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

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

Abstract—Caches are crucial components of modern processors; they allow high-performance processors to access data fast and, due to their small sizes, they enable low-power processors to save energy—by circumventing memory accesses. We examine efficient utilization of data caches in an adaptive memory hierarchy. We exploit data reuse through the static analysis of cache-line size adaptivity. We present an approach that enables the quantification of data misses with respect to cache-line size at compile-time using (parametric) equations, which model interference. Our approach aims at the analysis of perfect loop nests in scientific applications; it is applied to direct mapped cache and it is an extension and generalization of the Cache Miss Equation (CME) proposed by Ghosh et al. (1999). Part of this analysis is implemented in a software package, STAMINA. We present analytical results in comparison with simulation-based methods and we show evidence of both the expressiveness and the practicability of the analysis.

Index Terms—Cache-line size adaptivity, spatial locality, interference, parameterized loop nests.

1 Introduction

In modern uniprocessor systems, the memory hierarchy is an important concern for 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 percent of the total chip power [1], [2]. In recent years, there has been a great endeavor to engineer several levels of cache for the exploitation of performance and power. In particular, we have studied the effect of adaptivity in cache subsystems and we have built an architecture as a prototype that enables static and dynamic adaptation of memory hierarchy: its configuration and policies [3]. In this paper, we turn our attention to (compiler-driven) cache-line size adaptation of direct mapped data caches [4], [3]. In fact, the architecture changes the cache-line size dynamically (by hardware monitoring or application instruction) during the execution of the application. To fully exploit 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 minimum energy dissipation. The related work on cache behavior analysis can be distinguished in profiling-based and static approaches.

Profiling is an approach that uses the direct measure of performance as feed-back to drive the fine-tuning of some architecture parameters. The main goal is to improve performance of an application when applied on a representative input [5]. The approach is flexible and it can be used for the analysis of the whole application, as well as part of it. However, profiling has two limitations: The performance of an application is often dependent on the inputs and, of course, the analysis cannot be faster than the execution of the application itself.

Static approaches are basically independent of the *inputs* and, thus, the analysis can be performed just once at compile time. In particular, static approaches analyze mostly perfect loop nests and these loop nests are ubiquitous in scientific applications. (As reported by Ghosh et al. [6], 244 loop nests are statically analyzable, 289 are parameterized loop nests, and 189 are not analyzable—Table I, page 707—for SPECfp 95 benchmarks.) In fact, static approaches model data-cache misses of a memory reference in a perfect loop nest by using **cache miss equations** (CME) [6]. When the CMEs are defined for a given memory reference and a loop nest, every iteration in the loop nest (or a sampled version such as in [7], [8]) is checked as to whether or not it satisfies the equations. If an iteration satisfies the equations, then the memory reference has a cache miss at that particular iteration. Thus, the approaches count the solutions of the equations to achieve an estimation of the number of cache misses. As an extension of this idea, Vera and Xue [9] propose an approach to analyze the whole program based on their cache-miss solver developed by the same group [8]. For parameterized loop nests, the authors (both Ghosh et al. and Vera and Xue) suggest that the approach can be applied at runtime in similar fashion, because the parameters are known. However, there are two limitations in the current static approaches. First, the loop nest bounds must be known at compile time. This is not realistic (e.g., 289 loop nest in SPECfp) because they are often parameterized. Also, even if the analysis is performed at runtime, it may be impractical because these loop nests can be very large. Second, the analyzable loops are *sensitive* to tiling loop transformation. For example, if tiling is

- *P. D'Alberto, A. Nicolau, and A. Veidenbaum are with the Computer Science Department, University of California at Irvine, Irvine, CA 92697. E-mail: {paolo, nicolau, alexv}@ics.uci.edu.*
- *R. Gupta is with the Computer Science and Engineering Department, University of California at San Diego, La Jolla, CA 92093. E-mail: rgupta@ucsd.edu.*

Manuscript received 20 Dec. 2002; revised 2 Apr. 2004; accepted 20 Sept. 2004.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 118021.

performed on the three-loop-algorithm for matrix multiplication and the tile sizes do not evenly divide the loop bounds, the inner loop bounds cannot be represented by affine functions. The resulting nest is not analyzable.

To attack and overcome these limitations, we propose a static approach to investigate perfect (parameterized) loop nests and to determine the relation between cache-line size and number of misses on a per-nest-base for a direct-mapped cache. The analysis result is annotated in the code and it can be used at runtime to set the line size. The approach is especially suitable for applications with references having reuse within a few iterations in the inner loop and exploiting spatial locality [10].

A well-known paper on optimizing for data locality and parallelism exploitation is by Kennedy and McKinley [11]. In practice, they assume that there is little or no interference for a small number of iterations in the innermost loop. Spatial locality is exploited, if any is available in the inner loop, under the assumption that cache misses are independent of any interference. The authors propose different loop optimizations (i.e., loop permutations) to exploit maximum spatial and temporal locality in the innermost loop. The examples presented in this paper, as many other loop nests in real applications, do not satisfy McKinley and Kennedy's assumption. Cache interference can be the major contributor of cache misses in inner loops. Instead, our approach considers such interference and, in practice, the two approaches are *orthogonal*.

The paper is organized as follows: In Section 2, we use an example to outline the common pitfalls of the current approaches and *intuitively* introduce the reader to our approach. In Section 3, we introduce the notations about loop nests, cache equations, and parameterized loop nests. In Section 4, we introduce the theoretical framework and our approach. Finally, in Section 5, we show the results of our analysis for three representative examples.

2 Our Approach: An Overview

In this section, we present the novel contribution of our approach using a simple example. We break down the problem and the solution—as our approach does—in order to present the following three points: first, the challenges that current analysis tools face determining data cache misses, second, the terminology that is adopted in this paper, third, a quantitative and informal application of our approach—we shall see a rigorous notation and analysis in Sections 3 and 4.

Consider the example shown in Fig. 1. The two memory references $A[i][j+start]$ and $B[i][j]$ in the inner loop body are affine functions of the loop indices, that is, the indices i and j. The indices are represented as a vector $\mathbf{k} = (i,j)^t$: The first entry is the outermost index, the second entry is the innermost index. A particular iteration of the loop nest is simply identified by $\mathbf{k}_0 = (i_0, j_0)^t$.

The memory references exploit spatial reuse in the inner loop. Two consecutive accesses to matrix A (i.e., $A[i][j+start]$ and $A[i][j+1+start]$) and to matrix B tend to exploit spatial locality. We describe this reuse property by the vector $\mathbf{r} = (0,1)^t$. The reuse vector is relative to an iteration, that is,

```
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 <1024-100 */
}
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);
}
```

Fig. 1. Example: paramterized loop bounds and interference. Matrices A and B have the same number of columns but a different number of rows. The matrices are in row-major format and they are consecutively stored—as their declaration suggests. The procedure foo has two parameters, m and $start$. The first parameter m specifies the size of the iteration space. The second parameter, $start$, is an offset to access the columns of matrix A.

the cache line read at iteration $(i_0, j_0)^t$ will be read again at the next iteration $(i_0, j_0+1)^t$ (i.e., $\mathbf{r} = (i_0, j_0+1)^t - (i_0, j_0)^t$).[1]

When the two references of matrices A and B at an iteration $(i_0, j_0)^t$ are mapped to the same cache line, there is interference in the cache. The cache interference prevents the spatial reuse as the same line may be reloaded. Without cache interference, we can estimate the number of cache misses as $2m^2/\ell$, where $\ell = L/8$ is the number of double precision float numbers in a cache line, L cache-line size.

Let us consider the order of memory accesses in the loop body as follows: A read of A precedes a read of B, which precedes a write of A. The read of A has spatial reuse $(0,1)^t$ and temporal reuse $(0,0)^t$. The read of B has spatial reuse only. The temporal and spatial reuse of A is not exploited when the access to B is mapped to the same cache line; in other words, when the address of $B[i][j]$ is the address of $A[i][j+start]$ plus a multiple of the cache size and an offset no larger than the cache-line size at the iteration specified by $\mathbf{k} = (i,j)^t$ (where $0 \leq i, j < m$). We model interference by the following equation:

$$\begin{aligned}&B_{-1} + 8{,}192i + 8j = \\ &B_{-1} + 16{,}384{,}000 + 8{,}192i + 8j + 8start + n\mathcal{C} + q.\end{aligned}$$

The constant B_{-1} is the start addresses of B; the constant $\mathcal{C} = 16 * 1{,}024$ is the cache size; the variable n has positive integer values and q has integer value so that $|q| < L$. We simplify the equation to: $16{,}384{,}000 - 8start = n\mathcal{C} + q$. The set of inequalities defines a parameterized *polyhedron*.[2]

When $8start \mathbf{\ mod\ } \mathcal{C} < L$, we have a solution (e.g., for $n = 1{,}000$ and $q = 8start \mathbf{\ mod\ } \mathcal{C}$). The solution of the equation stands for a cache interference. The interference prevents the cache-line reuse and we have a cache miss.

Note that the optimal cache-line size and the number of cache misses are a function of the parameter $start$. The optimal line size is $L_{opt} = 8start \mathbf{\ mod\ } \mathcal{C}$ (i.e., no

1. Note that the reuse depends on no parameters.

2. Static approaches based on the one proposed by Ghosh et al. are not practical for large polyhedra. The analysis must be repeated for each parameter value.

cache interference).[3] The number of cache misses is $M = 2m^2(L-\Delta)/L$; the term m^2 specifies the number of iterations; the constant 2 is the number of references we analyze; the last term $(L-\Delta)/L$, where $\Delta = 8start \textbf{ mod } \mathcal{C}$, specifies the fraction of accesses that effect cache misses caused by cache-line underutilization.[4]

Now, consider matrix $B[100][512]$ instead of $B[100][1,024]$. The interference equation is

$$16,384,000 - 4,096i = 8start + n\mathcal{C} + q.$$

When i is 0, it is the previous equation. Both memory references interfere in the cache for the first m iterations, when $8start \textbf{ mod } \mathcal{C} < L$. For $i = 1$ and for the same values of *start*, there is no interference. Indeed, we have interference every four iterations of i. We define this ratio as *interference density*, denoted by $\rho = \frac{512*8}{C} = 1/4$. In the presence of cache interference, the number of cache misses is $2m^2\rho\frac{(L-\Delta)}{L}$.[5]

The main idea of our approach is to decouple the estimate of cache misses from the loop iteration space so that the approach can be fast even for large loop nests. Our approach combines a static symbolic analysis with an efficient and practical implementation. We use SUIF 1.3 and the framework developed by Ghosh et al. for the determination of eligible loop nests, memory references, reuse vectors, and for the manipulation of CMEs. We use *Polylib* for the estimation of the total number of iterations (e.g., m^2) and representation of parameterized polyhedra. We developed the software package STAMINA: It sorts the memory references as a function of their reuse vectors (i.e., temporal and spatial reuse, length); it determines their interference densities and it computes the total number of cache misses for each loop nest in an application. STAMINA annotates the original code with directives for the adaptation of the cache-line size for each eligible loop nest.

3 Notation and Interference Density

In this section, we introduce the notation and terminology used.

A **perfect loop nest** composed of d loops determines a set of integral points in $\mathbb{N}^d$. Each point is denoted by a column vector: $\mathbf{i} = (i_0, \ldots, i_{d-1})^t$; the first component (i.e., i_0) is associated with the outermost loop and the last component (i.e., i_{d-1}) is associated with the innermost loop. The loop order specifies a **lexicographic order** (as in [6]). In fact, a point $\mathbf{u}$ **precedes** a point $\mathbf{v}$, denoted by $\mathbf{u} \lhd \mathbf{v}$, if there exists an index t, $0 \le t \le d-1$, such that $u_n = v_n$ for every $n < t$ and $u_t < v_t$. When $\mathbf{v} = \mathbf{u}$ or $\mathbf{v} \lhd \mathbf{u}$, we use the notation $\mathbf{u} \trianglelefteq \mathbf{v}$. A **partial order**[6] between two points $\mathbf{v}$ and $\mathbf{u}$ is defined as follows: A point $\mathbf{u}$ is **smaller than** a point $\mathbf{v}$, denoted by $\mathbf{u} < \mathbf{v}$, when $v_n \le u_n$ for every index n, $0 \le n \le d-1$, except at least one index k such that $u_k < v_k$. For example, a point $\mathbf{u}$ determines a unique bounded polyhedron: $P = \{\mathbf{v} | \mathbf{0} \le \mathbf{v} \le \mathbf{u}\}$. Note that, if $\mathbf{v} < \mathbf{u}$, then $\mathbf{v} \lhd \mathbf{u}$, but not vice versa. (For example, $(1,1) < (2,2)$ and $(1,1) \lhd (2,2)$, and $(1,2) \lhd (2,1)$, but $(1,2) < (2,1)$ is not defined!)

We define an **iteration space** as a bounded polyhedron: $S_\mathbf{p} = \{\mathbf{i} | \mathbf{0} \trianglelefteq \mathbf{i} \trianglelefteq \mathbf{n}\}$, where $\mathbf{n}$ is $\mathbf{Ai} + \mathbf{Bk} + \mathbf{Cp}$ with $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ being constant matrices of size $d \times d$, $\mathbf{k}$ is a vector of constants, and $\mathbf{p}$ is a vector of parameters. The parameter $\mathbf{p}$ does not affect the shape of the iteration space, but only its cardinality. For example, consider $S_\mathbf{p}$ defined as $\{\mathbf{i} | 0 \trianglelefteq \mathbf{i} \lhd (p,p)^t, \}$. The iteration space $S_\mathbf{p}$ has cardinality $|S_\mathbf{p}| = p^2$, which is a function of p, and it has a square shape in $\mathbb{N}^2$ independently of any value of $\mathbf{p}$.

An **interval** is a set $P^\mathbf{r}(\mathbf{s}) = \{\mathbf{v} \in S_\mathbf{p} | \mathbf{s} - \mathbf{r} \lhd \mathbf{v} \trianglelefteq \mathbf{s}\}$, where $\mathbf{s}, \mathbf{s} - \mathbf{r} \in S_\mathbf{p}$ and $\mathbf{0} \trianglelefteq \mathbf{r}$. The cardinality of an interval is a function of $\mathbf{s}$. The cardinality of an interval represents the number of iterations separating the iteration point $\mathbf{s} - \mathbf{r}$ and the iteration point $\mathbf{s}$. In short, we specify distance as the cardinality of an interval, $|P^\mathbf{r}(\mathbf{s})|$. When $\mathbf{r} = \mathbf{e}_{d-1} \equiv (0, \ldots, 0, 1)$,[7] we have $|P^\mathbf{r}(\mathbf{s})| \le 1$. An interval, as well as an iteration space, is the composition of disjoint elementary rectilinear polyhedra. Note that these polyhedra can be just one point, where the vertices merge into one. This property assures that the determination of the distance of any interval is computable and that an Ehrhart polynomial exists [12], [13], [14], [15].

A reference R in the body of a loop nest has **temporal reuse** if, in different iterations $\mathbf{u}$ and $\mathbf{v}$, the reference accesses the same memory location $A_d[R(\mathbf{u})] = A_d[R(\mathbf{v})]$ [16]. We represent reuse by a vector $\mathbf{r}$ such that we have $A_d[R(\mathbf{u})] = A_d[R(\mathbf{u} + \mathbf{r})]$ for every iteration point $\mathbf{u}$. When the address of a reference is an affine function, that is, $A_d[R(\mathbf{u})] = \mathbf{l}^t(\mathbf{Mu} + \mathbf{b})$, the reuse vector is a point in the **null space** of matrix $\mathbf{M}$—i.e., $\mathbf{Mr} = \mathbf{0}$.[8] A reference has **spatial reuse** if the reference accesses—in different iterations—the same cache line. Note that temporal reuse is a particular case of spatial reuse. We have **group temporal** and **group spatial reuse** when different references exploit temporal and spatial locality among each other—during the computation.

For example, consider a matrix $A[100][100]$ stored in row-major format and starting at address 0x0. Consider a reference $R_A = A[u_0 + u_1][u_1]$ in a loop nest composed of two loops. We have

$$A_d[R_A(\mathbf{u})] = (100, 1)^t \left(\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \mathbf{u} + (0,0)^t \right).$$

In practice, R_A has spatial reuse and reuse vector $(0,1)^t$, but it has no temporal reuse because

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \mathbf{u} = \mathbf{0}$$

only when $\mathbf{u} = \mathbf{0}$ ($\mathbf{null}(A) = \emptyset$).

While we carry on the computation of the loop nest and one reuse of a reference is accomplished, we have a hit in

3. Polylib achieves an equivalent result.
4. Profiling approaches use a black-box approach about the application, therefore they should test all possible values of *start* just to be confident of the performance measurements
5. Polylib achieves an equivalent result, but it has to determine the solution of the equation for 512 different values of *start* and then it has to solve a system of 512 unknowns. This is a limitation of Ehrahrt's polynomial approach, rather than a Polylib limitation.
6. Also known as geometrical order, it does not always define an order between two iteration points.
7. The vector $\mathbf{e}_i$, the ith column vector of the identity matrix $I \in \mathbb{N}^d$.
8. Note that linear parameters do not affect the null space.

cache because the same reference is reused successfully. Otherwise, the memory reference may have been evicted from the cache and a miss may happen. The reuse $\mathbf{r}$ of a reference $R_A(\mathbf{u})$ is **prevented** when either a reference $R_B(\mathbf{s})$ with $\mathbf{s} \in P^r(\mathbf{u})$ interferes with the reference $R_A(\mathbf{u})$ or the iteration $\mathbf{u} - \mathbf{r}$ does not belong to the space.9

In general, the prevention of a reuse by another memory reference does not mean that we have a miss in cache. A reference may have multiple reuse vectors and, to have a miss in cache, all reuses must be prevented. We model the prevention of a reuse by an interference equation as follows.

Given two array references R_A—*interferer*—and R_B—*interferee*, we define an **interference equation** as:

$$E_{\mathbf{r}} \equiv \begin{cases} \mathbf{a}^t\mathbf{u} + a_{-1} = \mathbf{b}^t\mathbf{s} + b_{-1} + nC + q + \mathbf{d}^t\mathbf{p} \\ \\ \text{with } \mathbf{u} \in P^{\mathbf{r}}(\mathbf{s}), \mathbf{s} \in S_{\mathbf{p}}, n \neq 0, |q| < L, \\ \text{and with } L \text{ cache-line size,} \end{cases} \tag{1}$$

where $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{d}$ are constant vectors; the affine function for R_A is $\mathbf{a}^t\mathbf{i} + a_{-1}$ and the affine function for R_B is $\mathbf{b}^t\mathbf{i} + b_{-1}$; the parameter vector is $\mathbf{p}$ and the reuse vector for R_B is $\mathbf{r}$; the cache size is a constant C; the free variable n is not zero; the offset in the cache is $|q| \leq L - 1$; the cache-line size is L. The set of constraints is defined as **definition domain**.

An interference equation is always represented by an equality constraint—Diophantine equation—and by a definition domain in which the unknowns are defined [6], [17]. For example, in Fig. 1, the interference equation for interferer A and interferee B is as follows:

$$E_1 \equiv \begin{cases} b_{-1} + (1024, 8)(s_0, s_1)^t + nC + q + (0, 8)(m, start)^t \\ = a_{-1} + (1,024, 8)(u_0, u_1)^t \\ \\ \text{with } \mathbf{u} = \mathbf{s}, \mathbf{s} \in S_{\mathbf{p}}, n \neq 0, |q| < L, \\ \text{and with } L \text{ cache-line size.} \end{cases} \tag{2}$$

We model a direct-mapped cache, so, when the interference equation has a solution, we have cache interference and, thus, we have cache misses. Otherwise, if the equation has no solution and the interferee has only one reuse vector, then we have a hit.

When $|P^{\mathbf{r}}(\mathbf{s})| = 1$, we simplify (1). When $\mathbf{r} = \mathbf{e}_{d-1} \equiv (0, \dots, 1)$, $\mathbf{u} = \mathbf{s} - \mathbf{e}_{d-1}$ ($\mathbf{u} = \mathbf{s}$ when $\mathbf{r} = \mathbf{0}$), we isolate the term $nC + q$ as follows:10

$$E_{\mathbf{e}_{d-1}} \equiv \begin{cases} c_{-1} + \mathbf{c}^t\mathbf{s} = nC + q \\ \\ \text{with } \mathbf{c} = \mathbf{a} - \mathbf{b}, \mathbf{s} \in S_{\mathbf{p}} \\ \text{and } c_{-1} = a_{-1} + a_{d-1} - b_{-1} - \mathbf{d}^t\mathbf{p}. \end{cases} \tag{3}$$

For example, we simplify (2) as follows:

$$E_1 \equiv \{ -168,384,000 - 8start = nC + q. \tag{4}$$

We define the **interference density**, denoted by ρ_E, as the ratio of the number of points in the iteration space, for which the equation E has solution, over the total number of iteration points. For example, in (4), ρ_{E_1} is 1 (if $8start < L$).

9. Spatial reuse is prevented when a different line is accessed.

10. Note that we do not repeat the definition of the domains for the unknowns n and q.

Property 1. *If, in (3),* $E_{\mathbf{e}_{d-1}}$*, a solution exists and* $\mathbf{c} = C\mathbf{m}$*, then* $\rho_{E_{\mathbf{e}_{d-1}}} = 1$.

Proof. Because a solution exists in (3), a point $\mathbf{v}$, an integer n_0, and an integer q_0 exist for which $c_{-1} + \mathbf{c}^t\mathbf{v} = n_0C + q_0$. We substitute $\mathbf{c}$ with $C\mathbf{m}$ to obtain $c_{-1} + C\mathbf{m}^t\mathbf{v} = n_0C + q_0$. For any point $\mathbf{s} \in S_{\mathbf{p}}$, we find an integer g such that $c_{-1} + C\mathbf{m}^t\mathbf{s} = gC + q_0$ (e.g., $g = n_0 + \mathbf{m}^t(\mathbf{v} - \mathbf{s})$). Therefore, a solution exists and $\rho_{E_{\mathbf{e}_{d-1}}} = 1$. □

We can simplify (3) further because the element c_k, which is a multiple of the cache size (i.e., $c_k \textbf{ mod } C = 0$), does not contribute to the interference density:

$$E_{mod} \equiv \begin{cases} f_{-1} + \mathbf{f}^t\mathbf{s} = nC + q \\ \\ \text{where } f_k = c_k \textbf{ mod } C, \forall k \in [-1, d-1] \\ \text{and } \mathbf{s} \in S_{\mathbf{p}}. \end{cases} \tag{5}$$

In practice, $\rho_{E_{\mathbf{e}_{d-1}}}$ for (3) is equal to $\rho_{E_{mod}}$ for (5).

Property 2. *For the general case in (1), we have* $\rho_{E_{\mathbf{r}}} < \min(1, \max_{\mathbf{s} \in S_{\mathbf{p}}, \mathbf{p}} \rho_{E_{mod}} * |P^{\mathbf{r}}(\mathbf{s})|)$.

Proof. For every $\mathbf{s} \in S_{\mathbf{p}}$, we break the interval $P^{\mathbf{r}}(\mathbf{s})$ in smaller intervals with unit distance. We have up to $\max_{\mathbf{s} \in S_{\mathbf{p}}} |P^{\mathbf{r}}(\mathbf{s})|$ unit intervals. We consider each interval independently and we determine its interference density. Every interval has interference density, $\max_{\mathbf{s},\mathbf{p}} \rho_{E_{mod}}$. □

Property 2 states that we can determine the interference density for a rather complex interval using an estimate based on unit intervals. We shall present in Section 4.3 a technique that estimates $\rho_{E_{mod}}$ and it is independent of any parameter $\mathbf{p}$ and any iteration point in the iteration space $\mathbf{s}$. Furthermore, when the reuse vectors are short, the reuse intervals have short distance and, therefore, we have a simple and tight estimation. McKinley and Temam present strong evidence that short reuse are common in scientific computations [10]. We assume that the target of our analysis are applications with short reuse vectors—mostly spatial reuse.

4 Parameterized Loop Analysis

In this section, we introduce our approach in a top-down fashion describing the organization of our software package STAMINA, Fig. 2. In Section 4.1, we introduce the trade off between spatial reuse and cache interference and we propose our model for the representation of cache misses as a function of both cache-line size and interference density. In Section 4.2, we present how we model interference as set of interference equations. In Section 4.3, we discuss the computation of the interference density based on a simplified analysis of the interference equations. In Section 4.3.1, we introduce a more accurate analysis of the interference density based on the theory of affine equations using unimodular transformations [17].

4.1 Spatial Reuse versus Interference: Optimal Cache-Line Size

Ideally, without cache interference, an application having spatial locality is able to exploit a large cache-line size by

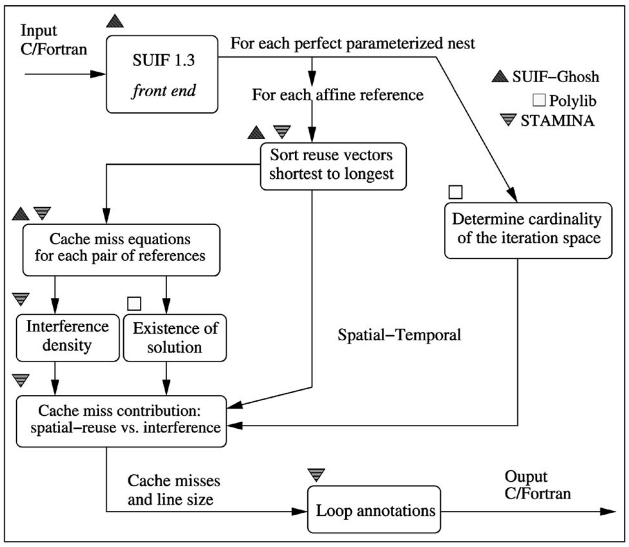

Input C/Fortran
SUIF 1.3 front end
For each perfect parameterized nest
For each affine reference
SUIF–Ghosh
Polylib
STAMINA
Sort reuse vectors shortest to longest
Determine cardinality of the iteration space
Cache miss equations for each pair of references
Interference density
Existence of solution
Spatial–Temporal
Cache miss contribution: spatial–reuse vs. interference
Cache misses and line size
Loop annotations
Ouput C/Fortran

Fig. 2. STAMINA.

reducing cache misses by virtue of fewer memory accesses. However, a large cache-line size may increase interference, which may impede the spatial locality exploitation and, in the worst case, it may increase cache misses. For some applications, we find it acceptable to have an increase of cache misses due to interference as long as the overall performance improves due to fewer communications to and from the cache.

For one memory reference R, we estimate the total contribution to the interference density, $\eta_R(L)$, by distinguishing three different cases and considering two contributions.

$$\eta_R(L) = \begin{cases} \min(1, \frac{s}{\ell} + \mu_R(L)) & \text{if } \mu_R(L) < 1 \text{ and} \\ & \text{spatial\&temporal reuse,} \\ \min(1, \mu_R(L)) & \text{if temporal reuse only,} \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

We define $\eta_R(L)$ in (6) as the **spatial-temporal interference density per memory reference**.

The memory reference can have **spatial** and **temporal** reuse, that is, a reference has reuse of the same cache line and reuse of the same element located in a cache line. If a reference R has spatial reuse and there is no interference, we estimate a miss every $\frac{\ell}{s}$ access(es)—i.e., iteration(s). The interference density is $\frac{s}{\ell}$, where ℓ is the line size in data elements (i.e., $\ell = L/8$ when an element is a double) and s is the length of the spatial reuse in elements. This contribution to the interference density is due to the spatial reuse only and notice that it is a monotonically decreasing function in L. Spatial reuse is an artificial reuse, which is introduced by the memory architecture configuration. A spatial reuse may be prevented because of the access of a different cache line and not because of interference. In fact, any other (longer) reuse may be satisfied and, instead of a cache miss, we could achieve a cache hit. For spatial reuse, it would be convenient to consider the effect of longer (temporal) reuse as well (see Ghosh et al. [18]).

If there is interference, part of the reuse can be prevented and we can have a larger contribution to the interference density. The factor $\mu_R(L) \in [0,1]$ is the estimate of interference density due to cache interference only, **interference density per memory reference**, that is, how other references displace reference R from the cache. When $\mu_R(L) = 1$, it means that interference is so high that no reuse is possible. The factor $\mu_R(L)$ is a monotonically increasing function. We shall see how to determine $\mu_R(L)$ in Section 4.2.

If a reference R does not have reuse of any kind, then $\eta_R(L) = 0$. If there is no reuse, there is no interference. If there is a **cold miss**,[11] it is unavoidable in this framework for every cache-line size.

Finally, the estimate of the number of cache misses, due to one memory reference, is simply $|S_\mathbf{p}|\eta_R(L)$ (i.e., $|S_\mathbf{p}|$ is the number of iterations in the loop nest). We explain shortly how we use $\eta_R(L)$ to estimate the number of cache misses as a function of the cache-line size. Suppose we have z memory references in a loop nest. We sort the references and we label them by using a unique integer according to the following criterion: Reference R_i, with $0 \leq i < x$, has spatial reuse and reference R_j, with $x \leq j < z$, has temporal reuse. An upper bound on the number of cache misses is given in (7).

$$\begin{aligned} |Misses| &\leq |S_\mathbf{p}|\epsilon(L) \\ |S_\mathbf{p}|\epsilon(L) &= |S_\mathbf{p}| \sum_{i=0}^{z-1} \eta_{R_i}(L) \\ &= |S_\mathbf{p}| \left(\sum_{i=0}^{n-1} \eta_{R_i}(L) + \sum_{i=n}^{m-1} \mu_{R_i}(L) \right). \end{aligned} \tag{7}$$

Because the function $|S_\mathbf{p}|$ is independent of the cache-line size, the minimum number of cache misses is a function of $\epsilon(L) = \sum_{i=0}^{n-1} \eta_{R_i}(L)$. In practice, we seek the optimal cache-line size that minimizes $\epsilon(L)$ and we do it by a linear search for increasing values of L (i.e., $L = 8, 16, 32, 64, 128, 256$ bytes).

4.2 Interference Density per Memory Reference

In this section, we introduce two important concepts and estimates: the interference existence and the interference density per memory reference (i.e., $\chi_E(L)$ and $\mu(L)$, respectively).

Consider an interference equation E_{mod}—as in (5). We define **interference existence** as a 0-1 function expressing whether or not the equation E_{mod} has integer solutions:

$$\chi_{E_{mod}}(L) = \begin{cases} 1 & \text{if } E_{mod} \text{ has a solution} \\ 0 & \text{otherwise,} \end{cases} \tag{8}$$

where L is the cache-line size.

A CME solver, as it counts the number of integer solutions of an interference equation, resolves the existence problem as well. However, a solver may be designed for the existence problem only; in fact, Omega test is an example of such a solver [19], [20]. Note that, in the worst-case scenario, searching for one solution is as hard as counting all integer solutions.

Currently, we deploy Polylib, which applies a linear search in parameterized polyhedra to find whether or not an integer solution exists. In Sections 5.3 and 5.2, we present an example showing the way we use the interference

11. The first time a reference is read, we have a cache miss and it is defined as cold miss.

existence to achieve an accurate estimate of the number of cache misses.

In the following, we present our approach for the determination of the interference density per memory reference. We outline the approach, describing the following three possible scenarios:

1. Consider a memory reference R_A with one reuse vector $\mathbf{r}$ and with k interferers R_{B_i}, $0 \leq i < k$. For each pair of references R_A and R_{B_i}, we determine the interference equation E_i, we estimate the interference density, and we compute the interference existence (i.e., ρ_{E_i} and $\chi_{E_i}(L)$). Then, we determine the contribution of each interferer R_{B_i} independently and we add their contributions:
$$\mu(L) = \sum_{i=0}^{k} \rho_{E_i} \chi_{E_i}(L). \tag{9}$$
2. Consider a memory reference, R_A, and one interferer, R_B. The reference R_A has m reuse vectors $\{\mathbf{r}_i\}_{0,m-1}$ such that $\mathbf{r}_{m-1} \lhd \ldots \lhd \mathbf{r}_0$. Every reuse vector $\mathbf{r}_i$ is associated with an interval $P^{\mathbf{r}_i}(\mathbf{s})$ and, for every $i > j$, we have $P^{\mathbf{r}_i}(\mathbf{s}) \subset P^{\mathbf{r}_j}(\mathbf{s})$. In particular, we have that $\cap_{i=0}^{m} P^{\mathbf{r}_i}(\mathbf{s}) = P^{\mathbf{r}_{m-1}}(\mathbf{s})$. We consider the shortest reuse vector only (i.e., $\mathbf{r}_{m-1}$) and, therefore, we consider the shortest interval only (e.g., $P^{\mathbf{r}_{m-1}}(\mathbf{s})$) because, if the shortest reuse is prevented, all reuses are prevented and there is a cache miss; otherwise, the shortest reuse is exploited and there is no miss in the cache—however, other reuse may be prevented. This is equivalent to the first case with one interferer: $\mu(L)$ is $\rho_E \chi_E(L)$ (e.g., in (9) with $k = 1$).
3. Consider a reference R_A with k interferers R_{B_i} and m reuse vectors $\{\mathbf{r}_i\}_{0,m-1}$ such that $\mathbf{r}_{m-1} \lhd \ldots \lhd \mathbf{r}_0$. Every reuse vector $\mathbf{r}_i$ is associated with an interval $P^{r_i}(\mathbf{s})$ and, for every $i > j$, we have $P^{\mathbf{r}_i}(\mathbf{s}) \subset P^{\mathbf{r}_j}(\mathbf{s})$. In particular, we have that $\cap_{i=0}^{m} P^{\mathbf{r}_i}(\mathbf{s}) = P^{\mathbf{r}_{m-1}}(\mathbf{s})$.

   For each pair of references R_A and R_{B_i}, we determine the interference equation E_i for the shortest reuse only, therefore for the shortest interval (e.g., $P^{\mathbf{r}_{m-1}}(\mathbf{s})$). If the shortest reuse is prevented, all reuses are prevented and there is a cache miss; otherwise, the shortest reuse is exploited and there is no miss in cache. In fact, in (9), we model this case as well.

The number of cache misses for a direct mapped cache is up to $|S_\mathbf{p}|\mu(L)$. For a k-way associative cache, we may estimate the number of cache misses as $|S_\mathbf{p}|\lfloor\frac{\mu(L)}{k}\rfloor$. In practice, our estimate/approach is independent of the approach proposed by Chatterjee et al. [21] for associative caches. However, there are three common features we summarize as follows: First, both approaches model cache misses using polyhedra, thus they do not convey any information on the temporal distribution of the cache interference; second, both are approximations—not upper bounds; third, we may use these estimations to determine statically the minimum associativity that circumvents cache interference altogether.

The interference equations model cache interference in an interval. This interval must be a valid interval in the iteration space. Otherwise, no analysis is performed. For example, given a reuse vector $\mathbf{r}$, our approach does not analyze the set of iterations:
$$P_B^{\mathbf{r}} = \{\mathbf{j} | \mathbf{j} \in S_\mathbf{p} \cap (\mathbf{j} - \mathbf{r}) \notin S_\mathbf{p}\}. \tag{10}$$
We can rewrite (10) as the union of nonintersecting elementary rectangular sets, therefore we may use Polylib to compute its cardinality. If we count the number of iterations in this set, we determine a **confidence index**, which is used separately to assess whether the analysis has any contribution. In fact, the smaller the reuse vector is, the larger the iteration space investigated by our approach is, therefore the larger the number of cache misses we can determine through the interference density. For the examples we present in Section 5, we analyze 99.9 percent of the iterations.

4.3 Interference Density Analysis, STAMINA

In this section, we describe our approach to determine the interference density only from the equality of an interference equation—as in (5)—that we repeat here:
$$E_{mod} \equiv \begin{cases} f_{-1} + \mathbf{f}^t\mathbf{s} = nC + q \\ \\ \text{where } f_k = c_k \ \mathbf{mod}\ C \ \forall k \in [-1, d-1], \\ c_k = a_k - a_k, d_{-1} = a_{-1} - b_{-1} - \mathbf{d}^t\mathbf{p}, \\ \text{and with } \mathbf{s} \in S_\mathbf{p} \end{cases} \tag{11}$$
(i.e., we consider $f_{-1} + \mathbf{f}^t\mathbf{s} = nC + q$ only).

Theorem 4.1 states the main result of this section—the interference density is simply a function of the cache size and cache-line size: $\rho_E \leq \frac{1}{2^{d-1}}\frac{2L}{C}$. To prove this result, we start by showing that the **solutions space**—e.g., the iteration points where a interference equation has a solution—is a regular structure in a rational domain. This structure envelopes all integer solutions and it has an extremely regular organization in cells—or tiles. We determine the interference density by computing the ratio of volumes, that is, we determine the *volume* of the solutions over the *volume* of a solution cell.

We begin with the definition of inner product and inverse vector. The **inner product** of two vectors $\mathbf{u}$ and $\mathbf{v}$ is the vector $\mathbf{s} = \mathbf{u} \cdot \mathbf{v}$ such that $s_k = u_k v_k$. The vector $\mathbf{1} = (1, \cdots, 1)^t$ is the identity vector for the inner product (i.e., $\mathbf{v} \cdot \mathbf{1} = \mathbf{1} \cdot \mathbf{v} = \mathbf{v}$). For every nonzero rational vector $\mathbf{v} \in \mathbb{Q}^d$— i.e., $v_k \neq 0$—there is one and only one inverse vector, denoted by $\mathbf{v}^{-1} \in \mathbb{Q}^d$, such that $\mathbf{v}^{-1} \cdot \mathbf{v} = \mathbf{1}$.

From here on, we denote by $\mathbf{i}_0$ the smallest rational solution for equation E_{mod}. We now describe a regular structure that models the solution space. We define a **grid** as a set of points:
$$\mathcal{G}(\mathbf{i}_0) = \{\mathbf{j} | \mathbf{j} = \mathbf{i}_0 + C\mathbf{f}^{-1} \cdot \mathbf{s}, \text{such that } \mathbf{s} \in \mathbb{N}^d\}. \tag{12}$$

We define a **grid cell** as a d-dimensional rectangle determined by the $2 + d$ vertices $\mathbf{i}_0 + C\mathbf{f}^{-1} \cdot \mathbf{s}$,
$$\mathbf{i}_0 + C\mathbf{f}^{-1} \cdot (\mathbf{s} + \mathbf{e}_0), \ldots, \mathbf{i}_0 + C\mathbf{f}^{-1} \cdot (\mathbf{s} + \mathbf{e}_{d-1}),$$
and

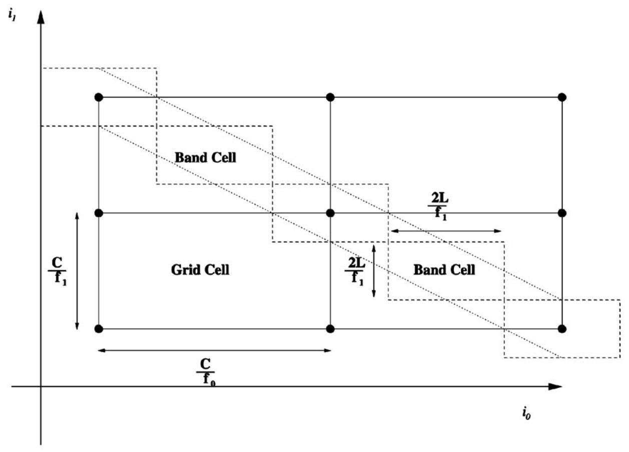


Fig. 3. Grid cells and band cells in a plane. In a two-dimensional space, grid cells and band cells are rectangles. Note that three bands pass through a grid cell.

$$\mathbf{i}_0 + C\mathbf{f}^{-1} \cdot \left(\mathbf{s} + \sum_{j=0}^{d-1} \mathbf{e}_j\right),$$

for any $\mathbf{s} \in \mathcal{G}(\mathbf{i}_0)$. Given an integer u, we define a **band** as the following set of rational points:

$$\mathcal{B}(u) \equiv \{\mathbf{b} | -L < u + \mathbf{f}^t\mathbf{b} < L \text{ with } \mathbf{b} \in \mathbb{Q}^d \text{ and } u \in \mathbb{N}\}. \tag{13}$$

We define a **band cell** as the set of rational points:

$$\mathcal{BC}(u) \equiv \{\mathbf{b}|(-L = u + \mathbf{f}^t\mathbf{b} \cup L = u + \mathbf{f}^t\mathbf{b}) \cap \cap (\forall k \neq j, b_k = 0 \text{ and } j \in [0, d-1])\}. \tag{14}$$

For every grid point in $\mathcal{G}(\mathbf{i}_0)$, we determine a band, that is, $\mathcal{B}(n_0C + f_{-1})$. Every point in the band is a solution and, in particular, it has the same value for the variable n. The grid and the bands represent a regular structure (see Fig. 3, for a two-dimensional example). We use the band cells to express the volume of the bands, therefore of the solution number in a grid cell; we eventually determine the interference density for a single grid cell as representative for the entire space determining their volume ratio.

Considering an example as in Fig. 3, a grid cell 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. In a two-dimensional space, the grid cell has an area, in a three-dimensional space, it has a volume. In general, we use the term **volume** to indicate the same *quantity-concept* for any dimension.[12]

Property 3. *Every grid cell has volume* $\frac{C^d}{\prod_{k=0}^{d-1} f_k}$.

We now determine how many bands cross a grid cell and then we determine their volumes. A band is determined by two $(d-1)$-dimensional spaces and, by construction, it passes through grid points. For any grid cell, there is only one band splitting the cell in two so that two vertices are apart. We have three bands crossing a grid cell.

12. We avoid the use of the term *space* because we use it in another context, that is, iteration space.

In the following property, we state how many band cells we may find in a grid cell; therefore, we have an estimate of the volume of a band intersecting a grid cell.

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

Proof. Consider a grid cell with size $\frac{C}{f_k}$, $0 \leq k < d$, in a d-dimensional space (i.e., in a three-dimensional space, it is a cube). The projection of a band on any $(d-1)$-dimensional space has a number of band cells as $\frac{1}{2^{d-1}} \prod_{k \neq j} \frac{C}{f_k} / \frac{2L}{f_k}$ (i.e., in a three-dimensional space, we have three projections on three planes; on each plane, the band cell projections are $\frac{1}{2}(\frac{C}{f_k}/\frac{2L}{f_k}) * (\frac{C}{f_j}/\frac{2L}{f_j})$ with $j \neq k$). □

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

When we have an estimate of the volume of a band cell and we have the number of bands cells, we have an estimate of the volume of a band. The last step is to show that this regular structure, made of a grid and bands, is dense as it contains all integer solutions.

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

Proof. By definition, $Cn_0 + q_0 = \mathbf{f}^t\mathbf{i}_0$ and $Cn_1 + q_1 = \mathbf{f}^t\mathbf{z}$, without loss of generality consider $n_1 > n_0$. A band is a space for which each rational point is a solution for the equation with same value of n, we prove the lemma as soon as we show that $\mathbf{p}$ exists so that $Cn_2 + q_2 = \mathbf{f}^t(\mathbf{i}_0 + C\mathbf{f}^{-1} \cdot \mathbf{p})$ and $n_2 = n_1$.

We have $Cn_2 + q_2 = \mathbf{f}^t\mathbf{i}_0 + \mathbf{f}^tC\mathbf{f}^{-1} \cdot \mathbf{p}$, that is, $Cn_2 + q_2 = \mathbf{f}^t\mathbf{i}_0 + C\mathbf{1}^t\mathbf{p}$. We determine n_2:

$$n_2 = \left\lfloor \frac{\mathbf{f}^t\mathbf{i}_0 + C\mathbf{1}^t\mathbf{p}}{C} \right\rfloor.$$

We obtain

$$n_2 = \left\lfloor \frac{\mathbf{f}^t\mathbf{i}_0}{C + \mathbf{1}^t\mathbf{p}} \right\rfloor = \left\lfloor \frac{\mathbf{f}^t\mathbf{i}_0}{C} \right\rfloor + \mathbf{1}^t\mathbf{p} = n_0 + \mathbf{1}^t\mathbf{p}.$$

So, $n_2 = n_1$ when $\mathbf{1}^t\mathbf{p} = n_1 - n_2$. There is always such a vector $\mathbf{p}$. □

Finally, we state and prove our estimate for the interference density.

Theorem 4.1. *If, in (5),* E_{mod}*, a solution exists and* f_k **mod** $C \neq 0$, $\forall k \in [0, d-1]$ *and* $C \geq 2L$*, then* $\rho_E \leq \frac{1}{2^{d-1}} \frac{2L}{C}$.

Proof. By Lemma 1, the grid and the bands on the grid constitute a dense solution space. Every integer solution is in it. The density is computed on a grid cell as the ratio of the volume of a band intersecting a cell over the volume of a grid cell. By Properties 4 and 5, there are $\frac{1}{2^{d-1}}\left(\frac{C}{2L}\right)^{d-1}$ band cells of volume $\frac{(2L)^d}{\prod_{k=0}^{d-1} f_k}$ in a grid cell. By Property 3, a grid cell has volume $\frac{C^d}{\prod_{k=0}^{d-1} f_k}$. Then, we have $\rho_E \leq \frac{1}{2^{d-1}} \frac{2L}{C}$. □

4.3.1 *Interference Density Analysis, Refined*

In this section, we present a more detailed analysis of the interference density; we refine the analysis for the integer domain. First, we present some considerations on the approach to solve Diophantine equations [17]. Second, we apply this approach when both the variable n and l are assigned to values, therefore they are constant terms of the equation. We then consider the case when only the variable l is assigned to a value. At last, we present the final result of this section in Theorem 4.4.

Consider the coefficients in (5). In fact, E_{mod} is a Diophantine equation. For the solution of Diophantine equations, we may use the **GCD test** [17]. We determine the great common divisor of the coefficients of the equations —$g = gcd(d_{-1}, d_0, \ldots, d_{d-1}, C, 1)$—and we verify whether the constant factor of the equation, for example ζ, is evenly divided by g. If $\zeta \textbf{ mod } g = 0$, the equation may have a solution—we need to check the domain; otherwise, the equation has no solution.

Because the free variable l has 1 as coefficient, the $gcd(d_{-1}, d_0, \ldots, d_{d-1}, C, 1) = 1$, the **GCD test** is inconclusive: we cannot conclude whether or not there is any solution. We need to solve the system and verify the constraints on the definition domain.

Banerjee presents a general approach to determine all integer solutions for Diophantine equations—without parameters, using the theory of unimodular matrices. We consider whether or not there is solution for arbitrary values of $n = n_0$ and $l = l_0$. The constant value will be $\zeta = f_{-1} + n_0 C + l_0$, which includes the parameters as well. We can rewrite (5) as follows:

$$E_b \equiv \{ \zeta = \mathbf{f}^t\mathbf{s}. \tag{15}$$

If g is $gcd_{k\in[0,d-1]}(f_k)$ and $gcd(g, \zeta)$ is not 1, then there exists a unimodular matrix $\mathbf{U}$ so that all the solutions are determined by the following expression:

$$E_{sol} \equiv \begin{cases} \mathbf{i} = \mathbf{U}^t\mathbf{s} \\ \\ \text{where } \mathbf{s} = (\zeta/g, s_1, s_2, \ldots, s_{d-1}) \\ \text{and } s_k \in \mathbb{N}, \forall k \in [1, d-1] \\ \text{and where } \mathbf{U} \in \mathbb{R}^{d\times d} \text{ is unimodular} \\ \text{and } \mathbf{U}\mathbf{f} = (g, 0, \ldots, 0)^t. \end{cases} \tag{16}$$

A matrix $\mathbf{U}$ is **unimodular** when it is an upper triangular matrix and it has $|det(\mathbf{U})| = 1$.[13] A unimodular matrix $\mathbf{U}$ is a linear transformation, it always has an inverse matrix $\mathbf{U}^{-1}$ (i.e., such that $\mathbf{U}^{-1}\mathbf{U} = \mathbf{I}$, $\mathbf{I}$ the identity matrix), and $\mathbf{U}^{-1}$ is unimodular as well. Banerjee presents an effective technique in Algorithm 2.1 for the determination of $\mathbf{U}$.

The matrix $\mathbf{U}$ is a 1-1 mapping between the iteration space S and a space $\mathbb{T}$, where $S, \mathbb{T} \subset \mathbb{N}^d$. In $\mathbb{T}$, the solution space is the plane $s_0 = \zeta/g$; the number of integer solutions in $\mathbb{T}$ is as many as in S and all solutions in S are in a plane.

For example, if $\mathbb{T} = \mathbb{N}^2$, the solution space is a line $\mathbf{s} = (\zeta/g, s_0)^t$ and the minimum distance between any two points in $\mathbb{T}$ is 1, that is, $(\zeta/g, 1) - (\zeta/g, 0) = (0, 1)$. Consider the equation $6i_0 - 4i_1 = 10$.[14] The solutions are:

$$\mathbf{i}^t = (5, s_0)\begin{bmatrix} 1 & 1 \\ 2 & 3 \end{bmatrix}. \tag{17}$$

Two solutions in $\mathbb{T}$, such as $(5, 0)$ and $(5, 1)$, are mapped on two solutions in S, $(5, 5)$ and $(7, 3)$, respectively. We notice that, in $\mathbb{T}$, the distance is 1, but, in S, the distance is more than 1. We want to determine the interference density in the original space—i.e., S—using some of the properties of matrix $\mathbf{U}$.[15] Indeed, we estimate the distance computing the volume of the d-dimensional rectangle that has the two solutions as opposite vertices. To do so, we estimate the size of the rectangle as follows:

We define

$$h_i = \left| \max_{j\in[1,d-1]} u_{j,i} - \min_{j\in[1,d-1]} u_{j,i} \right|, \tag{18}$$

where $u_{j,i}$ is the element in the ith column and jth row in $\mathbf{U}$. Intuitively, the product $\prod_{i=0}^{d-1} h_i$ is a lower bound to the distance between two solutions in S.

Theorem 4.2. *If equation $E_b \equiv \zeta = \mathbf{f}^t\mathbf{s}$ has a solution, then the interference density is at most $\rho_{E_b} \leq 1/(\prod_{i=0}^{d-1} h_i)$.*

Proof. Consider the solutions $\mathbf{s} + \mathbf{e}_1, \ldots, \mathbf{s} + \mathbf{e}_{d-1}$. These solutions are mapped to $\mathbf{U}^t\mathbf{s} + \mathbf{U}^t\mathbf{e}_1, \ldots, \mathbf{U}^t\mathbf{s} + \mathbf{U}^t\mathbf{e}_{d-1}$, that is, $\mathbf{U}^t\mathbf{s} + \mathbf{u}_1, \ldots, \mathbf{U}^t\mathbf{s} + \mathbf{u}_{d-1}$. The solutions are the $d-1$ vertices of a bounded region and $\prod_{i=0}^{d-1} h_i$ is a lower bound to the number of integer points in the region. □

When n is not an arbitrary value but it is a variable, the equation may have solutions for different values of n (i.e., the equation is $\zeta = Cn + \mathbf{f}^t\mathbf{s}$). For each solution of n, there is a different parallel plane in $\mathbb{T}$. As long as the planes are far apart, Theorem 4.2 holds. Otherwise, the interference density may be reevaluated as the following theorem states.

Theorem 4.3. *If an integer $j \in [0, d-1]$ exists such that $h_j > \frac{C}{f_j}$, then $\rho_{E_b^d} \leq \max_k \frac{2h_k f_k}{C} \rho_{E_b}$.*

Proof. For any n, the solution space is a set of parallel planes. We determine the image of the planes and of the set of points $\mathbf{s} + \mathbf{e}_1, \ldots, \mathbf{s} + \mathbf{e}_{d-1}$ in S. We note that, in one dimension, the distance between any two planes is (asymptotically) C/f_i and, in $\mathbf{U}^t\mathbf{s} + \mathbf{u}_1, \ldots, \mathbf{U}^t\mathbf{s} + \mathbf{u}_{d-1}$, there can be at most $\max_{k\in[0,d-1]} \frac{2h_k f_k}{C}$ planes intersecting. Each plane contributes just one integral solution. By Theorem 4.2, the proof follows. □

The last case is when, for every solution of n, there are different solutions of l. The following theorem estimates the interference density in this scenario.

Theorem 4.4. *If we have a set J so that $h_j < 2L$ with $j \in J$, then $\rho_E \leq ((\frac{2L}{C})^{|J|} \prod_{j\in J} \frac{1}{h_j}) * \rho_{E_b^{(k=d-|J|)}}$.*

Proof. In this case, each variable in J satisfies the equation in an interval of size C at most $2L/h_i$ (with $i \in J$) times, therefore, with density $\frac{2L}{Ch_i}$. We restrict the investigation on the other $d - |K|$ variables and we apply Theorem 4.3. □

13. Where $det(\mathbf{U})$ is the determinant of matrix $\mathbf{U}$.
14. Example 3.5 [17].
15. Because $\mathbf{U}$ is unimodular, we have that $|det(\mathbf{U})| = |det(\mathbf{U}^{-1})| = 1$ and, therefore, we cannot use the determinant to achieve any estimation for the interference density—at least directly.

```
#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
    }
}
```

Fig. 4. SWIM: calc1() in C code. We introduce comment lines above and below any instructions; those comment lines present STAMINA assumptions on the memory references. The comment above an instruction presents the reference numbers (RN) and the comment below an instruction presents the order in which the memory references are issued.

5 STAMINA Implementation Results

The reuse and interference analysis is implemented in the software package STAMINA (abbreviation for Static Modeling of Interference And reuse). It is built on top of an SUIF 1.3 compiler adapting the analysis developed by Ghosh et al. [6] and using *Polylib* [13], [12], [15]. In this section, we consider three cases to explore three important aspects of our analysis. We analyze loop nests presenting: first, parameterized loop bounds; second, only self-interference among memory accesses; and, last, parameterized loop bounds, parameterized memory accesses, and tiling.

STAMINA presents the result of the analysis in two forms (or types): a numeric and a symbolic form.

Numeric form: The output is a table with two contributions—two rows:

- A row is the contribution at **compile time**. It presents the estimation of interference as a function of the cache-line size, at compile time only. We identify the entries in such a row by $\epsilon_{ct}(L)$.
- A row is the contribution at **runtime**. It presents the estimation of the interference as a function of the numeric value of the parameters. We identify the entries in such a row by $\epsilon_{rt}(L)$.

This distinction between compile time and runtime is extremely helpful for an optimizing compiler: A compiler may use the quantitative measure and decide whether or not any adaptation is worth pursuing. In other words, if the contribution at runtime is overall negligible, we can set the optimal line size at compile time; otherwise, we may introduce annotations to the original code and drive adaptation at runtime.

Symbolic form: We represent the effect of the cache-line size by a symbolic function. We insert code computing the symbolic function as header of the loop nest and we evaluate it at runtime, before the loop nest execution.

TABLE 1
Simulation of the Data-Cache Misses Due to 10 Calls to *calc1()*; L = Cache-Line Size in Bytes, DCMR = Data Cache Miss Rate

L	32	64	128	256
DCMR	11.916 %	5.960%	2.982%	1.493%

Spatial locality is fully exploited in calc1().

We assume that the scheduling of the references (i.e., loads and stores) follows two criteria. First, the computation is performed so as to minimize the number of temporaries [22] for each statement. Second, a reference may be loaded once or more in the inner loop. We assume the final scheduling from the source code only because it is very difficult to retain high-level information from the source code to the assembly code and vice versa (e.g., after all optimizations such as scheduling and register allocation). For example, we label each reference with an integer and we assume a possible reference schedule. This schedule is automatically determined and it is used for the interference analysis. (Note, this is not a limitation of the approach but of the implementation.) We assume the data cache is direct mapped of size 16KB.

5.1 Case A: SWIM-SPEC 2000

The first application is `swim` from SPEC 2000. It has a main loop with four function calls. Each function has a loop nest for which the loop bounds are parameters introduced at runtime. We present results for two of these loop nests.

In Fig. 4, we present one of the loop nests in C language. We analyze the interference for two different matrix sizes, the **reference** size $1,335 \times 1,335$ and the power of two $1,024 \times 1,024$. Our analysis states that, for the former, there is no interference for any cache-line size, but, for the latter, there is interference among all references and all cache-line sizes.

Due to the number of equations, it is very difficult to verify the accuracy of the analysis by hand. We simulate 10 of the 800 calls to the $calc1$ routine using **cachesim5** from **Shade** [23]. The routine is compiled with $gcc/3.1$ with the $-O2$ flag on. The simulation results for matrix size $1,335 \times 1,335$ confirm our analysis as shown in Table 1.

The case for power of two matrices is confirmed as well, but not reported here.

A more interesting case is in procedure calc2(), see Fig. 5. STAMINA determines that reference 16, $CU(I+1,J)$, interferes with reference 8, $H(I+1,J)$, when the line size is larger than 128B. The software determines the trade off between spatial locality exploitation and interference, but, even though only two references are interfering, the optimal line size proposed is 128B. Note that the analysis is able to indicate which references are involved and when there is interference.

Using the Shade simulator, we validate our analysis as shown in Table 2.

The execution of SWIM with reference input (matrices of size $1,335 \times 1,335$) takes 1 hour on a Sun Ultra 5, 450MHz. Any full simulation takes at least 50 times more. In contrast, our analysis takes less than one minute for each loop nest, whether or not there is interference (i.e., for SWIM, our analysis takes less than 5 minutes).

```
      SUBROUTINE CALC2
C
C        COMPUTE NEW VALUES OF U,V,P
C
      IMPLICIT REAL*8   (A-H, O-Z)
      PARAMETER (N1=1335, N2=1335)

      COMMON  U(N1,N2), V(N1,N2), P(N1,N2),
     *        UNEW(N1,N2), VNEW(N1,N2),
     1        PNEW(N1,N2), UOLD(N1,N2),
     *        VOLD(N1,N2), POLD(N1,N2),
     2        CU(N1,N2), CV(N1,N2),
     *        Z(N1,N2), H(N1,N2), PSI(N1,N2)
C
      COMMON /CONS/ DT,TDT,DX,DY,A,ALPHA,ITMAX,MPRINT,M,N,MP1,
     1              NP1,EL,PI,TPI,DI,DJ,PCF
      TDTS8 = TDT/8.D0
      TDTSDX = TDT/DX
      TDTSDY = TDT/DY

C SPEC removed CCMIC$ DO GLOBAL
      DO 200 J=1,N
      DO 200 I=1,M
C           0          1
         UNEW(I+1,J) = UOLD(I+1,J)+
C                     2          3          4           5
     1        TDTS8*(Z(I+1,J+1)+Z(I+1,J))*(CV(I+1,J+1)+CV(I,J+1)
C              6       7                  8        9
     2        +CV(I,J)+CV(I+1,J))-TDTSDX*(H(I+1,J)-H(I,J))
C    #   2 3 4 5 6 7 8 9 1 0
C       10            11                  2         12
         VNEW(I,J+1) = VOLD(I,J+1)-TDTS8*(Z(I+1,J+1)+Z(I,J+1))
C              13          14        15      16
     1        *(CU(I+1,J+1)+CU(I,J+1)+CU(I,J)+CU(I+1,J))
C                      17        9
     2        -TDTSDY*(H(I,J+1)-H(I,J))
C    # 2 12 13 14 15 16 17 9 11 10
C        18          19                16        15
         PNEW(I,J) = POLD(I,J)-TDTSDX*(CU(I+1,J)-CU(I,J))
C                       5         6
     1        -TDTSDY*(CV(I,J+1)-CV(I,J))
C     #  16 15 5 6 19 18
 200  CONTINUE
      RETURN
      END
```

Fig. 5. SWIM: calc2() in Fortran. We introduce comment lines above and below any instructions, those comment lines present STAMINA assumptions on the memory references. The comment above an instruction presents the reference numbers (RN) and the comment below an instruction presents the order in which the memory references are issued.

5.2 Case B: Self-Interference

We now consider the case when an application has self-interference. Self-interference happens when two references of the same array, or the same reference in different iterations, interfere in cache. The example, Fig. 6, is the composition of six loops with only one memory reference in each.

Each memory reference has a different spatial reuse and the reuse vector is *long*. Each loop accesses a matrix by row and updates a small part of it. Even though the matrix access is done by row, instead of by column, spatial locality may be exploited because of the matrix size. In practice, the number of columns for each matrix is chosen so that each loop has a different optimal line size.

For example, in LOOP 0, a cache-line size of 8 Bytes does not have any self-interference and a cache-line size of 16B has spatial reuse; for larger line size, there is always interference because elements in two contiguous rows share the same cache line.

STAMINA recognizes that the spatial reuse goes across one iteration of the outermost loop. In the current implementation, it fixes the value of the interference density at $\rho = 1$ (STAMINA assumes that there is a capacity miss because, in general, the distance is not a constant and it cannot be compared to the cache size). For this particular case, we achieve a tight estimation. In general, we achieve an overestimation. Notice that the existence of interference plays the main role; it discriminates when there is interference and when to count the interferences. In Table 3, we report the results of the analysis.

TABLE 2
Simulation of the Data-Cache Misses Due to 10 Calls to *calc2()*; L = Cache-Line Size in Bytes, DCMR = Data Cache Miss Rate

L	32	64	128	256
DCMR	11.968 %	6.739%	**3.371**%	4.091%

Optimal cache-line size is 128B.

```
#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++)   // LOOP 0
    for (k = 0; k < CACHE_SIZE / 16 k++)
      A[k][j]++;

  for (j=0;j<8;j++)   // LOOP 1
    for (k = 0; k < CACHE_SIZE / 32; k++)
      B[k][j]++;

  for (j=0;j<16;j++)  // LOOP 2
    for (k = 0; k < CACHE_SIZE / 64; k++)
      C[k][j]++;

  for (j=0;j<32;j++)  // LOOP 3
    for (k = 0; k < CACHE_SIZE / 128; k++)
      D[k][j]++;

  for (j=0;j<64;j++)  // LOOP 4
    for (k = 0; k < CACHE_SIZE / 256; k++)
      E[k][j]++;

  for (j=0;j<128;j++) // LOOP 5
    for (k = 0; k < CACHE_SIZE / 512; k++)
      F[k][j]++;

}
```

Fig. 6. Case B: self-interference.

5.3 Case C: Matrix Multiply

In the previous cases (Sections 5.1 and 5.2), the optimal cache-line size is set at compile time and, therefore, the analysis returns a numeric-form result. In this section, we present a case where the analysis returns a symbolic-form result to comply with the dynamic behavior of the application.

The examples are simple and we check the accuracy of the analysis manually. At the same time, the problem size is large and it is not practical in an exhaustive collection of simulations.

We analyze a variation of the common *ikj*-matrix-multiply algorithm [24], see Fig. 7. Matrices A, B, and C are square matrices and, in particular, matrices B and C are power of two. We choose the size of the matrices so that, if there is interference due the reference to A, it is rare. The index computation for A is parameterized $(0 \leq n \leq 64)$.[16] Due to the upper bounds of the parameters, A does not interfere with any other matrix. Even if it could, the interference density would be small. We distinguish two

16. Note that we can handle larger cases; this is to yield a clearer example.

TABLE 3
STAMINA's Result for Self-Interference Example

	Line	8	16	32	64	128	256
Loop 0	$\epsilon_{ct}(L)$	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**

Loops 4 and 5 have no interference for any line size, the output is set to zero. In bold face, we present the optimal ϵ per cache-line size and loop.

different contributions: at compile time, $\epsilon_{ct}(L)$, at runtime, $\epsilon_{rt}(L)$. We have $\epsilon_{rt}(L) = 0$ for any L and

$$\epsilon_{ct}(\{8, 16, 32, 64, 128, 256\}) = \{2.00, \mathbf{1.00}, 2.00, 2.00, 2.00, 2.00\}.$$

The reference to A does not interfere with the references to C and to B for $0 \leq n, m \leq 64$. It would only if we use larger values for the parameters. The suggested optimal cache-line size is 16 Bytes. We simulate the number of cache misses for some values of m, n (only a subset of the possible 64^2 pairs is presented) and for different cache-line sizes. The experimental results are in Table 4.

STAMINA proposes 16B as optimal cache-line size because it currently assumes the interference density as $\rho = 1$ for every line size. However, ρ is $1/2$ for L = 32B and the two cache-line sizes (16B and 32B) are equally good and a larger cache line may improve overall performance. In practice, simulation results suggest that a cache-line size of 32B is optimal for a negligible difference (Table 4). A solution to this problem is presented shortly, in the next example, where we represent the cache misses as a symbolic expression of the cache-line size.

We analyze the blocked version of matrix multiplication, see Fig. 8. We analyze only the loop nest in the procedure *ikj_mm* and we find that $\epsilon_{ct}(L) = 0$ for any L and

$$\epsilon_{rt}(\{8, 16, 32, 64, 128, 256\}) = \{2.00, 2.00, 2.00, 2.00, 2.01, 2.03\}.$$

TABLE 4
Data Cache Misses for Matrix Multiply, Fig. 7, Using *shade* Cache Simulator

m	n	miss L=16B	miss L=32B	miss L=64B
4	4	5670	3740	2856
8	8	6107	4160	**3531**
12	12	7304	5330	**5011**
16	16	9645	**7632**	8800
20	20	13532	**11507**	13818
24	24	19355	**17309**	23355
28	28	27480	**25397**	33922
32	32	38283	**36159**	51881
36	36	52373	**50493**	70606
40	40	69782	**68011**	99709
44	44	91390	**90106**	128561
48	48	116546	**115064**	170124
52	52	146286	**144598**	209300
56	56	181488	**180018**	267866
60	60	221808	**220361**	321279
63	63	260740	**260418**	380464

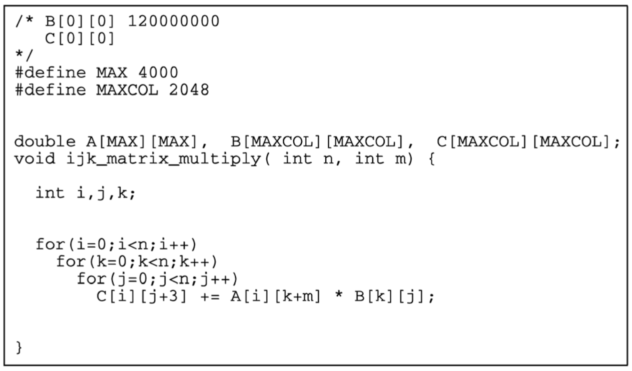
We present cache misses only for cache-line size 16B, 32B, and 64B (cache-line size 8B and 128B are omitted).

Every reference interferes with every other reference. The interference due to matrix A is negligible since the matrix access 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 cache-line size is not set once per loop nest, it is determined at runtime.

We expect to have a symbolic form of the type $\epsilon(L) = \eta_{R_C}(L) + \eta_{R_B}(L) + \eta_{R_A}(L)$. We know that, in this particular case, $\eta_{R_A} < L/16,384 * 2 \sim 0$. STAMINA produces a symbolic output where C_0 is $16,384$, Δ is $|8n - 8m| \% C_0$, and $\mathbb{1}(x)$ is 1 is $x \geq 0$ and 0 otherwise (where % is the C-language remain operator):

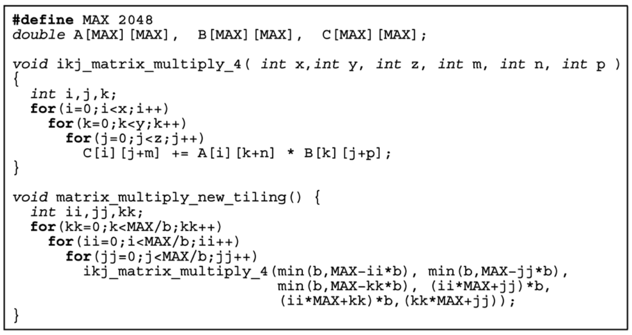
$$\epsilon(L) = 2 \min\left(1, \mathbb{1}(\Delta - L)\frac{8}{L} + \mathbb{1}(8 - \Delta)\frac{8 - \Delta}{L}\right), \tag{19}$$

which has minimum when L is 16B and 32B. In fact, reference C has spatial reuse and it may interfere with B, mainly: $\eta_{R_C}(L) = \mathbb{1}(L - \Delta)\frac{L-\Delta}{L} + \frac{8}{L}$. For example, when $n = m = 0$, references R_C and R_B interfere at any iteration and no optimal line size exists; otherwise, if $m = 3$ and



```
/* 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];


}
```

Fig. 7. Matrix multiply. There are two parameters: n and m. The first affects the loop bounds and the latter affects the access offset on matrix A. We assume that $0 < n, m < 64$.



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

void ikj_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++)
        ikj_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));
}
```

Fig. 8. Tiling matrix multiplication. We have six parameters: x, i, and k are used to specify the loop bounds, m, n, and p are used to modify the access to matrices C, A, and B, respectively.

$n = 0$ (notice that this is the example in Fig. 7), the optimal line size is 32B. Automatically, the symbolic form and the numerical form are used to insert a function driving adaptation in the source code before the loop nest.

For the example in Fig. 7, the analysis takes up to two minutes. For the blocked matrix multiplication in Fig. 8, the analysis takes more than 8 hours on a Sun Ultra 5 450MHz. The difference of the execution times is expected. For the former case, the existence test has to investigate a relatively small iteration space. For the latter case, the search for the existence of the integer solution is extremely time consuming because we need to search a space of $2,048^9$ points.[17]

6 Conclusion

We present a fast approach to statically determine the effect of the data cache-line size on the performance of scientific applications. We use the static cache model introduced by Ghosh et al. [6] 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 perfect loop nests on direct mapped cache. Experimental results demonstrate the accuracy and efficiency of our approach.

Acknowledgments

The authors wish to thank Vincent Loechner, Somnath Ghosh, Dan Hirschberg, and the members of AMRM project. In particular, they heartily thank Arun Kejariwal. They helped on Ehrhart polynomials and the existence test, cache miss equation determination, interference estimation, and moral/technical support, respectively. Financial support has been in part provided by DARPA/ITO under contract DABT63-98-C-0045.

References

[1] M. Kamble and K. Ghose, "Analytical Energy Dissipation Models for Low-Power Caches," *Proc. Int'l Symp. Low Power Electronics and Design,* pp. 143-148, 1997.

[2] K. Ghose and M. Kamble, "Reducing Power in Superscalar Processor Caches Using Subbanking, Multiple Line Buffers and Bit-Line Segmentation," *Proc. 1999 Int'l Symp. Low Power Electronics and Design,* pp. 70-75, 1999.

[3] A. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji, "Adaptive Cache Line Size to Application Behavior," *Proc. Int'l Conf. Supercomputing,* pp. 145-154, June 1999.

[4] P. van Vleet, E. Anderson, L. Brown, J. Baer, and A. Karlin, "Pursuing the Performance Potential of Dynamic Cache Line Sizes," *Proc. Int'l Conf. Computer Design (ICCS '99),* 1999.

[5] X. Ji, D. Nicolaescu, A. Veidenbaum, A. Nicolau, and R. Gupta, "Compiler-Directed Cache Assist Adaptivity," *Proc. Third Int'l Symp. High Performance Computing (ISHPC),* pp. 84-104, Oct. 2000.

[6] S. Ghosh, M. Martonosi, and S. Malik, "Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior," *ACM Trans. Programming Languages and Systems,* vol. 21, no. 4, pp. 703-746, July 1999.

[7] S. Ghosh, M. Martonosi, and S. Malik, "Automated Cache Optimizations Using CME Driven Diagnosis," *Proc. Int'l Conf. Supercomputing,* pp. 316-326, 2000. citeseer.nj.nec.com/ghosh00automated.html,

[8] X. Vera, J. Llosa, A. González, and N. Bermudo, "A Fast and Accurate Approach to Analyze Cache Memory Behavior (Research Note)," *Lecture Notes in Computer Science,* vol. 1900, pp. 194-, 2001. citeseer.nj.nec.com/vera00fast. html.

[9] X. Vera and J. Xue, "Let's Study Whole-Program Cache Behaviour Analytically," *Proc. Eighth Int'l Symp. High Performance Computer Architecture (HPCA '02),* pp. 176-185, 2002. citeseer.ist.psu.edu/article/vera02lets.html.

[10] K. McKinley and O. Temam, "A Quantitative Analysis of Loop Nest Locality," *Proc. Seventh Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 94-104, 1996.

[11] K. Kennedy and K. McKinley, "Optimizing for Parallelism and Data Locality," *Proc. Sixth Int'l Conf. Supercomputing,* pp. 323-334, 1992.

[12] P. Clauss, "Counting Solutions to Linear and Nonlinear Constraints through Ehrhart Polynomials: Applications to Analyze and Transform Scientific Programs," *Proc. 10th Int'l Conf. Supercomputing,* pp. 278-285, 1996.

[13] P. Clauss and V. Loechner, "Parametric Analysis of Polyhedral Iteration Spaces," *Proc. Int'l Conf. Application Specific Array Processors,* Aug. 1996.

[14] P. Clauss, "Handling Memory Cache Policy with Integer Points Countings," *Lecture Notes in Computer Science,* vol. 1300, pp. 285-293, Aug. 1997.

[15] P. Clauss, "Advances in Parameterized Linear Diophantine Equations for Precise Program Analysis," Technical Report ICPS RR 98-02, Universität Louis Pasteur, Strasbourg, Sept. 1998.

[16] M. Wolfe and M. Lam, "A Data Locality Optimizing Algorithm," *Proc. ACM SIGPLAN '91 Conf. Programming Language Design and Implementation,* June 1991.

[17] U. Banerjee, *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer Academic, 1993.

[18] S. Ghosh, M. Martonosi, and S. Malik, "Precise Miss Analysis for Program Transformations with Caches of Arbitrary Associativity," *Proc. Eighth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 228-239, 1998.

[19] W. Pugh, "A Practical Algorithm for Exact Array Dependence Analysis," *Comm. ACM,* vol. 35, no. 8, Aug. 1992.

[20] W. Pugh, "Counting Solutions to Presburger Formulas: How and Why," *Proc. ACM SIGPLAN 1994 Conf. Programming Language Design and Implementation,* pp. 121-134, 1994.

[21] S. Chatterjee, E. Parker, P. Hanlon, and A. Lebeck, "Exact Analysis of the Cache Behavior of Nested Loops," *Proc. ACM SIGPLAN 2001 Conf. Programming Language Design and Implementation,* pp. 286-297, 2001.

[22] A. Aho, R. Sethi, and J. Ullman, *Compilers, Principles, Techniques and Tools.* Addison-Wesley, 1986.

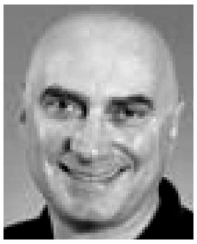
[23] R. Cmelik and D. Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling," technical report, Sun Microsystems Laboratories, 1993.

[24] G. Golub and C. van Loan, *Matrix Computations.* The Johns Hopins Univ. Press, 1996.

Paolo D'Alberto is currently a PhD student in computer science at the University of California at Irvine. His research interests include compiler design and hardware-software optimizations for divide-and-conquer algorithms.

17. The deployment of watch-dogs may be advised at this time, arguably, if a solution is difficult to find, then the interference density should be small and negligible.

Alexandru Nicolau received the PhD degree from Yale University in 1984. He is a professor of computer science at the University of California at Irvine. He has worked on various aspects of parallelizing compiler technology for over 20 years. His current research interests are in programming and operating system support for highly configurable parallel processing. He led the DARPA supported compiler projects PROMIS and COPPER on ILP and compiler optimizations for low power. He is a member of the IEEE.



Alexander Veidenbaum received the PhD degree from the University of Illinois at Urbana-Champaign in 1985. He is an associate professor of computer science at the University of California at Irvine. His interests have concentrated on new approaches to reducing and tolerating memory latency. The overall goal of his research is to increase system performance and/or minimize hardware cost and power consumption. His current projects focus on high-performance uniprocessor design, scalable and parallel systems, and embedded and adaptive systems. He is a member of the IEEE.

Rajesh Gupta received the PhD degree in electrical engineering from Stanford University in 1994. He is a professor of computer science at the University of California at San Diego (UCSD). He led the Adaptive Memory Reconfiguration Management project supported by the DARPA DIS Program. The AMRM project devised methods to build and automatically exploit adaptivity in the memory system for peak memory efficiency across a range of data intensive applications. He holds the Qualcomm Endowed Chair in Embedded Microsystems at UCSD. He is a fellow of the IEEE.