ICS

TECHNICAL REPORT

Line Size Adaptivity Analysis of Parameterized Loop Nests for Direct Mapped Data Cache

Paolo D'Alberto, Alexandru Nicolau,
Alexander Veidembaum and **Rajesh Gupta**

email:{*paolo,nicolau,alexv,rgupta*}*@ics.uci.edu*

UCI-ICS Technical Report #01-42

Information and Computer Science
University of California, Irvine

Line Size Adaptivity Analysis of Parameterized Loop Nests for Direct Mapped Data Cache

Paolo D'Alberto Alexandru Nicolau Alexander Veidembaum
Rajesh Gupta

Information and Computer Science
University of California at Irvine *
email:{*paolo,nicolau,alexv,rgupta*}*@ics.uci.edu*

UCI-ICS Technical Report #01-42

Abstract

Caches are an important part of architectural and compiler high performance and low-power strategies by reducing memory accesses and energy per access. In this paper, we examine efficient utilization of data caches in an adaptive memory hierarchy. We focus on the optimization of data reuse through the static analysis of line size adaptivity. We present a framework that enables the quantification of data misses with respect to cache line size at compile-time using (parametric) equations modeling interference. The framework considers both expressiveness and practicability of the analysis. Part of this analysis is implemented in a software package STAMINA. Experimental results demonstrate effectiveness and accuracy of the analytical results compared to alternative simulation based methods.

*Supported by AMRM DABT63-98-C-0045

Contents

1 Introduction

In modern uniprocessor systems the memory hierarchy is an important concern of performance, area and energy. It is also the component requiring most of the die area in systems-on-chip and it is the principal power consumer, accounting for as much as 20-50% of the total chip power [19, 16]. In recent years, there has been a great effort on the engineering of several levels of cache, to reduce the impact on performance/power of caches. The focus of our work on memory hierarchy is adaptivity in cache subsystems. We have built an architecture that enables static and dynamic adaptation of memory hierarchy: its configuration and policies [28]. In this paper, we focus on (compiler-driven) data cache line size adaptation [27, 1, 28]. In fact, the architecture is able to change dynamically the line size (by hardware monitoring or application instruction) during the execution of the application. To exploit fully the potential of this adaptation, we need a way to target it, that is, (statically) determine the application cache behavior to trace adaptation for maximum performance and/or minimum energy dissipation.

Related work on cache behavior analysis can be distinguished in *profiling-based* and *static* approaches. Profiling has been used to determine the memory behavior by direct measure. Varying some parameters of the architecture, the direct measure quantifies the variation of the memory performance [18]. The approach is flexible and it can be used for the analysis of the whole application as well as part of it. There are two limitations: the measure might be dependent from the inputs and, of course, the analysis cannot be faster than the execution of the application itself.

Static cache analyzers are independent from the *inputs* and focus on the analysis of perfect loop nests [15, 29]. In [15], the authors propose to model the cache misses of memory references by equations, *Cache Miss Equations*. Then every iteration (or a sampled version) in the loop nest is checked whether it satisfies the equations or it does not. The approaches count the solutions of the equations to achieve an estimation of the number of cache misses. There are two limitations in the current static approaches: 1) The loop nest bounds must be known at compile time. This is not realistic because they are often parameterized and it is not practical, because they can be very large. 2) The analyzable loops are *sensitive* to tiling loop transformation. For example, if tiling is performed on the three-loop-algorithm for matrix multiplication and the tile sizes do not divide evenly the loop bounds, the inner loops bounds cannot be represented by affine functions. The resulting nest is not analyzable.

To attack and overcome these limitations, in this paper we propose a static approach to investigate perfect loop nests and determine the relation between line size and number of misses on a per-nest-base. The analysis result is annotated in the code and it can be used at run time to set the line size.

The paper is organized as follows. In Section 2 we present the approaches available in

literature and also a motivating example. The example is used to point out the common pitfalls of the current approaches and an intuitive introduction to ours. In Section 3 we introduce notations about loop nests and cache equations. In Section 4 we introduce the theoretical frame work and our approach. Finally, in Section 5 we show the results of our analysis for three representative examples.

2 Related Work

In this paper we analyze the cache behavior of *perfect loop nests* in scientific applications. The basic block, the subject of our investigation, is a loop nest so that each index variable has positive values; it starts from zero; its increment step is always one and its upper bound is either constant or an affine function of the bounds and index variables of outer loops. Memory references are in the inner loop and their address computations are linear functions of the loop index variables [21] (this condition can be relaxed w.l.o.g.). Parameterized loop nests are basically perfect loop nests with parameters introduced in the affine function of the upper bounds and in the linear functions of address computations (e.g. [8]). The parameter values are constant during the execution of the loop nest, but they may vary after each execution. We use also the term of *iteration space* to identify the set of iterations of the loop nest, which can be considered a *bounded integer polyhedron.* We use the term of *iteration point*, to specify one particular iteration.

The goal of the analysis is to determine the cache behavior in terms of misses/hits of any loop nest (in an application) and drive (software/hardware) adaptation to improve performance. In this section we give a first introduction to the related work, the main ideas and implementations of static approaches, symbolic approaches and profiling-based approaches.

The Static Analysis is applied at compile time on perfect loop nests. For each memory reference is determined a set of requirements (expressed as inequalities and equations) representing the conditions for which the memory reference is cause of a miss/hit in cache. A static model of the cache behavior is based on evaluation of these conditions for each reference and the computation of the sum $\sum_{\bar{\mathbf{i}}} \delta(\bar{\mathbf{i}})$. $\delta(\bar{\mathbf{i}})$ is 1 if $\bar{\mathbf{i}}$ is a step of the computation for which all the conditions for a cache miss are satisfied. It is 0 otherwise ([15, 29, 6]). This is not a direct counting of the misses or hits. The accuracy of the model is based on the accuracy of the conditions used to determine a hit/miss.

For the analysis of parameterized loop nests at compile time, it is possible to use the static analysis, i.e. it is performed for each possible value of the parameters. The static approach is the most natural enumeration approach, it is accurate but it is also *the least interesting* for parameterized loop nests [5] (for an advanced and complete investigation refer to Stanley's book [24] and Barvinok's tutorial [5, 3]). Or for a different approach

see Pugh's symbolic approach [22] based on the Omega Test [23].

Among the *interesting ones* there is the approach by the French mathematicians Eugene Ehrhart (1906-2000). The original work proposes the exact enumeration of integer points in a polyhedron with only one *parameter* and in two dimensions. Ehrhart approach permits to have a closed formula so that the number of integer points is known for every value of the parameter. The extension to a general number of parameters and dimensions is proposed by Clauss et al [8, 10, 11, 9, 4]: Ehrhart polynomials of degree k are defined for rational polyhedra[1] in $\mathcal{Q}^k$. Using the authors' terminology, the set of inequalities describing the constraints of a polyhedron is called *the implicit form* of a polyhedron. An equivalent one is the *explicit form*, which is the collection of *vertices*, *rays* and *lines* (see [30]). The determination of the explicit format takes an exponential number of steps $O(2^k)$ (also the space complexity is affected). The characteristics of Ehrahart polynomials are easier to see from the explicit format. Indeed, Ehrhart polynomial are an extension of the multi variable polynomials where the coefficients can be scalars and arrays. The arrays represent a set of possible coefficients. The size of the arrays is function of the *vertices* coordinates. If the vertices are integral points, the coefficients of the polynomial are scalars. If the vertices are rational, the size of the arrays is the largest denominator of every vertex coordinates. The number of the iteration points is computed directly for a subset of values of the parameters and a linear system solution is sought. The number of systems and the size of the systems to solve is function of the number of coefficients to determine.

The Ehrhart approach is feasible and elegant but it is impractical for common cases in cache modeling (mostly rational vertices with large coefficients). We use *polylib* to manipulate parameterized polyhedra and Ehrhart's polynomials to estimate the number of iteration points in a parameterized loop nest (mostly integral vertices).

If the measure of cache misses is done at run time there is no hassle about parameters or cache modeling. Profiling is based on direct counting of misses/hits (most of the times by simulation tools and lately by hardware register counters) during the execution of the application. The statistics obtained are used as feedback for further optimizations in the compilation of the application. There are two basic distinctions w.r.t. static approaches. 1) The final counting is oblivious of the cause of cache misses. Even if the measure is aware of the cause, it is not carried on as final result. It is used to determine the effect of the variations of a tunable parameter (e.g. line size). 2) Profiling is based on the assumption that the future executions of the application have the same *characteristics.*

Examples of profiling-based approaches driving adaptation can be found for example in [1, 27, 28]. At run time, the hardware monitors the execution of an application and

[1] A rational polyhedron is a polyhedron so that the coordinates of every point can be represented by the ratio of two integer numbers

adapts the line size to the application's needs. The monitoring can be over the whole execution or a sampled part. The hardware is able to measure the *real* misses and, based on a heuristic, make some assumptions on the best line size. The authors in [28, 26, 25, 14] investigate two approaches: *Adaptive Fetch Line Size* [28] (AFL) and *Adaptive Line Size* [25] (ALS). ALS is the first approach introduced. The approach tailors the line size per reference: each memory reference is associated with a line size; references with spatial locality tend to have large line size; references with temporal locality tend to have short line size. The line size determination is performed on a reference miss, using information collected during the execution. AFL can be considered a simplification of ALS. At any time there is only one line size for every reference. The line size is fixed for an interval time and the optimal line size is a compromise among all the references.

In the following, we sketch an example of problem, Figure 1, to explain the challenges we tackle in a more realistic scenario. The example cannot be analyzed fully by any

```
extern double A[2000][1024],B[100][1024];

void foo(int m, int start) {
  int i,j;
  for (i=0;i<m;i++)                /* 0<=m<100 */
    for (j=0;j<m;j++)
      A[i][j+start] += B[i][j];   /* 0<= start <10024-100 */
}
void update(int start) {
  int start1=0; /* compile time */
  int start2;
  int startin;

  start2 = start+2; /* not really at compile time */
  foo(50,start1);
  foo(50,start2);

  scanf(``\%d'',\&startin); /* run time */
  foo(50,startin);
}
```

Figure 1: Motivating example: parameterized loop bounds and interference

approaches (static or profiling) and no optimal line size can be determined. We describe why the example is so problematic and how our approach does handle it. The simplicity of the example must not mislead the reader, because it is representative for more complex and meaningful cases in Section 5.

We use the C language to describe our motivating example. Matrixes A and B have same number of columns but different shape (different number of rows). They are in row major format and they are consecutively stored. The procedure `foo` has two parameters m and *start*. References $A[i][j]$ and $B[i][j]$ have spatial reuse because the matrix accesses are column wise in the inner loop. The computation tends to use

the data in a cache line fully. Very conveniently, if there is interference between A and B, it may happen between consecutive iterations in the inner loop. We consider the case when B interferes with A. The interference equation is $B_{-1} + 8B_1 i + 8j = A_{-1} + 8A_1 i + 8j + 8start + n\mathcal{C} + l$, where B_1 and A_1 are the number of column of B and A respectively (read the equation as follows "*there is interference when the address of $B[i][j]$ is the address of $A[i][j + start]$ plus a multiple of the cache and an offset of size no larger than the cache line size at iteration specified by i and j*") and substituting the numbers in we get: $2000 * 1024 * 8 = 8start + n\mathcal{C} + l$. If $8start \bmod \mathcal{C} < L$, then the equation has solution and there is always interferences between the two references. Therefore the optimal line size for this example is $L = 8start \bmod \mathcal{C}$. This result can be obtained using *polylib*: the Ehrhart polynomial exists, for $8start \bmod \mathcal{C} < L$. This is called also *definition domain.*

Static approaches are not able to analyze the example because the parameters of the procedures affect the dimension of the iteration space and also the interference equation. They are not known at compile time. Even if an range of possible values is known, the problem size makes impractical a full investigation.

Profiling is not able to find the optimal line size as well. Given a training set specifying two values for `startin`, i.e. 1 and 16, the profiling approach correctly can determine that the optimal line sizes are 8 and 128, respectively. But it has to choose one. Either choice is not optimal for any other input.

If we change the number of rows of the matrixes, we are able to see another characteristic of interference: `extern double A[2000][1024],B[100][512]`. The interference equation changes: $2000 * 1024 * 8 - 512 * 8i = 8start + n\mathcal{C} + l$. When $i = 0$ we have the previous equation. The first m iterations A and B interfere, if $8start \bmod \mathcal{C} < L$. When $i = 1$, for the same values of *start*, there is no interference. We can see there is interference every four iterations on i ($i = k\frac{C}{512*8}$ with $\mathcal{C} = 16K$). The interference is not random but is not evenly distributed either. If we determine the Ehrhart polynomial, it can have 512 coefficients (the number of coefficients can be polynomial in the dimension of the problem).

In the example proposed, the interference may happen at every iterations or rarely, it depends on the matrix sizes. Inspired by the example, we investigate a quantitative measure of interference as the *density of interference.* When matrix B has 1024 columns the interference density is 1, that is, if there is interference then it is at every iteration. When B has 512 columns the density is no more than $\frac{1}{4}$. The interference density is based only on the interference equation coefficients. We do not take in account any parameters. The parameters come to play for the existence of the interference.

3 Background

A perfect loop nest composed of k loops [21] determines a set (*iteration space*) of integral points (*iteration points*), in $\mathbb{N}^k$. The loop order specifies a strong order between any two iteration points $\bar{\mathbf{j}} = (j_0, \ldots, j_{k-1})$ and $\bar{\mathbf{i}} = (i_0, \ldots, i_{k-1})$. $\bar{\mathbf{i}}$ precedes $\bar{\mathbf{j}}$, and we indicate as $\bar{\mathbf{i}} \lhd \bar{\mathbf{j}}$, if it exists a $0 \leq l \leq k-1$ so that $i_n = j_n$ for every $n < l$ and $i_l < k_l$.

Informally, the first coordinate i_0 of an iteration point $\bar{\mathbf{i}}$ is associated with the outer loop of the nest and the last coordinate i_{k-1} is associated with the inner loop. The inequality for two points $\bar{\mathbf{i}} \trianglelefteq \bar{\mathbf{j}}$ is verified if either they coincide or $\bar{\mathbf{i}} \lhd \bar{\mathbf{j}}$. A geometrical order can be inferred too. $\bar{\mathbf{i}}$ is smaller than $\bar{\mathbf{j}}$, $\bar{\mathbf{i}} < \bar{\mathbf{j}}$, if $i_n \leq j_n$ for every n but a l so that $i_l < j_l$. Graphically, a point $\bar{\mathbf{i}}$ in the iteration space determines a unique bounded polytope ($\{\bar{\mathbf{j}} | \bar{\mathbf{j}} \leq \bar{\mathbf{i}}\}$), a point is smaller than another if its bounded polytope is contained in the other. It is easy to see that if $\bar{\mathbf{i}} < \bar{\mathbf{j}}$ then $\bar{\mathbf{i}} \lhd \bar{\mathbf{j}}$, but not vice versa. A formal definition of iteration space is the following.

Definition 1 *The iteration space is a bounded polytope (lattice). The iteration space is the polytope* $\{\bar{\mathbf{i}} | \bar{\mathbf{0}} \trianglelefteq \bar{\mathbf{i}} \trianglelefteq N(\bar{\mathbf{n}})\}$, *in short* $P_{\bar{\mathbf{i}} \leq N(\bar{\mathbf{n}})}$, *where* $N(\bar{\mathbf{n}}) = A\bar{\mathbf{n}} + B\bar{\mathbf{p}}$ *with* A *and* B *matrices of size* $k \times k$ *and* $\bar{\mathbf{p}}$ *is a vector of constant parameters.*

Given a point $\bar{\mathbf{t}} \in P_{\bar{\mathbf{i}} \leq N(\bar{\mathbf{n}})}$ and a vector $\bar{\mathbf{r}}$, it is common to investigate the bounded polytope $P^r(\bar{\mathbf{t}}) = \{\bar{\mathbf{j}} \in P_{\bar{\mathbf{i}} \leq N(\bar{\mathbf{n}})} | \bar{\mathbf{t}} - \bar{\mathbf{r}} \lhd \bar{\mathbf{j}} \trianglelefteq \bar{\mathbf{t}}\}$. It is an interval in the iteration space. Indeed, $P^r(\bar{\mathbf{t}}) = P_{\bar{\mathbf{i}} \trianglelefteq \bar{\mathbf{t}}} - P_{\bar{\mathbf{t}} - \bar{\mathbf{r}} \lhd \bar{\mathbf{t}}}$ is the difference of two parameterized polytopes containing the origin. A property which is used all along is the partition-ability of a polytope into *rectangles*. We define the polytope $P_{condition} = \{\bar{\mathbf{i}} | \bar{\mathbf{0}} \trianglelefteq \bar{\mathbf{i}}$ *and "condition" is true*$\}$.

Property 1 *The following three expressions are true.*

1. $P_{\bar{\mathbf{i}} \trianglelefteq \bar{\mathbf{t}}} = P_{(i_0 < t_0)} \cup P_{(i_0 = t_0 \cap i_1 < t_1)} \cup \cdots \cup P_{(i_0 = t_0 \cap \cdots \cap i_{k-2} = t_{k-2} \cap i_{k-1} \leq t_{k-1})} = \cup_{j=0}^{d-1} P_{(\cap_{l=0}^{j-1} i_l = t_l \cap i_j \leq t_j)}$
2. $P_{\bar{\mathbf{t}} - \bar{\mathbf{r}} \lhd \bar{\mathbf{i}}} = P_{(t_0 - r0 < i_0)} \cup P_{(t_0 = i_0 \cap t_1 - r_1 < i_1)} \cup \ldots = \cup_{j=0}^{d-1} P_{(\cap_{l=0}^{j-1} i_l = t_l - r_l \cap i_j < t_j - r_j)}$,
3. $P^r(\bar{\mathbf{t}}) = \cup_{j=0}^{d-1} (P_{(\cap_{l=0}^{j-1} i_l = t_l \cap i_j \leq t_j)} - P_{(\cap_{l=0}^{j-1} i_l = t_l - r_l \cap i_j < t_j - r_j)})$

Proof: The first two expressions follow by the definition of precedence. Since the interval $P^r(\bar{\mathbf{t}})$ defines the iterations points that belong to $P_{\bar{\mathbf{i}} \trianglelefteq \bar{\mathbf{t}}}$ and do not belong to $P_{\bar{\mathbf{t}} - \bar{\mathbf{r}} \lhd \bar{\mathbf{i}}}$, the last expression is verified. ♢

The decomposition is very useful when a property must be found in the interval $P^r(\bar{\mathbf{t}})$, because the problem can be decomposed on *simplicial rectangular bounded polytopes*. The decomposition assures that the problem is computable.

As introduced in [31], a reference R_B of a vector B has temporal reuse if in different iteration points the same memory location is accessed: $Addr(R_B(\bar{\mathbf{i}})) = Addr(R_B(\bar{\mathbf{j}}))$. The reuse is summarized by a vector $\bar{\mathbf{r}}$ so that for every iteration point $\bar{\mathbf{i}}$, $Addr(R_B(\bar{\mathbf{i}})) = Addr(R_B(\bar{\mathbf{i}}+\bar{\mathbf{r}}))$. The address of a reference is a linear function: $Addr(R_B(\bar{\mathbf{i}})) = \mathbf{B}\bar{\mathbf{i}}+\bar{\mathbf{u}}$. The reuse vector is a vector in the *kernel* of matrix $\mathbf{B}$, indeed $\mathbf{B}\bar{\mathbf{r}} = 0$. Spatial reuse is attained when elements of the same line is accessed. Temporal reuse is a particular case of spatial reuse.

Reuse vector of a memory reference is statically determined by its index computations. If during the computation the reuse is satisfied, we have a hit in cache, otherwise a miss may happen: the reuse $\bar{\mathbf{r}}$ of a reference $R_A(\bar{i})$ is *prevented* if either a reference $R_B(\bar{\mathbf{t}})$ with $\bar{\mathbf{i}}-\bar{\mathbf{r}} \trianglelefteq \bar{\mathbf{t}} \trianglelefteq \bar{\mathbf{i}}$ interferes with reference R_A, or the iteration $\bar{\mathbf{i}}-\bar{\mathbf{r}}$ is not point in the iteration space. In general, if a reuse vector is prevented does not mean that during the computation we have a miss in cache. Indeed, a reference may have multiple reuse vectors and, to have a miss in cache, all reuses must be prevented. In practice, if we analyze the cache behavior of a memory reference by a partial number of reuse vectors, the analysis is conservative: all cache misses are determined correctly, but some hits can be mistakenly determined as misses. The interference between memory references is investigated by the *Cache Miss Equation* (CME) model (see [15]).

Definition 2 *Given two array references R_A (*interferer*) and R_B (*interferee*) of the arrays A and B respectively, we define the following integer equation.*

$$E(R_A, R_B, \bar{\mathbf{t}}, \bar{\mathbf{r}}, \bar{\mathbf{p}}) \equiv \begin{cases} \mathbf{A}\bar{\mathbf{i}} = \mathbf{B}\bar{\mathbf{t}} + n\mathcal{C} + l + D\bar{\mathbf{p}} \\ \textit{with } \bar{\mathbf{i}} \in P^r(\bar{\mathbf{t}}) \textit{ and } \bar{\mathbf{t}} \in P_{\bar{\mathbf{i}} \leq N(\bar{\mathbf{n}})} \end{cases} \tag{1}$$

$E(R_A, R_B, \bar{\mathbf{t}}, \bar{\mathbf{r}}, \bar{\mathbf{p}})$ is the interference equation. *$\mathbf{A}$ and $\mathbf{B}$ are integer matrixes of size $1 \times k$. $\mathbf{A}$ is the index matrix for the interferer and $\mathbf{B}$ for the interferee. $\mathcal{C}$ is the cache size in bytes, $|l| < L - 1$ is the offset in the cache line and L is the cache line size, always in bytes. The free variable $n \neq 0$ describes the distance in number of cache size blocks between the two interfering references. $\bar{\mathbf{p}}$ is a vector of parameters. $\bar{\mathbf{r}}$ is a reuse vector for R_B.*

We model a direct mapped cache, therefore if the equation has solution, there is interference and there is a miss. If there is no solution and in case of only one reuse vector, it is a hit. A k-way cache can handle k interferences without a miss. We should count the interferer references at least L bytes apart and determine if a miss happens. In our case, the reuse vectors of inner loop references, if any, are very short [20]. Therefore the interferer has a few chances to interfere with different memory locations. It is also possible to check interference only with the leader reference of a group with spatial reuse [31, 15].

4 The Parameterized Loop Analysis

In this section we introduce a quantitative approach to determine cache interferences at compile time. Our focus is to achieve a quantitative measure for the misses that are dependent from the line size. We focus on the determination of interferences and, indirectly, on a subset of *cold misses* and *capacity misses*. Some authors suggest that the line size may remove some capacity and cold misses, because, fetching more data, it may alleviate the number of misses. We agree, but only when there is spatial reuse.

We organize the Section as follows: shortly we observe the interference equation (Equation 1) and simplify it in a more concise format. The simplified equation is subject of two types of analysis to determine interference density. 1) In the rational domain, Section 4.2, the very elegant result is found in Theorem 4.1: an upper bound is found to the interference density in function of only the cache size and line size. This estimation is used in STAMINA. 2) In the integer domain, Section 4.3, the main result is found in Theorem 4.4. These two sections are independent to each other and self contained. They are similar and the reader may choose to read either one, to have a grasp on the subject and leave the other for a more careful reading. In Section 4.4 we discuss the existence of solution for an integer equation. Corollary 4.2 summarizes the relation between the interference density, interference existence and miss ratio for two memory references. In Section 4.5 we present how to reduce the general case of interference to the simplest case: only two memory references and only one reuse vector are involved. In Section 4.6 we introduce the goal function, i.e. the function we want to minimize. We conclude this section with a brief discussion on the complexity of our approach.

4.1 Interference Equation Simplification

Consider the parameters of Equation 1 as constant, we can write the following interference equation (where the terms A_{-1} and B_{-1} absorb the parameters constant terms).

$$E(A, B, \bar{\mathbf{t}}, \bar{\mathbf{r}}) \equiv \begin{cases} A_{-1} - B_{-1} + \sum_{k=0}^{d-1}(A_k i_k - B_k t_k) = n\mathcal{C} + l \\ \text{with } \bar{\mathbf{i}} \in P^r(\bar{\mathbf{t}}) \text{ and } \bar{\mathbf{t}} \in P_{\bar{\mathbf{i}} \leq N(\bar{\mathbf{n}})} \end{cases} \tag{2}$$

The interference density of an equation is the ratio of iteration points where the interference equation is satisfied over all the iteration points. We identify with $0 \leq \rho_E \leq 1$ the interference density. When the reuse vector is short (distance equal to 1), and this is often the case for code optimized to exploit spatial locality, ρ_E has a very simple and elegant approximation. The Equation 2 can be simplified further as:

$$AB_{-1} + \sum_{k=0}^{d-1}(AB_k t_k) = n\mathcal{C} + l \text{ with } AB_k = A_k - B_k. \tag{3}$$

Again AB_{-1} may absorb some of the constant terms due to $\bar{\mathbf{t}} \pm 1 = \bar{\mathbf{i}}$.

Property 2 *If Equation 3 has solution and* $AB_k \bmod \mathcal{C} = 0$, $k \in [0, d-1]$, *then* $\rho_E = 1$.

Proof: $\lfloor \frac{AB_{-1} + \sum_{k=0}^{d-1}(AB_k t_k)}{\mathcal{C}} \rfloor = \lfloor \frac{AB_1}{\mathcal{C}} + \frac{\sum_{k=0}^{d-1}(AB_k t_k)}{\mathcal{C}} \rfloor = \lfloor \frac{AB_1}{\mathcal{C}} + \sum_{k=-1}^{d-1}(\frac{AB_k}{\mathcal{C}} t_k) \rfloor = \lfloor \frac{AB_1}{\mathcal{C}} \rfloor + \sum_{k=-1}^{d-1} n_k t_k$. The contribution of the iteration points will be always a multiple of $\mathcal{C}$. If there is solution, there is for every iteration point. $\diamondsuit$

By Property 2 we need to focus on the following equation.

$$E_{mod}(A, B, \bar{\mathbf{t}}, \bar{\mathbf{r}}) \equiv AB_{-1}^m + \sum_{k=0}^{d-1}(AB_k^m t_k) = n\mathcal{C} + l \text{ and } AB_k^m = AB_k \bmod \mathcal{C} \tag{4}$$

We distinguish rational solutions and integer solutions. The rational domain offers a neat and elegant formulation for the estimation of interference density, which is pretty much independent from the equation coefficients. The integer domain requires more work, but it has similar property even though it is more notational.

4.2 Interference Density, Rational Domain

The existence of integer solution assures the existence of rational solutions. We are going to describes a rational solution space such as if there is integer solution, it contains every integer solution. We then determine the density in the rational space. We introduce some definitions. Given $\bar{\dot{\mathbf{q}}}$ the smallest rational solution to Equation 4 we define as *grid* the solution points $\mathcal{G}(\bar{\mathbf{q}}) = \{\bar{\mathbf{g}} |$ for any k $g_k = \dot{q}_k + \frac{C}{AB_k^m} p_k$ with p_k natural number$\}$. We define as *grid cell* the smallest polydron that has all vertices in the grid. Given an integer l, we define as *band* the set of rational points $\mathcal{B}(l) = \{\Delta\bar{\mathbf{b}} | -L < l + \sum_{k=0}^{d-1} AB_k^m \Delta b_k < L\}$. Note that the origin always belongs to a band when $-L < l < L$. We define as *band cell* the polyhedron determined by the vertices obtained as intersection of the two hyper-planes: $-L = l + \sum_{k=0}^{d-1} AB_k^m \Delta b_k$ and $L = l + \sum_{k=0}^{d-1} AB_k^m \Delta b_k$ with the lines $\forall k \neq j, \Delta b_k = 0$ for any $0 \leq j \leq d-1$.

For every grid point we can determine a band $(\mathcal{B}(l + AB_{-1}^m))$. Every point in the band has the same solution value for n. In the band we can distinguish different band cells. The space determined by the grid and the bands on the grid points is dense as formalized in the following Lemma.

Lemma 1 *For any integer solution* $\bar{\mathbf{z}}$, *there is a grid point in the band passing through* $\bar{\mathbf{z}}$.

Proof: By definition, $AB^m_{-1} + \sum_{k=0}^{d-1} AB^m_k z_k = n_z \mathcal{C} + l_z$; we can represent $z_k = p_{z_k} \frac{\mathcal{C}}{AB^m_k} + \gamma z_k$ where $\gamma z_k = z_k \bmod \frac{\mathcal{C}}{AB^m_k}$. Therefore, $AB^m_{-1} + \sum_{k=0}^{d-1} AB^m_k (p_{z_k} \frac{\mathcal{C}}{AB^m_k} + \gamma z_k) = AB^m_{-1} + \mathcal{C} \sum_{k=0}^{d-1} p_{z_k} + \sum_{k=0}^{d-1} AB^m_k \gamma z_k = n_z \mathcal{C} + l_z$. We have that $n_z = \sum_{k=0}^{d-1} p_{z_k} + \lfloor \frac{AB^m_{-1} + \sum_{k=0}^{d-1} AB^m_k \gamma z_k}{\mathcal{C}} \rfloor$ and $l_z = (AB^m_{-1} + \sum_{k=0}^{d-1} AB^m_k \gamma z_k) \bmod \mathcal{C}$. We can see that $-(d-1) \leq \lfloor \frac{AB^m_{-1} + \sum_{k=0}^{d-1} AB^m_k \gamma z_k}{\mathcal{C}} \rfloor \leq d-1$ because for every k we have $AB^m_k \gamma z_k = z_k \bmod \mathcal{C}$. There are several points in the neighborhood of $\overline{\mathbf{z}}$ and in the grid that have the same solution in n, just increasing some of p_{z_k}. These points are in the band passing through $\overline{\mathbf{z}}$. ♢

For any grid cell there is only one band splitting the cell in two, so that two vertices are apart. The band is determined by two $d-1$-dimensional spaces or hyper-plane and by construction pass through grid points. In a 2-dimensional space the grid is a rectangle and the band is a line crossing the grid cell on only two grid points. Two different bands are crossing the remaining two vertices. See Figure 2 for an example in a 2-dimensional space. Now we are ready to determine the solution density.

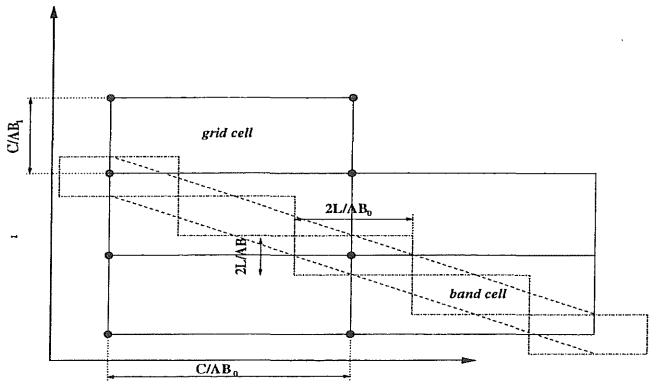

Figure 2: Grid cells and band cells in a plane. In a 2-dimensional space the grid cell is a rectangle and the band is between two lines crossing the grid cell on only two grid points. Two different bands are crossing the remaining two vertices.

Property 3 *Every grid cell has up to* $\frac{\mathcal{C}^d}{\prod_{k=0}^{d-1} AB^m_k}$ *solution points.*

Property 4 *Every grid cell intersects three bands and up to* $\frac{1}{2^{d-1}} (\frac{\mathcal{C}}{2L})^{d-1}$ *band cells.*

Proof: Consider the grid cell with sizes $\frac{\mathcal{C}}{AB^m_k}$ in a d-dimensional space (i.e. in a 3-dimensional space it is a cube). The projection of any band on any $d-1$-dimensional space (i.e. in a 3-dimensional space it determines a triangle rectangle) has $\frac{1}{2^{d-1}} \prod_{k \neq j} \frac{\mathcal{C}}{AB^m_k} / \frac{2L}{AB^m_k}$ solutions points. ♢

Property 5 *Every band cell has at most* $\frac{(2L)^d}{\prod_{k=0}^{d-1} AB_k^m}$ *solution points.*

Theorem 4.1 *If Equation 3 has solution and* $AB_k \bmod C \neq 0$, $\forall k \in [0, d-1]$ *and* $C \geq 2L$, *then* $\rho_E \leq \frac{1}{2^{d-1}} \frac{2L}{C}$.

Proof: By Property 2 we can focus on the Equation 4. By Lemma 1 the grid and the bands on the grid is a dense solution space. Every integer solution is in it. The density is computed on a grid cell as the ratio of solution points in a band intersecting a cell over the size of the grid cell. By Property 4 and 5, there are $\frac{1}{2^{d-1}}(\frac{C}{2L})^{d-1}$ band cells of size $\frac{(2L)^d}{\prod_{k=0}^{d-1} AB_k^m}$ in a grid cell. By Property 3 a grid cell has size $\frac{C^d}{\prod_{k=0}^{d-1} AB_k^m}$. Then we have the proof $\rho_E \leq \frac{1}{2^{d-1}} \frac{2L}{C}$ ♢

When the reuse vector has distance h, it is possible to write the Equation 2 as a system of h equations. Each equation differs for a constant term. We approximate the density as $\rho = \max(1, h\rho_E)$, counting each equation separately. This is a over estimation.

4.3 Interference Density, Integer Domain

In this section we investigate a more detailed analysis of the integer solutions of Equation 4. It is introduced as completion of the previous section but it is not used in the current implementation. In the rational domain, the solution density is introduced for a rational space and defined as the ratio of area/volumes. It may happen that the solution space determined by a band can be a rational number smaller than one. In an integral space has no meaning (there is either zero or at least one integer solution). In this section we show that the integral solution space has properties that are similar to the rational space.

To make this section self contained, we need to recall some of the theory of unimodular matrix transformations as well as elementary number-theoretic [2, 13]. To do so we introduce some notations for the operation "*great common divisor*" [13]. If a evenly divides b, we write that $a|b$. The greatest common divisor of two integers a and b is the largest integer d so that $d|a$ and $d|b$, and it is defined as $d = gcd(a, b)$. $gcd(a, b)$ is also the smallest positive integer of the set $\{ax + by | x, y \in \mathbb{Z}\}$. In [13], it is presented a variation of the Euclid algorithm to determine not only the $gcd(a, b)$ but also the integers x_0 and y_0 so that $d = ax_0 + by_0$.

General Solution by Unimodular Transformation U**:** given Equation 4, we consider the solutions for an arbitrary value of $n = n_0$ and $l = l_0$. The constant value will be $c = AB_{-1}^m + n_0 C + l_0$. The gcd-test (e.g. in [2]) can be used to verify if an equation has integer solution. Indeed, there is integer solution if and only if $g = gcd(AB_0^m, \ldots, AB_{d-1}^m)$ and $g|c$.

In [2], the author presents a general approach to determine all the integer solutions of an integer equation. We report the basic steps to determine the solution space. The equation 4 is represented in [2] in matrix-form as follows: $c = \overline{\mathbf{t}}^t \overline{\mathbf{A}}$ where c is an integer scalar, $\overline{\mathbf{t}}^t$ is a row vector (matrix of sizes $1 \times d$) and $\overline{\mathbf{A}}$ is a column vector (a matrix of size $d \times 1$). If $g|c$ then there is an matrix U, so that all the solutions are determined by the following expression:$\overline{\mathbf{x}} = (c/g, t_1, t_2, \ldots, t_{d-1})^t U$ where t_i are arbitrary integers and U is a unimodular matrix so that $U\overline{\mathbf{A}} = (g, 0, \ldots, 0)$. The column vector $(g, 0, \ldots, 0)$ is also an *echelon matrix* of size $d \times 1$, [2]. A matrix U is *unimodular*, if $|det(U)| = 1$. The matrix U is a linear transformation $U : \mathbb{T} \to \mathbb{X}$ with $\mathbb{X} = \mathbb{T} = \mathbb{Z}^d$ mapping the d-dimensional integer domain in itself; it exists the inverse matrix U^{-1}; it is a one to one mapping.

Shape of U: in [2], the author presents an algorithm (Algorithm 2.1) for the echelon reduction of vector $\overline{\mathbf{A}}$ and determination of matrix U. We reformulate the algorithm in the following. The matrix U of size $n \times n$ is determined in $n-1$ steps. In each step, a single row of U is determined, starting from row $n-1$ to row 1 (that is $U_{n-1,*}$ and $U_{1,*}$, "$*$" is wild; row 0 is computed for free in the last step). We observe the the i-th step we determine $\{U_{n-i,n-j}\}_{j \leq i+1}$ so that:

$$\begin{aligned} \textstyle\sum_{j \leq i+1} U_{n-i,n-j} A_{n-j} &= U_{n-i,n-i-1} A_{n-i-1} + x \textstyle\sum_{j<i-1} U^1_{n-i,n-j} A_{n-j} \\ &= U_{n-i,n-i-1} A_{n-i-1} + x g_{n-i} = 0 \end{aligned} \tag{5}$$

By the Euclid algorithm we compute:

$$\begin{aligned} \textstyle\sum_{j \leq i+1} U^1_{n-i-1,n-j} A_{n-j} &= U^1_{n-i-1,n-i} A_{n-i-1} + x \textstyle\sum_{j \leq i-1} U^1_{n-i,n-j} A_{n-j} \\ &= U^1_{n-i-1,n-i} A_{n-i-1} + x g_{n-i+1} = gcd(A_{n-i}, g_{n-i-1}) = g_{n-i} \end{aligned} \tag{6}$$

Property of unimodular matrix is that elements in any rows are relatively prime and elements in columns are relative prime too. Indeed, the row elements computed at any step are relatively prime by construction. The Euclid algorithm, which is applied to two rows, determines coefficients that are relatively prime as well. The matrix U is upper triangular with all the elements of first lower diagonal different from zero.

We can see that at any time the variation of Euclid algorithm can be applied on two integers, instead of the entire two rows, reducing the complexity of the algorithm from the original $2\log(\min(A_{n-1}, A_{n-2})) + \sum_{i=0}^{n-2}(n-i+1)\log g_i$ (the Euclide algorithm to compute $gcd(a,b)$ with $a > b$ has complexity $O(\log b)$) to the $\frac{n(n-1)}{2} + \log(\min(A_{n-1}, A_{n-2})) + \sum_{i=0}^{n-2} \log g_i$. The computation exploit the right associativity and it has the typical *fish spine* shape. Using the left and right associativity (of the operation), we can always reorganize the computation as binary tree, exploiting parallelism.

Use of U: the unimodular matrix U is a mapping between an iteration space in $\mathbb{X}$ (image) and an equivalent one $\mathbb{T}$. In $\mathbb{T}$ the solution space is defined as the plane $t_0 = c/g$. The number of integer solutions in $\mathbb{T}$ are as much as in $\mathbb{X}$ and all solutions are in a plane. In $\mathbb{T}$ the plane is dense, there are 2^{d-1} "*contiguous*" integer solution points. In $\mathbb{X}$ the image is stretched, the solution points may not be contiguous. We investigate how convex spaces in the solution plane without integer solution in $\mathbb{T}$ are mapped in $\mathbb{X}$. A *whole* W is specified by a set of solutions S for which there is no integer point $\overline{\mathbf{v}}$ in W such as $\overline{\mathbf{v}} = \sum_{\overline{\mathbf{s}}_k \in S} \lambda_k \overline{\mathbf{s}}_k$ with $0 \leq \lambda_k < 1$ and $\sum_k \lambda_k = 1$. In other words, a whole in the solution plane is a convex space without internal integer points (the points of the set are also the convex hull). If W is a whole in $\mathbb{T}$, its image $W * U = V$ in $\mathbb{X}$ is the set of integer points $\{\overline{\mathbf{v}} | \overline{\mathbf{w}}^t U = \overline{\mathbf{v}}^t \text{ and } \overline{\mathbf{w}} \in W\}$.

Theorem 4.2 *If W is a whole in $\mathbb{T}$ then $V = W * U$ is a whole in $\mathbb{X}$.*

Proof: By contradiction, suppose there exists a $\overline{\mathbf{v}} \in V$ so that $\overline{\mathbf{v}}^t = \sum_{\overline{\mathbf{v}}_k \in V} \lambda_k \overline{\mathbf{v}}_k^t$ with $0 \leq \lambda_k < 1$ and $\sum_k \lambda_k = 1$. U is a one to one mapping and therefore $\overline{\mathbf{v}}^t U^{-1} = \overline{\mathbf{w}}^t$ is in W. $\overline{\mathbf{v}}^t U^{-1} = \sum_{\overline{\mathbf{v}}_k \in V} \lambda_k \overline{\mathbf{v}}_k^t U^{-1} = \sum_{\overline{\mathbf{w}}_k \in W} \lambda_k \overline{\mathbf{w}}_k^t$, with $0 \leq \lambda_k < 1$ and $\sum_k \lambda_k = 1$. W would not be a whole. ♢

In $\mathbb{T}$, there are an infinite number of wholes, all of the same size (2^{d-1} points) but translated in space. Since U is a linear transformation and unimodular, the images of any whole has same number of integer points and same shape but they differ by a translation. Most interestingly, this is true even if we translate the entire plane, e.g. varying the value of c. Without loss of generality we can focus on just one of them.

The solution plane is a $(d-1)$-dimensional space immersed in a d-dimensional space. The whole W in $\mathbb{T}$ has a stretched image V in $\mathbb{X}$. We investigate a quantitative measure of the number of integer points in the d-dimensional space associated with only one whole. Note that if we are able to achieve such a measure we are able to achieve an estimation of the solution density in the integer domain. C_D, *cube of* D, is the smallest hypercube in $\mathbb{Z}^d$ that contains D. In fact, the cube of a set D is fully determined by its *defective sizes*. If we use the common notation of $\overline{\mathbf{e}}_i$ for the column vector with all zero but the i-th element, the defective size l_i of a cube C_D is $|\max_{\overline{\mathbf{d}} \in D} \overline{\mathbf{e}}_i^t \overline{\mathbf{d}} - \min_{\overline{\mathbf{d}} \in D} \overline{\mathbf{e}}_i^t \overline{\mathbf{d}}|$. Note that the correct sizes should be $l_i + 1$. We define as expansion factor of a set D the product $\prod_{i=0}^{d-1} l_i$ obtained by the defective sizes of C_D.

Two wholes share no internal points (because they do not have any) and, in this simple scenario, so do their cubes. The expansion factor is introduced to achieve an estimation of the internal volume of D when $D \subset C_D$.

Theorem 4.3 *Given a whole W in $\mathbb{T}$, the cube of $V = W * U$ has defective sizes* $l_i = |\max_{J \subseteq [1,i+1]} \sum_{k \in J} U_{k,i} - \min_{J \subseteq [1,i+1]} \sum_{k \in J} U_{k,i}|$.

Proof: By construction, W is $\{\overline{\mathbf{w}} = (\frac{c}{g}, z_1, \ldots, z_{d-1}) |$ every z_i is either 0 or 1$\}$. Then using the definition of cube of V, the sizes of C_V are determined by the elements in the column of U. $\diamondsuit$

Corollary 4.1 *If the equation $c = \overline{\mathbf{x}}^t A$ has solution, then there is a matrix U so that the general solution is $\overline{\mathbf{x}} = (\frac{c}{g}, t_1, \ldots, t_{d-1})^t U$ and the solution density is at most $\rho_d \leq (\prod_{i=0}^{d-1} l_i)^{-1}$ where l_i is $|\max_{J \subseteq [1,i+1]} \sum_{k \in J} U_{k,i} - \min_{J \subseteq [1,i+1]} \sum_{k \in J} U_{k,i}|$.*

There is only one solution plane and cubes of different wholes do not intersect. The density ratio can be estimated in the cube, which is the inverse of the expansion factor.

Putting all together: when n is not an arbitrary value but it is a variable, the equation may have solution for different values of n, each one of them describing a parallel plane. As long as the planes are far apart, the whole cubes do not intersect. But it may be that they interfere. The following Lemma enumerates the interference density in this new scenario.

Lemma 2 *If V is a whole in $\mathbb{X}$ and for every $i \in [0, d-1]$ we have C_V and there exists a l_j so that $l_j > \frac{C}{A_j^m}$, then $\rho_E^d \leq \max_k \frac{2l_k AB_k^m}{C} \rho_d$.*

Proof: For any n, the solution space is a set of parallel planes. On one dimension, the distance between any two planes is (asymptotically) C/A_i^m. There can be at most $\max_{k \in [0,d-1]} \frac{2l_k AB_k^m}{C}$ planes intersecting the inside of the cube. Each plane contributes with just one integral solution. Indeed, any solutions in a particular plane will belong to different cubes. Then from Corollary 4.1 the lemma follows. $\diamondsuit$

The last scenario is when for every solution of n there are different values for l. This is the case of two references that interfere for almost every iteration points in one (or more) dimensions. In these dimensions, we can give a very simple estimation and "*remove*" them from the equation and investigate the equation so modified as in the previous cases. The following Theorem estimates the interference density in this scenario.

Theorem 4.4 *Given a whole cube C_V in $\mathbb{X}$ and for every $i \in K \subset [0, d-1]$ we have $l_i > 2L$, then $\rho_E \leq ((\frac{2L}{C})^{|K|} \prod_{j \in K} \frac{1}{l_j}) * \rho_E^{(k=d-|K|)}$*

Proof: In this case we have $|K|$ variables so that each satisfies the equation in an interval of size C at most $2L/l_i$ (with $i \in K$) times, therefore with density $\frac{2L}{Cl_i}$. We restrict the investigation on the other $d - |K|$ variables, and we apply Lemma 2. $\diamondsuit$

As we did in the other domain, when the reuse vector has distance h, it is possible to write the Equation 2 as a system of h equations. Each equation differs for a constant term. We approximate the density as $\rho = \max(1, h\rho_E)$, counting each equation separately.

4.4 Interference Existence

The interference density is based only on the observation of the interference equations but it does not take in account the definition domain of the variables involved in the equation itself. The existence of a solution is still to be found (e.g. [23, 6, 9]).

The *interference existence* for an equation is a function $\chi_{P(L)}$ where $P(L)$ is a polyhedron determined by the interference equation and for which the line size L is parameter. If $P(L_i)$ has an integral solution, $\chi_{P(L_i)} = 1$; if it has not integral solution $\chi_{P(L_i)} = 0$.

χ is a monotone increasing function. Indeed, if $L_0 \leq L_1$, then $\chi_{P(L_0)} \leq \chi_{P(L_1)}$. If there is interference for a line size L_i, there is interference for any larger line size. A preliminary test to verify that an integer equation has solution is by the $gcd()$ test (e.g. [2]). But the variable l in the interference equation (Equation 4) has coefficient one, the test is always verified. If we consider l as a constant and assign any of the values in $(-L, L)$, it becomes a constant term of the equation with AB^m_{-1}. The equation has no solutions if g is $gcd(C, A^m_0, \ldots, A^m_{d-1})$ and AB^m_{-1} **mod** $g = h$ with $h \notin (-L, L)$. If $h \in (-L, L)$, the definition domain must be checked further to investigate the existence of solutions (STAMINA uses a functionality of *polylib* [30] it is an exhaustive search).

To conclude the section we summarize the main result in the following Corollary.

Corollary 4.2 *If $m > 0$ is the number of coefficients in Equation 3 so that $(A_i - B_i)$* **mod** *$C \neq 0$, then the cache miss ratio is at most $\rho\chi_{P(L)}$ where ρ is the interference density (either one in Theorem 4.1 or Theorem 4.4) and χ is the interference existence of the equation. If $m = 0$, the cache miss ratio is at most $\chi_{P(L)}$.*

4.5 Reduction to Single Reference Interference

The simplest case we need to analyze is the following. There is an iteration space with $|I|$ iteration points; there is a reference with only one interference equation E (one reuse vector of size h and one interferer); the interference polyhedron is function of the line size and it is denoted as $P(L_i)$. The number of misses is the following $M = |I|\rho\chi_{P(L)}$. We show shortly our approach to reduce the general case to the simplest case (see [15]). The approach is constructive .

When there is a reference R_A with multiple interferer, k, and R_A has just one reuse vector $\bar{\mathbf{r}}$, we indicate the density for each equation as ρ_i and the solution existence function as $\chi_{P_i(L)}$ $(0 \leq i \leq k)$. The interferences due to different interferers are independent to each other, we can add their contribution. $\mu(L) = \sum_{i=0}^{k} \rho_i \chi_{P_i(L)}$, we identify the function $\mu(L)$ as *interference density per reference.* The upper bound to the number of misses for a direct mapped cache is $|I|\mu(L)$. If we would model a m-way associative cache, we could consider as estimation of the number of misses $|I|\lfloor\frac{\mu(L)}{m}\rfloor$. This is an

approximation, not an upper bound, because the interference density does not give any information on the temporal distribution of the misses.

When there are only two references R_A (interferee) and R_B (interferer). R_A has multiple reuse $\{\bar{\mathbf{r}}_i\}_{0,m-1}$ so that $\bar{\mathbf{r}}_0 > \bar{\mathbf{r}}_1 > \cdots > \bar{\mathbf{r}}_{m-1}$. Every reuse vector $\bar{\mathbf{r}}_i$ is associated with a bounded polytope $P^{r_i}(\bar{\mathbf{t}})$ so that $P^{r_i}(\bar{\mathbf{t}}) \supset P^{r_j}(\bar{\mathbf{t}})$ with $i < j$, and it is easy to see that $\cap_{i=0}^{k} P^{r_i}(\bar{\mathbf{t}}) = P^{k}(\bar{\mathbf{t}})$. We will consider only the shortest reuse. If the reuse is prevented, there is a miss. If it is not, there is not a miss. This is the simplest case. [2]

When there is a reference R_A with multiple interferers and R_A has multiple reuse $\{\bar{\mathbf{r}}_i\}_{0,m-1}$ so that $\bar{\mathbf{r}}_0 > \bar{\mathbf{r}}_1 > \cdots > \bar{\mathbf{r}}_{m-1}$. Every reuse vector $\bar{\mathbf{r}}_i$ is associated with a bounded polytope $P^{r_i}(\bar{\mathbf{t}})$. A set of equations $E_0 \ldots E_{n-1}$ represent the interferences with different references. For each equation we consider only the shortest reuse vector. We reduce to the case of a reference with single reuse vector and multiple interferers.

The approach investigates a subset of the points in the iterations space. Indeed, a reuse may be prevented and cause a miss, if the previous iteration is in the iteration space. Given a reuse vector $\bar{\mathbf{r}}$, the iterations non inspected are $P = \{\bar{\mathbf{j}} \in P_{\bar{\mathbf{i}} \leq \bar{\mathbf{n}}} | (\bar{\mathbf{j}} - \bar{\mathbf{r}}) \notin P_{\bar{\mathbf{i}} \leq \bar{\mathbf{n}}}\}$. It can be written as the union of non intersecting simplicial rectangular sets $P_B(\bar{\mathbf{r}}) = P_{(n_0 - r_0 < j_0)} \cup P_{(n_0 - r_0 \geq j_0, n_1 - r_1 \leq j_1)} \cdots P_{(n_0 - r_0 \geq j_0, n_1 - r_1 \geq j_1 \ldots n_k - r_k \geq j_k)}$. The measure of the points investigated $(P_{\bar{\mathbf{i}} \leq \bar{\mathbf{n}}} - P_B(\bar{\mathbf{R}}))$ over the number of points in the iteration space $(P_{\bar{\mathbf{i}} \leq \bar{\mathbf{n}}})$ is an estimation of the confidence of the analysis. The confidence of the analysis is based also on orientation of the reuse vector. Reuse across the inner loop is more likely to be prevented, because of the distance.

4.6 Interference and Reuse: Optimal Line Size

In this section we present an approach to determine the best line size L_{opt} for given loop nest, I, and a set of references $\{R_i\}$ with $0 \leq i \leq m-1$ in the inner loop.

As result of several analysis steps: we can determine the reference reuse vectors and the type of reuse, i.e. *spatial* or *temporal*. If a reference has spatial reuse and no interference is present, the reference has a miss every $\frac{\ell}{s}$ accesses, where ℓ is the line size in data elements and s is the length of the spatial reuse in elements. If Interference is present some of the reuse can be prevented. In the following we formalize the trade off between spatial reuse and interference as $\eta(L) = \frac{s}{\ell} + \mu(L)$ if $\mu(L) < 1$, otherwise $\eta(L) = \mu(L)$. This is the *miss density* for spatial reuse, when the line size is L, the number of elements per line is ℓ. In general, $\eta(L)$ is not monotone increasing. It may have a single valley and it has always a single minimum, because combination of a monotone decreasing function and a monotone increasing function. It is always possible to label the references so that R_i with $0 \leq i \leq n-1$ are references with spatial reuse

[2] In general, even if the shortest reuse is prevented it may be there is no miss, a longer reuse is satisfied.

and R_i with $n \leq i \leq m-1$ with temporal reuse. The density of the misses for the loop nest is defined as follows: $\epsilon(L) = \sum_{i=0}^{n-1} \eta_i(L) + \sum_{i=n}^{m-1} \mu_i(L)$. In general $\epsilon(L)$ may have one valley and has one minimum.

Now we have a quantitative tool to measure the effect of line size on cache performance, the number of misses. Indeed, $|I|\epsilon(L)$ is the number of misses for which the line size has any effect. The determination of the optimal line size is the line size for which $\epsilon(L)$ is minimum.

5 STAMINA Implementation Results

The reuse and interference analysis is implemented in the software package StaMInA (*Sta*tic *M*odeling of *In*terference *A*nd reuse as a part of AMRM compiler suite). It is built on top of SUIF 1.3 compiler adapting the code developed in [15] and using *polylib* [30, 9, 8, 10]. We consider three examples to explore three important aspects of our analysis.

5.1 Swim from SPEC 2000

`swim` is a scientific application. It has a main loop with four function calls. Each function has a loop nest for which the bounds are parameters introduced at run time. For sake of exposition, we present the analysis for the main loop nest of one procedure *calc*1() (Figure 3 written in C language). We analyze the interference for two different matrix sizes, the reference size 1335×1335 and the power of two 1024×1024. For the reference size, there is no interference for any cache line. For power of two matrices there is always interference. The execution of SWIM with reference input takes 1hr on a sun ultra 5, 450MHz. Any full simulation takes at least 50 times more. Even the single loop simulation is time consuming. Our analysis takes less than one minute for each routine whether there is interference or there is no interference.

Due to the number of equations to verify, it is very difficult to verify by hand the accuracy of the analysis. We simulate 10 of the 800 calls to the *calc*1 routine using *cachesim5* from *Shade* [12]. The simulation results confirm our analysis.

5.2 Self Interference

We now consider self interference. Self interference happens when two references of the same array, or the same reference in different iterations, interfere in cache. The example, Figure 6, is the composition of six loops with only one memory reference in each. Each memory reference has a different spatial reuse and it is very long. STAMINA recognizes

```
#define N1 1335
#define N2 1335

extern  double U[N1][N2], V[N1][N2], P[N1][N2],UNEW[N1][N2], VNEW[N1][N2],
        PNEW[N1][N2], UOLD[N1][N2],  VOLD[N1][N2], POLD[N1][N2],
        CU[N1][N2], CV[N1][N2], Z[N1][N2], H[N1][N2], PSI[N1][N2];

extern double D0, DX, DY;


void calc1(int M, int N) {


  int i,j;
  double FSDX,FSDY;

  for (i=0;i<M;i++)
    for (j=0;j<N;j++) {
      //      RN 0         =       1        2       3
      CU[i+1][j] = D0*(P[i+1][j]+P[i][j])*U[i+1][j];
      //C      #  1 2 3 0
      //C      RN 4                 5        2       6
      CV[i][j+1] = D0*(P[i][j+1]+P[i][j])*V[i][j+1];
      //C      # 5 2  6 4
      //C      RN 7                      8          6                9
      Z[i+1][j+1] = (FSDX*(V[i+1][j+1]-V[i][j+1])-FSDY*(U[i+1][j+1]
      /* C      RN        3          2       1        10        5    */
                    -U[i+1][j]))/(P[i][j]+P[i+1][j]+P[i+1][j+1]+P[i][j+1]);
      //      # 8 6 9 3 2 1 10 5 7
      //        RN 11        2          3        3        12      12
      H[i][j] = P[i][j]+D0*(U[i+1][j]*U[i+1][j]+U[i][j]*U[i][j]
     //      RN        9          9        13      13
                          +V[i][j+1]*V[i][j+1]+V[i][j]*V[i][j]);
      //      #  3 12  9 13 2 11
    }
}
```

Figure 3: SWIM: calc1() in C code, in the comment lines the reference number and the order of the references are specified.

that the interval between reuses is after one iteration of the outer loop. It computes the reuse distance and, in the current implementation, it fixes the value of the interference density at $\rho = 1$. It assumes there is a miss due to capacity (in general the distance is not a constant and it cannot be compared to the cache size). For this particular case, it is a tight estimation. In general it is an over estimation. The existence of interference plays the main role, it discriminates when there is interference and when to count the interferences. In Table 1, we report the results of the analysis.

5.3 Tiling and Matrix Multiply

We analyze two variations of the common ijk-matrix-multiply algorithm (e.g. [17]). In Figure 4 the size of matrix A is not a power of two, but it is for B and C. The size of A has been chosen so that if there is interference due the reference on A, it does not happen very often. The index computation for A is parameterized ($0 \leq m \leq 64$ and $0 \leq n \leq 64$). Accesses on matrix C interfere with the accesses on B. Due to the upper bounds we choose for the parameters, A does not interfere with any other matrix. Even if it could, the interference density would be small. We are able to distinguish

Loop 0	Line	8	16	32	64	128	256
	$\epsilon_{ct}(L)$	0.50	0.25	1.00	1.00	1.00	1.00
Loop 1	$\epsilon_{ct}(L)$	0.50	0.25	0.12	1.00	1.00	1.00
Loop 2	$\epsilon_{ct}(L)$	0.50	0.25	0.12	0.06	1.00	1.00
Loop 3	$\epsilon_{ct}(L)$	0.50	0.25	0.12	0.06	0.03	1.00
Loop 4	$\epsilon_{ct}(L)$	0.00	0.00	0.00	0.00	0.00	0.00
Loop 5	$\epsilon_{ct}(L)$	0.00	0.00	0.00	0.00	0.00	0.00

Table 1: Self interference example. Loop four and five have no interference dependent from the line size, the output is set to zero

two different contribution: at compile time, $\epsilon_{ct}(L)$, and at run time, $\epsilon_{rt}(L)$. $\epsilon_{rt}(L) = 0$ for any L and $\epsilon_{ct}(\{8, 16, 32, 64, 128, 256\}) = \{2.00, 1.00, 2.00, 2.00, 2.00, 2.00\}$. Reference on A does not interfere with C and B with $0 \leq n, m \leq 64$. It would if we use lager parameters values. We can see that the suggested line size is 16B. This example has been introduced to show a case where the optimal line reduces interference and it is smaller than the common 32B line. Let us consider a more interesting example, where we analyze the tiled version of matrix multiplication Figure 5. We analyze only the loop nest in the procedure *ijk_matrix_multiply_4*, and the result of the analysis is that $\epsilon_{ct}(L) = 0$ for any L and $\epsilon_{rt}(\{8, 16, 32, 64, 128, 256\}) = \{2.00, 2.00, 2.00, 2.00, 2.01, 2.03\}$. Every matrix interferes with every matrix. The interference due to matrix A is negligible since is an invariant for the inner loop. The interference between C and B can be at every iteration point. There is no interference whenever $|m - n| \mathbf{mod}\ \mathcal{C} = L$. This example is very peculiar because the line size is not set once for loop nest, it is determined at run time.

In the example in Figure 4 the analysis takes no more than two minutes. For the example in Figure 5 it takes more than 8 hrs, on a Sun ultra 5 450MHz. The difference of the execution times is expected. Most of the time is spent in the search for the existence of the integer solution. This is our performance bottleneck and it will be subject of further investigations/optimizations.

6 Summary and Future Work

We present a fast approach to statically determine the line size effect on the cache behavior of scientific applications. We use the static cache model introduced in [15] and we present an approach to analyze parameterized loop bounds and memory references. The approach is designed to investigate the trade-off between spatial reuse and interferences of loop nests on direct mapped cache. Experimental results demonstrate the

```
/* B[0][0] 120000000
   C[0][0]
*/
#define MAX 4000
#define MAXCOL 2048


double A[MAX][MAX],  B[MAXCOL][MAXCOL],  C[MAXCOL][MAXCOL];
void ijk_matrix_multiply( int n, int m) {

  int i,j,k;


  for(i=0;i<n;i++)
    for(k=0;k<n;k++)
      for(j=0;j<n;j++)
        C[i][j+3] += A[i][k+m] * B[k][j];


}
```

Figure 4: Matrix Multiply. Two parameters: loop bounds and A offset. The parameters n and m up to 64

accuracy and efficiency of our approach. We plan to expand our implementation to consider multi-way associative caches and to improve the performance of the existence test, by applying the *gcd*-test as proposed in [2].

7 Acknowledgment

The authors wish to thank Vincent Loechner, Somnath Ghosh, Dan Hirschberg and the members of AMRM project. They helped on Ehrhart polynomials and the existence test, cache miss equation determination, interference estimation and moral/technical support, respectively. Financial support for this research was provided by DARPA/ITO under contract DABT63-98-C-0045.

```
#define MAX 2048
double A[MAX][MAX],  B[MAX][MAX],  C[MAX][MAX];

void ijk_matrix_multiply_4( int x,int y, int z, int m, int n, int p ) {
  int i,j,k;

  for(i=0;i<x;i++)
    for(k=0;k<y;k++)
      for(j=0;j<z;j++) {
        C[i][j+m] += A[i][k+n] * B[k][j+p];

}


void matrix_multiply_new_tiling() {
  int ii,jj,kk;

  for(kk=0;k<MAX/b;kk++)
    for(ii=0;i<MAX/b;ii++)
      for(jj=0;j<MAX/b;jj++)
        ijk_matrix_multiply_4(min(b,MAX-ii*b), min(b,MAX-jj*b),
                              min(b,MAX-kk*b), (ii*MAX+jj)*b,
                              (ii*MAX+kk)*b,(kk*MAX+jj));
}
```

Figure 5: Tiling of Matrix Multiply. 6 parameters: loop bounds and A,B and C offsets. The first procedure describes the computation on a tile.

References

[1] E. Anderson, T. Van Vleet, L. Brown, J. Baer and A.R. Karlin, "On the Performance Potential of Dynamic Cache Line Sizes". *Technical Report UW-CSE-99-02-01.*

[2] U. Banerjee, *Loop Transformations for Restructuring Compilers The Foundations.* Kluwer Academic Publishers, January 1993

[3] A. Barvinok, "A polynomial Time Algorithm for Counting Integral Points in Polyhedra when the Dimension is Fixed". *Mathematics of Operations Research*, vol. 19, 1994, N 4, pag, 769.

[4] A. Barvinok, "Computing the Ehrhart Polynomial of a Convex Lattice Polytope". *Discrete and Computational Geometry*, vol. 12, 1994, pag. 35-48.

[5] A. Barvinok and J.E. Pommersheim, "An Algorithmic Theory of Lattice Points in Polyhedra".*Manuscript.*

[6] N. Bermudo, X. Vera, A. Gonzales and J. Llosa, "An Efficient Solver for Cache Miss Equations", *ISPASS* 2000 c

[7] Ph. Clauss and B. Meister, "Automatic Memory Layout Transformation to Optimize Spatial Locality in Parameterized Loop Nests", *4th Annual Workshop on Interaction between Compilers and Computer Architectures, INTERACT-4*, Toulouse (France), January 2000.

[8] Ph. Clauss, "Advances in parameterized linear diophantine equations for precise program analysis", *[ICPS RR 98-02]*, September 1998.

[9] Ph. Clauss, V. Loechner, "Parametric Analysis of Polyhedral Iteration Spaces", *research report ICPS 96-04, IEEE Int. Conf. on Application Specific Array Processors, ASAP'96*, Chicago, Illinois, August 1996.

[10] Ph. Clauss, "Counting Solutions to Linear and Nonlinear Constraints through Ehrhart polynomials: Applications to Analyze and Transform Scientific Programs", *research report ICPS 96-03, 10th ACM Int. Conf. on Supercomputing, ICS'96*, May 1996.

[11] Ph. Clauss, "Handling Memory Cache Policy with Integer Points Countings", *Euro-Par'97*, Passau, August 1997, LNCS 1300, p. 285-293.

[12] B. Cmelik and D. Keppel, "Shade: a fast instruction-set simulator for execution profiling", *Proceedings of the 1994 conference on Measurement and modeling of computer systems*, 1994, Pages 128 - 137

[13] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to Algorithms*. The MIT Press.

[14] H. Du, P. D'Alberto and R. Gupta, "Memory Adaptation Techniques: an Unified Overview across Benchmark Suites", *Technical Report* #01-41.

[15] S. Ghosh, M. Martonosi and S. Malik, "Cache Miss Equations: a Compiler Framework for Analyzing and Tuning Memory Behavior", *ACM Transactions on Programming Languages and Systems*, Vol. 21, No. 4, July 1999, Pages 703-746.

[16] K. Ghose and M.B. Kamble "Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation", *Proceedings 1999 international symposium on Low power electronics and design*, 1999, Pages 70 - 75

[17] G.H. Golub, C.F. Van Loan *Matrix Computations*, Johns Hopkins Series in the Mathematical Sciences.

[18] X. Ji, D. Nicolaescu, A. Veidembaum, A. Nicolau and R. Gupta, "Compiler-Directed Cache Assist Adaptivity". *ICS Techincal Report* #00 17, May 2000.

[19] M.B. Kamble and K. Ghose "Analytical Energy Dissipation models for Low-power Caches", *Proceedings of the 1997 international symposium on Low power electronics and design*, 1997, Pages 143 - 148

[20] K.S. McKinley and O. Temam, "A Quantitative Analysis of Loop Nest Locality". *APLOS VII* 10/96 MA, USA.

[21] S.S. Muchnick, *Advanced compiler design implementation*, Morgan Kaufman.

[22] W. Pugh, "Counting Solutions to Presburger Formulas: How and Why", *SIGPLAN Programming language issues in software systems* 94-6/94 Orlando, Florida, USA

[23] W. Pugh "A practical algorithm for exact array dependence analysis", *Communications of the ACM* Volume 35 , Issue 8 (August 1992)

[24] R.P. Stanley, *Enumerative Combinatorics, Volume I*, Cambrige Studies in Advanced Mathematics 49.

[25] W. Tang, A. Veidenbaum, A. Nicolau, R. Gupta, "Cache with Adaptive Fetch Size", *ICS Technical Report* #00-16, April 2000.

[26] W. Tang, A. Veidenbaum, A. Nicolau, R. Gupta, "Adaptive Line Size Cache", *ICS Technical Report* #99-56, Nov. 1999.

[27] . P. Van Vleet, E. Anderson, L. Brown, J. Baer and A.R. Karlin, "Pursuing the Performance Potential of Dynamic Cache Line Sizes", *Int. Conference on Computer Design (ICDD'99)* October 1999.

[28] A.V. Veidenbaum, W. Tang, R. Gupta, A. Nicolau and X. Ji, "Adaptive Cache Line Size to Application Behavior", *In Proceedings of International Conference on Supercomputing (ICS)*. June 1999, pp.145-154.

[29] X. Vera, J. Llosa, A. Gonzales and N. Bermudo, "A Fast and Accurate Approach to Analyze Cache Memory Behavior", *EUROPAR 2000.*

[30] D.K. Wilde, "A library for Doing Polyhedral Operations", *Publication interne* N 785, 1993

[31] M.E. Wolf and M.S. Lam, "A data Locality Optimizing algorithm", *Proc. of the ACM SIGPLAN'91 Conference on programming languages design and implementation*, Toronto, Ontario, Canada, June 26-28, 1991, pages 30-44.

```
#define CACHE_SIZE 16384

int A[CACHE_SIZE /16][(CACHE_SIZE+16)/4];
int B[CACHE_SIZE / 32][(CACHE_SIZE+32)/4];
int C[CACHE_SIZE / 64][(CACHE_SIZE+64)/4];
int D[CACHE_SIZE / 128][(CACHE_SIZE+128)/4];
int E[CACHE_SIZE / 256][(CACHE_SIZE+256)/4];
int F[CACHE_SIZE / 512][(CACHE_SIZE+512)/4];


int
main ()
{
  int i,j,k,l;
  int step;
  l = 0;


  for (j=0;j<4;j++) {
    for (k = 0; k < CACHE_SIZE / 16 k++)
      A[k][j]++;
  }

  for (j=0;j<8;j++) {
    for (k = 0; k < CACHE_SIZE / 32; k++)
      B[k][j]++;
  }

  for (j=0;j<16;j++) {
    for (k = 0; k < CACHE_SIZE / 64; k++)
      C[k][j]++;
  }


  for (j=0;j<32;j++) {
    for (k = 0; k < CACHE_SIZE / 128; k++)
      D[k][j]++;

  }

  for (j=0;j<64;j++) {
    for (k = 0; k < CACHE_SIZE / 256; k++)
      E[k][j]++;
  }

  for (j=0;j<128;j++) {
    for (k = 0; k < CACHE_SIZE / 512; k++)
      F[k][j]++;
  }

  return 0;
}
```

Figure 6: Self Interference and analysis results