Improving data locality by chunking

Cédric Bastoul
Laboratoire PRiSM
Université de Versailles Saint Quentin
45 avenue des États-Unis
78035 Versailles Cedex, France
cedric.bastoul@prism.uvsq.fr

Paul Feautrier
INRIA
Domaine de Voluceau
Rocquencourt - B.P. 105
78153 Le Chesnay Cedex, France
paul.feautrier@inria.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. Modeling the traffic between levels is difficult; this observation has led to the use of heuristics 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 temporal locality in the case of self and group-reuse. It can be applied to any static control program with arbitrary dependences. The optimizer has been implemented and applied to non-trivial programs. We present experimental evidence that the amount of cache misses is drastically reduced with corresponding energy saving and performance improvements.

Keywords

Temporal locality, cache memory, program transformations

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 soon 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 [6], 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 [2]) for the most common linear algebra and signal processing algorithms. This method 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 [20, 15] according to some cost model [17], then by tiling the resulting loop nest [21] with tiles having a carefully chosen size [8]. Basically, this method applies only to perfect loop nests in which dependences are nonexistent or have a special form (fully permutable loop nests). Another, data-centric [13], approach starts from a memory cell and tries to build the slice of the program 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 [10]. This program class include a large range of problems which are discussed in depth by Xue [22]. The properties of such programs can be summarized in this way: (1) control statements are the Fortran **do** loop with affine bounds and **if** conditional with affine conditions (in fact control can be more complex, see [22]); (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; (4) subroutine and function calls have been inlined.

All methods cited earlier are based on a heuristic cost model. Let us consider for instance two accesses to the same mem-

ory cell. It seems probable that the longer the time interval between these accesses, the higher is the probability of the first reference to be evicted from the cache. 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 descripbes our implementation and experimental results. Section 6 compare 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 and executed in order of increasing numbers. 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 . 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 and multiple references. In this example, the order of the operations

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

(a) source program

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

(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

Figure 1: Running example

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. One can notice 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 do not aim to find *new* transformations but to find the *right* transformation automatically.

3. COMPUTING CHUNKING FUNCTIONS

The quality of a chunking system 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. In such a system, each chunk footprint fits in the cache and each memory cell appears in as few footprints as possible. To be able to generate the target code, we are looking for affine chunking functions. 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:

$$\theta_S(x) = Tx + k.$$

T is the chunking matrix of dimension $g \times \rho(S)$ with $\rho(S)$ the number of loops surrounding S and k a constant vector; the choice of the value of g is postponed till section 3.2. 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 which the chunking functions must satisfy to minimize the traffic. Section 3.2 explains how to build the functions under such constraints. Section 3.3 shows how to modify the functions in such a way that the transformation is legal for dependences. Lastly, section 3.4 gives the constraints which have to be satisfied by the chunking functions in order to achieve group-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 remains, hence we propose the following simplifications:

- conflict misses don't change the order of magnitude of the traffic; this assumption is satisfied by fully associative caches and is close to be by modern caches with high associativity; any discrepancy 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. In these cases, it would be useless to fiddle with constant factors or worse, units in the last decimal place. In some cases, e.g. when self-reuse has already been exploited, one can only improve the constant factors; 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. 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:

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

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⟩.

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

Note that there is no need to insert a flush instruction between chunks provided that the replacement mechanism always selects data from previous chunks for eviction. This is true for the LRU and FIFO policies, but not for RANDOM.

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.

The orders of magnitude of the cardinals of sets describing footprints (1) and traffic (2) are known: if the value of each component of x is an integer in a segment of length m, then:

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

where $\begin{pmatrix} T \\ F \end{pmatrix}$ is a matrix composed of the matrix T for its first rows and of the matrix F for the next rows.

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. l and k are not arbitrary; it is easy to check that:

$$\begin{cases} 0 \leq \text{rank } (T) \leq \rho(S) \\ \max\left(\text{rank } F, \text{rank } T\right) \leq \text{rank } \begin{pmatrix} T \\ F \end{pmatrix} \\ \text{rank } \begin{pmatrix} T \\ F \end{pmatrix} \leq \min\left(\rho(S), \text{rank } T + \text{rank } F\right) \end{cases} \quad (3)$$

Let us consider one statement with n array accesses, the subscript matrix of the i^{th} access being F_i. We can enumerate all tuples $\left\langle \text{rank } T, \text{rank } \begin{pmatrix} T \\ F_i \end{pmatrix} \text{ for } 1 \leq i \leq n \right\rangle$ which satisfy the constraints (3). 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 of size $O\left(m^l\right)$ fits in the cache if $l \leq \alpha$. 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 with one reference, building a matrix T with rank v such that rank $\begin{pmatrix} T \\ F \end{pmatrix} = w$ is always possible, provided that v and w have compatible values. To do it, we compose a generating matrix having the basis vectors of $\ker F$ as column vectors, which we extend to a non singular matrix. We then compute the inverse of the generating matrix. T is made of v rows of the inverse, completed with null rows if necessary. The process is more formally described in the algorithm in figure 2.

Algorithm Construction: Build a matrix under rank constraints.

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

Output: a matrix T respecting the rank constraints.

1. Compute B, a basis of $\ker F$ and complete it to a basis of $N^{\rho(S)}$.
2. Build the generating matrix G:
   (a) For i from 1 to $\rho(S)$:
   i^{th} column of $G = i^{th}$ vector of B.
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) For i from $v+1$ to g:
   i^{th} row of $T = \vec{0}$.

Figure 2: Algorithm Construction

Let us demonstrate that this algorithm builds a matrix T that answers 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, 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 generating matrix must have for each reference exactly $\rho(S) - w_i$ vectors of a basis of $\ker F_i$ for a total of at most v vectors. Such a matrix doesn't always exist. The choice of vectors to be included in the generating matrix is essential. We can guide this choice by adding for each reference as many vectors from a preceding reference as possible. If a solution doesn't 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 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 $T1$ having the best properties corresponds to the tuple $\langle 1, 1 \rangle$, it will generate footprint sizes of $O\left(n^1\right)$ and a traffic of $O\left(n^1\right)$; one 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 modified 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 system is legal if and only if:

$$\forall S, R \in I_{\mathcal{P}}, \quad S[x]\, \delta_{\mathcal{P}} R[y] \Rightarrow \theta(S[x]) \leq \theta(R[y]). \qquad (4)$$

In this formula, if the chunking function is many-dimensional, $\leq$ has to be interpreted as lexicographic ordering. This can be done in the following way: the problem $u \geq 0$ where u is n-dimensional can be replaced by n problems

$$u_1 \geq 1, \begin{cases} u_1 \geq 0 \\ u_2 \geq 1 \end{cases} \cdots \begin{cases} u_1 \geq 0 \\ \vdots \\ u_n \geq 0 \end{cases}. \qquad (5)$$

Let us call ϵ any one of the 0-1 vectors on the right hand sides of (5). (4) can be split into n problems

$$S[x]\, \delta_{\mathcal{P}} R[y] \Rightarrow \theta(S[x]) + \epsilon \leq \theta(R[y]), \qquad (6)$$

where $\leq$ is now componentwise ordering. Let us write

$$\theta_{Si}(S[x]) = t_{Si} x + k_{Si}$$

for the i^{th} component of θ_S; equivalently, t_{Si} is the i^{th} row of the matrix T_S and k_{Si} is the i^{th} component of the vector k_S. The Farkas algorithm [11] allows one to eliminate x and y from (6) and other similar constraints. The result is a system of linear inequalities which we write

$$\Phi(\mathcal{T}, \mathcal{K}, \epsilon) \geq 0, \qquad (7)$$

where $\mathcal{T}$ and $\mathcal{K}$ are the concatenation of the t_{Si} and k_{Si}. There is no *a priori* reason for (7) to be satisfied by the chunking matrices as constructed by the previous algorithm. However, we are free to modify them as long as we do not change their rank properties. We are also free to adjust the k_S, as they have no impact on the footprints and traffic (at least asymptotically). We choose first to replace the first row of T_S by a linear combination of all rows:

$$t'_{S1} = \sum_{i=1}^{g} c_{Si} t_{Si}.$$

When substituted into (7), this gives a new system of linear constraints in the c_{Si}, which we try to solve with any linear programming code. If the problem has a solution, we apply the same algorithm to the next row. If not, 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 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. One has finally $\theta_{S1}\left(\begin{bmatrix} i \end{bmatrix}\right) = \begin{bmatrix} i \end{bmatrix}$ and $\theta_{S2}\left(\begin{bmatrix} i \\ j \end{bmatrix}\right) = \begin{bmatrix} j+n \end{bmatrix}$. ∎

3.4 Group-reuse

There is group-reuse when two statements, $S1$ and $S2$, access the same array `A` throught 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 $F_2x_2 = F_1x_1$, and this reuse is exploited if these two operations are in the same chunk:

$$\forall x_1 \forall x_2, F_2x_2 - F_1x_1 = \vec{0} \Rightarrow T_2x_2 - T_1x_1 = \vec{0}. \qquad (8)$$

Observe that this constraint has the same shape as a dependence constraint. If $F_2x_2 = F_1x_1$, then $S1[x_1]$ and $S_2[x2]$ 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 (8), it is similar but more restrictive than the right-hand side of (4).As a consequence, we can give a more precise result:

THEOREM 1. *(8) 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: $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$. Formula (8) 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 hyperplans where one point belonging to the first one necessarily belongs to the second one too. Therefore the first one is a subspace 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 $\begin{pmatrix} F_2 & -F_1 \end{pmatrix} = M\begin{pmatrix} T_2 & -T_1 \\ Q & \end{pmatrix}$ with M a matrix such that $\det M \neq 0$ (the system don't change by linear transformations), and $\begin{pmatrix} T_2 & -T_1 \\ Q & \end{pmatrix} = M^{-1}\begin{pmatrix} F_2 & -F_1 \end{pmatrix}$. Let us write M^{-1} as $\begin{pmatrix} N \\ N' \end{pmatrix}$ where N' is the submatrix made with the b last lines of M^{-1}. 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 (8) that the linear subsspace $G = \{x_2 - x_1 | F_1x_1 - F_2x_2 = 0\}$ 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.

Example Let us consider the following source code:

```
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 simple control centric method will estimate that there is no self reuse and no exploitable group-reuse. In fact there is good reuse between the two statements for a part of the array `A` as shown by the figure 3. In this example, there

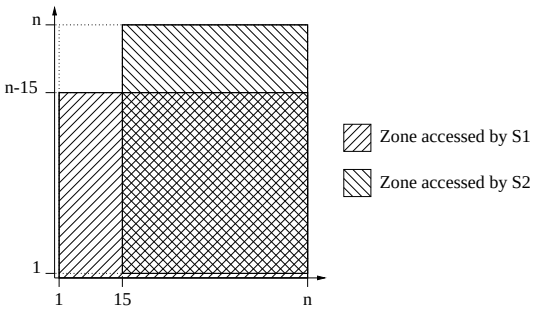

Figure 3: Accessed zones of `A`

is no dependence, then we can use the trivial solution of $\begin{pmatrix} T_2 & -T_1 \end{pmatrix} = N\begin{pmatrix} F_2 & -F_1 \end{pmatrix}$, 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
```

■

The selection of pairs of references offering a good group-reuse is an interesting problem. On one hand, it is certainly not possible to satisfy all constraints for all possible pairs. Hence, there is a need to find a priority order on the sets of constraints according to the potential benefits. On the other hand, improving group-locality is less interesting than improving self locality: it can't change the order of magnitude of the traffic. But adding constraints can complicate the chunking functions and as a consequence the target code. There is a need to evaluate which constraints can give a performance benefit in spite of the control overhead (this question has no sense when energy is the critical resource). It is quite easy to know if there exists group-reuse between a pair of references: it is sufficient to find an integral solution to the system of constrains consisting of conjunction of the equality of the subscrpts, the iteration domains domains and the contexts. It is much harder to compare the numbers of integral solutions that the different systems have. This question amounts to the well known problem of counting integral points in polyhedra. There are exact solutions when the parameters have fixed values [4, 7]. When there is just one unfixed parameter, it is still possible to compare the parametric numbers [7]. But in the general case, the use of heuristics is needed.

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. Because the input problem is a static control program, the execution domain of each statement can be represented as a polyhedron [14]. 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. This problem was first solved by Ancourt and Irigoin [1] for the simple case of Z-polyhedra with unit lattice. They used the Fourier-Motzkin elimination technique to compute loop bounds. For more complex situation, the best solution is the Quilleré et al. one [18]. Their technique generates each loop level by separating the polyhedra until they are disjoint on the current dimension, then recursively generating loop nests that scan each of them and lastly sorting polyhedra in order to respect the execution order. This method is well adapted to the chunking problem provided we generalize it somewhat. We have to deal with sequential inner loops, and we have to optimize the code in the case of imperfect loop nests. Our resulting code is quite efficient.

Example Let us continue the example of section 3.3. The polyhedra describing the execution domains of $S1$ and $S2$ result from the study of the original code. One complete them with the chunking dimension c and the chunking constraints. The constraint systems describing the iteration domains are:

$S1$ constraint system

$$\begin{cases} c - i = 0 \\ -i + n \geq 0 \\ i - 1 \geq 0 \end{cases}$$

$S2$ constraint system

$$\begin{cases} c - j - n = 0 \\ -i + n \geq 0 \\ i - 1 \geq 0 \\ -j + m \geq 0 \\ j - 1 \geq 0 \end{cases}$$

On the first dimension c, polyhedra are already disjoint: the first one covers $1 \leq c \leq n$ while the second one covers $n+1 \leq c \leq n+m$, hence there is no need to separate or aggregate them. As a consequence, there will be one loop nest per statement; the recursion on each of them is then trivial. Lastly, we must order the loop nests in such a way that they respect the execution order. It is easy to see that the first polyhedron must precede the second one. The resulting code is the one shown in figure 1(b) as the target program. ■

5. IMPLEMENTATION AND RESULTS

From the chunking function calculation to the code generation, our method is completely automated. The *chunky* prototype implements the full process in C except for the dependence calculation and application of the Farkas lemma which it still uses a Maple solver. The dependence correction and code generation make an intensive use of polyhedral operations, as implemented in the Polylib[1] [19] and PIP[2] [9].

This prototype allows us to test various 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. L1 is a 4-way set associative cache

[1]The Polylib is freely available under GNU license at `http://icps.u-strasbg.fr/PolyLib`

[2]PIP is freely available under the GNU public license at `http://www.prism.uvsq.fr/~paf`

with a miss penalty of 3 cycles. L2 is an 8-way set associative cache with a miss penalty of 44 cycles. Both cache levels are non-blocking and have a line size of 32 bytes. To make the best evaluations, we choose to use the hardware counters of the Pentium III to compare the number of cache misses [5]. Figure 4 shows the evolutions of the number of cache misses for the original and target versions of the running example (see figure 1), according to the value of the parameter m. The ratio m/n is set to 64 in order to better

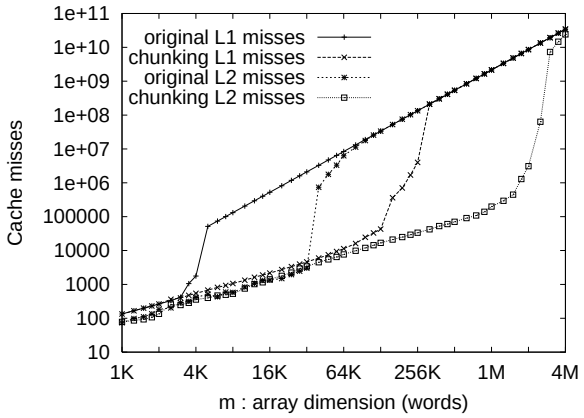

Figure 4: Cache misses for the running example

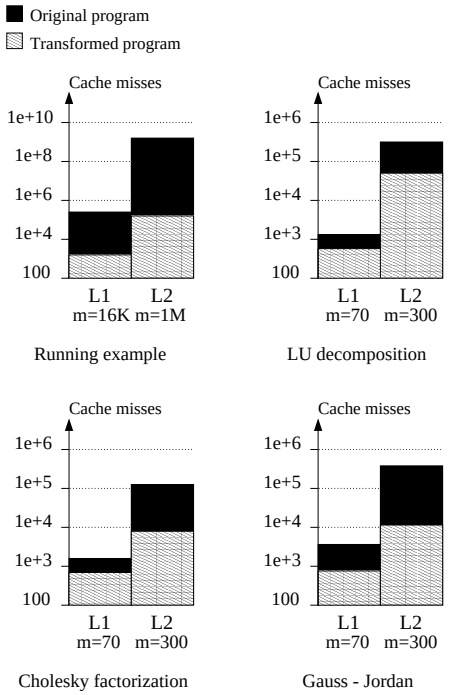

Figure 5: Experimental results (log. scale)

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. Since chunking reorders operations, it can influence the spatial locality. For better comparison, we have selected by hand the layout giving the best results. This improvement has been applied both to the original program and to the chunked code. We plan to automate this task in the near future. The results are presented with logarithmic scale. As for the running example, chunking can reduce the number of cache misses by more than one order of magnitude. For instance, chunking cuts down the L1 cache misses of a Cholesky factorization on $70 * 70$ arrays by 63% and L2 cache misses by 92% on $300 * 300$ arrays. Since the number of cache misses is one of the main factors of energy dissipation [6], this cache miss reduction implies a significant improvement. At the same time, performance can be increased: in the Cholesky factorization case, we obtain a speedup of 35% for $300 * 300$ arrays and of some percents for $70 * 70$ arrays. The performance/energy rate is then greatly improved.

Despite the high theoretical complexity of many of our methods (for instance, parametrized linear programming solvers, polyhedral manipulations, and code generator have exponential complexities), the prototype seems to offer good performance. The reason is that the main parameters are loop nest depths and array dimension numbers which are usually small. To give an idea, the chunking of a Cholesky factorization with 7 statements, a maximal loop nest depth of 3 and a maximal array dimension number of 2 requires about 20 seconds on the test machine. Most of the time is spent in Maple code and we have many reasons to think that a better implementation will significantly improve the prototype performance. Nevertheless, the question of scalability remains, and will be tested on a larger benchmark suite.

6. RELATED WORK

The effort of research to create effective locality optimizing compilers began with Wolf and Lam [20] 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 [21] 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). Li [15] generalizes the framework of unimodular matrices [3] 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. [17] 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* [12], 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, and in addition, both [20] and [17] deal with spatial reuse while we don't. Alternatively to these control centric techniques, Kodukula et al. [13] 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 [13]. 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, it is not based on heuristics and the proposed transformation can't spoil the temporal locality, in the worst case it leaves the original code intact. Next, it can be applied to any static control program without other limitations. Lastly, there is no requirement on dependences and it is often possible to make a transformation legal without decreasing its quality. The method is completely automated, and requires nothing besides the original code but the relative sizes of the cache and data. The proposed optimizations remain stable for large size variations.

First results are very encouraging. Nevertheless, there remain several kinds of problems for which we need to extend our method before we can compete with the classic optimization techniques. We are currently working on tiling which seem to be the natural continuation of our approach. Intuitively, tiling is a question of aggregating small chunks or splitting big ones. We must also deal with spatial locality improvement. A step in that direction is the work of Loechner, Meister and Clauss [16], which is based on precise counting of memory accesses. Lastly, we must deal with programs which have static control regions but have not static control *in toto*. Our method can be adapted to local memories (or software managed caches) at the price of more attention to footprint layout.

8. REFERENCES

[1] C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In *3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, june 1991.

[2] 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.

[3] U. Banerjee. Unimodular transformations of double loops. pages 192–219, Irvine, august 1990.

[4] A. Barvinok. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Mathematics of Operations Research*, 19(4):769–779, november 1994.

[5] R. Berrendorf and H. Ziegler. PCL - The Performance Counter Library: a common interface to access hardware performance counters on microprocessors. Technical Report FZJ-ZAM-IB-9816, Forschungszentrum Jlich, 1998.

[6] F. Cathoor, S. Wuytack, and al. *Custom memory managament methodology*. Kluwer Academic Publishers, 1998.

[7] P. Clauss. Handling memory cache policy with integer points counting. In *Euro-Par'97 European Conference on Parallel Processing*, pages 285–293, Passau, august 1997.

[8] 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.

[9] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[10] P. Feautrier. Dataflow analysis of scalar and array references. *International Journal of Parallel Programming*, 20(1):23–53, february 1991.

[11] 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.

[12] 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.

[13] 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.

[14] D. Kuck. *The Structure of Computers and Computations*. John Wiley & Sons, Inc., 1978.

[15] W. Li. *Compiling for NUMA parallel machines.* PhD thesis, Cornell University, 1993.

[16] V. Loechner, B. Meister, and P. Clauss. Precise data locality optimization of nested loops. *Journal of Supercomputing*, 21(1):37–76, january 2002.

[17] 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.

[18] 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.

[19] D. Wilde. A library for doing polyhedral operations. Technical report, IRISA, 1993.

[20] 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.

[21] M. Wolfe. Iteration space tiling for memory hierarchies. In *3rd SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, december 1987.

[22] J. Xue. Transformations of nested loops with non-convex iteration spaces. *Parallel Computing*, 22(3):339–368, 1996.