrightarrow{i_0})},\ l \bmod N = s\ \}.$$

Hence it is the set of memory lines that map on set s by the (non linear) application): $E_N : l \longmapsto l \bmod N$. Again even though E_N is not linear, it is not difficult to handle this problem: we just add a new variable α in the system, and the new equation: $l = s + \alpha \times N$.

It is straightforward to check that the number of l satisfying the new system is exactly the number of pairs (l, α) that are solutions, since when α is fixed there is a one-to-one correspondance between l and α.

It should be noted that this scheme assumes cyclic memory mapping on lines. One could however also try to find another allocation of memory space onto the cache. We did not consider it yet.

2.4 Conclusions for static window computation and parameterization.

As presented above, we thought originally that generalized reference windows could be computed directly by the Polylib library, and yield the parameterized Erhart polynom. However, even very simple problems happen to produce overflow computations, or computations that do not finish. There are two main reasons for that. The first one is that to sequentialize the time, we introduce *large coefficients that make overflows*. As already mentioned, this first problem should be solved by using the GNU-MP infinite precision library. A second more delicate problem is related to the way the Erhart polynom is computed; first its form is determined, then its coefficients are computed by enumerating points in some basic cases, and solving the resulting equations. GRW happens to have generally a small number of points, and surprisingly to not vary very much in the interesting cases. This produces *degenerate cases* of polyhedra that are not handled in the current version of the Polylib. The way to solve this problem is to consider polynomials with lower degree.

Generalized Window Computation appeared much more complex than planned. Some features that we believed were available in the Polylib library were actually not yet implemented, because of time or because theoretical problems were not yet solved. Even though we thought that Generalized Window Computation were more complex to handle because of spatial locality and associativity, we found that the latter do not actually give specific problems. We are therefore coming back to the basic case of Reference Window and are considering different routes. For instance, using approximations like the ones described in [7], before applying on lines and sets, add linear parameters one by one, such as array padding, and for the non linear cases, instantiate the parameters before window computing. We are also aware of a new environment that has connected the PIP library [8] and the Polylib library. This environment now enables to compute projections of parameterized polyhedra.

Another route that we will also consider is enumerating, possibly by using the same methods as the ones used for the Cache Miss Equations.

The whole GRW computation is not completely performed by the MHAOTEU tools. The current version of GRW generates the equations and inequations defining GRWs, that are then passed to the Polylib library (figure 5).

Because static parameterized computation of GRW could not be realized yet, we have also started to consider computing it dynamically. This work is presented in the next section.

3 Dynamic computation of GRW

Because static tools were not yet available and we actually wanted to evaluate the usefulness of GRW, we have also considered dynamic evaluation of GRW. For that we have designed a

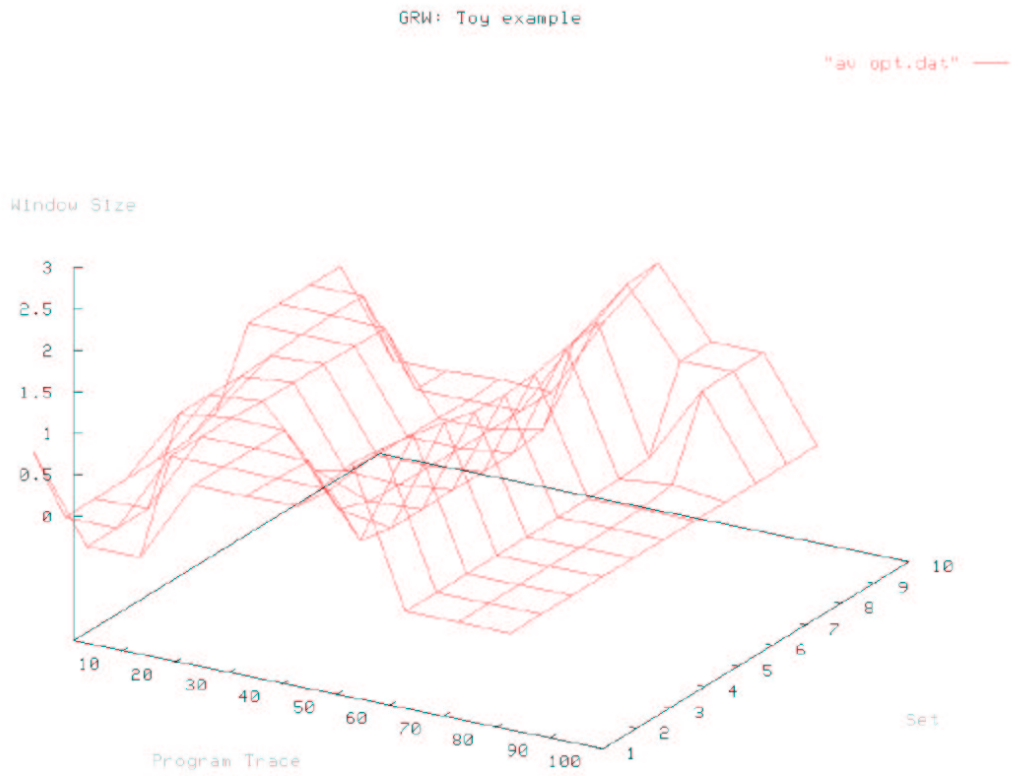


Figure 2: Window before optimization

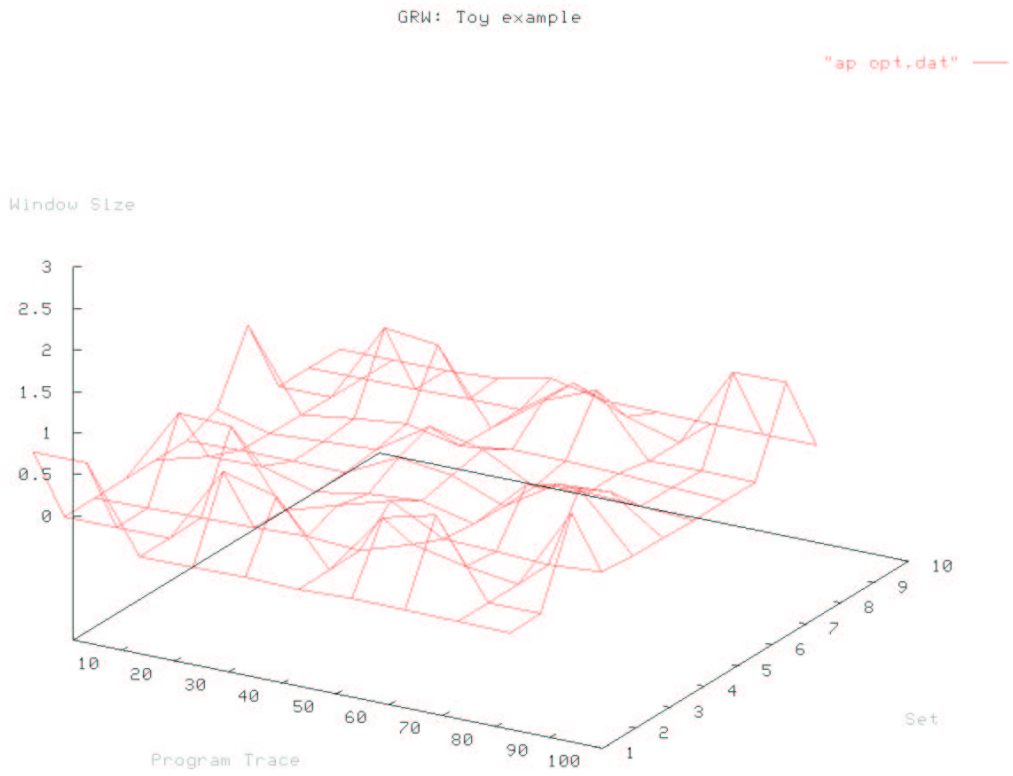


Figure 3: Window after optimization

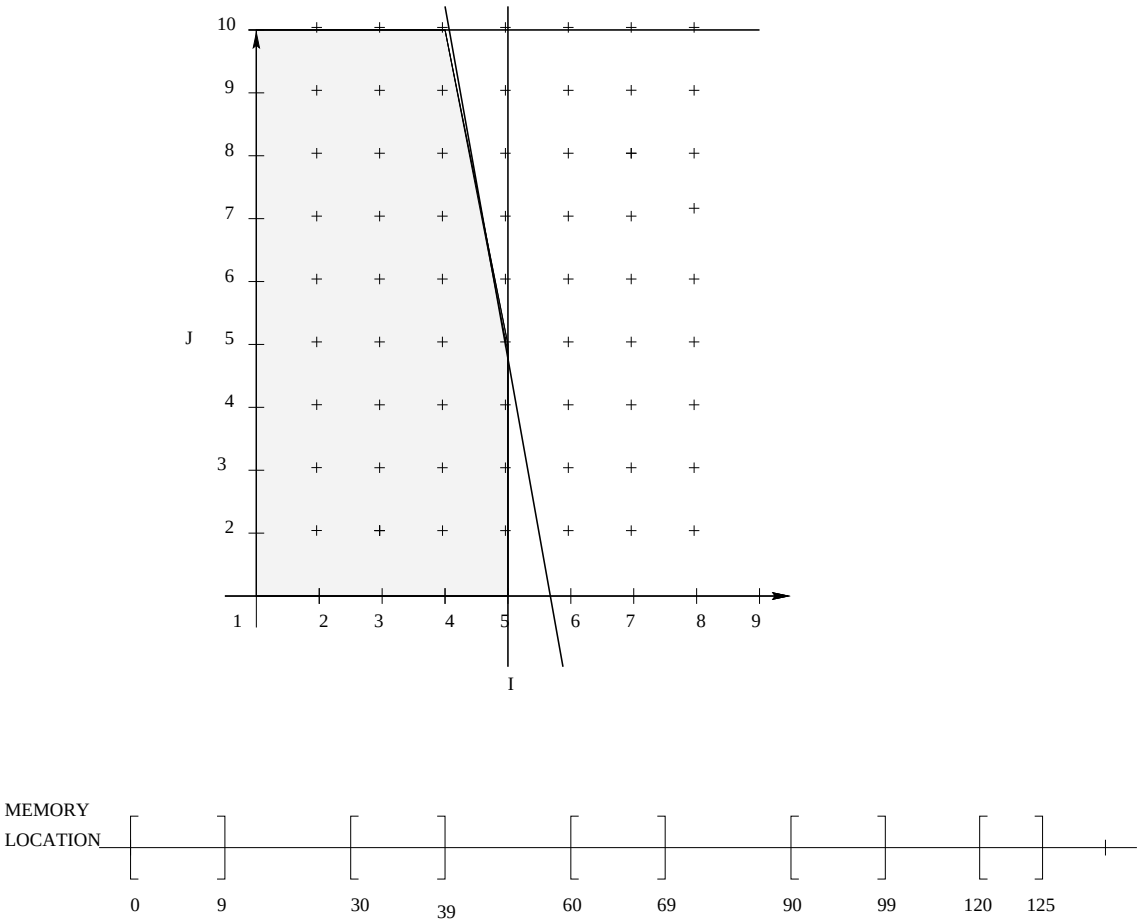


Figure 4: Domain “before” and its projection over memory

Figure 5: GRW server

```
telnet ariane 6666
Trying 128.93.39.1...
Connected to ariane.inria.fr.
Escape character is ' \^{}\{{]}\}'.
initiate grw
OK
infile yee.f
OK
procedure YEE
OK
loop 104:10
OK
loop 106:10
OK
line 107
OK
ref- 3
OK
ref+ 4
OK
set 0
OK
go
OK
terminate
OK
Connection closed by foreign host.
Exit 1
ariane:   >
15 8
 0 0 0 0 0 8 -1 0
 1 1 0 0 0 0 0 -2
2 -1 0 0 0 0 0 25
1 0 1 0 0 0 0 -2
 1 0 -1 0 0 0 0 25
1 -24 -1 0 0 0 0 250
2 0 0 1 0 0 0 -2
1 0 0 -1 0 0 0 25
 1 0 0 0 1 0 0 -2
1 0 0 0 -1 0 0 25
1 0 0 24 1 0 0 -250
1 1 101 0 0 0 -4 40702
 1 -1 -101 0 0 0 4 -40699
1 0 0 1 101 0 -4 40701
1 0 0 -1 -101 0 4 -40698
```

tool called dgrw (for dynamic generalized reference window computation). We first present the dgrw tool and show an example of use of dgrw, actually analysis of the application code from INRIA.

3.1 dgrw

dgrw is a tool that instruments programs in order to analyse behaviour of generalized reference windows. The goal is twofold: first to see whether we can correlate window size and number of cache misses/execution time; second to show graphically to the programmer the data locality in his/her program.

In a first pass, dgrw dumps variables lifetime in a file that is analysed in a second pass. Automatic instrumentation is performed by using the Sage++ transformation tools. In the second pass GRWs are represented graphically. On figure 6, axis represent sets and time. Points that are drawn are those where the window size is larger than respectively 1, 2, 3 or 4. The user can then point one set for instance and see the evolution over time of the GRW on this set (figure 7). Then he/she can also ask for those variables that are alive in this set (figure 8), and there are links to the source line where the current variable is referenced. GRW computation is done in the source program; however one can consider either logical addresses of arrays (base address computed from memory allocation given in declarations) or physical addresses or arrays, computed dynamically.

The graphical tool used here is R, that is a system for statistical computation and graphics. It consists of a language plus a run-time environment with graphics, a debugger, access to certain system functions, and the ability to run programs stored in script files.

Previously, we also attempted to use gnuplot (figures 2 and 3) and also Scilab for 3D graphics.

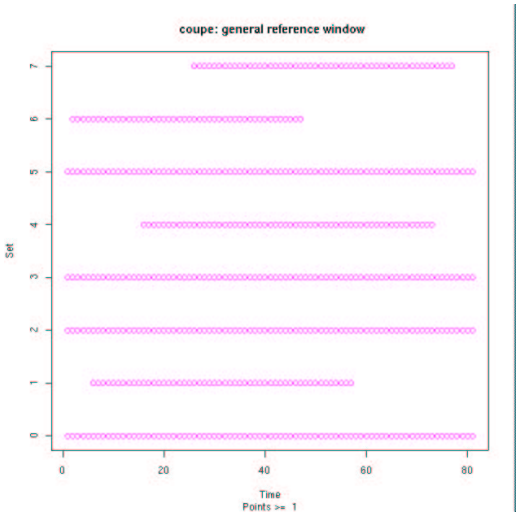
The current tool performs two passes. This enforces keeping a lot of information between the two passes, and is space consuming. We are now considering to design dgrw like the CVT performance debugger, and directly compute GRW on line.

3.2 Example of application

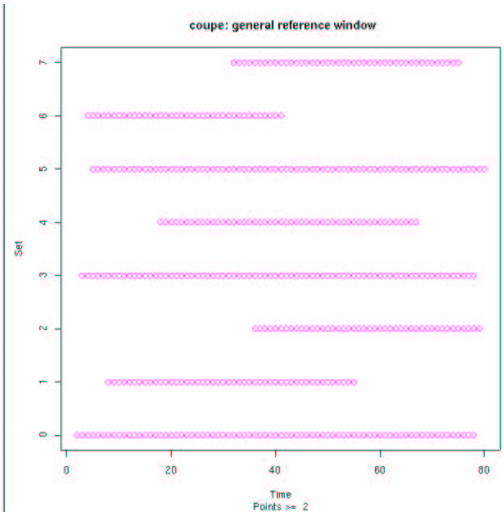
On the INRIA Osiris application code, we identified loops in procedure yee_par to be the most time consuming ones on both UltraSparc and Alpha. By using dgrw we found that GRW size was equal to one most of the time during these loops, except for the second loop where it became equal to two in some cases: this means that two variables are competing for the same set. Since cache is direct mapped on the Alpha processor, this means that conflict misses occur. It was clear that conflict was due to not reusing some variable (reuse $Hx(I,J) \rightarrow HX(I,J-1)$), as the set was used meanwhile by some other variable $sigma(I,J)$. By unroll-and-jamming twice the outermost loop, execution time dropped from 7 seconds to 6 seconds. And it could be verified that GRW size was then always less than one. More details are given in deliverable M2D3.

4 Conclusion

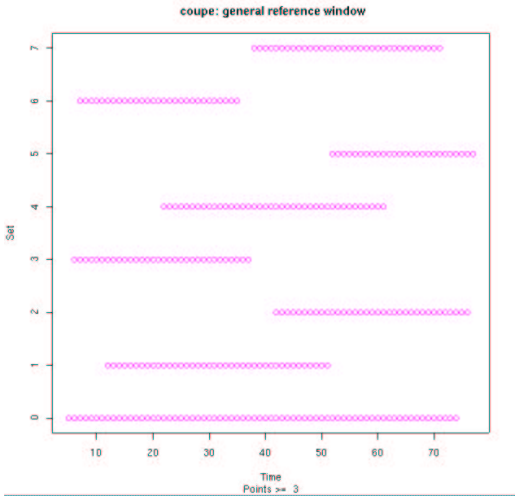
Static Generalized Reference Window computation was more complex than expected. We have presented in this report the main problems encountered when using the Polylib library,



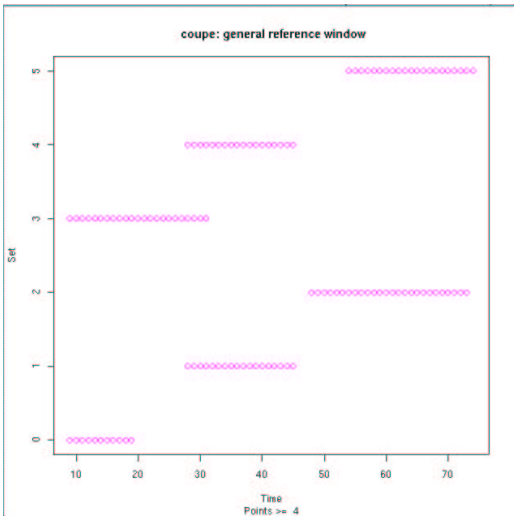
(a) GRW cut 1



(b) GRW cut 2



(c) GRW cut 3



(d) GRW cut 4

Figure 6: Window cuts

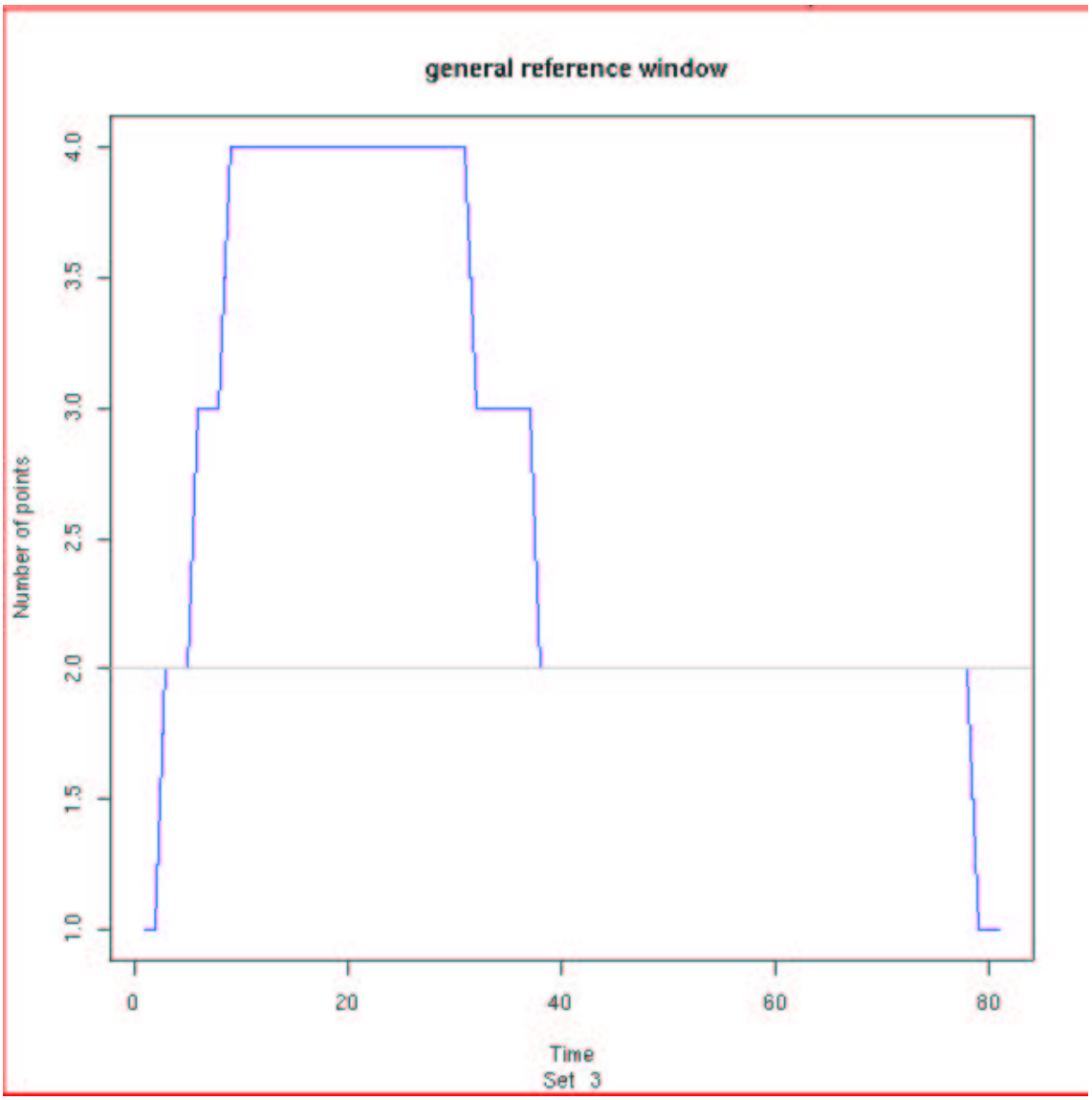
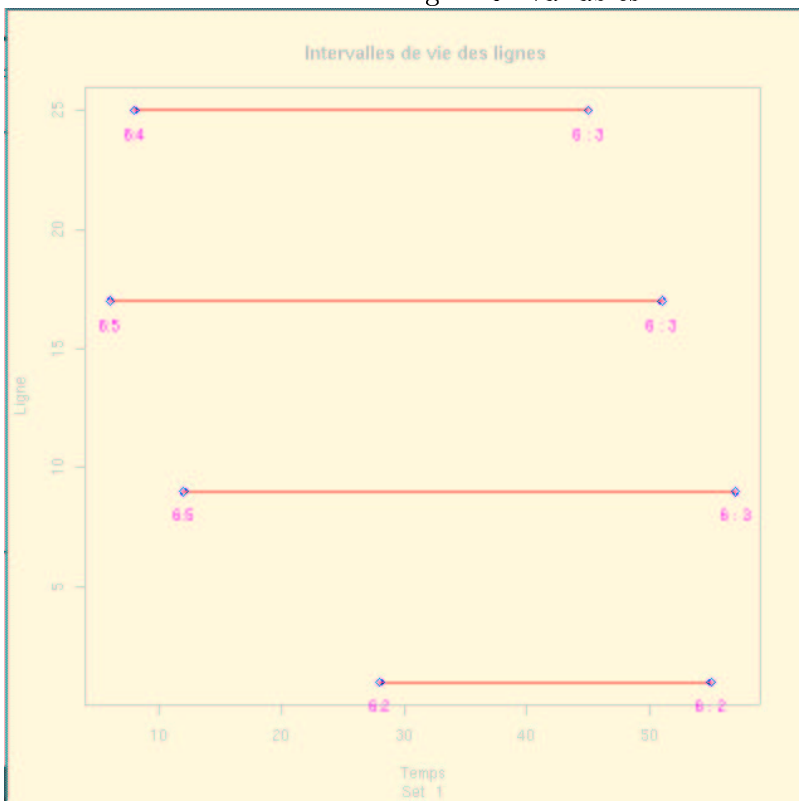


Figure 7: Variation over time of GRW on set 3

Figure 8: Variables lifetime



basically using large numbers that cause overflows, and the fact that GRWs are often degenerate, so that there are not enough data for computing coefficients of the Erhart polynomial. Surprisingly, we found that the heart of problems was not lying in mapping memory onto cache (and introducing integer division and modulo operation), this is why we plan to come back to implementing a simple version of Reference Windows, since General Reference Windows can be deduced very easily from Reference Windows. For the parameterized case, that is of most interest for choosing which transformation to apply, next versions of the Polylib library will be used, basically a combination of the PIP parametric linear systems solver and the Polylib library, we will also consider how to take into account specificities of the problem. Finally we still believe that GRWs are very convenient and meaningful criteria for analysing data locality in programs, and we have designed a tool for computing GRWs dynamically. We are now planning to extend it for computing GRWs on line and improving the graphical output, to possibly make it a graphical performance debugging tool, just like CVT.

Part II
Cache Miss Equations

5 Introduction

Data caches are key to overcome the well-known memory bottleneck. To take the best advantage of caches, it is necessary to exploit as much as possible the locality that exists in the source code. Sometimes programmers do it by themselves, in other cases compilers transform the source code automatically. Either way, a locality analysis tool is required in order to identify the sections of the code that are responsible for most penalties and to estimate the benefits of code transformations. Several methods, such as simulators or compilers heuristics, can estimate this information. The problem is that simulators are very slow whereas heuristics can be very imprecise.

CME is another method that accurately describes the cache behavior. Even though the computation cost of generating CME 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.

CME allow us to study each reference in a particular iteration point independently of all other memory references. Deciding whether a reference causes a miss or a hit for a given iteration point is equivalent to deciding whether it belongs to the polyhedra defined by the CME.

We present efficient techniques to count the number of integer points inside the polyhedra defined by the CME. By exploiting some intrinsic properties of the particular types of polyhedra generated by CME, we reduce the complexity of the algorithm, which results in very high speed-ups.

Our proposal is based on estimating the result of the CME by means of sampling techniques. This technique is very fast and accurate, and the confidence of the error can be choosen. For example, for a loop nest with one million of iteration points, studying only 1000 iteration points allows us to know the number of misses with an error less than 5% with a 95% confidence.

We show that the proposed method can compute the miss ratio of most SpecFP95 benchmarks just in a few seconds on a typical workstation. This opens the possibility to include this analysis framework in production compilers in order to support many optimizations.

The rest of the document is organized as follows. In section 2 can be found a brief description of Cache Miss Equations. Section 3 describes some criteria for detecting empty CME polyhedra, as well as it presents a technique to compute whether a reference is a miss for a particular iteration point. Section 4 describes the algorithm used to compute the CME using samples of the iteration space. Section 5 shows the results obtained for the SPECfp benchmarks and demonstrates the accuracy of the proposed approach. Finally, section 6 draws the main conclusions of this work.

6 Cache Miss Equations Overview

This section describes the algorithm for generating miss equations. For our algorithm we need the reuse vectors obtained from relatively standard reuse analysis [17]. We will explain the information needed from reuse analysis as we describe our algorithm.

We consider only perfectly nested loops such that all array references are contained within the innermost loops. The algorithm can be extended to handle even some imperfectly nested loops if they have only a single basic block in between the loops in a nest. The loop nest is assumed to consist entirely of FOR loops or DO loops. We assume that subscript expressions of array references and the bounds of a loop index are affine functions of the enclosing loop indices. These constraints are not too restrictive in practice as most array references and loop bounds satisfy this. We have also assumed that the loops contain no conditional expressions. We consider loops with only constant step values as is true for virtually all loops found in numerical codes. Finally, we have considered each loop nest separately ignoring any inter-nest effects. We plan to do inter-nest analysis in the future. Currently we can generate number of misses for both direct-mapped caches and arbitrary set-associative organizations [11].

In order to describe our analysis steps in a concise mathematical form we represent a loop nest of depth n as a finite convex polyhedron of the n-dimensional iteration space $\mathbb{Z}^n$, bounded by the loop bounds [4, 17]. Each iteration in the loop corresponds to a node in the polyhedron and is called an iteration point. Every iteration point is identified by its index vector $\vec{\imath} = (i_1, i_2, \ldots, i_n)$, where i_l is the loop index of the lth loop in the nest with the outermost loop represented by the first dimension.

In order to find out whether a reference misses in the cache in a particular loop iteration we need to know whether the memory line is being accessed for the first time or whether it is reusing a previously accessed memory line [1]. If it is reusing a previously accessed line we need to know when it was last accessed and the reference that accessed it. Once we have the information about the reuse we can check if any intervening memory access evicts the memory line from the cache before it can be reused; this would result in a cache miss. If a reuse results in a cache hit we say that the reuse is realized. The central idea behind the CM equations is to find the loop instances at which reuse does not result in cache hits.

If a reference accesses the same memory line in iterations $\vec{\imath}_1$ and $\vec{\imath}_2$, where $\vec{\imath}_1 \prec \vec{\imath}_2$, we say that there is reuse in direction $\vec{r} = \vec{\imath}_2 - \vec{\imath}_1$ and $\vec{r}$ is called a reuse vector [17]. For the moment, let us assume that each reference has only a reuse vector. We denote all the misses represented by the equations for a reuse vector as misses along that reuse vector[2]. In Section 6.3 we show how all the reuse vectors interact to decide the cache misses for a reference.

The CM equations[3] generated here are of two types, namely, the cold miss equations and the replacement miss equations. The cold miss equations represent the cold or compulsory misses–misses that occur on the first reference to a memory line. The replacement miss equations represent all other misses including both capacity and conflict misses [13].

[1] A memory line refers to a cache line sized block in the memory, while a cache line refers to the actual cache block to which a memory line is mapped.

[2] A miss is said to occur along a reuse vector at those iteration points where, ignoring other reuse vectors, the data reuse is not realized.

[3] The term equation has been used loosely to represent a set of simultaneous equalities or inequalities.

For each reference in the loop nest
  For each reuse vector of this reference
    Generate a cold miss equation
    For each reference in the loop nest
      Generate a replacement miss equation

Figure 9: Algorithm to generate all CMEs in a loop nest

Figure 9 summarizes the process to generate the CM equations.

We assume that $f_1, \ldots, f_m, g_1, \ldots, g_m$ are integer values. For each induction variable i_k $(k = 1, \ldots, m)$, ub_k and lb_k stand for the upper and lower bounds of this variable in the iteration space.

6.1 Compulsory Equations

Cold miss equations are formed by investigating the situations when a memory line is brought into the cache for the first time.

As each loop nest is treated in isolation, we assume none of the data accessed in a loop nest is already present in the cache before it starts execution. For each reference, we form a cold miss equation which captures all the cold misses along each reuse vector.

6.1.1 Cold Miss Equations

These equations describe the iteration points where a reuse does not hold because the reference reuses(for a spatial or temporal reuse vector) from an iteration point outside the iteration space. These polyhedra are defined over the iteration space. This means the only variables that appear in their definition (in the linear inequalities that characterize the set), are the induction variables. The Cold Miss equations put a restriction on the possible values of one of the variables inside the iteration space.

(CM)

$$\begin{array}{ll} i_l \leq d, & \text{for a fixed l} \in [1, \ldots, m] \\ lb_k \leq i_k \leq ub_k, & k = 1 \ldots m \end{array}$$

where i_l corresponds to the l-th variable of the iteration space, $d \in \mathbb{Z}$. The first equation represents an additional restriction on one of the variables. Note that this equation could introduce a lower bound of the variable i_k, instead of an upper bound. The other $2m$ constraints determine the iteration space.

6.1.2 Cold Miss Bounds

These equations describe the iteration points where a reuse is not realized because the reference reuses data that is mapped in a different cache line (spatial reuse). These polyhedra are defined over $\mathbb{R}^{m+1}$, where m is the dimension of the iteration space. A new variable z is introduced for linearity reasons [5]. In fact, there is a version of the cache miss equations

that ignores this variable [10], but we focus on the more precise model that includes it. The equations have the following form:

(CMB)

$$\begin{array}{l} f_1 i_1 + f_2 i_2 + \cdots + f_m i_m - Lz \geq LB_1 \\ f_1 i_1 + f_2 i_2 + \cdots + f_m i_m - Lz \geq LB_2 \\ f_1 i_1 + f_2 i_2 + \cdots + f_m i_m - Lz \leq UB \\ lb_k \leq i_k \leq ub_k, \qquad k = 1 \ldots m \end{array}$$

where $LB_1, LB_2, UB \in \mathbb{Z}$, and L stands for the cache line size. Note that the three first equations can also have the form

$$\begin{array}{l} f_1 i_1 + f_2 i_2 + \cdots + f_m i_m - Lz \geq LB \\ f_1 i_1 + f_2 i_2 + \cdots + f_m i_m - Lz \leq UB_1 \\ f_1 i_1 + f_2 i_2 + \cdots + f_m i_m - Lz \leq UB_2 \end{array}$$

where $LB, UB_1, UB_2 \in \mathbb{Z}$. One of these constraints is redundant so, the polyhedron can be expressed as follows:

(CMB)

$$\begin{array}{l} f_1 i_1 + f_2 i_2 + \cdots + f_m i_m - Lz \geq LB \\ f_1 i_1 + f_2 i_2 + \cdots + f_m i_m - Lz \leq UB \\ lb_k \leq i_k \leq ub_k, \qquad k = 1 \ldots m \end{array}$$

6.2 Replacement Miss Equations

All replacement miss equations are formed by a single method, regardless of whether it is due to self or cross interference and whether the reuse vector is temporal or spatial. Given a reference, replacement equations represent the interferences with any other reference. For each pair of references (R_A and R_B), the following expression gives the condition for a cache set contention:

$$\begin{array}{rcl} Cache_Set(\vec{\imath})_{R_A} & = & Cache_Set(\vec{\jmath})_{R_B} \\ \vec{\jmath} & \in & \mathcal{J} \end{array}$$

where $\mathcal{J}$ represents the iteration points between $\vec{\imath}$ (the current one) and the iteration point from which R_A reuses. In a *k-way* set associative cache, there are k cache lines in every cache set, so k distinct contentions are needed before a cache miss occur for reference R_A.

Replacement polyhedra have the most complicated topology among CME polyhedra. A Replacement polyhedron is contained in $\mathbb{R}^{2m+3}$. $2m$ of its variables $(i_1, \ldots, i_m, j_1, \ldots, j_m)$ refer in some way to the iteration space and the remaining variables (b, n and z) are artificial and have been introduced, as in the case of the Cold Miss Bounds, for linearity reasons. Replacement Equations have the following form:

(RCM)

$$
\begin{array}{l}
-b - Cn + f_1 i_1 + \cdots + f_m i_m + g_1 j_1 + \cdots + g_m j_m = A \\
Lz - b + g_1 j_1 + \cdots + g_m j_m \geq B_1 \\
Lz + g_1 j_1 + \cdots + g_m j_m \leq B_2 \\
Lz + g_1 j_1 + \cdots + g_m j_m \geq B_1 \\
Lz - b + g_1 j_1 + \cdots + g_m j_m \leq B_2 \\
n \neq 0 \\
p_k \leq i_k - j_k \leq q_k, \qquad k = 1 \ldots m \\
lb_k \leq i_k \leq ub_k, \qquad k = 1 \ldots m
\end{array}
$$

where L stands for the cache line size and C for the cache size $A, B_1, B_2 \in \mathbb{Z}$ and $p_k, q_k \in \mathbb{Z}, \forall k = 1, \ldots, m$.

The condition $n \neq 0$ is split into $n \leq -1$ and $n \geq 1$. Each of these inequalities combined with the remaining inequalities above define a different polyhedron.

For some value of k, the variable i_k might not appear in one or both of the inequations

$$
\begin{array}{lcl}
i_k - j_k & \leq & q_k \\
i_k - j_k & \geq & p_k
\end{array}
$$

This fact does not introduce significant changes in the techniques proposed. Thus, we will assume the general form of Replacement Equations presented previously.

As the coefficient of the variable b in the equation is -1, this variable can always be isolated in the equation and expressed as a function of the other variables:

$$b = f_1 i_1 + f_2 i_2 + \cdots + f_m i_m + g_1 j_1 + g_2 j_2 + \cdots + g_m j_m - Cn - A$$

Substituting this expression of b in the inequations, a more simple form of the Replacement Equations is obtained:

(RCM)

$$
\begin{array}{l}
Lz + Cn + f_1 i_1 + \cdots + f_m i_m \geq AL \\
Lz + Cn + f_1 i_1 + \cdots + f_m i_m \leq AU \\
Lz + g_1 j_1 + \cdots + g_m j_m \geq BL \\
Lz + g_1 j_1 + \cdots + g_m j_m \leq BU \\
n \neq 0 \\
p_k \leq i_k - j_k \leq q_k, \qquad k = 1 \ldots m \\
lb_k \leq i_k \leq ub_k, \qquad k = 1 \ldots m
\end{array}
$$

where $AU, AL, BU, BL \in \mathbb{Z}$.

6.3 Finding Cache Misses from CM Equations

CM equations are useful because they allow for a systematic way to generate all the cache misses of a loop nest from them. This section describes the algorithm to generate all the cache misses of a loop nest from its CM equations.

As described above, for every reference we generate a set of equations for each of its reuse vectors. Every reuse vector has one equation for the cold misses and an equation for

each reference in the nest that represent the conflicts with that reference along that reuse vector.

Each equation represents a convex polyhedron in $\mathbb{R}^n$, where n depends on the type of equation. The integer points inside each convex polyhedron represent the potential cache misses (the number of points is the number of potential cache misses). This leads us to consider several ways for computing them:

- **Solver** Given a reference R with m reuse vectors and n_k equations for the kth reuse vector, the polyhedron that contains all the iteration points that result in a miss is [9]:

  $$Set_Misses = \cap_{k=1}^{m} \cup_{j=1}^{n_k} Solution_Set_Equation_j$$

  For this approach we need to count the number of points inside the polyhedra, which is a NP-hard problem. Writing the set of misses as a function of the complementary sets, we have that

  $$Set_Misses = \cup_{j=1}^{n_k} \cap_{k=1}^{m} Solution_Set_Equation_j^C$$

  where $Solution_Set_Equation_j^C$ represents the set of all points in $\mathbb{Z}^n$ that are not solution to the jth equation. Thus, according to measure theory, the union of s sets can be computed as follows:

  $$\begin{aligned} \mu(\cup_{i=1}^{s} A_i) \quad = \quad & \mu(A_i) + \ldots + \mu(A_s) - \\ & - \sum_{i \neq j} \mu(A_i \cap A_j) + \\ & + \sum_{i \neq j \neq k} \mu(A_i \cap A_j \cap A_k) + \\ & + \ldots + \\ & (-1)^{(i-1)} \mu(\cap_{i=1}^{s} A_i) \end{aligned}$$

  where $\mu(P)$ is the number of points inside polyhedron P. As the expression shows, the number of polyhedra that must be counted is 2^s, making this problem unfeasible due to its huge computing time.

- **Traversing the iteration space** Given a reference, all the iteration points are tested independently, studying the equations in order: from the equations generated by the shortest reuse vector to the equations generated by the longest one [12]. For this approach, we need to know whether a polyhedron is empty after substituting the iteration point in the equation. This is still a NP-Hard problem, however only $s * number_of_points$ polyhedra must be analyzed.

Our proposal builds upon the second method because it works for both direct-mapped and set-associative organizations whereas the first one can only be applied to direct-mapped caches.

```
ALGORITHM:
Ask for confidence and interval width
S= Generate_sample
Loop_misses=0
For each reference R
   Generate equations
   Ref_misses=0
   For each i⃗ ∈ S
      if R misses in i⃗ then
         Ref_misses++
   Ref_ratio=Ref_misses / #sample
   Loop_misses+=Ref_misses
Loop_Ratio=Loop_misses / (#sample*#ref)
```

Figure 10: Algorithm for obtaining cache misses

7 Optimizing CME polyhedra

In this section, we present efficient techniques to count the number of integer points inside the polyhedra defined by the CME. By exploiting some intrinsic properties of the particular types of polyhedra generated by CME, we reduce the complexity of the algorithm, which results in very high speed-ups. We show that the proposed technique can compute the miss ratio of most SpecFP95 benchmarks just in a few seconds on a typical workstation. This opens the possibility to include this analysis framework in production compilers in order to support many optimizations.

7.1 Background

This section reviews the definition of convex polyhedra, as well as a general method to compute the number of integer points inside of it and a technique to identify whether it is empty.

7.1.1 Definitions

Definition 2.1 Given the points $x_1, \ldots, x_n$ and scalars $\lambda_1, \ldots, \lambda_n$, we define a *convex combination* of $x_1, \ldots, x_n$ as $\sum_{i=1}^{n} \lambda_i x_i$ where $\sum_{i=1}^{n} \lambda_i = 1$ and all $\lambda_i \geq 0$.

Definition 2.2 A *vertex* of a set K is any point in K which cannot be expressed as a convex combination of any other distinct points in K.

Definition 2.3 A set K is *convex iff* every convex combination of any two points in K is also a point in K.

Definition 2.4 A *convex polyhedron* P is the intersection of a finite family of closed linear half-spaces of the form $\{\vec{x} | \vec{a}\vec{x} \geq c\}$ where a is a non-zero row vector and c is a scalar constant.

We only consider bounded convex polyhedra because polyhedra defined by Cache Miss Equations are convex and always bounded. Since the points of a convex bounded polyhedron can be expressed as a convex combination of its vertices, a polyhedron is fully described by its vertices. Therefore, the polyhedra can be given either by a system of linear constraints or a set of vertices [16].

Definition 2.5 We define the *real domain* of a variable x in a polyhedron P as the range of real values it takes inside of P.

Definition 2.6 We define the *integer domain* of a variable x in a polyhedron P as the range of integer values it takes inside of P.

7.1.2 Empty Polyhedra

Solving the CME requires to compute whether some polyhedra are empty. A polyhedron is considered empty when it does not contain any integer point, although it may contain real points. Counting the number of integer points in empty polyhedra takes often as much time as doing it for non empty polyhedra. Thus, a criteria for identifying empty polyhedra may be desirable.

For a given polyhedron, if there exists a variable that cannot take any integer value, there will not be any integer points inside the given polyhedron. This gives us a general criterion for detecting empty polyhedra: for each variable x_k, its definition domain $[a_k, b_k]$ in the polyhedron is calculated. Let lb_k and ub_k be the lower and upper bounds of the correspondent integer domain. If $ub_k < lb_k$ then the polyhedron is empty, since this happens when there is no integer value inside the real domain $[a_k, b_k]$ of the variable.

7.1.3 Counting integer points

If the previous criterion does not allow to determine the non-existence of integer points inside a polyhedron, the number of integer points inside it is counted until one point is found. The method for counting presented next is based on the fact that the vertices of a polyhedron are extreme points. This implies that the greatest and smallest values that any variable can take inside a polyhedron can be found in the vertices. Therefore, the computation of the domain of a variable can be done using its vertices.

Let P be a polyhedron in $\mathbb{R}^p$. We take a variable x_i and calculate its integer domain $[lb_i, ub_i]$. Then, for every integer value z from this domain, we consider the *(p-1)-dimensional* polyhedra that result from giving the variable x_i the value z. This process is repeated recursively, until we have polyhedra defined only by one variable.

Let $P_1^1, \ldots, P_M^1$ be these polyhedra. The number of integer points inside one of them is $ub - lb + 1$, where ub and lb are the upper and lower bounds of the corresponding variable. The total number of integer points in the polyhedron is obtained by adding the points of $P_1^1, \ldots, P_M^1$.

Remarks

1. The selection of the variable to be fixed is not irrelevant. Since in general the domain of a variable in a polyhedron is a function of the other variables, we take every time the variable that has the smallest definition domain in order to minimize the number

of nodes in the recurrence tree. Although we do spend some time in choosing the variable, this criterion helps us to reduce the time consumed by counting the number of integer points inside the polyhedron.

2. The domains of the variables are calculated as follows:
   Let x_k be the variable whose domain we want to determine. Let V_P be the set of vertices of P. Then the bounds of the integer domain of the variable in P are:

$$lb_k = \lceil \min_{v=(v_1,\ldots,v_n)\in V_P} v_k \rceil$$

$$ub_k = \lfloor \max_{v=(v_1,\ldots,v_n)\in V_P} v_k \rfloor$$

   Unfortunately, computing the vertices of a polyhedron is an NP-hard problem. Our approach avoids this expensive phase of the computation.

7.2 Removing Empty Polyhedra

The complexity of both methods mentioned before, is a function of the number of CME polyhedra. For this reason it is interesting to reduce the number of them. This section presents criteria for detecting empty CME polyhedra.

The general criterion that has been presented in section 7.1.2 does not detect all empty polyhedra. In order to increase the number of detected empty polyhedra, specific criteria for each type of polyhedron have been developed. These criteria rely on the structure of the equations and their interpretation in terms of the cache behavior.

7.2.1 Cold Miss Equations

Since each of these polyhedra consists of the iteration space and an additional restriction on one of the variables, it will be empty if the restriction is incompatible with the iteration space. If the additional restriction has the form $i_l \leq d$ and $d < lb_l$, there is a contradiction between the two conditions and we conclude that the polyhedron is empty. The same happens when the restriction has the form $i_l \geq d$ and $d > ub_l$.

7.2.2 Cold Miss Bounds

Recall that the Cold Miss Bounds polyhedra are of the form

(CMB)

$$\begin{array}{l} LB \leq -Lz + f_1 i_1 + f_2 i_2 + \cdots + f_m i_m \leq UB \\ \quad lb_k \leq i_k \leq ub_k, \qquad k = 1 \ldots m \end{array}$$

Since the domains of $i_1, \ldots, i_m$ are explicitly given, and they are not constrained by any other equation, the only variable that might have a domain without integer values inside is variable z. Let us observe the constraints involving variable z.

$$f_1 i_1 + \cdots + f_m i_m - UB \leq Lz \leq f_1 i_1 + \cdots + f_m i_m - LB$$

Let be

$$z_{max} = \frac{\max_{(i_1,\ldots,i_m)\in I}\{f_1 i_1 + \cdots + f_m i_m\} - LB}{L}$$
$$z_{min} = \frac{\min_{(i_1,\ldots,i_m)\in I}\{f_1 i_1 + \cdots + f_m i_m\} - UB}{L}$$

where I stands for the iteration space. Then, the integer domain of the variable z in the polyhedron (CMB) is

$$[\lceil z_{min} \rceil, \lfloor z_{max} \rfloor] \cap \mathbb{Z}$$

If there are no integer values inside this interval, the (CMB) polyhedron is empty. This condition is sufficient, but not necessary. That is, even if the domain of z contains integer values, the polyhedron might be empty.

7.2.3 Replacement Equations

Different criteria to detect empty Replacement polyhedra have been developed. In this case, not only the information given by the equations is considered, but also its interpretation in terms of the cache behavior.

- Convex Regions

  In a Replacement polyhedron, there is a subset of equations which relates the variables i_k and j_k for $k = 1, \ldots, m$.

  $$\begin{array}{l} i_k - j_k \geq p_k \\ i_k - j_k \leq q_k \\ k = 1, \ldots m \end{array}$$

  These equations appear from the division in convex regions of the domain of the variables $j_1, \ldots, j_m$ [9]. In order to detect empty Replacement polyhedra, it is checked whether these constraints are consistent with the fact that $\vec{i}$ and $\vec{j}$ must belong to the iteration space.

- $Mem_{R_A} - Mem_{R_B}$ and the variable n have different sign.

  The Replacement equations result from the following identity:

  $$Mem_{R_A}(\vec{i}) - Mem_{R_B}(\vec{j}) = Cn + b$$

  where R_A and R_B are the references whose interferences are being studied, C stands for cache size, n stands for the distance between R_A and R_B in cache size units, and b is the difference between the offset of each reference with respect to the beginning of their respective lines.

  Since the placement of the two references R_A and R_B in the memory is fixed, their relative position will not change, so that $Mem_{R_A}(\vec{i}) - Mem_{R_B}(\vec{j})$ has constant sign for all $\vec{i}$, $\vec{j}$.

Besides, this sign must be the same as the sign of the variable n, as this variable represents the distance, in terms of cache size, between the two references. A Replacement polyhedron is empty if the range of feasible values of the expression $Mem_{R_A}(\vec{i}) - Mem_{R_B}(\vec{j})$, (which depends on the variables $i_1, \ldots, i_m$ and $j_1, \ldots, j_m$), causes a contradiction with the restriction that determines the sign of the variable n.

- Incompatible range of $Mem_{R_A} - Mem_{R_B}$ with the restriction of the variable n

  The first two equations of (RCM) are:

$$\begin{array}{l} AL \leq Lz + Cn + f_1 i_1 + \cdots + f_m i_m \leq AU \\ BL \leq Lz + g_1 j_1 + \cdots + g_m j_m \leq BU \end{array}$$

  which are equivalent to

$$\begin{array}{l} AL \leq Lz + Cn + f_1 i_1 + \cdots + f_m i_m \leq AU \\ -BU \leq -Lz - g_1 j_1 - \cdots - g_m j_m \leq -BL \end{array}$$

  Then, the following expression holds:

$$\begin{aligned} AU - BL &\geq f_1 i_1 + \ldots + f_m i_m - g_1 j_1 - \ldots - g_m j_m - Cn && (1) \\ AL - BU &\leq f_1 i_1 + \ldots + f_m i_m - g_1 j_1 - \ldots - g_m j_m - Cn && (2) \end{aligned}$$

  Depending on the constraint on the variable n, one of the two inequalities is chosen and gives a criterion to detect the emptiness of the polyhedron.

  * $n \leq -1$

    As this restriction gives an upper bound of n, that is a lower bound of $-n$, we consider the second restriction

$$\begin{aligned} AU - BL &\geq f_1 i_1 + \ldots + f_m i_m - g_1 j_1 - \ldots - g_m j_m - Cn \\ &\geq f_1 i_1 + \ldots + f_m i_m - g_1 j_1 - \ldots - g_m j_m + C \\ &\geq \min_{\vec{i} \in I, \vec{j} \in J} \{f_1 i_1 + \ldots + f_m i_m - g_1 j_1 - \ldots - g_m j_m\} + C \end{aligned}$$

  * $n \geq 1$

    In this case the considered inequation is the first one.

$$\begin{aligned} AL - BU &\leq f_1 i_1 + \ldots + f_m i_m - g_1 j_1 - \ldots - g_m j_m - Cn \\ &\leq f_1 i_1 + \ldots + f_m i_m - g_1 j_1 - \ldots - g_m j_m + C \\ &\leq \max_{\vec{i} \in I, \vec{j} \in J} \{f_1 i_1 + \ldots + f_m i_m - g_1 j_1 - \ldots - g_m j_m\} + C \end{aligned}$$

  where I stands for the iteration space and J for the domain of $(j_1, \ldots, j_m)$.

  In each of these cases, if the constraint does not hold, the polyhedron is empty.

- The variable n cannot take integer values.

  The inequations (1) and (2) are used in order to compute the domain of the variable n. If it contains no integer points, the polyhedron is empty.

7.3 Analyzing Iteration Points

This section shows some methods for knowing whether an iteration point $\vec{i_0}$ fulfills a CME. This problem is equivalent to finding out whether the resulting polyhedron after substituting the variables $i_1, \ldots, i_m$ with the values given by the iteration point is empty.

7.3.1 Cold Miss Equations

Let $\vec{i_0} = (i_{01}, i_{02}, \ldots, i_{0m})$, be the iteration point we want to study. The only inequality it might not verify is

$$i_l \leq d \tag{3}$$

as the others are just a characterization of the iteration space. So, $\vec{i_0}$ is a point from the given Cold Miss polyhedron iff its l-th component verifies inequality (3).

7.3.2 Cold Miss Bounds

In section 6.1.2 we showed that these equations have the following form:

(CMB)

$$\begin{array}{l} LB \leq -Lz + f_1 i_1 + f_2 i_2 + \cdots + f_m i_m \leq UB \\ lb_k \leq i_k \leq ub_k, \qquad k = 1 \ldots m \end{array}$$

When an iteration point $\vec{i_0}$ is substituted, a *1-dimensional* polyhedron is obtained. Deciding whether $\vec{i_0}$ verifies the equations is equivalent to deciding whether the *1-dimensional* polyhedron

(CMB')

$$LB' \leq -Lz \leq UB'$$

is empty, where $LB' = LB - f_1 i_{01} - \cdots - f_m i_{0m}$ and $UB' = UB - f_1 i_{01} - \cdots - f_m i_{0m}$.

The real domain of the variable z, $\left[\frac{UB'}{L}, \frac{LB'}{L}\right] \subset \mathbb{R}$, is first computed, and then the integer domain of z is obtained from its real domain. By comparing its bounds, it is determined whether it is empty.

7.3.3 Replacement Equations

In this section we first present a criterion for determining whether a Replacement polyhedra is empty. This criterion does serve for all empty Replacement polyhedra. Therefore, for those that are not filtered out by the previous criteria, the number of points inside them will be counted until one point is found.

Detecting Empty Polyhedra

Let $\vec{i_0} = (i_{01}, i_{02}, \ldots, i_{0m})$ be an iteration point. The polyhedron (RCM') obtained after substituting $\vec{i_0}$ in the equations of a Replacement polyhedron has the following form:

(RCM')

$$\begin{array}{c} AL' \leq Lz + Cn \leq AU' \\ BL \leq Lz + g_1 j_1 + \cdots + g_m j_m \leq BU \\ n \geq 1 \\ q'_k \leq j_k \leq p'_k, \qquad k = 1 \ldots m \end{array}$$

This polyhedron is empty if there exists no integer combination of the variables $j_1, j_2, \ldots, j_m, z, n$, that verifies the equations. By construction, $BU - BL = AU - AL = L - 1$, where L is the cache line size. Then, $AU' - AL' = L - 1$.

Note that as we are only interested in integer solutions, the inequalities $AL' \leq Lz + Cn \leq AU'$ are equivalent to the following system of Diophantine equations:

$$Lz + Cn = D, \qquad D = AL', \ldots, AU', D \in \mathbb{Z}$$

According to the linear Diophantine equations theory, an equation of the form $Lz + Cn = D$ has a solution iff $Gcd(C, L)$ divides D [2]. Since C represents the cache size, $C = N * L$ for a certain $N \in \mathbb{N}$. It results that

$$Gcd(C, L) = L$$

Therefore, the previous equation $Lz + Cn = D$ has a solution iff L divides D and thus, the system of equations 7.3.3 has a solution iff there exists any value of D in $[AL', AU']$ that is multiple of L.

Since $AU' - AL' = L - 1$, the interval $[AL', AU'] \quad mod \quad L$ contains all the values of $\mathbb{Z}_L$[4]. As the previous interval can be written as $AL' + [0, L - 1]$ we have that

$$\forall x \in \mathbb{Z}_L \quad \exists D \in [AL', AU'] \mid x \equiv D \quad mod \quad L$$

In fact, as the number of integer points in $[AL', AU']$ is L, the relation defined before is bijective. For this reason, since $0 \in \mathbb{Z}_L$, there will always be only one value of D for which the equation $Lz + Cn = D$ has a solution. Let $D0$ be that value. Computing it is very simple: consider $D1 = \frac{AL'}{L}$ as a first approximation of $D0$. If $D1$ is an integer value, we take $D0 = AL'$. Otherwise,

$$D0 = \lceil \frac{AL'}{L} \rceil * L$$

Then, (RCM') can be written as follows:

(RCM')

$$\begin{array}{c} Lz = D0 - Cn \\ BL - D0 \leq -Cn + g_1 j_1 + \cdots + g_m j_m \leq BU - D0 \\ n \geq 1 \\ q'_k \leq j_k \leq p'_k, \qquad k = 1 \ldots m \end{array}$$

[4] $\mathbb{Z}_L = \mathbb{Z} \quad mod \quad L$

Let be $BL' = BL - D0$ and $BU' = BU - D0$. Then,

(RCM')

$$\begin{aligned} BL' \leq -Cn + g_1 j_1 + \cdots + g_m j_m \leq BU' \\ n \geq 1 \\ q'_k \leq j_k \leq p'_k, \qquad k = 1 \ldots m \end{aligned}$$

We assume that $q'_k \neq p'_k, \quad \forall k$. Otherwise, we substitute the value of the corresponding j's in the inequalities and we obtain a polyhedron with the same structure, but lower dimension.

Let us observe the system of linear Diophantine equations corresponding to the first two inequations above:

$$-Cn + g_1 j_1 + \cdots + g_m j_m = E, \qquad E = BL', \ldots, BU' \quad E \in \mathbb{Z}$$

Each of these equations has solution $iff\ G = Gcd(g_1, \ldots, g_m, C)$ divides E. However, neither the existence nor the uniqueness of a value $E0$ for which the corresponding equation has a solution is ensured. The computation of $E0$ is done in an similar way to the computation of $D0$. Let be $E1 = \frac{BL'}{G}$ a first approximation of $E0$. If $E1 \in \mathbb{Z}$, then $E0 = BL'$. Otherwise,

$$E0 = \lceil \frac{BL'}{G} \rceil * G$$

$E0$ is the smallest integer value of E, greater than BL', for which an equation of the form $-Cn + g_1 j_1 + \cdots + g_m j_m = E$ has a solution. If $E0$ does not belong to the domain of E, that is if $E0 > BU'$, then (RCM') is empty.

On the other hand, a feasible value of E does not ensure that the polyhedron is not empty and in this case, the number of integer points inside the polyhedron must be counted. Due to the particular form of this polyhedron, the number of integer solutions can be computed in a more efficient way than for general polyhedra. The proposed approach is described below.

Counting Integer Points

In this section, two different methods for counting the Replacement polyhedra that have not been detected as empty by the previous criterion will be described. They are based on the general method presented in section 7.1.3, extented with a new technique to compute the domains of the variables.

From the definition of (RCM') we can derive several conclusions that are interesting for both methods:

- For every feasible combination of $(j_1, \ldots, j_m)$ the domain of the variable n is:

$$\left[\frac{BU' - g_1 j_1 - \cdots - g_m j_m}{C}, \frac{BL' - g_1 j_1 - \cdots - g_m j_m}{C} \right]$$

  Since C is a multiple of L and $BU' - BL' = L - 1$, the length of this interval is:

$$\frac{BU' - BL'}{C} = \frac{L-1}{NL} < 1$$

Thus, given a $\vec{j_0}$ belonging to the domain of the variables $j_1, \ldots, j_m$, if there exists any feasible integer value of n, it is unique.

- The domains of the variables $j_1, \ldots, j_m$ are explicitly given in the expression of the polyhedron, so they do not need to be calculated. The domain of the variable n can be calculated by means of the two next inequations:

$$g_1 j_1 + \cdots + g_m j_m - BU' \leq Cn \leq g_1 g_1 + \cdots + g_m g_m - BL' \tag{4}$$

Let us define

$$n_{max} = \frac{\max_{(j_1,\ldots,j_m)\in J}\{g_1 j_1 + \cdots + g_m j_m\} - BL'}{C}$$
$$n_{min} = \frac{\min_{(j_1,\ldots,j_m)\in J}\{g_1 j_1 + \cdots + g_m j_m\} - BU'}{C}$$

where J stands for the domain of $\vec{j} = (j_1, \ldots, j_m)$. Then, the integer domain of the variable n in the polyhedron (RCM') is

$$[\lceil n_{min} \rceil, \lfloor n_{max} \rfloor] \cap \mathbb{Z}$$

We can thus conclude that the domains of all variables are easily computed and the explicit computation of the vertices is not needed.

The main difference between the two methods presented next for counting the points of Replacement polyhedra is that the first method can only be used when the variable to be fixed is $j_1, \ldots, j_m$, while the second one can always be applied. However, the second method has a lower computation cost.

1. **Method 1**

   In this method only the variables $j_1, \ldots, j_m$ are considered for counting. Since the domains of these variables are independent, they only need to be calculated once and the order for them can be determined at the beginning. Besides, the upper and lower bounds of these variables are given explicitly in the equations, which results in a linear cost for computing their domains.

   In order to stop counting when a certain combination of some of the variables j_k already gives an empty polyhedron, we use a version of the criterion given in section 7.3.3.

   Thus, before counting we need to:

   (a) Order the variables by the length of their domains.

   (b) Compute the Gcd of the coefficient of a variable j_k with the Gcd of the coefficients of the previous variables in the given ordering.

As the only variable whose domain will change during the process is n, it is only necessary to study the existence of integer points in its domain to detect empty polyhedra while counting.

2. **Method 2**
   Since the domains of the variables $j_1, \ldots, j_m$ may change when the variable n is fixed, the order in which the variables will be fixed cannot be determined at the beginning. Thus, the real domain of these variables must be recalculated every time. This is done in a similar way to the computation of the domain of n in the initial polyhedron (see 7.3.3): For every variable j_k, its greatest and lowest values given by the two inequations (eq. 4) are calculated. The actual domain of this variable is the intersection between this interval and the explicit domain given by the equations of the polyhedron.

   In order to detect empty polyhedra for this method, the search of empty integer domains must be done for all the variables.

8 Solving CME using sampling

Our proposal is based on estimating the result of the CME by means of sampling techniques. This technique is very fast and accurate, and the confidence of the error can be choosen. For example, for a loop nest with one million of iteration points, studying only 1000 iteration points allows us to know the number of misses with an error less than 5% with a 95% confidence. This section describes the algorithm used to compute the CME using samples of the iteration space.

8.1 Sampling

Our proposal builds upon the second method to solve the CME (traversing the iteration space: section 6.3). This approach to solve the CME allows us to study each reference in a particular iteration point independently of all other memory references. Based on this property, a small subset of the iteration space can be analyzed, reducing heavily the computation cost. In particular, we use random sampling to select the iteration points to study, and we infer the global miss ratio from them. This sampling technique cannot be applied to a cache simulator. A simulator cannot analyze an isolated reference, since it requires information of all previous references.

8.2 Statistical overview

In this section, the statistical techniques used will be presented, as well as their application to the CME equations.

8.2.1 Discrete random variables

Random variables are functions defined over the probability space [6]. These functions are often used to describe some phenomena.

Let $\mathcal{S} = (\Omega, \mathcal{A}, P)$ be a probability space (where Ω is the sample space, $\mathcal{A} \subset \wp(\Omega)$[5] , and P is the probability function). Let X be a random variable (RV) defined over $\mathcal{S}$. X is a discrete random variable when the image set is finite or numerable. There are several RV that have been deeply studied because of their importance and the number of usual phenomena that they describe. Now we review two of them that will be used by our model.

Let X be a real discrete random variable:

- It follows a Bernoulli distribution ($X \sim B(p)$) when the image set has only two elements. Bernoulli describes the aleatory experience in which only two things can happen: success or miss. We define $\mathcal{T} \subset \Omega$ as the set of results obtained that we consider as 'success'. Thus:

$$\begin{array}{rcl} X : \Omega & \longrightarrow & \mathbb{R} \\ \omega & \longmapsto & \left\{ \begin{array}{l} 0 \iff w \notin \mathcal{T} \\ 1 \iff w \in \mathcal{T} \end{array} \right. \end{array}$$

  The probability $P[X = 0]$ is p. Therefore, the probability $P[X = 1]$ is $q = 1 - p$, since $p + q$ must be 1.

- Binomial distribution (represented by $X \sim Bin(n, p)$) models phenomena where n different and independent experiments modeled by Bernoulli take place. This RV represents the number of successes.

  Once $\mathcal{T} \subset \Omega$ is defined, we obtain:

$$\begin{array}{rcl} X : \Omega^n & \longrightarrow & \mathbb{R} \\ (\omega_1, \ldots, \omega_n) & \longmapsto & card\{i | \omega_i \in \mathcal{T}\} \end{array}$$

  The probability $P[X = k]$, $k = 0 \ldots n$ represents the probability that k experiments over the n succed. Thus,

$$P[X = k] = \left(\begin{array}{c} n \\ k \end{array} \right) p^k (1 - p)^{(n-k)}.$$

8.2.2 Parameters estimation

Sometimes it is desired to study any characteristic of a large set of elements (also called population), but it is impossible due to its size. In these cases it is interesting to reduce the problem size. A subset can be analyzed and the values obtained inferred to the population. In our case, we model the number of misses with a Binomial-RV.

Let $X \sim Bin(n, p)$, and assume that p (the probability of success) is unknown. The way to obtain an aproximation of p is to evaluate the behavior of a subset of the population (called sample), obtaining the empiric value of the parameters that describe the sample-RV and to infer these values to the population-RV.

Let $\mathcal{Q} \subset \Omega^n$ be the sample, $N = card(\Omega^n)$ and $k = card(\mathcal{Q})$. The value $\hat{p}$ is defined as

$$\hat{p} = \frac{successes \in \mathcal{Q}}{k}$$

[5] $\wp(X)$ is the set of all the possible subsets of X

If the sample is randomly chosen among the population, the RV that describes the behavior of the sample is $Y \sim Bin(k, \hat{p})$, and we have that [6].

$$\frac{(\hat{p} - p)}{\sqrt{\frac{pq}{k}}} \sim N(0,1)$$

provided that the sample does not contain repeated elements and the following conditions hold:

- $\frac{k}{N} \leq 0.05$
- $\hat{p}k \geq 5$ and $\hat{q}k \geq 5$
- $k \geq 30$

Once a confidence percentage [7] is chosen, a confidence interval for the value of p is given by the following expression[8]:

$$p \in \hat{p} \pm z_{\frac{\alpha}{2}} \sqrt{\frac{\hat{p}\hat{q}}{k}}$$

8.3 CME particularization

This subsection describes how the above statistical techniques are used to analyze the CME.

We represent a perfectly nested loop of depth n with known bounds as a finite convex polyhedron of the n-dimensional iteration space $\mathbb{Z}^n$. We are interested in finding the number of misses this loop nest produces (said m). In order to obtain it, for each reference belonging to the loop nest we defined RV that returns the number of misses. Below, we show that this RV follows a Binomial distribution. Thus, we can use the statistical techniques in the previous section to compute the parameters that describe it.

For each memory instruction, we can define a Bernoulli-RV $X \sim B(p)$ as follows:

$$\begin{array}{lcl} X : Iteration \quad Space & \longrightarrow & \mathbb{R} \\ \vec{\imath} & \longmapsto & \{0,1\} \end{array}$$

such that $X \equiv 1$ if the memory instruction results in a miss for iteration $\vec{\imath}$, $X \equiv 0$ otherwise. Note that X describes the experiment of chosing an iteration point and checking whether the memory instruction produces a miss for it, and p is the probability of success. The value of p is $p = \frac{m}{N}$, where N is the number of iteration points.

Then, we repeat N times the experiment, using a different iteration point in each experiment, obtaining $X_1, \ldots, X_N$ different RV-variables. We note that:

- All the $X_i, i = 1 \ldots N$ have the same value of p.
- All the $X_i, i = 1 \ldots N$ are independent.

[6] Z∼N(0,1) is the Normal or Gauss distribution

[7] e.g: if the percentage is 95%, it represents that for 95 out of every 100 different samples, $\hat{p}$ will belong to the confidence interval

[8] $\alpha = 1 - confidence$

I.Width	Confidence	#Points	Min. Population
0.05	0.90	656	13120
	0.95	1082	21640
	0.99	2164	43280
0.10	0.90	164	3280
	0.95	270	5400
	0.99	541	10820
0.15	0.90	72	1440
	0.95	120	2400
	0.99	240	4800

Table 1: Sample statistics

The variable $Y = \sum X_i$ represents the total number of misses in all N experiments. This new variable follows a binomial distribution with parameters Bin(N,p) [6] and it is defined over all the iteration space. By generating random samples over the iteration space, we can infer the total number of misses as shown in section 8.2.2.

8.4 Generating samples

In this subsection, we discuss the methodology used to obtain samples. The key issues to obtain a good sample are:

- It is important that all the population is represented in the sample.
- The size of the sample.

In our case, we have to keep in mind another constraint: the sample cannot have repeated iteration points (one iteration point cannot result in a miss twice).

To fulfil these requirements, we use *Simple Random Sampling* [14]. The size of the sample is set according to the required width of the confidence interval and the desired confidence (see section 8.2.2). Table 1 shows some possible values for those parameters. For each of them, it is shown the required size of the sample and the minimum size of the population according to the statistical requirements. If we ask for the best accurary (i.e the narrowest interval and the highest confidence), the size of the sample is about 2000 points, and the iteration space must exceed 43000 points. In most numeric programs loop nests have iteration spaces bigger than the values shown in the table. Otherwise, the iteration space is small enough to test all the iteration points in it.

9 Evaluation

This section describes the implementation of the CME and the approach to estimate their solutions. Examples of their use will be given, as well as the relationship between the accuracy and the computing time required by sampling-based methodology.

SPEC	Proposed	Polylib	speed-up
applu	36.70	1933.87	16.24
apsi	1.30	17.31	29.24
hydro2d	3.16	47.51	23.31
mgrid	235.75	5495.10	11.13
su2cor	3.08	34.27	15.03
swim	3.00	48.71	13.32
tomcatv	9.60	280.75	52.69

Table 2: Execution time (in seconds) for detecting empty polyhedra.

9.1 CME polyhedra

In this section we evaluate the performance of the proposed method to deal with CME polyhedra.

9.1.1 Empty Polyhedra

First we evaluate the effectiveness of our proposal for detecting empty polyhedra. Figure 12 compares our method with the technique of the Polylib [16] for detecting empty polyhedra. Only Replacement polyhedra have been considered, as their evaluation is the most time consuming among all CME polyhedra. The first column shows the number of Replacement polyhedra obtained for each SpecFP95 program analyzed. The second column depicts the number of empty polyhedra detected by our approach, whereas the third column shows the number of empty Replacement detected through the Polylib. We can see that our approach detects a significatly higher number of empty polyhedra. This is due to the fact that Polylib only detects real empty polyhedra.

Table 2 shows the execution time required by both methods. Due to the complexity of the computation of the vertices of a polyhedron, our proposal is much faster than Polylib's technique. The complexity of the proposed method is $O(m)$, whereas the complexity of the Polylib's techniques is $O(m^{\lfloor \frac{m}{2} \rfloor})$, where m is the nesting level of the loop nest.

```
parameter (N)
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
```

Figure 11: Matrix multiply algorithm

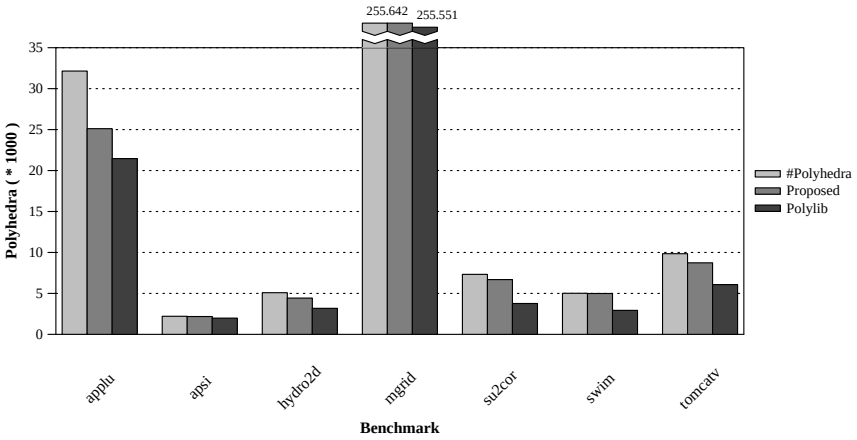


Figure 12: Empty Polyhedra

N	Miss %	Sim. Time	CME. Time	Samp. Time
1000	6.31	2h37m	211h	6s
100	2.19	9s	10m	5s
10	0.9	0.09s	0.4s	N/A

Table 3: Matrix multiply results

9.1.2 Analyzing Iteration Points

In order to evaluate the techniques proposed for analyzing iteration points, we implemented two different solvers of the Cache Miss Equations based on traversing the iteration space as described in 8.1. We only consider the remaining polyhedra after using our method for detecting empty polyehdra.

Figure 13 shows the execution time for both implementations. The first column shows the execution time for the proposed method, whereas the second column corresponds to an algorithm that uses the Polylib functions. This second algorithm counts the number of integer points inside the polyhedra by means of the general method for counting presented in section 7.1.3. It uses Polylib functions for computing the vertices of the polyhedra.

The speed-up of our approach is very important, due to the different complexities of both algorithms. It is between 8 and 12.5 times faster than the Polylib method. The difference between these two algorithms relies on the approach to compute the domains of all variables. The proposed method does it with a complexity of $O(m^2)$. The algorithm based on the Polylib is split into two steps: first the vertices of the polyhedron are computed with a complexity of $O(m^{\lfloor \frac{m}{2} \rfloor})$. Then, by means of the vertices, the domains of the variables are computed with a complexity of $O(m * \#vertices)$.

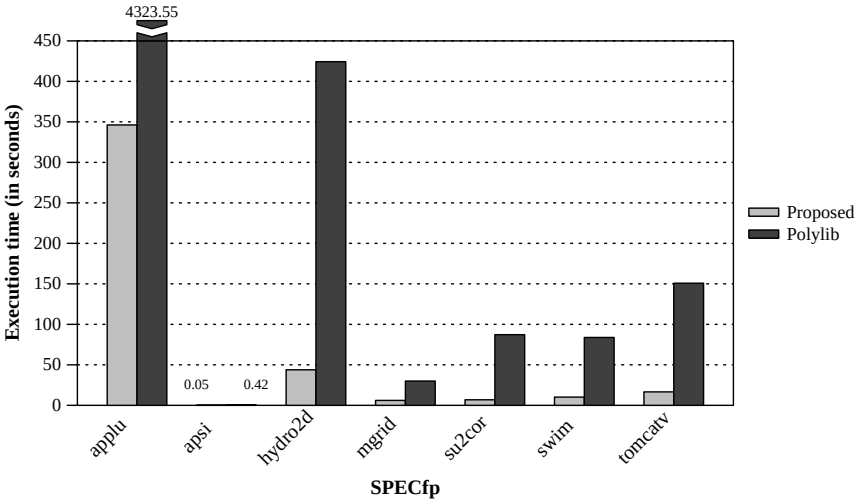


Figure 13: Execution Time

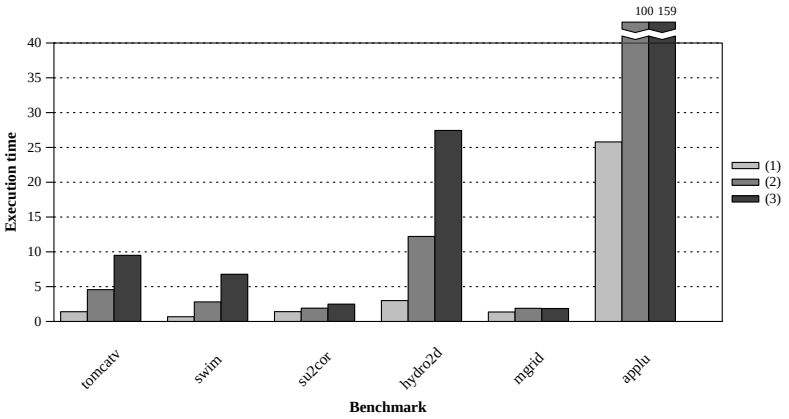


Figure 14: Execution time for direct mapped caches.

9.2 Solving CME

9.2.1 Implementation

CME have been implemented for fortran codes through the Polaris Compiler [15] and the Ictineo library [1]. They allow us to obtain all the compile-time information needed to generate the equations.

We have implemented the algorithm described in [12] to obtain the number of misses of each reference. The resulting algorithm can be seen in fig. 10.

Due to CME restrictions, only perfectly nested loops in which the array subscript expressions are affine functions are analyzed [9].

Both direct-mapped and set-associativity caches with LRU replacement policy are supported.

SPEC	#LN	%VLN	#Ref	#R.V	#Eq.
tomcatv	9	100%	10.6	30.8	261
swim	22	59%	10.5	18.5	33
su2cor	13	84%	14.0	25.8	296.4
hydro2d	42	76.1%	7.1	11	196
mgrid	8	100%	29	627.5	70229
applu	55	100%	10.2	24.3	286.7
Average	18.6	77.25	13.6	123	11883.7

LN: loop nests. *VLN*: analyzed loop nests. *Ref.*: references. *R.V*: reuse vectors. *Eq.*: equations.

Table 4: Sampling over SPECfp

SPEC	(1)	(2)	(3)
tomcatv	9	9	9
swim	13	13	13
su2cor	11	11	10
hydro2d	32	32	32
mgrid	8	8	6
applu	55	51	51

(1) stands for $\alpha = 0.05$ and interval width=0.10
(2) stands for $\alpha = 0.05$ and interval width=0.05
(3) stands for $\alpha = 0.01$ and interval width=0.05

Table 5: Loops analyzed

9.2.2 Example

We will first illustrate the effectiveness of the sampling approach by means of the matrix multiply program (fig. 11) for a 32K direct mapped cache. Table 3 shows the results obtained for an interval width of 0.05 and a 99% confidence. For large problem sizes (N=1000), simulators are very slow and solving CME through the classic analysis (section 6.3) is not feasible, whereas we obtain the same results as the simulator[9] in a few seconds. For small problems (N=100), sampling is faster than simulators. In the smallest one (N=10), the size of the sample must be all the iteration space, obtaining the same computing cost as the classic analysis, which is quite low due to the small number of points.

9.2.3 Performance evaluation

Next, we evaluate the accuracy of the proposed method and the speed/accuracy trade-offs for direct mapped caches. Finally, we also show some results demonstrating the effectiveness of the proposal for set-associative caches.

The loop nests considered are obtained from the SPECfp95, choosing for each program the most time consuming loop nests that in total represent between the 60-70% of the

[9]In this particular case, the central point of the confidence interval practically coincides with the simulation result

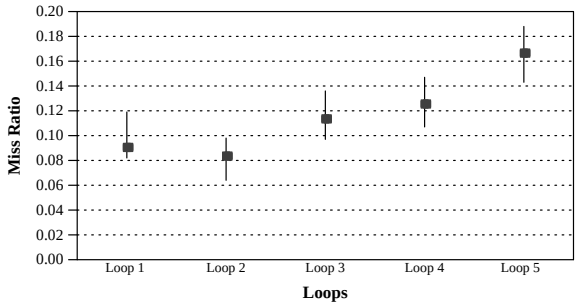


Figure 15: Tomcatv miss ratio.

execution time using the reference input data. The simulation values are obtained using a trace driven cache simulator. The traces have been obtained by instrumenting the program with Ictineo. For the evaluation of the execution time, an Origin2000 has been used.

Four SPEC programs have not been evaluated for the following reasons:

- **125.turb3d and 141.apsi** The loop nests that represent the 65% of the execution have not enough iteration points to use sampling.
- **145.fpppp** The section of the code that represents the main part of the execution time does not contain perfectly nested loops.
- **146.wave5** The array subscripts are function of other arrays, and thus the CME cannot be obtained.

The CME have been generated for all the other SPECfp programs assuming a 32K cache of arbitrary associativity. Table 4 shows the number of loop nests and the percentatge of them in which we can apply sampling [10]. The next three columns illustrate the size of the problem. The table shows that on average, each loop nest contains 13.5 references and 123 reuse vectors, and the number of equations per loop nest for a direct mapped cache is 11883.7. The number of equations depends on the organization of the cache, since the number of misses is not the same.

We have experimented with different accuracy configurations. Table 5 shows the different configurations (from less accurate to more accurate) and the number of loops for which sampling can be used. For more accurate configurations less loops can be analyzed using sampling because there are not enough points in the iteration space. In these cases the whole iteration space must be traversed. However this situation arises only for small iteration spaces.

Figure 14 shows the time in seconds required to analize the SPECfp programs for the 3 configurations for a direct mapped cache. Note that the analysis time is reasonable in all cases and that the more accurate configurations require more time since more points are considered. From our experiments we consider that a confidence of 95% and an interval of 0.05 is a good trade-off between analysis time and accuracy.

Figure 15 shows the results obtained for the tomcatv SPECfp program (for a 95% confidence and an interval width of 0,05) for a direct mapped cache. For each loop nest,

[10] At least, 200 points must be tested

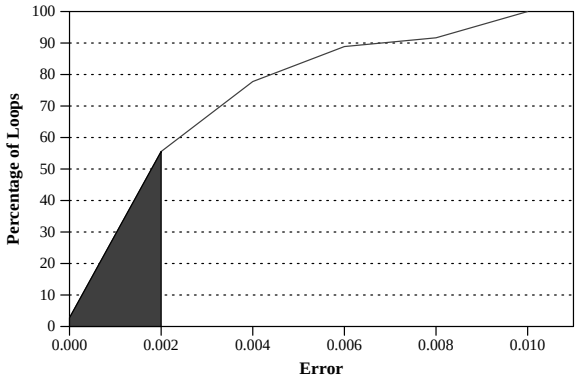


Figure 16: Sampling error for direct caches.

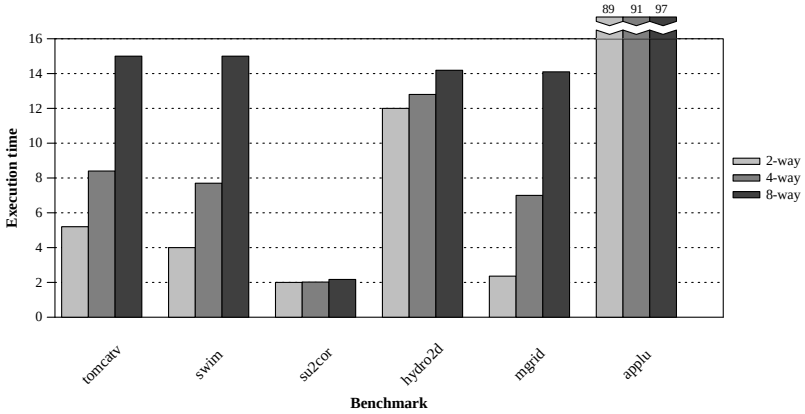


Figure 17: Execution time for set-associativity cache.

the vertical segment represents the obtained confidence interval, and the square point the miss ratio obtained using a simulator. In all cases, as well as for the other SPECfp programs, the actual miss ratio belongs to the interval obtained from our analysis.

Finally, figure 16 depicts the cumulative distribution of the difference between the miss ratio and the central point of the confidence interval (also known as empirical estimation) for all the programs, with a 95% confidence and an interval width of 0,05. Y axis represents the percentage of loop nests that have an error less or equal than the corresponding value in the X axis. This graph shows that the absolute differences between the actual miss ratio and the miss ratio obtained from our analyzer is usually lower than 0.002 and never higher than 0.01.

Set-associative caches

Figure 17 shows the time in seconds required to analize the SPECfp programs for three different organizations of set-associative caches (95% confidence and an interval width of 0,05). Although in general analysis time is higher than for direct mapped caches, the analysis time is reasonable.

Figure 18 depicts the cumulative distribution of the difference between the miss ratio and the empirical estimation for all the programs, with a 95% confidence and an interval width of 0,05. As in the case of direct mapped caches, the absolute differences between the actual miss ratio and the miss ratio obtained from our analyzer is usually lower than 0.002 and never higher than 0.01.

10 Conclusions

Cache Miss Equations are a static analysis approach to determine the memory behavior. They provide an analytical and precise description of the cache memory behavior. The main drawback of CME is that solving them to know the exact number of misses is an NP-hard problem that makes them unfeasible for the most of applications.

In this document we propose the use of sampling techniques to solve CME. First, we propose some techniques that exploit some intrinsic properties of the particular polyhedra generated by CME. These techniques significantly reduce the complexity of the algorithms and result in speed-ups of more than one order of magnitude for the SpecFP95 benchmarks. On the other hand, with statistic techniques we can perform memory analysis extremely fast independently of the size of the iteration space. For instance, it takes the same time (6 seconds) to analize a matrix by matrix loop nest of size 100x100 than one of size 1000x1000. On the contrary it takes 9 seconds to simulate the first case and more than 2 hours to simulate the second.

In addition, the use of sampling along with inference teory allows the user to set the desired confidence and the width of the result interval. The bigger the accuracy the bigger the set of points to analyze and therefore more time is required to perform the analysis. In this way the user can trade accuracy for speed at his will.

In our experiments we have found that, using a confidence of 95% and an interval width of 0.05, the absolute error in the miss ratio was smaller than 0.002 in 65% of the loops from the SPECfp programs and was never bigger than 0.01. Furthermore, the analysis time for each program was usually just a few second and never more than 2.3 minutes.

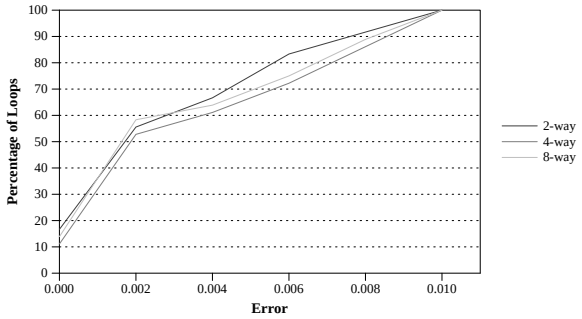


Figure 18: Sampling error for set-associativity caches.

References

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

[2] Utpal Banerjee. *Loop transformations for restructuring compilers.* Kluwer academic publishers, 1993.

[3] Cristina Barrado, Carles Ciuraneta, Christine Eisenbeis, Jordi Garcia, Antonio González, Josep Llosa, Andry Randrianatoavina, Jesus Sánchez, François Thomasset, and Xavier Vera. Basic performance analysis. Deliverable M1.D1 of the MHAOTEU ESPRIT project n0 24942, décembre 1998.

[4] P. Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs. In *Proc. Int'l Conf. on Supercomputing*, May 1996.

[5] Philippe Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: applications to analyze ans transform scientific programs. In *ICS96*, 1996.

[6] M.H. DeGroot. *Probability and statistics.* Addison-Wesley, 1998.

[7] Christine Eisenbeis, William Jalby, Daniel Windheiser, and François Bodin. A strategy for array management in local memory. *Mathematical Programming*, 63:331–370, 1994. Special Issue on Applications of Discrete Optimization in Computer Science.

[8] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, September 1988.

[9] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: an analytical representation of cache misses. In *ICS97*, 1997.

[10] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. In *ACM Transactions*, 1998.

[11] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the 8th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, California, October 4-7 1998.

[12] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *ASPLOS98*, 1998.

[13] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Transactions on Computers*, 38(12):1612–1630, Dec. 1989.

[14] Moore; McCabe. *Introduction to the Practice of Statistics.* Freeman & Co, 1989.

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

[16] D.K. Wilde. A library for doing polyhedral operations, 1993.

[17] M.E. Wolf and M.S. Lam. A data locality optimizing algorithm. In *Procs. of Conf. on Programming Language Design and Implementation (PLDI'91)*, pages 30–44, June 1991.