Improving Data Locality by Chunking

Cédric Bastoul[1] and Paul Feautrier[2]

[1] Laboratoire PRiSM, Université de Versailles Saint Quentin
45 avenue des États-Unis, 78035 Versailles Cedex, France
cedric.bastoul@prism.uvsq.fr
[2] École Normale Supérieure de Lyon
46 Allée d'Italie, 60364 Lyon, France
paul.feautrier@ens-lyon.fr

Abstract. Cache memories were invented to decouple fast processors from slow memories. However, this decoupling is only partial, and many researchers have attempted to improve cache use by program optimization. Potential benefits are significant since both energy dissipation and performance highly depend on the traffic between memory levels. But modeling the traffic is difficult; this observation has led to the use of heuristic methods for steering program transformations. In this paper, we propose another approach: we simplify the cache model and we organize the target program in such a way that an asymptotic evaluation of the memory traffic is possible. This information is used by our optimization algorithm in order to find the best reordering of the program operations, at least in an asymptotic sense. Our method optimizes both temporal and spatial locality. It can be applied to any static control program with arbitrary dependences. The optimizer has been partially implemented and applied to non-trivial programs. We present experimental evidence that the amount of cache misses is drastically reduced with corresponding performance improvements.

1 Introduction

Technological advances in the realization of integrated chips result in faster clocks for processors, and in larger capacity for memory. In consequence, if nothing is done, processors will starve because their memory systems cannot supply data at the required speed. Memory hierarchies are a good solution to this problem: they are cheap and efficient, at least for ordinary programs and situations. Nevertheless, their efficiency decreases dramatically for scientific computing and signal processing codes, where large data sets are accessed according to highly regular patterns. Next, their temporal behavior is difficult to predict; this forbids their use in systems with hard real time constraints. Lastly, moving data from level to level uses a lot of power, which renders them unsuitable for embedded systems.

A lot of work has been devoted to improving the behavior of memory hierarchies. There are two kinds of approaches for this problem. The first approach consists in designing highly optimized libraries (LAPACK is a good example [1]) for the most common linear algebra and signal processing algorithms. This method

G. Hedin (Ed.): CC 2003, LNCS 2622, pp. 320–334, 2003.

often gives the best results, provided the source problem and the target architecture are within the scope of the available library. The second approach tries to optimize the source program at compile time. This method is not restricted to a given set of algorithms and can be adapted, with minor modifications, to any memory hierarchy architecture. The present work belongs to the later approach.

Most optimizing compilers try to transform the source program in order to improve the behavior of the memory hierarchy. The basic principle is to regroup all accesses to a given memory cell, in order to take a maximum advantage of possible reuses. This is obtained first by applying loop transformations [15,11] according to some cost model [13], then by tiling the resulting loop nest [16] with tiles having a carefully chosen size [4]. Basically, this method applies only to perfect loop nests in which dependences are non-existent or have a special form (fully permutable loop nests). Another data-centric [9] approach starts from a memory cell and tries to build the slice that accesses this cell. Here again, dependences greatly complicate the transformation process.

As said above, previous methods require most of the time severe limitations on the input program. Our work can be applied to a wide application domain since we do not lay down any requirement on dependences provided that the program has static control [5]. This program class includes a large range of problems which are discussed in depth by Xue [17]. The properties of such programs can be summarized in this way: (1) control statements are **do** loops with affine bounds and **if** conditionals with affine conditions (in fact control can be more complex, see [17]); (2) arrays are the only data structures, and their subscripts are affine; (3) affine bounds, conditions and subscripts depend only on outer loop counters and structure (or size) parameters.

All methods mentioned earlier are based on a heuristic cost model. Let us consider for instance two accesses to the same memory cell. It seems probable that the longer the time interval between these accesses is, the higher the probability of the first reference to be evicted from the cache is. Hence, loop transformations aim at moving these references to neighboring iterations of some innermost loop. Our technique is based on an estimate of the memory traffic, and tries to find the loop transformation that minimizes this estimate, under the constraint that all dependences are satisfied. This technique, which we call *chunking* is presented in section 2. Section 3 explains how to construct good chunking functions for a given program. Section 4 deals with the problem of code generation when the chunking functions are given. Section 5 describes our implementation and experimental results. Section 6 compares chunking to other approaches. We then conclude and discuss future work.

2 Chunking

The principle of our method is to partition the set of operations of a program in subsets small enough that their accessed data fit in the cache: the *chunks*. The program is then executed chunk by chunk, as if there was a cache flush between each of them. These subsets must be such that their sequential execution is

equivalent to the execution of the original program. In practice, chunks will be numbered then executed in order of increasing numbers. A chunk number will be assigned to each operation, i.e. to each instance of each statement. In other words, for each statement S we seek a *chunking function* θ_S associating a chunk number $\theta_S(x)$ to each iteration vector x. The original operations will be rescheduled accordingly to these chunking functions. We present in figure 1 an example of chunking of a simple program. We assume as input hypothesis that n array elements can fit in the cache, but m cannot. Such a simple code yet exhibits several difficulties: non-perfect loop nest, dependences between different statements, parameters and multiple references. In this example, the order of the

```
do i=1, n
  a(i) = i                          ! S1
  do j=1, m
    b(j) = b(j) + a(i)              ! S2
  enddo
enddo
```

(a) source program

$$\theta_{S1}\left(\left[\,i\,\right]\right) = \left[\,i\,\right]; \theta_{S2}\left(\begin{bmatrix} i \\ j \end{bmatrix}\right) = \left[\,j+n\,\right]$$

(b) chunking functions

```
do c=1, n
  a(c) = c                          ! S1
enddo
do c=n+1, n+m
  do i=1, n
    b(c-n) = b(c-n) + a(i)          ! S2
  enddo
enddo
```

(c) target program

Fig. 1. Running example

operations has been modified for a maximal use of temporal locality, according to the chunking functions in figure 1(b). In the target program, c gives the number of the current chunk. This example will be used for illustration throughout this paper. It can be noticed that the code can be restructured in the same way by conventional loop distribution, loop permutation and skewing. Chunking is set in the framework of the polytope model and every chunking can be broken down in a succession of well known transformations. In fact, chunking does not aim to find *new* transformations but to find the *right* transformation automatically.

3 Computing Chunking Functions

The quality of a chunking can be assessed by using two valuations. First, the *footprint size* which is the number of memory cells accessed by the operations of a chunk. Next, the *traffic* which is the number of data movements between main and cache memories. We want to build an optimal chunk system *i.e.* where each chunk footprint fits in the cache and the traffic is minimal. To be able to generate the target code, we are looking for affine chunking functions. Subsequently, for an operation $S[x]$, instance of the statement S with the iteration vector x in the iteration domain D_S, the chunk number can be written:

$$\theta_S(x) = T_S x + k_S.$$

T_S is a matrix called the *chunking matrix*; its dimensions are $g \times \rho(S)$ with $\rho(S)$ the number of loops surrounding S. The choice of the value of g is postponed till section 3.2. k_S is a constant vector. Chunking functions are calculated in several steps which are discussed in the next sections. In section 3.1 we show how to compute an asymptotic evaluation of the traffic with respect to the chunking functions. Then we exhibit the constraints that the chunking functions must satisfy to minimize the traffic. Section 3.2 explains how to find all the functions verifying such constraints. Section 3.3 shows how to choose the functions in such a way that the transformation is legal for dependences. Lastly, section 3.4 and 3.5 gives respectively the constraints which have to be satisfied by the chunking functions in order to achieve group-locality and spatial-locality.

3.1 Asymptotic Evaluation

It is hard to find an accurate solution to the traffic evaluation problem for a particular cache type. Modeling the replacement mechanism is quite difficult, but it is bypassed by chunking. However, several difficulties remain, hence we propose the following simplifications on our cache and memory models:

- conflict misses do not change the order of magnitude of the traffic; this assumption is satisfied by fully associative caches and is close to be satisfied by modern caches with high associativity; most discrepancies can be compensated by using an effective cache size smaller than the real one;
- we will be satisfied with asymptotic evaluation of the traffic; in many cases, program transformations can change the order of magnitude of the traffic, then it would be useless to fiddle with constant factors or worse, units in the last decimal place; in some cases, *i.e.* when self-reuse has already been exploited, only the constant factors can be improved; the question of deciding if a more precise evaluation can influence the target code is left for future work.

In our model, it is possible to make estimates of footprint sizes and traffic with respect to the chunking functions. Considering a statement S, an array A

and a subscript function f, the footprint generated by this reference is the set of memory cells accessed during the chunk execution:

$$\mathcal{F}_{S,A,f}(t) = \left\{ f(x) \mid x \in D_S, \theta_S(x) = t \right\}. \tag{1}$$

Let us suppose that the cache is empty at the start of a chunk and that its footprint fits in the cache. Then any cells in the footprint is copied once to the cache at some time during the execution of the chunk and stays there until the termination of the chunk. Hence the traffic can be estimated as the number of pairs ⟨data, chunk number⟩:

$$\mathcal{T}_{S,A,f} = \text{Card } \left\{ \left\langle f(x), \theta_S(x) \right\rangle \mid x \in D_S \right\}. \tag{2}$$

Since input programs have static control, subscript functions are affine and can be written $f(x) = Fx + a$, where F is the subscript matrix of dimension $\rho(A) \times \rho(S)$, with $\rho(A)$ the dimension of array A, and a a constant vector.

Theorem 1. *Let* $H = \left\{ Ux \mid Vx = 0, x \in D \right\}$ *be a set where* U *and* V *are arbitrary integral matrices of the right dimension, and where* D *is a bounded full dimensional domain such that the value of each component of the vector* x *is an integer in a segment of length* m*. Then* Card H *is of the order of* m^l *with* $l = \text{rank } \begin{pmatrix} U \\ V \end{pmatrix} - \text{rank } V$.

Proof. Let us first study the dimension of the subspace $K = \left\{ Ux \mid Vx = 0 \right\}$. This corresponds to the rank of the application f from $\ker V$ to Im U that associates Ux to x. According to a well known algebraic theorem, we have dim $\ker V = \text{rank } f + \dim \ker f$. As $\ker f = \ker U \cap \ker V$, it follows:

$$\text{rank } f = \dim \ker V - \dim (\ker U \cap \ker V).$$

Since D is such that the value of each component of x is an integer in a segment of length m, it follows that each component of Ux also is integral and belongs to a segment of length proportional to m. Hence, the size of H is of the order of m^l. Since dim $\ker V + \text{rank } V =$ number of column of V, we have finally Card H is of the order of m^l with $l = \text{rank } \begin{pmatrix} U \\ V \end{pmatrix} - \text{rank } V$. ∎

The orders of magnitude of the cardinals of sets describing footprints (1) and traffic (2) are directly given by theorem 1. The asymptotic size of footprints are found with V as T and U as F, and considering the traffic, with V as the null matrix and U as the block matrix $\begin{pmatrix} T \\ F \end{pmatrix}$ composed of the matrix T for its first rows and of the matrix F for the next rows. If the value of each component of x is an integer in a segment of length m, we have:

$$\text{Card } \mathcal{F}_{S,A,f}(t) = O\left(m^l\right), \text{with } l = \text{rank } \begin{pmatrix} T \\ F \end{pmatrix} - \text{rank } T,$$

$$\mathcal{T}_{S,A,f} = O\left(m^k\right), \text{with } k = \text{rank } \begin{pmatrix} T \\ F \end{pmatrix}.$$

These evaluations depend on F which can be extracted by analysis of the source code and T which is the unknown of the problem. Thus we can find the constraints that T has to satisfy in order that the footprints fit in the cache and the traffic is minimal.

Let us consider one statement with n array accesses, the subscript matrix of the i^{th} access being F_i. All tuples $\left\langle \text{rank } T, \text{rank } \begin{pmatrix} T \\ F_i \end{pmatrix} \text{ for } 1 \leq i \leq n \right\rangle$ corresponding to the possible sets of constraints can be enumerated. We need to know the cache size C and an estimate of the size parameter m. We then determine an integer α such that $m^\alpha \leq C$. A footprint component of size $O\left(m^{l_i}\right)$ fits in the cache if $l_i \leq \alpha$. We can thus eliminate all tuples for which this condition is not satisfied, and we can rank the remaining ones in order of increasing traffic. It then remains to try building a T which satisfies the rank condition of the best tuple. If this is proved to be impossible, we start again with the next tuple.

3.2 Building Chunking Matrices

Thanks to the evaluations, we know which rank constraints must be satisfied by the chunking matrices to minimize the traffic. In this section, we show how to build such matrices, at first when the corresponding statement includes only one reference. Then, we show that there always exists a chunking matrix such that each associated footprint fits in the cache.

For a statement S with one reference, it is always possible to find a matrix T such that rank $T = v$ and rank $\begin{pmatrix} T \\ F \end{pmatrix} = w$, provided that v and w have compatible values (i.e. $\rho(S) \geq w \geq v$). The building process is described by the algorithm in figure 2. From the returned matrix T, we can generate the set of matrices with the required properties: the set of CT matrix where C is a matrix of full row rank. We will choose in this set the matrices in order to satisfy additional constraints described in section 3.3 and 3.4.

Let us demonstrate that this algorithm builds a matrix T that answers the requirements. Since the matrix T is composed of v linearly independent rows, the constraint rank $T = v$ is satisfied. These rows are those of G^{-1} from $\rho(S) - w + 1$ to $\rho(S) - w + v$. Hence, the kernel of T is generated by the column vectors of G from 1 to $\rho(S) - w$ and from $\rho(S) - w + v + 1$ to $\rho(S)$. The kernel of $\begin{pmatrix} T \\ F \end{pmatrix}$ is the intersection of the kernel of T with the kernel of F, hence it is generated by the $\rho(S) - w$ first column vectors of G and the constraint rank $\begin{pmatrix} T \\ F \end{pmatrix} = w$ is satisfied. As for the choice of g, the number of rows of T, it is clear that bordering a matrix by null rows does not change its rank. Since when reordering the program it is useful to have all chunking function of the same dimension, we may take $g = \max \rho(S)$.

The generalization to n references implies the combination of n constraints: rank $\begin{pmatrix} T \\ F_i \end{pmatrix} = w_i$ for $1 \leq i \leq n$. The matrix G must have for each reference

Construction Algorithm: Build a matrix under rank constraints.

Input: the subscript matrix F and the rank constraints $\operatorname{rank} T = v$ and $\operatorname{rank} \begin{pmatrix} T \\ F \end{pmatrix} = w$.
Output: a matrix T respecting the rank constraints.

1. Compute a basis of $\ker F$ and complete it to a basis of $N^{\rho(S)}$.
2. Let G be the matrix of these vectors (vectors added to complete to a basis of $N^{\rho(S)}$ are the last columns).
3. Compute G^{-1}, inverse of G.
4. Build matrix T:
   a) For i from 1 to v:
      i^{th} row of $T = (\rho(S) - w + i)^{th}$ row of G^{-1}.
   b) Complete T with null rows.

Fig. 2. Construction Algorithm

exactly $\rho(S) - w_i$ vectors of a basis of $\ker F_i$ for a total of at most v vectors. Such a matrix does not always exist. The choice of vectors to be included in the matrix G is essential. We can guide this choice by adding for each reference as many vectors from a preceding reference as possible. If a solution does not exist for a tuple, then we try to find another one for the next more interesting tuple.

A chunking matrix such as each footprint fits in the cache always exists. The hardest constraint for the footprints is to have a size in $O\left(m^0\right)$, and the last tried possibility will be the tuple $\langle \rho(S), w_i = \rho(S) \text{ for } 1 \leq i \leq n \rangle$. The corresponding chunking generates for the i^{th} reference footprint sizes of $O\left(m_i^0\right)$ and the maximal traffic of $O\left(m_i^{\rho(S)}\right)$. Its solution $T = Id$ always exists and is the trivial chunking where there is one chunk per operation.

Example 1. Let us consider the source code in figure 1. We assume that `a` is an array of `n` cells which fits in the cache and `b` is an array of `m` cells which does not fit in the cache. Then, the acceptable orders of magnitude for the footprints size are $O\left(n^1\right)$ and $O\left(m^0\right)$. The program has two statements:

- the statement $S1$ has just one reference to the array `a` with the index matrix $F_{S1,1} = \begin{bmatrix} 1 \end{bmatrix}$; the matrix T_{S1} having the best properties corresponds to the tuple $\langle 1, 1 \rangle$, it will generate footprint sizes of $O\left(n^0\right)$ and a traffic of $O\left(n^1\right)$; the algorithm builds $T_{S1} = \begin{bmatrix} 1 \end{bmatrix}$;
- the statement $S2$ has two references, the first one to the array `a` with the index matrix $F_{S2,1} = \begin{bmatrix} 1\,0 \end{bmatrix}$ and the second one to the array `b` with the index matrix $F_{S2,2} = \begin{bmatrix} 0\,1 \end{bmatrix}$; the matrix T_{S2} having the best properties would correspond to the tuple $\langle 1, 2, 1 \rangle$, it would generate footprint sizes of

$O\left(m^0 + n^1\right)$ and a traffic of $O\left(m^1 + n^2\right)$; the construction is possible and gives $T_{S2} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$.

3.3 Legality

Since chunking reorders operations, it must satisfies dependences. In this section, we explain how chunking functions can be chosen in such a way that the transformation satisfies dependences. We will show that there always exists a valid solution which satisfies the constraints described in previous sections.

Chunks are numbered in the order they will be executed, and inside each of them, operations are executed in the original sequential order. Let us consider $I_{\mathcal{P}}$, the statement set of the program $\mathcal{P}$, and $\delta_{\mathcal{P}}$, the dependence relation on $\mathcal{P}$; a chunking is legal if and only if:

$$\forall S, R \in I_{\mathcal{P}}, \quad S[x]\, \delta_{\mathcal{P}} R[y] \Rightarrow \theta_S(x) \leq \theta_R(y). \tag{3}$$

There is no *a priori* reason for (3) to be satisfied by the chunking matrices as constructed by the algorithm in previous section. However, we are free to modify them as long as we do not change their rank properties. We are also free to adjust the constant vectors k, as they have no impact on the footprints and traffic (at least asymptotically). Thus, for any statement S, the chunking function can be written

$$\theta_S(x) = C_S T_S x + k_S,$$

where C_S is a matrix of full row rank. We use the Farkas algorithm [6] to solve (3) and to find the set of all C_S and k_S. If the problem has no solution, we declare a failure and try the next best traffic/footprint combination.

A legal solution such as the footprints fit in the cache always exists. It corresponds to the worst solution, in which all the chunking matrices are identity matrices. In this case, the original program is not modified. This possibility must always be left open, since it might happen that the source program is already optimal.

Example 2. Let us continue the example of section 3.2. The chunking functions associated to the proposed matrices are:

$$\theta_{S1}\left(\begin{bmatrix} i \end{bmatrix}\right) = \begin{bmatrix} 1 \end{bmatrix} \begin{bmatrix} i \end{bmatrix} + \begin{bmatrix} 0 \end{bmatrix} = \begin{bmatrix} i \end{bmatrix};\ \theta_{S2}\left(\begin{bmatrix} i \\ j \end{bmatrix}\right) = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} j \\ 0 \end{bmatrix}.$$

These functions do not describe a valid chunking: the dependence from $S1$ to $S2$ is not satisfied. For instance, the operation $S2 \begin{bmatrix} 2 \\ 1 \end{bmatrix}$ is executed in chunk number 1 whereas the operation $S1 \begin{bmatrix} 2 \end{bmatrix}$ on which it depends is executed later, in chunk number 2. Our method makes it possible to correct this chunking so that all the dependences are respected and the quality is preserved. The correction suggested by our prototype is the following one:

$$\theta_{S1}\left(\begin{bmatrix} i \end{bmatrix}\right) = \begin{bmatrix} 1 \end{bmatrix} \begin{bmatrix} i \end{bmatrix} + \begin{bmatrix} 0 \end{bmatrix} = \begin{bmatrix} i \end{bmatrix};\ \theta_{S2}\left(\begin{bmatrix} i \\ j \end{bmatrix}\right) = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} n \\ 0 \end{bmatrix} = \begin{bmatrix} j+n \\ 0 \end{bmatrix}.$$

To homogenize the chunking functions, one can add null dimensions, or remove them if they are null for all the functions, since this does not change the ranks. We have finally $\theta_{S1}\left(\left[\, i \,\right]\right) = \left[\, i \,\right]$ and $\theta_{S2}\left(\begin{bmatrix} i \\ j \end{bmatrix}\right) = \left[\, j + n \,\right]$.

3.4 Group-Reuse

There is group-reuse when two statements, $S1$ and $S2$, access the same array `A` through indexing matrices F_1 and F_2 (for the sake of readability, we will use homogeneous coordinates in this section). There is reuse if there exists iteration vectors x_1 and x_2 such that $F_2 x_2 = F_1 x_1$, and this reuse is exploited if these two operations are in the same chunk:

$$\forall x_1 \forall x_2, F_2 x_2 - F_1 x_1 = \mathbf{0} \Rightarrow T_2 x_2 - T_1 x_1 = \mathbf{0}. \tag{4}$$

Observe that this constraint has the same shape as a dependence constraint. If $F_2 x_2 = F_1 x_1$, then $S1[x_1]$ and $S_2[x_2]$ are in dependence. This dependence may be a read-read dependence, which may not be taken into account in other circumstances, but which exists nevertheless. As to the right-hand side of (4), it is similar but more restrictive than the right-hand side of (3). As a consequence, we can give a more precise result:

Theorem 2. *(4) is true iff* $\begin{pmatrix} T_2 & -T_1 \end{pmatrix} = N \begin{pmatrix} F_2 & -F_1 \end{pmatrix}$ *where* N *is a matrix of full row rank.*

Proof. Let x be the concatenation of vectors x_1 and x_2. Formula (4) can be written

$$\forall x, \begin{pmatrix} F_2 & -F_1 \end{pmatrix} x = 0 \Rightarrow \begin{pmatrix} T_2 & -T_1 \end{pmatrix} x = 0.$$

$\begin{pmatrix} F_2 & -F_1 \end{pmatrix} x = 0$ and $\begin{pmatrix} T_2 & -T_1 \end{pmatrix} x = 0$ describe two sets where one point belonging to the first one necessarily belongs to the second one too. Therefore the first one is a subset of the second one. So it can be written as the second one with b additional constraints:

$$\begin{pmatrix} F_2 & -F_1 \end{pmatrix} x = 0 \Leftrightarrow \begin{cases} \begin{pmatrix} T_2 & -T_1 \end{pmatrix} x = 0 \\ Qx = 0 \end{cases}$$

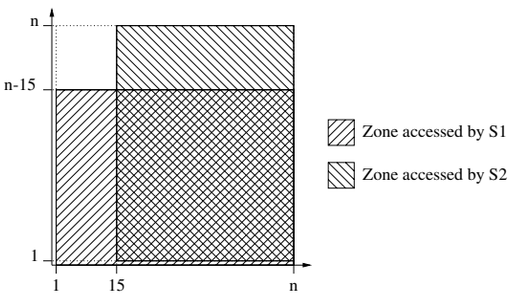
then $\begin{pmatrix} T_2 & -T_1 \\ & Q \end{pmatrix} = M \begin{pmatrix} F_2 & -F_1 \end{pmatrix}$ with M a matrix such that $\det M \neq 0$ (the system is not modified by linear transformations). Let us write M as $\begin{pmatrix} N \\ N' \end{pmatrix}$ where N' is the matrix made with the b last lines of M. Now we have $\begin{pmatrix} T_2 & -T_1 \\ & Q \end{pmatrix} = \begin{pmatrix} N \\ N' \end{pmatrix} \begin{pmatrix} F_2 & -F_1 \end{pmatrix}$ and finally $\begin{pmatrix} T_2 & -T_1 \end{pmatrix} = N \begin{pmatrix} F_2 & -F_1 \end{pmatrix}$. ∎

The unknowns are the entries of N, which define the linear transformations to apply to $\begin{pmatrix} F_2 & -F_1 \end{pmatrix}$ in such a way that the chunking functions respect the dependences. This is clearly the same problem as the correction for dependences in section 3.3. We solve them at the same time, by adding the necessary constraints

(a set of constraints by pairs of references in which group-reuse is detected) to the initial problem. This theory, which does not assume that group-reuse is associated to constant dependences, can even be used for "self-group-reuse", when the two accesses to `A` are in the same statement. Here, we deduce from (4) that the linear subspace $G = \{x_2 - x_1 | F_1 x_1 - F_2 x_2 = 0\}$ is included in the kernel of $T = T_1 = T_2$.. It is easy to find a basis for G by gaussian elimination techniques. The resulting vectors can be taken into account when building the chunking matrices. Improving group-locality do not change the order of magnitude of the traffic. It can divide the traffic generated by n references by a factor of n.

Example 3. Let us consider the source code in figure 3(a). All control centric

```
do i=1, n
  do j=5, n-10
    C(i,j) = A(i,j-5)  ! S1
    D(i,j) = A(j+10,i) ! S2
  enddo
enddo
```



(a) sample code (b) Accessed zones of `A`

Fig. 3. Example of group reuse

methods will estimate that there is no self reuse and no exploitable group-reuse. The reason is that they fail to consider non uniformly generated references (uniformly generated references are such as their subscript functions differ in at most the constant term [7]). In fact there is good reuse between the two statements for a part of the array `A` as shown by the figure 3(b). In this example, there is no dependence, then we can use the trivial solution of $(T_2 - T_1) = N(F_2 - F_1)$, that is $T_1 = F_1$ and $T_2 = F_2$. Therefore, the chunking functions will be :

$$\theta_{S1}\left(\begin{bmatrix} i \\ j \end{bmatrix}\right) = \begin{bmatrix} i \\ j-5 \end{bmatrix}; \theta_{S2}\left(\begin{bmatrix} i \\ j \end{bmatrix}\right) = \begin{bmatrix} j+10 \\ i \end{bmatrix}.$$

This transformation leads to the target code below. The group-locality is now maximal: in the shared zone of `A`, the two statements access the same memory cell during the same iteration.

```
do c1=1, 14
  do c2=0, n-15
    C(c1,c2+5) = A(c1,c2)        ! S1
  enddo
enddo
do c1=15, n
  C(c1,5) = A(c1,0)              ! S1
```

```
  do c2=1, n-15
    C(c1,c2+5)  = A(c1,c2)       ! S1
    D(c2,c1-10) = A(c1,c2)       ! S2
  enddo
  do c2=n-14, n
    D(c2,c1-10) = A(c1,c2)       ! S2
  enddo
enddo
```

3.5 Spatial-Reuse

There is spatial reuse for a reference if it accesses data on the same cache line during different iterations. As for group locality, improving spatial locality do not change the order of magnitude of the traffic. It can divide the traffic generated by a reference by a factor of d, where d is the cache line length in words. Spatial locality is achieved if the operations accessing the same cache line are in the same chunk. Let us consider a reference to an array A with the subscript function F. Let i be the number of the major dimension of A, i.e. the dimension with data lines ordered successively in memory. Then spatial locality is achieved for A if the operations accessing the memory cells of the major dimension are in the same chunk. In other words, spatial locality is achieved if $F_{i,\cdot} \in \ker T$.

This constraint is added in the T construction algorithm seen in section 3.2 by asking for a more accurate choice of vectors to be included in the matrix G. If the new constraint prevents the construction of T, we can try with another line of the subscript function and suggest the corresponding data layout transformation. This result can be compared with the Kandemir et al. method [8], where both loop and data transformations are used to improve spatial locality. Chunking does not require a non-singular transformation matrix, but it can achieve spatial locality only for a given loop level. However, in practive results are often alike.

4 Code Generation

Code generation is the last step to the final program. It is often ignored in spite of its impact on the target code quality. We must ensure that a bad control management does not spoil performance, for instance by producing redundant guards or complex loop bounds. An outline of the resulting code is a loop on the number of chunks L which contains the chunk operations. If the chunk numbers are vectors, we have as many surrounding loops as chunking dimensions.

Because the input problem is a static control program, the bounds on statement iteration spaces can be specified by a set of linear inequalities defining a polyhedron [10]. In the chunking case, we change the scanning order of this polyhedron by substitution of the original dimensions by chunking dimensions. The code generation is then a well known Z-polyhedron scanning problem. At present, the best solution is the Quilleré et al. one [14]. Their method is well adapted to the chunking problem provided we generalize it somewhat. We have implemented an extended version, CLooG, which can handle sequential inner loops and imperfect loop nests. Our resulting code is quite efficient.

5 Experimental Results

We are implementing our approach in the *chunky*[1] source-to-source optimizing tool. This prototype implements at present the process from the chunking function calculation to the code generation, but without group and spatial locality improvement support. This prototype already allows us to present preliminary results for some important non-trivial problems. The experiments were conducted on a PC workstation with a Pentium III processor running at 1GHz. This processor comes with two cache levels: a split first level (L1) for instructions and data of 16KB each and an unified second level (L2) of 256KB. Figure 4 shows the evolutions of the number of cache misses observed with hardware counters for the original and target versions of the running example (see figure 1), according to the value of the parameter m.

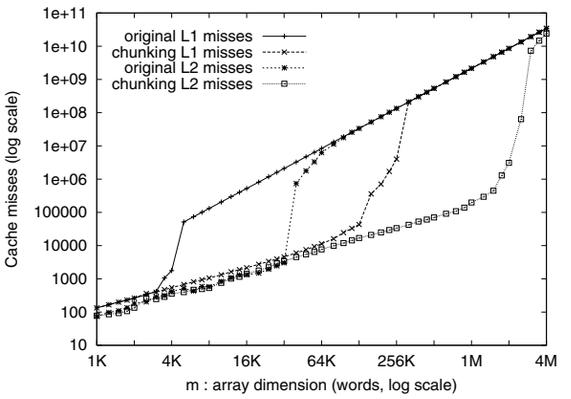


Fig. 4. Cache misses for the running example

The ratio m/n is set to 64 in order to better show the impact of our method. The number of cache misses sharply grows when the array b becomes larger than a cache level in the original program. The chunked program has a better behavior. The miss growth comes later, when the input hypothesis are no longer satisfied, *i.e.* when the array a cannot fit in the cache. We have observed the same phenomenon on most of the programs with good data reuse we have tested. Some experimental results on well known problems are shown in figure 5. The compiler option was O3 for the original programs, but O1 for the transformed programs in order to prevent any compiler optimization that can disturb the chunking. As for the running example, chunking can reduce the number of cache misses by more than one order of magnitude. This cache miss reduction can imply a significant performance improvement. The speedup is better with big problems. Since the

[1] Parts of Chunky are freely available at `http://www.prism.uvsq.fr/~cedb`

problem	array size (words)	missdown (%)	speedup (%)
running example	16K	99.1 (L1)	7
	1M	99.9 (L2)	427
LU decomposition	$80 * 80$	79.3 (L1)	2
	$256 * 256$	84.1 (L2)	43
Cholesky factorization	$80 * 80$	70.3 (L1)	2
	$256 * 256$	85.5 (L2)	46
Gauss-Jordan	$80 * 80$	70.2 (L1)	-13
	$256 * 256$	93.1 (L2)	26

Fig. 5. Experimental results

miss penalty for an L2 miss is of the order of 10 times an L1 miss, these results are not surprising. The situation of Gauss-Jordan for $80 * 80$ arrays shows how it is necessary to avoid control overheads. In this (rare) case, despite the attention given to code generation and a significant cache miss reduction, our method fails to improve performance on small problems. The point of view is quite different when the critical resource is energy, like in embedded systems. Cathoor et al. [3] show that data movements in the hierarchy is one of the main cause of energy consumption. In this case, a cache miss reduction is always a benefit.

6 Related Work

The effort of research to create effective locality optimizing compilers began with Wolf and Lam [15] and their *data locality optimizing algorithm*. This algorithm applies unimodular transformations to loop nests in order to maximize locality, according to evaluations of legal loop transformations relevance. Then it applies tiling [16] to the innermost loops. In comparison, our approach is applicable to a wider range of programs since in one hand we do not require perfect nests or nests such as they can be made perfect. And on the other hand because we do not require that dependences must have any simplified shape (Wolf and Lam algorithm needs that the dependence vectors be lexicographically positive). Moreover, to make perfect loops and to tile imply severe control overhead while we minimize it thanks to an accurate code generation method.

Li [11] generalizes the framework of unimodular matrices [2] by using linear, non-unimodular transformations to change the iteration space. We expect our algorithm will find more accurate transformations in practice since Li's transformation and dependence types are quite simple: the transformations do not handle parameters and the only case discussed is the one where dependences are represented by distance vectors.

McKinley et al. [13] propose a technique based on a detailed cost model that drives the use of loop permutation, fusion and distribution. They apply the basic transformations according to a definite order, while this strategy can be ineffective for some problems. To find which is the best application order of

the transformations for a given program is known to be very hard. Chunking bypasses this difficulty because it unifies all kind of linear transformations in a single framework. For group-reuse, McKinley et al. consider the classic case of *uniformly generated references* [7], with small restrictions. We propose to go beyond this case by optimizing group-locality between non uniformly generated references when they are in different statements. In compensation, chunking processing is heavier than the McKinley et al. algorithm.

Alternatively to these control centric techniques, Kodukula et al. [9] propose a data centric approach that plans to act on data movement directly, rather than as a side-effect of control flow manipulations. Our work shares many features with [9]. Both papers are set in the framework of the polytope model, and aim at partitioning the code in pieces which are (almost) free of cache misses. Both techniques transform the code by well known transformations (loop exchange, loop skewing ...): the problem is not to invent *new* transformations, but to find the *right* transformation for a given program. There are however several important differences. Kodukula et al. start from the following intuition: once a datum has been brought into the cache, it is beneficial to execute all operations which access this datum. Our approach is different since we start from an estimate of the traffic and try to minimize it. In both cases we have to find a transformation legal for dependences. But while Kodukula et al. can just check if their transformation respects dependences, we have integrated the legality in the transformation construction. Lastly, while Kodukula et al. use an arbitrary array blocking, we show that significant improvements can be obtained without blocking. Testing whether blocking can improve our results is left for future studies.

7 Conclusion

In this article, we have presented a method based on traffic evaluations for data locality improvement. It exhibits many advantages. First of all, the computed solution always fulfills the memory requirements imposed. Next, it can be applied to any static control slice of a program. Lastly, there is no requirement on dependences and we compute the space of all legal transformations directly. The method requires nothing besides the original code but the relative sizes of the cache and data.

First results are very encouraging and make us believe that our technique is a new significant way to achieve data locality automatically for a large amount of problems. Moreover, chunking seems to be well adapted to several extensions and we plan to obtain even better theoretical and practical results. We are currently working on tiling which seems to be the natural continuation of our approach. Intuitively, tiling is a question of aggregating small chunks or splitting big ones. We are also working on a more accurate solution for spatial locality improvement. A step in that direction is the work of Loechner, Meister and Clauss [12], which is based on precise counting of memory accesses. Lastly, we must deal with programs which have static control regions but do not have static control *in toto*. Locality optimization have the nice property that there is no

need of applying it to far away statements, since the hope of having reuse in this situation is very small. Hence chunking can be applied locally, i.e. to loop nests or small subroutines, and there is no danger of an excessive compilation time. Our method can be adapted to local memories (or software managed caches) at the price of more attention to footprint layout.

References

1. E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK User's Guide, Third Edition.* SIAM, 1999.
2. U. Banerjee. Unimodular transformations of double loops. In *Advances in Languages and Compilers for Parallel Processing*, pages 192–219, Irvine, august 1990.
3. F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom memory managament methodology.* Kluwer Academic, 1998.
4. S. Coleman and K. McKinley. Tile size selection using cache organization and data layout. In *ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 279–290, La Jolla, june 1995.
5. P. Feautrier. Dataflow analysis of scalar and array references. *International Journal of Parallel Programming*, 20(1):23–53, february 1991.
6. P. Feautrier. Some efficient solutions to the affine scheduling problem, part I: one dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, october 1992.
7. D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memories management by global program transformation. *Journal of Parallel and Distributed Computing*, (5):587–616, 1988.
8. M. Kandemir, J. Ramanujam, and A. Choudhary. Improving cache locality by a combination of loop and data transformations. *IEEE Transactions on Computers*, 48(2):159–167, february 1999.
9. I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *ACM SIGPLAN'97 Conference on Programming Language Design and Implementation*, pages 346–357, Las Vegas, june 1997.
10. D. Kuck. *The Structure of Computers and Computations.* John Wiley & Sons, Inc., 1978.
11. W. Li. *Compiling for NUMA parallel machines.* PhD thesis, Cornell Univ., 1993.
12. V. Loechner, B. Meister, and P. Clauss. Precise data locality optimization of nested loops. *Journal of Supercomputing*, 21(1):37–76, january 2002.
13. K. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, july 1996.
14. F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, october 2000.
15. M. Wolf and M. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 30–44, New York, june 1991.
16. M. Wolfe. Iteration space tiling for memory hierarchies. In *3rd SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, december 1987.
17. J. Xue. Transformations of nested loops with non-convex iteration spaces. *Parallel Computing*, 22(3):339–368, 1996.