The Journal of Supercomputing, 21, 37–76, 2002

Precise Data Locality Optimization of Nested Loops

VINCENT LOECHNER, BENOÎT MEISTER AND
PHILIPPE CLAUSS {loechner, meister, clauss}@icps.u-strasbg.fr

ICPS/LSIIT, Université Louis Pasteur, Strasbourg, Pôle API, Bd Sébastien Brant, F-67400 Illkirch France

Abstract. A significant source for enhancing application performance and for reducing power consumption in embedded processor applications is to improve the usage of the memory hierarchy. In this paper, a temporal and spatial locality optimization framework of nested loops is proposed, driven by parameterized cost functions. The considered loops can be imperfectly nested. New data layouts are propagated through the connected references and through the loop nests as constraints for optimizing the next connected reference in the same nest or in the other ones. Unlike many existing methods, special attention is paid to TLB (Translation Lookaside Buffer) effectiveness since TLB misses can take from tens to hundreds of processor cycles. Our approach only considers active data, that is, array elements that are actually accessed by a loop, in order to prevent useless memory loads and take advantage of storage compression and temporal locality. Moreover, the same data transformation is not necessarily applied to a whole array. Depending on the referenced data subsets, the transformation can result in different data layouts for a same array. This can significantly improve the performance since *a priori incompatible* references can be simultaneously optimized. Finally, the process does not only consider the innermost loop level but all levels. Hence, large strides when control returns to the enclosing loop are avoided in several cases, and better optimization is provided in the case of a small index range of the innermost loop.

Keywords: data locality, memory hierarchy, cache performance, TLB effectiveness, spatial and temporal locality, loop nests, parameterized polyhedron, Ehrhart polynomial

1. Introduction

Efficient use of memory resources is a significant source for enhancing application performance and for reducing power consumption for embedded processor applications [19]. Nowadays, computers include several levels of memory hierarchy in which the lower levels are large but slow (disks, memory, ...) while the higher levels are fast but small (caches, registers, ...). Hence, programs should be designed for the highest percentage of accesses to be made to the higher levels of memory. To accomplish this, two basic principles have to be considered, both of which are related to the way physical cache and memory are implemented: spatial locality and temporal locality. Temporal locality is achieved if an accessed memory location is accessed again before it has been replaced. Since a cache miss or page fault for a single data element will bring an entire cache line into cache or page into main memory, if several closely located memory locations are accessed before the cache line or page is replaced, then spatial locality is achieved. Taking advantage of spatial and temporal locality translates to minimizing cache misses,

TLB (translation lookaside buffer) misses and page faults, and thus increases performance.

The TLB usually holds only a limited number of pointers to recently referenced pages. On most computers, this number ranges between 8 and 256. Few microprocessors have a TLB reach larger than the secondary cache when using conventional page sizes. Many microprocessors like the MIPS R12000/R10000, Ultrasparc II and PA8000 use selectable page sizes per TLB entry in order to increase the TLB reach. For example, the MIPS R10000 processor TLB supports 4K, 16K, 64K, 256K, 1M, 4M and 16M page sizes per TLB entry, which means the TLB reach ranges from 512K if all entries use 4K pages to 2G if all entries use 16M pages. Unfortunately, while many processors support multiple page sizes, few operating systems make full use of this feature [21]. Current operating systems usually use a fixed page size.

Since a TLB miss can take from tens to hundreds of cycles, the use of large-stride data accesses must be refrained. For example, a stride of the same size as a page can generate a TLB miss every access. In order to minimize cache misses, data should ideally be processed sequentially as it is stored in memory, or conversely, should be stored in the same order as it is processed. Arrays that are allocated to memory in a row-major order in C and in a column-major order in Fortran can be processed with stride-one, taking ideally advantage of spatial locality.

Many recent works have provided advances in loop and data transformation theory. By using affine representation of loops, several loop transformations have been unified into a single framework using a matrix representation of these transformations [25]. These techniques consist either in unimodular [3] or non-unimodular [15] iteration space transformations as well as tiling [13, 23, 24]. Although there has been less attention paid to data transformations, there have been some investigations of the effect of data transformations combined with loop transformations. However, the only data layouts that have been considered are row-major or column-major storage [2, 6, 11] and data transformations have been restricted to be unimodular [14].

More recently, Kandemir et al. [12] and O'Boyle and Knijnenburg [17] have proposed a unifying framework for loop and more general data transformations. In [17], the authors propose an extension to nonsingular data transformations. Unfortunately, these approaches do not use any symbolic analysis in order to evaluate the array sizes, the amount of reused data, nor the number of iterations, and do not derive symbolic transformations. Moreover, the derived data transformations only consider the innermost loop level and only a single data layout is associated with each array. Such pre-determined data layouts favor one axis of the index space over the others and adjacent data in the unfavored directions become distant in memory [5, 21]. Hence, such layouts can yield large strides generating TLB misses when control returns to an enclosing loop, and also generate many useless cache loads. In the same way, loop transformations that are dedicated to temporal locality optimization in the innermost loop for some references, do not consider the other references even though temporal localization of reused data sequences could have been made in outer loop levels, resulting in significant savings in cache and TLB misses.

Additionally, when a given data layout is determined by optimizing one reference in a loop, this layout is propagated to all the other references to the same array in the program. Other loop transformations may be needed to take advantage of this layout. But such transformations may not be valid due to data dependences or may be incompatible with other references. We show that in such cases, it can still be possible to improve spatial locality if there is a significant subset of iterations referencing array elements that are not referenced in the originally optimized loop.

In this paper, transformations are expressed as a function of several parameters such as the size of the data set or the parameterized loop bounds. Spatial locality is improved by computing new access functions to array elements. Such a function translates to a parameterized data layout depending on the loop bounds or on the array sizes. When it has been determined beneficial, different access functions may be associated with disjoint data subsets for a given array. Since arrays are allocated to memory as one-dimensional arrays, they are directly represented that way in the transformed program. Temporal locality is improved by transforming the iteration space of a loop that can be imperfectly nested in such a way that the innermost loops will access the same array element. If such optimization is not possible for all references, then temporal localization of reused data sequences is made at some outer loops level.

Our techniques use the polytope model [10] with an intensive use of its parametric extensions [9] and of their implementations in the polyhedral library *PolyLib* [22].

Some representative examples are presented in Section 2, where some "classically" optimized loop nests are compared with loop nests optimized using our approach. Section 3 introduces the general model of loop nests and references. Then we present in Section 4 our loop transformation framework, dedicated to temporal locality optimization and to previously defined data layout adequacy. Our data layout transformation technique is presented in Section 5. Both Sections 4 and 5 are organized by first presenting the case of one unique reference to a given array and then presenting the more general case of several references. The global optimization algorithm driven by parameterized loop cost functions is detailed in Section 6, while the consequences resulting from our optimizations on parallelized code are discussed in Section 7. Finally, conclusions and future objectives are given in Section 8.

2. Motivation of our approach

Some motivating examples are given in this section, showing the usefulness of our approach. Measurements have been made on a 300 MHz R12000 processor with a 32 KB L1 data cache, a 8 MB L2 unified cache and a TLB of 1024 entries tracking 4 MB of memory. Both caches are two-way set-associative and nonblocking. We used the performance evaluation tool `perfex` which uses hardware registers to count the number of primary and secondary cache misses and the number of TLB misses occurring during an execution. It is shown that loops and references that one would consider as sufficiently optimized can still be significantly improved.

The original and the transformed versions were compiled with the `-O3` option. Unfortunately, due to its limited optimization range, the compiler is unable to unroll loops, to take advantage of blocking or data prefetch when loop bounds are not constant and references are not affine. We contend that such automatic optimizations are still possible for such cases and we expect future compilers (in particular for EPIC processors) to be able to generate them. In any case, the same optimizations that are made in the original program can be made in the transformed program, giving a similar improvement in efficiency. Hence, from the perspective of an implementation of the method within a compiler, such optimizations could be done before our transformations. In order to get comparable executable codes, the options `-LNO:ou=1:prefetch=0:blocking=off` were added to the compiling command line, to prevent loop unrolling, prefetching and blocking in both versions.

In examples given below, the transformations are applied by considering references to a given array A. For each loop, we consider that a complete optimization is made relative to the whole program. The global program optimization process is presented in Section 6. Performance results were obtained by launching programs with instantiated loop bounds and array sizes. The choice of these values is determined by the available memory size and to the wish for a reasonable computation time. Performance improvements obviously increase as these values become larger.

Let us look at the first example in Figure 1. This example is representative of cases where the innermost loop is very small compared to the enclosing loop, and the number of referenced array elements in the innermost loop is also very small compared to the array size. The original loop seems to be well optimized since $A(j, i)$ is accessed with stride-one in the innermost loop. But when index i is incremented, the next array element is accessed with stride $N - 4$. Such a situation can generate a TLB miss at each iteration of the enclosing loop and also generate cache misses since useless data are loaded and occupying cache lines that could have been

Original or traditionally transformed program	Our approach

`real A(N,N),Y`
`.`
`.`
`do i = 1,N`
`  do j = 1,4`
`  Y=Y+A(j,i)`
`  enddo`
`enddo`
`.`
`.`
	`real A(N*N),Y`
	`.`
	`.`
	`do i = 1,N`
	`  do j = 1,4`
	`  Y=Y+A(4*(i-1)+j)`
	`  enddo`
	`enddo`
	`.`
	`.`

for N=8000, # L1 data cache misses: 8,196 # L2 data cache misses: 7,828 # TLB misses: 8,337	for N=8000, # L1 data cache misses: 4,200 # L2 data cache misses: 1,017 # TLB misses: 8

Figure 1. First motivating example.

loaded with active data. For this loop, we perform the data transformation defined by:

$$A(i, j) \rightarrow A(i + 4(j - 1)) \qquad \text{for any } 1 \leq i \leq 4 \text{ and } 1 \leq j \leq N$$

The resulting data layout is ideal since data are stored in memory in the same order as they are accessed. The measurements show an important decline of TLB misses, more than 7 times less secondary data cache misses and half fewer primary data cache misses.

The second example in Figure 2 is representative of cases where incompatible references occur in a loop. Traditional optimization processes would choose to consider just one of them. The resulting loop would cause useless cache loads, TLB misses and non-local accesses through the reference that was not optimized. But a further analysis shows that the data that are accessed by each reference are not the same: the two sets of referenced data are disjoint and array A can be allocated to memory using two different data layouts. We derive and perform the data transformations defined by:

$$A(i, j) \rightarrow A(i(i - 1)/2 + j) \qquad \text{for any } 1 \leq i \leq N \text{ and } 1 \leq j \leq i$$

$$A(i, j) \rightarrow A(i + j(j - 1)/2 + N(N + 1)/2 + 1) \qquad \text{for any } \begin{cases} 1 \leq j \leq N \\ 0 \leq i < j \end{cases}$$

Afterwards, both references yield local accesses in the resulting loop. The measurements show an impressive decline of the number of TLB misses and primary data cache misses.

In the third example in Figure 3, the data dependences prevent a loop interchange that could optimize references to array A. Hence, traditional techniques will result

Original or traditionally transformed program	Our approach

`real A(0:N,N),Y`
`.`
`.`
`do i = 1,N`
`  do j = 1,i`
`  Y=Y+A(i,j)+A(j-1,i)`
`  enddo`
`enddo`
`.`
`.`
	`real A(N*(N+1)),Y`
	`.`
	`.`
	`do i = 1,N`
	`  do j = 1,i`
	`  Y=Y+A(i*(i-1)/2+j)`
	`  +A(i*(i-1)/2+j+N*(N+1)/2)`
	`  enddo`
	`enddo`
	`.`
	`.`

for N=8000, # L1 data cache misses: 35,947,086 # L2 data cache misses: 3,225,567 # TLB misses: 32,500,787 computation time: 9.48s	for N=8000, # L1 data cache misses: 8,011,091 # L2 data cache misses: 1,998,885 # TLB misses: 8,813 computation time: 0.88s

Figure 2. Second motivating example.

Original or traditionally transformed program	Our approach
`real A(N,N),X(0:N,N+1)`	`real A(N*N),X(0:N,N+1)`
`.`	`.`
`.`	`.`
`do i = 1,N`	`do i = 1,N`
`  do j = 1,i`	`  do j = 1,i`
`    X(i,j)=A(i,j)+`	`    X(i,j)=A(i*(i-1)/2+j)`
`    X(i-1,j+1)+X(i-1,j-1)`	`    +X(i-1,j+1)+X(i-1,j-1)`
`  enddo`	`  enddo`
`enddo`	`enddo`
`.`	`.`
`.`	`.`
`do i = 1,N`	`do i = 1,N`
`  do j = 1,i`	`  do j = 1,i`
`    X(i,j)=X(i,j)+A(j-1,i)`	`    X(i,j)=X(i,j)`
`    -X(i-1,j+1)+X(i-1,j-1)`	`    +A(i*(i-1)/2+j+N*(N+1)/2)`
`  enddo`	`    -X(i-1,j+1)+X(i-1,j-1)`
`enddo`	`  enddo`
`.`	`enddo`
`.`	`.`
	`.`
for N=5000,	for N=5000,
# L1 data cache misses: 29,549,526	# L1 data cache misses: 18,492,854
# L2 data cache misses: 1,641,311	# L2 data cache misses: 736,586
# TLB misses: 16,305,769	# TLB misses: 8,299,875
computation time: 5.06s	computation time: 2.70s

Figure 3. Third motivating example.

in optimizing just one reference at best. But, as in the previous example, both referenced data sets are disjoint, and two different data layouts can be derived:

$$A(i, j) \rightarrow A(i(i-1)/2 + j) \qquad \text{for any } 1 \leq i \leq N \text{ and } 1 \leq j \leq i$$

$$A(i, j) \rightarrow A(i + j(j-1)/2 + N(N+1)/2 + 1) \qquad \text{for any } \begin{cases} 1 \leq j \leq N \\ 0 \leq i < j \end{cases}$$

The measurements show about half fewer cache and TLB misses for the transformed loops.

The fourth example in Figure 4 exhibits two references that any traditional technique would not transform at all. However, we show that some misses can still be avoided. Unfortunately, both references are accessing data sets that are not disjoint: about half of array A is accessed by both references and the other half is only accessed by the second reference. Hence, this second half can take advantage of a better data layout. In such a case, our optimization process results in splitting the initial loop into two disjoint loops. The second half of the array is only accessed by

Original or traditionally transformed program	Our approach

`real A(2*N,N),Y`
`.`
`.`
`do i = 1,N`
`  do j = 1,N`
`  do k = 1,N`
`  Y=Y+A(k,i)+A(j+k,j)`
`  enddo`
`  enddo`
`enddo`
`.`
`.`
	`real A(2*N*N),Y`
	`.`
	`.`
	`do i = 1,N`
	`  do j = 1,N`
	`  do k = 1,N-j`
	`  Y=Y+A(N*(i-1)+k)`
	`  +A(N*(j-1)+j+k)`
	`  enddo`
	`  do k=N-j+1,N`
	`  Y=Y+A(N*(i-1)+k)`
	`  +A(j+k+(j*j-j)/2+N*N)`
	`  enddo`
	`  enddo`
	`enddo`
	`.`
	`.`

for N=2000, # L1 data cache misses: 1,009,841,909 # L2 data cache misses: 211,242,923 # TLB misses: 2,125,966 computation time: 145.45s	for N=2000, # L1 data cache misses: 1,007,564,287 # L2 data cache misses: 105,839,497 # TLB misses: 1,565,058 computation time: 111.88s

Figure 4. Fourth motivating example.

the second reference in the second loop. This transformation is fully explained in Section 5.3. The new data layouts are defined by:

$$A(i, j) \to A(i + N(j - 1)) \qquad \text{for any } 1 \le i \le N \text{ and } 1 \le j \le N$$

$$A(i, j) \to A(i(i - 1)/2 + j + N^2) \qquad \text{for any } \begin{cases} N + 1 \le i \le 2N \\ 1 \le j \le N \end{cases}$$

The measurements show a significant improvement in the number of TLB and L2 data cache misses.

One might think that the original loop of the example in Figure 5 is already optimized: stride-one accesses occur in the innermost loop for array A and references to array B take advantage of temporal and spatial locality. But this does not take into account an expensive phenomenon occurring with reference $A(j, k)$: since a high number of accessed data yields some unavoidable TLB misses, these misses are repeated each time index i is incremented. Hence, this reference needs to be temporally optimized prior to $B(i)$, in order to minimize the TLB misses. This optimization is done by interchanging loops such that index i is incremented in the innermost loop. Stride-one accesses still occur for array A with both enclosing loops. The new data layout is defined by:

$$A(i, j) \to A((i - 1)(i - 2)/2 + j) \qquad \text{for any } 1 \le i \le N \text{ and } 1 \le j \le i - 1$$

These experiments have shown an impressive reduction in the number of misses.

Original or traditionally transformed program	Our approach

`real A(N,N),B(N),Y`
`.`
`.`
`do i = 1,N`
`  do j = 1,N`
`  do k = 1,j-1`
`  Y=Y+A(k,j)+B(i)`
`  enddo`
`  enddo`
`enddo`
`.`
`.`
	`real A(N*N),B(N),Y`
	`.`
	`.`
	`do j = 1,N`
	`  do k = 1,j-1`
	`  do i = 1,N`
	`  Y=Y+A((j-1)*(j-2)/2+k)`
	`  +B(i)`
	`  enddo`
	`  enddo`
	`enddo`
	`.`
	`.`

for N=2000, # L1 data cache misses: 501,507,202 # L2 data cache misses: 68,387,761 # TLB misses: 1,039,018 computation time: 49.94s	for N=2000, # L1 data cache misses: 258,821 # L2 data cache misses: 59,696 # TLB misses: 315 computation time: 26.78s

Figure 5. Fifth motivating example.

In the sixth example (Figure 6), the original loop seems to be well stated: temporal locality occurs in the innermost loop for reference $A(j, i)$ and stride-one access occurs for reference $B(k, j)$. Even in the outer loops, stride-one accesses occur for reference $A(j, i)$ in loops j and i. But in this version, no attention is paid to temporal reuse generated by reference $B(k, j)$, since it is not possible to opti-

Original or traditionally transformed program	Our approach

`real A(N,N),B(N,N),Y`
`.`
`.`
`do i = 1,N`
`  do j = 1,N`
`  do k = 1,N`
`  Y=Y+A(j,i)+B(k,j)`
`  enddo`
`  enddo`
`enddo`
`.`
`.`
	`real A(N,N),B(N,N),Y`
	`.`
	`.`
	`do j = 1,N`
	`  do i = 1,N`
	`  do k = 1,N`
	`  Y=Y+A(j,i)+B(k,j)`
	`  enddo`
	`  enddo`
	`enddo`
	`.`
	`.`

for N=2000, # L1 data cache misses: 1,000,529,635 # L2 data cache misses: 207,921,082 # TLB misses: 1,054,908 computation time: 123.97s	for N=2000, # L1 data cache misses: 4,571,962 # L2 data cache misses: 249,337 # TLB misses: 1,051,019 computation time: 53.91s

Figure 6. Sixth motivating example.

mize both references simultaneously in the innermost loop. We argue that temporal optimization for reference $B(k, j)$ has to be considered at the next loop level: loops i and j are interchanged in order to optimize the temporal reuse of the smallest possible data sequence which is $B(1, j), B(2, j), \ldots, B(N, j)$. The measurements we made show significant savings in the number of cache misses.

The access functions generated by our technique are not affine. They are multivariate polynomials of the loop indices. One could think that the evaluations of such polynomials induce a computation overhead that will significantly slow down the program. But our many experiments have shown that this is not a significant issue at all: compilers transform any access function by computing an increment of the referenced variable index at each loop level. Such an increment is generally of low complexity and is often a constant integer value. It is equal to one when spatial locality is achieved.

3. Background

The iteration space of a loop nest of depth d is a d-dimensional convex polyhedron D where each point is denoted by a d-vector $I = (i_1, i_2, \ldots, i_d)$. Each i_k denotes a loop index with i_1 as the outermost and i_d the innermost loop. We will use $(i_1, i_2, \ldots, i_d)$ to denote an iteration as well as a point in the iteration space.

We denote by $l_m(i_1, i_2, \ldots, i_{m-1})$ (respectively $u_m(i_1, i_2, \ldots, i_{m-1})$) the lower (resp. the upper) bound for the loop of depth $m, m \leq d$. Such bounds are defined by parametric affine functions of the enclosing loop indices. They are of the form:

$$a_1 i_1 + a_2 i_2 + \cdots + a_{m-1} i_{m-1} + b_1 p_1 + b_2 p_2 + \cdots + b_q p_q + c$$

where $a_1, a_2, \ldots, a_{m-1}, b_1, b_2, \ldots, b_q, c$ are rational constants and $p_1, p_2, \ldots, p_q$ are integer parameters. Thus, the polyhedron corresponding to the iteration space is bounded by parameterized linear inequalities imposed by the loop bounds. The iteration space will be denoted by D_P with $P = (p_1, \ldots, p_q)$.

We assume that loops are normalized such that their step is one. A reference to an array element is represented by the pair (R, o) where R is the access matrix and o is the offset vector. Elements of R and o are integer numbers. A reference is an affine mapping $f(I) = RI + o$. For a reference to an m-dimensional array inside an n-dimensional loop nest, the access matrix is $m \times n$ and the offset vector is of size m. We are also able to handle parameterized references. Therefore elements of o can be affine combinations of integer parameters. References can occur in any loop of the nest since imperfectly nested loops are considered.

4. Loop transformations for temporal locality

This section describes how to perform temporal optimization given one loop. Each reference in the loop is associated with an accessed set of data, and all references are ordered by decreasing data set sizes. This choice of sorting criterium is explained in Section 6.

Notice that each data accessed by a temporally optimized reference can be stored in a register during the execution of the optimized inner loops. If accessed in read/write or read only, the data is stored in a register at the beginning of the temporal usage, and if accessed in write, it is written to memory at the end of the temporal usage.

Temporal locality is achieved by applying a transformation to the original loop. In this paper, we consider unimodular transformations, being equivalent to any combination of loop interchange, skewing and reversal (see [25] for references). In order to be valid, the transformation has to respect the dependences of the loop. This is described in subsection 4.1. An algorithm to compute the transformation matrix for achieving temporal locality is given in subsection 4.2. This subsection also describes the additional constraints that have to be considered in order to take advantage of the spatial organization of an array accessed in another loop.

4.1. Definitions and prerequisites

4.1.1. Unimodular transformations. Let T be a unimodular $(d+q+1) \times (d+q+1)$ matrix. This matrix defines a homogeneous[1] affine transformation t of the iteration domain as:

$$t: \quad D_P \subset \mathbb{Z}^d \quad \rightarrow \quad D'_P = t(D_P) \subset \mathbb{Z}^d$$

$$I \quad \mapsto \quad I' = t(I) \qquad \text{such that} \quad \begin{pmatrix} I' \\ P \\ 1 \end{pmatrix} = T \begin{pmatrix} I \\ P \\ 1 \end{pmatrix}$$

The transformed domain D'_P corresponds to a new scanning loop, obtained by applying the Fourier-Motzkin algorithm [4] to a perfectly nested loop. This algorithm computes the lower and upper bounds of each iteration variable I'_k as a function of the parameters and the variables $I'_1 \cdots I'_{k-1}$ only. The body of the loop also has to be transformed in order to use vector I': all references to vector I are replaced by $t^{-1}(I')$.

If the loop nest is not perfect then some more work has to be done. First, we have to compute the set of iteration domains corresponding to the original loop nest. All these iteration domains have to be defined in the same geometric space (same dimension and same variables). This can be done by using a variant of code sinking [25] presented in the following examples. Then, the transformation is applied to all these iteration domains. Finally, to get the resulting loop nest we apply Quilleré's algorithm [18], which constructs an optimized loop nest scanning several iteration domains simultaneously.

Example 1 The following loop nest, on the left of Figure 7, is an imperfect loop nest containing an instruction outside the innermost loop. In order to build the iteration domains, this instruction `S1` has to be put into the same depth of the loop nest. This is done by creating a new loop, containing only one iteration, to include instruction `S1`. The bounds of this new loop is the lower bound of the innermost loop minus 1. The resulting loop is shown on the right of the figure.

Original loop	Transformed loop

do i = 1, N
  do j = 1, N
  S1
  do k = 1, N
  S2
  enddo
  enddo
enddo
	do i = 1, N
	  do j = 1, N
	  do k = 0, 0
	  S1
	  enddo
	  do k = 1, N
	  S2
	  enddo
	  enddo
	enddo

Figure 7. Transforming a loop into a perfect loop nest, first example.

The final iteration domain is the union of the two iteration domains corresponding to each instruction, since they are disjoint. In this example, it is the convex domain:

$$D_N = \left\{ \begin{pmatrix} i \\ j \\ k \end{pmatrix} \in \mathbb{Z}^3 \;\middle|\; \begin{array}{l} 1 \le i \le N \\ 1 \le j \le N \\ 0 \le k \le N \end{array} \right\}$$

Example 2 In this second example Figure 8, we have two innermost non disjoint loop nests. In order to build two disjoint iteration domains, we have to shift one of the loops. Let us call l_0 and u_0 the lower and upper bounds for the first innermost loop, and l_1 and u_1 the bounds for the second innermost loop. For the two innermost loops to be disjoint, we can for example shift the second loop nest by $u_0 - l_1 + 1$, so

Original loop	Transformed loop

do i = 1, N
  do j = 1, N
  do k = 1, i
  S1
  enddo
  do k = j, N
  S2
  enddo
  enddo
enddo
	do i = 1, N
	  do j = 1, N
	  do k' = 1, i
	  k = k'
	  S1
	  enddo
	  do k' = i+1, N+i-j+1
	  k = k'-i+j-1
	  S2
	  enddo
	  enddo
	enddo

Figure 8. Transforming a loop into a perfect loop nest, second example.

this loop starts at $u_0 + 1$. In this example, we shift the second loop nest by $i - j + 1$:

The final iteration domain is the union of the two disjoint iteration domains:

$$D_N = \left\{ \begin{pmatrix} i \\ j \\ k' \end{pmatrix} \in \mathbb{Z}^3 \middle| \begin{array}{l} 1 \le i \le N \\ 1 \le j \le N \\ 1 \le k' \le N + i - j + 1 \end{array} \right\}$$

4.1.2. Data dependences and validity of loop transformations. From this point, by 'dependence' we will mean flow, anti-, and output dependences only. Input dependence does not play a role, since distinct reads of the same memory location can be done in no particular order. We denote by $\prec$ the lexicographical *lower than* operator and we define in the same way the operators $\preceq$, $\succ$, and $\succeq$. In the following, we denote by $v^{\perp}$ the transpose vector of v.

Let $\mathscr{D}$ be the set of *distance vectors* related to data dependences occurring in the original loop nest:

$$\delta \in \mathscr{D} \Leftrightarrow \begin{array}{l} \exists\, I, J \in D_P, \text{ with } I \preceq J, \text{ such that } J = I + \delta, \text{ and there} \\ \text{is a data dependence from iteration } I \text{ to iteration } J. \end{array}$$

Notice that all distance vectors are lexicographically non-negative.

The condition for equivalence between the transformed loop nest and the original loop nest is that [4]: $t(\delta) \succ 0$ for each positive δ in $\mathscr{D}$. Null dependence vectors correspond to loop-independent dependences. A distance vector δ in the original loop nest will become a distance vector $t(\delta)$ in the new one: in order for iteration $J' = t(J)$ to be executed after $I' = t(I)$, vector $t(\delta) = t(J - I) = J' - I'$ must be lexicographically positive.

4.2. Achieving temporal locality

In the first part of this section we describe how to find a transformation matrix T which optimizes one array reference. The second part extends the method to handle two references, to the same or to different arrays. Finally, we give a global algorithm to optimize a loop with multiple references. Our method, compared to other ones, performs an accurate optimization of all references through a step-by-step constructive algorithm.

4.2.1. Optimizing one reference. Let us consider an iteration domain D_P of dimension d, referencing one array through a homogeneous reference matrix R of size $(d' + q + 1) \times (d + q + 1)$, where d' is the dimension of the array and q the number of parameters. There is temporal reuse if the data accessed by the loop has smaller geometric dimension than the iteration domain. As a consequence, in order for the reference to be temporally optimized, the rank r of matrix R has to be lower than $(d + q + 1)$.

The algorithm to find a new scanning loop consists of determining a set of scanning vectors: for each level of the new loop, the associated scanning vector corresponds to the difference between two successive iterations. Let us call B the

matrix composed of these column vectors, outermost loop on the first column, innermost loop on the last one. This unimodular matrix is a basis of the iteration space. Its inverse matrix $T = B^{-1}$ is the transformation matrix to apply to the iteration domain in order to get the new loop, by the Fourier-Motzkin algorithm.

The inner loops have to scan the iterations accessing one data, and the outer loops to scan each of the accessed data. The computation of B consists of two steps: first we compute the basis B_T of the space where temporal reuse occurs, then the basis B_D of the space scanning each accessed data. Finally we have $B = \left(\begin{array}{c|c} (B_D|B_T) & 0 \\ \hline 0 & \mathrm{Id} \end{array}\right)$ in the homogeneous space: B_T corresponds to the inner loops and B_D to the outer loops.

Matrix B_T is computed as follows. The image by matrix R of the iteration space results in a polytope containing all the accessed data, and is called the *data space*. Each integer point d_0 of the data space, or each data, corresponds to a polytope to be scanned by the temporal inner loops. This polytope is computed by applying function preimage[2] R^{-1} to d_0, intersected with the iteration domain D_P. Let us call $\mathcal{H}$ the affine hull of this polytope. The column vectors of matrix B_T are the basis vectors of $\mathcal{H}$.

The leftmost vectors of matrix B scanning the accessed data, *i.e.* matrix B_D, is chosen in order for B to be unimodular and full rank, and to satisfy the remaining dependences. It can be computed using the Hermite normal form of matrix B_T.

If another loop, that has already been optimized, accesses the same array and if the data sets intersect, then spatial organization is constrained. In this case, in order for the innermost loop of the data iteration space to have stride-one accesses, the rightmost column vector of B_D is also constrained: for the innermost loop scanning this data space to access the data in the same order as the other loop, it must scan it along the same direction as the other loop.

Example 3 Consider the following loop nest:

```
do i = 1, N
    do j = 1, N
      do k = 1, i
          A[i+N, j+k-1] = f(i,j,k)
      enddo
    enddo
enddo
```

There is an output dependence for array A, of distance vector $\delta = (0, 1, -1)^{\perp}$. The reference matrix to variable A, is:

$$R = \begin{array}{c} \begin{array}{ccccc} i & j & k & N & 1 \end{array} \\ \left(\begin{array}{ccccc} 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & -1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{array}\right) \end{array} \begin{array}{c} x_0 \\ y_0 \\ N \\ 1 \end{array}$$

Each point $d_0 = \binom{x_0}{y_0}$ of the data space corresponds to the scanning of the following polytope:

$$R^{-1}d_0 \cap D_N = \left\{ \begin{pmatrix} x_0 - N \\ y_0 - k + 1 \\ k \end{pmatrix} \in D_N \right\},$$

where D_N is the iteration domain. The basis vector of the affine hull of this polytope is

$$B_T = \begin{pmatrix} 0 \\ -1 \\ 1 \end{pmatrix}.$$

The dependence is satisfied and B is unimodular by chosing

$$B_D = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}.$$

Finally, we get

$$T = B^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

The transformed loop nest is obtained by applying T to the iteration domain. Let us call $I' = (u, v, w)^{\perp} = t(I)$. The references to I in the loop nest have to be replaced by $t^{-1}(I')$. The resulting loop nest is given below. Notice that index w does not appear in the reference, which means that it has been temporally optimized:

```
do u = 1, N
    do v = 2, N+u
      do w = max(1,v-N), min(u,v-1)
          A[u+N, v-1] = f(u,v-w,w)
      enddo
    enddo
enddo
```

4.2.2. Optimizing two references. Let us consider an iteration domain D_P of size d, referencing two arrays through homogeneous reference matrices R_1 of size $(d_1 + q + 1) \times (d + q + 1)$ and R_2 of size $(d_2 + q + 1) \times (d + q + 1)$. Suppose that the first reference will be optimized prior to the second one. The method for chosing which reference to optimize first will be described in Section 6.

Property 1 *The number of inner loops that can be temporally optimized for two references, is* $d + q + 1 -$ Rank$(\binom{R_1}{R_2})$, *where* d *is the number of loops,* q *the number of parameters, and* R_1 *and* R_2 *the homogeneous reference matrices.*

The demonstration is obvious: the rank of $\binom{R_1}{R_2}$ is equal to the number of required loops to scan both data spaces plus $q+1$ in homogeneous space. Out of d loops, the number of remaining loops where temporal reuse can be realized for both variables is then $d + q + 1 -$Rank$(\binom{R_1}{R_2})$.

An immediate consequence of this property is that both references can be temporally optimized in the inner loops, if and only if the rank of matrix $\binom{R_1}{R_2}$ is lower than $(d + q + 1)$.

The method to build matrix T is similar to the algorithm presented in previous section: it consists of building a basis for the scanning loops represented as matrix B, from right to left. For both references to be temporally optimized in the inner loops, the algorithm selects as inner scanning directions the basis of the intersection of the two affine spaces where temporal reuse occurs. Matrix B is then completed from right to left, using the remaining directions where temporal reuse is optimal for one of the references.

1. Compute the parameterized polytopes scanning the loops where temporal reuse occurs for each reference: $D_i = R_i^{-1} d_i \cap D_P$, where d_i is a point of the data space, for $i = 1, 2$. Compute the affine hulls $\mathcal{H}_1$ and $\mathcal{H}_2$ of these two polytopes.
2. Compute the intersection $\mathcal{H}_1 \cap \mathcal{H}_2$. The basis vectors of this space are used as innermost scanning directions: they will take advantage of temporal reuse for both references.
3. Compute the vectors generating $\mathcal{H}_1 - \mathcal{H}_2$ and $\mathcal{H}_2 - \mathcal{H}_1$. Use these vectors alternatively to generate the next vectors of basis B. Each of these vectors will generate a scanning loop taking advantage of temporal reuse for only one of the references.
4. Complete B by scanning the rest of the iteration space, using the vectors generating the space: $\mathcal{H} - (\mathcal{H}_1 \cup \mathcal{H}_2)$, where $\mathcal{H}$ is the affine hull of D_P.

For basis B to be valid, the unimodularity and the dependences have to be checked. Transformation matrix T is finally obtained as above: $T = B^{-1}$.

Example 4 Consider the following 4-dimensional loop nest.

```
do i = 1, N
    do j = i, N
      do k = 1, N
          do l = 1, N
            A[k+l,j] = A[k+l,j] + B[i,j-i]
          enddo
      enddo
    enddo
enddo
```

There is a dependence for array A, of distance vector $\delta = (0, 0, 1, -1)^{\perp}$. According to Section 6, we choose to optimize temporal reuse for the first reference $A[k+l, j]$, since it has the largest data set ($2N^2 - N$ array elements, versus $\frac{1}{2}(N^2 + N)$ for the second one).
Let

$$R_1 = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{and} \quad R_2 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

$\mathcal{H}_1$ is generated by $(1, 0, 0, 0)^{\perp}$ and $(0, 0, -1, 1)^{\perp}$, and $\mathcal{H}_2$ is generated by $(0, 0, 1, 0)^{\perp}$ and $(0, 0, 0, 1)^{\perp}$.

$\mathcal{H}_1 \cap \mathcal{H}_2$ is generated by vector $(0, 0, -1, 1)^{\perp}$. This is the direction of the innermost scanning loop, where both references are optimized.

We choose as second one, the direction optimizing reference R_1 since the first reference has the largest data set: vector $(1, 0, 0, 0)^{\perp}$. The third one optimizes reference R_2; let us take for example vector $(0, 0, 1, 0)^{\perp}$. The last one will scan the data spaces; let us choose for example $(0, 1, 0, 0)^{\perp}$, in order for B to be full row and unimodular.

Finally we have

$$B = \left(\begin{array}{cccc|cc} 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right),$$

and

$$T = B^{-1} = \left(\begin{array}{cccc|cc} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right).$$

The dependence is satisfied. Let us call the new loop indices u, v, w, x, with $u = j$, $v = k + l$, $w = i$, and $x = l$. The new loop in u, v, w, x optimally scans both

references. The two inner loops access the same array element by the first reference. For the second reference the innermost x-loop accesses the same array element, and the v-loop reuses a subset of the data space corresponding geometrically to a line, scanned by the w-loop.

```
do u = 1, N
    do v = 2, 2N
      do w = 1, u
          do x = max(1,v-N), min(N, v-1)
            A[v,u] = A[v,u] + B[w,u-w]
          enddo
      enddo
    enddo
enddo
```

4.2.3. A global algorithm for multiple references optimization. Let us consider an iteration domain D_P of dimension d, referencing n arrays through reference functions R_i, $1 \leq i \leq n$. The first step of the algorithm is to compute the subspaces $\mathcal{H}_i$ where temporal reuse occurs for each reference i, as described in the previous subsection.

The algorithm consists of building a basis for the scanning loops represented as matrix B, from right to left. The rightmost scanning vector is chosen so that it optimizes as many references as possible. The algorithm then selects the next scanning vectors in order to optimize as many non yet optimized references as possible. In case of equality, the references having the largest data set sizes are chosen. After the initialization, an iterative process generates each scanning vector, from right to left in matrix B:

1. Order the n references in decreasing data set sizes.
2. Compute the linear spaces $\mathcal{H}_i$ for each reference, $1 \leq i \leq n$.
3. for $col = d$ downto 1 do

3a. Find the direction $\mathcal{T}$ that optimizes as many references as possible, in the set of references that have been optimized the least. This is done by computing successive intersections of the subspaces $\mathcal{H}_i$.

If there are no more references to optimize choose a direction such that it takes advantage of the spatial organization of a data if necessary, and such that B is unimodular.

3b. Put the vector $\mathcal{T}$ in the column col of matrix B. Check the dependences and the unimodularity of B; if this is not satisfied, go back to step 3a and choose another direction. Remove $\mathcal{T}$ from the subspaces $\mathcal{H}_i$ so that it will not be considered in a further step.

The unimodularity of B is checked as follows: at each step of the algorithm, the $d - col$ vectors generate a subspace; for the final matrix B to be unimodular, each integer point of this linear space should possibly be generated as an integer combination of the $d - col$ vectors. In other words, these vectors must generate a dense lattice subspace. This condition is verified if and only if the gcd of the

subdeterminants of order $d - col$ of these column vectors is 1. This property is based on a corollary of the Hermite normal form theorem ([20, corollary 4.1c]).

Example 5 Consider the following 4-dimensional loop nest.

```
do i = 1, N
    do j = 1, N
      do k = 1, N
          do l = 1, N
            A[k,j,i] = A[k,j,i] + B[l,j] + B[i,l]
          enddo
      enddo
    enddo
enddo
```

We choose to optimize temporal reuse for reference $A[k, j, i]$, since it has the largest data set (N^3). The two references to B have the same data set size (N^2). Let us call R_1 the reference to $A[k, j, i]$, R_2 and R_3 the references to $B[l, j]$ and $B[i, l]$ respectively. There is only one dependence, on variable A, of distance vector $(0, 0, 0, 1)^{\perp}$.

The first step of the algorithm consists of computing the subspaces of temporal reuse $\mathcal{H}_i$, for $i = 1, 2, 3$. $\mathcal{H}_1$ is generated by vector $(0, 0, 0, 1)^{\perp}$, $\mathcal{H}_2$ is generated by $(1, 0, 0, 0)^{\perp}$ and $(0, 0, 1, 0)^{\perp}$, and $\mathcal{H}_3$ is generated by $(0, 1, 0, 0)^{\perp}$ and $(0, 0, 1, 0)^{\perp}$.

The algorithm then selects four new scanning vectors in this order:

— vector $(0, 0, 1, 0)^{\perp}$ is chosen first, since it optimizes both references R_2 and R_3 (but not R_1).
— vector $(0, 0, 0, 1)^{\perp}$ optimizes then reference R_1 which has not been optimized yet. Notice that reference R_1 cannot be better optimized in a further step, since $\mathcal{H}'_1 = \mathcal{H}_1$.
— vector $(1,\ 0,\ 0,\ 0)^{\perp}$ optimizes reference R_2.
— vector $(0,\ 1,\ 0,\ 0)^{\perp}$ optimizes reference R_3.

Finally, we get:

$$B = \left(\begin{array}{cccc|cc} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right)$$

The dependence is satisfied The resulting loop is obtained by exchanging indices i with j, and k with l:

```
do j = 1, N
    do i = 1, N
      do l = 1, N
          do k = 1, N
            A[k,j,i] = A[k,j,i] + B[l,j]+B[i,l]
          enddo
      enddo
    enddo
enddo
```

Our experimental results show, after spatial optimization in both cases, a huge improvement of performance: for $N = 700$, the number of TLB's is about 2^{32} and overflows the hardware register for the original loop, versus $8.7 * 10^6$ for the transformed loop, on the R12000 processor. The transformed loop is more than 2 times faster than the original one.

5. Data layout transformations of active data

In this section, we present our data transformation method. Its objective is to derive new data layouts for certain references in order to get stride-one access for a maximum number of loop levels. Such a result is obtained by taking into account only data that is actually referenced, that is, the active data. The method is based on a geometric model of loops and array references.

5.1. The active data

Let $Y(d_0)$ be any element of an array Y referenced in a loop nest through a homogeneous reference matrix R. As presented in the previous section, the set of iterations referencing this array element $Y(d_0)$ is defined by:

$$P(d_0) = \left\{ I \in D_P \;\middle|\; R \begin{pmatrix} I \\ P \\ 1 \end{pmatrix} = \begin{pmatrix} d_0 \\ P \\ 1 \end{pmatrix} \right\} = D_P \cap PreImage\left(R, \left\{ \begin{pmatrix} d_0 \\ P \\ 1 \end{pmatrix} \right\}\right)$$

If $P(d_0)$ is empty, then $Y(d_0)$ is never referenced by the loop nest and never has to be loaded in the cache. Depending on how the data storage process is implemented in the compiler, it can be useful, for any given array element $Y(d_0)$, to know if it is effectively accessed and how many times. This information is given by the number of integer points in $P(d_0)$, that is, the *Ehrhart polynomial* $EP(d_0)$ of $P(d_0)$ [7, 9]. An Ehrhart polynomial is a parametric expression of the exact number of integer points contained in a parameterized polyhedron. It can be computed

using our program available at http://icps.u-strasbg.fr/Ehrhart/program/. Hence, if $EP(d_0) = 0$, then $Y(d_0)$ is never referenced by the loop nest.

If $P(d_0)$ is reduced to a single iteration, then array Y has no temporal reuse.[3] Stride-one access for any loop level will then be obtained by indexing any datum with the position of the iteration that references it, relative to the execution order: if the qth iteration references $Y(d_0)$, then $Y(d_0)$ will be mapped to address $b + q \times w$ in main memory, where b is the base address of array Y, and w is the size in bytes of an array element.

If $Y(d_0)$ is temporally reused, two cases can occur:

— The innermost loops have been temporally optimized for the reference under consideration as presented in Section 4: in this case, a new access matrix has been computed which is only applied to the indices of the outermost loops scanning the data space. Stride-one access is then obtained by indexing any datum in the same way by considering only those outermost loops, since temporal optimization resulted in stride-zero access for the remaining innermost loops. This case occurred while optimizing array A of the fifth example in Section 2;
— Not all innermost loops have been temporally optimized for the reference under consideration: In this case, we choose to consider the first iteration referencing $Y(d_0)$ to compute the new data layout, that is, the lexicographic minimum of $P(d_0)$. This minimum is computed using our geometric tools as described in [9]. Then, any datum is indexed in the same way by considering only iterations that reference such lexicographic minima. Since in the case of temporal reuse, the same data sequence is accessed several times, this choice ensures stride-one access each time such a sequence occurs. Moreover, if this data sequence is small enough to be loaded entirely in the cache, then temporal locality will also be achieved. This case will be illustrated with Example 7.

Example 6 In order to show the generality of the technique, let us consider the following example that will result in a rather complicated data layout. It consists in the following loop nest depending on a positive integer parameter N:

```
do i = 1, N
   do j = i, 2*i-1
     do k = i-j, i+j
       X = X + Y(i+j-1, i+k+2, j+k)
     enddo
   enddo
enddo
```

Let us focus on array Y. For the reference $Y(i + j - 1, i + k + 2, j + k)$, we have:

$$R = \begin{pmatrix} 1 & 1 & 0 & 0 & -1 \\ 1 & 0 & 1 & 0 & 2 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

The set of iterations referencing any array element $Y(i0, j0, k0)$ is defined by

$$\begin{aligned} P(d_0) &= P(i0, j0, k0) \\ &= \{(i, j, k) \in D_N | i + j - 1 = i0, i + k + 2 = j0, j + k = k0\} \\ &= \left\{(i, j, k) \in D_N \,\middle|\, i = \frac{i0 + j0 - k0 - 1}{2}, j = \frac{i0 - j0 + k0 + 3}{2}, \right. \\ &\qquad \left. k = \frac{-i0 + j0 + k0 - 3}{2}\right\}. \end{aligned}$$

Observe that for a given datum $Y(i0, j0, k0)$, $P(i0, j0, k0)$ is either empty, when the resulting values of $(i0 + j0 - k0 - 1)/2$, $(i0 - j0 + k0 + 3)/2$ or $(-i0 + j0 + k0 - 3)/2$ are not integer or do not belong to D_N, or contains a single iteration. The number of integer points in $P(i0, j0, k0)$ is:

$$EP(i0, j0, k0) = \begin{cases} 0 & \text{if } (\text{i0+j0+k0}) \mod 2 = 0 \\ 1 & \text{otherwise} \end{cases}$$

For example, since $EP(1, 3, 2) = 0$, $Y(1, 3, 2)$ is never referenced by the loop nest.

The set of active data is defined by:

$$Data_Y = \left\{ d_0 \,\middle|\, \begin{pmatrix} d_0 \\ p \\ 1 \end{pmatrix} = R \begin{pmatrix} I \\ p \\ 1 \end{pmatrix}, I \in D_P \right\} = RD_P$$

These data are associated with the points resulting from the affine transformation R of the iteration space. We have presented in [8] a method allowing the determination of the exact count of these active data. This set is generally not a convex polyhedron containing a regular lattice of points since the affine transformation of a lattice-polytope is not a lattice-polytope: "holes" may occur irregularly.

In our case however, an exact determination of $Data_Y$ is not necessary. The convex hull of $Data_Y$, $Conv(Data_Y)$, is sufficient in order to restrict the result to the only considered useful information. Hence data situated outside this set are assuredly not accessed. For this purpose, we use our parametric vertices finding program presented in [9, 16]. This program computes the parametric coordinates of the vertices of a convex parameterized polyhedron defined by some linear parameterized inequalities. The result is given as a set of convex adjacent domains of the parameters associated with some vertices. In order to compute $Conv(Data_Y)$, we compute the parametric vertices of the parameterized polyhedron $P(d_0)$. Hence, the domains of the parameters computed by the program gives the convex hull of the values taken by d_0, that is, $Conv(Data_Y)$.

Example 6 (*continued*) The set of active data is defined by $Data_Y = \{(i0, j0, k0) | i0 = i + j - 1,\ j0 = i + k + 2,\ k0 = j + k,\ 1 \le i \le N,\ i \le j \le 2i - 1,\ i - j \le k \le$

$i + j\}$. The convex hull of $Data_Y$, $Conv(Data_Y)$, is determined by computing the parametric vertices of $P(i0, j0, k0)$. The program gives the following answer:

$$Conv(Data_Y) = \left\{ \begin{pmatrix} i0 \\ j0 \\ k0 \end{pmatrix} \;\middle|\; \begin{array}{l} i0 + j0 \leq k0 + 2N + 1, \\ 3k0 + 7 \leq i0 + 3j0,\ j0 \leq k0 + 2, \\ i0 + j0 \leq 3k0 + 1,\ j0 + k0 \leq 3i0 + 5 \end{array} \right\}.$$

5.2. *Mapping array elements to memory*

The lexicographic minimum of the set $P(d_0)$ defines the first iteration referencing any array element $Y(d_0)$. If $P(d_0)$ contains one single point, this point defines the unique iteration referencing $Y(d_0)$. Let I_{min} be this point. The coordinates of $I_{min} = (I_{min,1}, I_{min,2}, \ldots, I_{min,n})$ are affine functions of d_0. The position of this iteration, relative to the execution order, gives the position index in main memory of the referenced datum $Y(d_0)$, relatively to the base address b of Y. This will ensure stride-one access since the datum referenced just before or just after $Y(d_0)$ will be contiguous to $Y(d_0)$ in memory.

The position of iteration I_{min} is determined by computing the number of iterations referencing an array element for the first time and occurring before I_{min}. The set of iterations occurring before I_{min} (included) is defined by:

$$P(I_{min}) = \{I \in D_P | I \preceq I_{min}\}$$

where $\preceq$ denotes the lexicographic order. In order to compute the number of iterations in $P(I_{min})$, the lexicographic inequality has first to be transformed into linear inequalities. This transformation will result in a decomposition of $P(I_{min})$ as a union of disjoint convex polyhedra in the following way:

$$P(I_{min}) = P_1(I_{min}) \cup P_2(I_{min}) \cup P_3(I_{min}) \cup \cdots \cup P_n(I_{min})$$

where

$$P_1(I_{min}) = \{I \in D_P | i_1 < i_{min,1}\}$$

$$P_2(I_{min}) = \{I \in D_P | i_1 = i_{min,2}, i_2 < i_{min,2}\}$$

$$P_3(I_{min}) = \{I \in D_P | i_1 = i_{min,1}, i_2 = i_{min,2}, i_3 < i_{min,3}\}$$

$$\ldots$$

$$P_n(I_{min}) = \{I \in D_P | i_1 = i_{min,1}, \ldots, i_{n-1} = i_{min,n-1}, i_n \leq i_{min,n}\}$$

Since I_{min} is defined as an affine function of d_0, $P(I_{min})$ is a union of polyhedra parameterized by d_0. By computing the Ehrhart polynomials $EP_q(d_0)$ of each of these polyhedra, and by summing the results, we obtain the number of iterations defined by $P(I_{min})$ and parameterized by d_0. This final result provides a mapping function of any array element $Y(d_0)$ to main memory, by giving its position index relatively to the base address b of array Y. It is expressed as the Ehrhart polynomial $EP(d_0)$ of the union of polyhedra $P(I_{min})$. Hence, any reference $Y(d_0)$ is transformed into $Y(EP(d_0))$ where the referenced array Y is now a one-dimensional array.

Example 6 (*continued*) The set $P(i0, j0, k0)$ contains at most one point defining a unique iteration referencing any array element $Y(i0, j0, k0)$. This iteration is defined by $I_{min} = ((i0+j0-k0-1)/2,\ (i0-j0+k0+3)/2,\ (-i0+j0+k0-3)/2)$. Hence, the number of iterations occurring before I_{min} is determined by computing the Ehrhart polynomial of the following union of polyhedra:

$$
\begin{aligned}
&P\left(\frac{i0+j0-k0-1}{2}, \frac{i0-j0+k0+3}{2}, \frac{-i0+j0+k0-3}{2}\right)\\
&\quad = \left\{(i,j,k) \in P \;\middle|\; i < \frac{i0+j0-k0-1}{2}, (i0, j0, k0) \in Conv(Data_Y)\right\}\\
&\quad \cup \left\{(i,j,k) \in P \;\middle|\; \begin{array}{l} i = \frac{i0+j0-k0-1}{2}, j < \frac{i0-j0+k0+3}{2}, \\ (i0, j0, k0) \in Conv(Data_Y) \end{array}\right\}\\
&\quad \cup \left\{(i,j,k) \in P \;\middle|\; \begin{array}{l} i = \frac{i0+j0-k0-1}{2}, j = \frac{i0-j0+k0+3}{2}, k \le \frac{-i0+j0+k0-3}{2}, \\ (i0, j0, k0) \in Conv(Data_Y) \end{array}\right\}
\end{aligned}
$$

On the first set, our program gives the following answer:

$$
\begin{aligned}
EP_1(i0, j0, k0) = &-\tfrac{1}{8}k0^3 + \tfrac{3}{8}j0k0^2 + \tfrac{3}{8}i0k0^2 - \tfrac{3}{4}k0^2 - \tfrac{3}{8}j0^2k0 - \tfrac{3}{4}i0j0k0\\
&+ \tfrac{3}{2}j0k0 - \tfrac{3}{8}i0^2k0 + \tfrac{3}{2}i0k0 - \tfrac{11}{8}k0 + \tfrac{1}{8}j0^3 + \tfrac{3}{8}i0j0^2\\
&- \tfrac{3}{4}j0^2 + \tfrac{3}{8}i0^2j0 - \tfrac{3}{2}i0j0 + \tfrac{11}{8}j0 + \tfrac{1}{8}i0^3\\
&- \tfrac{3}{4}i0^2 + \tfrac{11}{8}i0 - \tfrac{3}{4}
\end{aligned}
$$

on the second, the answer is:

$$EP_2(i0, j0, k0) = i0k0 - i0j0 + k0 - j0 + 2i0 + 2$$

and on the third:

$$EP_3(i0, j0, k0) = \tfrac{3}{2}k0 - \tfrac{1}{2}j0 - \tfrac{1}{2}i0 + \tfrac{3}{2}$$

Finally, the memory address of any array element $Y(i0, j0, k0)$ is given by:

$$
\begin{aligned}
&EP(i0, j0, k0) \times w + b\\
&\quad = (EP_1(i0, j0, k0) + EP_2(i0, j0, k0) + EP_3(i0, j0, k0)) \times w + b
\end{aligned}
$$

where b denotes the base address of array Y and w denotes the size of a datum. Hence, reference $Y(i + j - 1, i + k + 2, j + k)$ is transformed into the reference $Y\left(-i\left(i\left(\frac{5}{2} - i\right) + \frac{1}{2}\right) + j(j+1) + k + 1\right)$.

For example, let us consider 3 successive iterations: (3, 5, 8), (4, 4, 0) and (4, 4, 1). The referenced array elements in the original loop are respectively $Y(7, 13, 13)$,

$Y(7, 6, 4)$ and $Y(7, 7, 5)$. Using the array reference evaluation function we have just determined, it results in contiguous memory addresses for these array elements:

$$\begin{aligned} EP(7, 13, 13) &= 15 + 16 + 11 = 42 \\ EP(7, 6, 4) &= 42 + 0 + 1 \quad = 43 \\ EP(7, 7, 5) &= 42 + 0 + 2 \quad = 44 \end{aligned}$$

Hence, $Y(7, 13, 13)$, $Y(7, 6, 4)$ and $Y(7, 7, 5)$ will respectively be mapped to memory addresses $42 \times w + b$, $43 \times w + b$ and $44 \times w + b$.

The measurements we made on the R12000 for $N = 100$ give the following results:

	Original loop	Transformed reference
# L1 data cache misses	1,012,645	127,155
# L2 data cache misses	121,679	31,234
# TLB misses	1,042,231	152
Computation time	0.32 s	0.02 s

The resulting mapping function may be used by the compiler during the array reference evaluation process instead of using its default method of addressing an array element [1]. The process could be replaced by the evaluation of the previously computed access function EP.

The array reference function that we produce may seem at first to require expensive calculations. But our many experiments have shown that this is not at all a significant issue.

Example 7 Let us consider another example illustrating the case where the loop has not been temporally optimized for the reference under consideration although temporal reuse occurs. It consists of the following loop nest:

```
do i = 1, N
   do j = 1, 2*N
     do k = 1, N
       A(i,j) = A(i,j)+B(i,k)
     enddo
   enddo
enddo
```

According to our optimization selection criteria that will be presented in next section, references to array A prevent the temporal optimization of the reference to array B. Spatial locality will be improved for this latter reference.

An array element $B(i0, j0)$ is referenced by iterations of the form $(i0, j, j0) \in P(i0, j0), i0 \leq j \leq 2N$. Here, the lexicographic minimum $B(i0, j0)$ is obvious to obtain: $(i0, i0, j0)$. Hence, iterations (i, j, k) referencing such lexicographic minima are such that $i = j$. The number of such iterations occurring before iteration

$(i0, i0, j0)$ is determined by computing the Ehrhart polynomial of the following union of polyhedra:

$$\{1 \le i \le N, j = i, 1 \le k \le N, i < i0, 1 \le i0 \le N, 1 \le j0 \le N\}$$
$$\cup \left\{ \begin{array}{l} 1 \le i \le N, j = i, 1 \le k \le N, i = i0, j < i0, 1 \le i0 \le N, \\ 1 \le j0 \le N \end{array} \right\}$$
$$\cup \left\{ \begin{array}{l} 1 \le i \le N, j = i, 1 \le k \le N, i = i0, j = i0, k \le j0, 1 \le i0 \le N, \\ 1 \le j0 \le N \end{array} \right\}$$

The computation results in $EP(i0, j0) = N(i0 - 1) + j0$ and gives the following transformed loop:

```
do i=1, N
   do j = 1, 2*N
     do k=1, N
       A(((4*N+1-i)*i)/2+j-2*N)=A(((4*N+1-i)*i)/2+j-2*N)
           +B(N*(i-1)+k)
     enddo
   enddo
enddo
```

Observe that for the latter reference, the same data sequence is repeatedly accessed with stride-N each time index j is incremented.

When several references to a same array occur in a loop nest, different data layouts can be determined for this array as it was shown in the second and fourth examples of Section 2. It is also the case when references to a same array occur in different loop nests as it was shown in the third example of Section 2. These techniques are presented in the following two subsections.

5.3. Allocating arrays with different data layouts

For the sake of clarity, we first describe our technique for cases where only two references to a same array occur in one loop nest. The more general case where several references occur in one loop nest is presented at the end of this subsection.

Let us consider two references to an array Y, through the homogeneous access matrices R_1 and R_2, $R_1 \neq R_2$, in the same iteration space D_P. We denote by $Data_1$, respectively $Data_2$, the set of active data for reference R_1, resp. R_2:

$$Data_i = \left\{ d_0 \left| \begin{pmatrix} d_0 \\ P \\ 1 \end{pmatrix} = R_i \begin{pmatrix} I \\ P \\ 1 \end{pmatrix}, I \in D_P \right. \right\}, i = 1, 2$$

If $Data_1 = Data_2$, only one data layout that optimizes only one of both references can be determined. Otherwise, some data accessed by one reference are not

accessed by the other one. In this case, two different data layouts defined by two Ehrhart polynomials EP_1 and EP_2 are determined for $Data_1 \cup Data_2$. These data transformations can be applied in two ways, leading to different solutions:

— EP_1 is applied to all the data accessed by reference R_1 and EP_2 is applied to the data accessed by R_2 that are not accessed by R_1. In this case, data that are accessed by both references are not accessed with stride-one when accessed by the reference R_2.
— EP_2 is applied to all the data accessed by reference R_2 and EP_1 is applied to the data accessed by R_1 that are not accessed by R_2. In this case, data that are accessed by both references are not accessed with stride-one when accessed by the reference R_1.

The best solution is the one where the defavourable case occurs less: we compute for each solution the number of iterations where stride-one access does not occur for one of the references, and then choose the solution having the minimum number of these iterations. This computation is presented hereunder.

Let us call $T_i(I)$ the affine transformation defined by homogeneous transformation matrix R_i, for $i = 1, 2$. Then we have $Data_i = T_i(D_P)$. The set of iterations for which the data accessed by R_2 are also accessed by R_1 (for some other set of iterations) is defined by:

$$S_{12} = T_2^{-1}(T_1(D_P)) \cap D_P$$

Since T_2 is not necessarily invertible, T_2^{-1} denotes the *preimage*, that is, the inverse operation of image: given a domain D', $T_2^{-1}(D')$ is the domain which, when transformed by T_2, gives D'. This operation is implemented in the polyhedral library *Polylib* [22]. In the same way, $S_{21} = T_1^{-1}(T_2(D_P)) \cap D_P$ is the set of iterations for which the data accessed by R_1 are also accessed by R_2 for some iterations.

Both sets S_{12} and S_{21} are the iterations for which stride-one access does not occur for one of the references. Hence, the best solution is characterized by the smallest of these sets. For each set, we compute the Ehrhart polynomial giving its number of elements. Suppose that $\#S_{12} < \#S_{21}$. The best solution is the one where stride-one access always occurs for the first reference and occurs for the second reference only when it accesses data that are never accessed by the first reference.

In order to compute both different data layouts and the new loop nest, the original loop has to be split into two different loops $L1$ and $L2$: $L1$ scans the set of iterations S_{12} and $L2$ scans the set $D_P - S_{12}$. The data layout defined by EP_1 is computed from the original loop with the first reference, and the data layout defined by EP_2 is computed from $L2$ with the second reference. EP_1 is then applied to both references in $L1$ and to the first reference in $L2$, and EP_2 is applied to the second reference in $L2$. Moreover, in order to allocate consecutively in memory data accessed by EP_1 and data accessed by EP_2, the biggest index of the data accessed by EP_1 is added to EP_2. Let us now detail the fourth example of Section 2.

Example 8 Let $T_1(i, j) = (k, i)$ and $T_2(i, j) = (j + k, j)$. We have:

$$D_N = \{(i, j) \in \mathbb{Z}^2 | 1 \leq i \leq N, 1 \leq j \leq N, 1 \leq k \leq N\}.$$

The set of iterations for which the data accessed by $A(j+k, j)$ are also accessed by $A(k, i)$ is defined by:

$$S_{12} = T_2^{-1}(T_1(D_N)) \cap D_N$$
$$= \{(i, j) \in \mathbb{Z}^2 | 1 \leq i \leq N, 1 \leq j \leq N, 1 \leq k \leq N - j\}$$

In the same way, the set of iterations for which the data accessed by $A(k, i)$ are also accessed by $A(j+k, j)$ is defined by:

$$S_{21} = T_1^{-1}(T_2(D_N)) \cap D_N$$
$$= \{(i, j) \in \mathbb{Z}^2 | 1 \leq i \leq N, 1 \leq j \leq N, i + 1 \leq k \leq N\}$$

Computations of the Ehrhart polynomials of each set give the following answer:

$$\#S_{12} = \#S_{21} = \tfrac{1}{2}N^3 - \tfrac{1}{2}N^2 + N$$

Therefore, both solutions are equivalent. Let us choose the one characterized by S_{12}. The resulting loop nest is obtained by splitting the innermost loop indexed by k into two loops where index k ranges from 1 to $N - j$ for the first one and from $N - j + 1$ to N for the second one.

The new data layout for reference $A(k, i)$ is computed from the original loop as described in the previous subsections: $EP_1(i0, j0) = N(j0 - 1) + i0$. The new data layout for reference $A(j + k, j)$ is computed from the second innermost loop of the new loop nest: $EP_2(i0, j0) = i0(i0 - 1)/2 + j0$. Since the data with the biggest index accessed by the first reference is $A(EP_1(N, N)) = A(N^2)$, N^2 is added to $EP_2(i0, j0)$. The resulting loop nest is shown in Figure 4 of Section 2.

It is important to mention that such a loop splitting can always be done without altering the data dependences, since the initial order of the iterations is not affected at all: any loop inside a considered loop nest is split into several successive loops such that the included instructions are executed in the same order as in the original loop, and without modifying any of the enclosing loops.

We consider now the case where a given array is accessed by r ($r > 2$) references in the loop nest. For each reference to an array Y through access matrix R_i, the active data set is $Data_i$, $i = 1..r$, with $R_i \neq R_j$ for any $i \neq j$. In order to present our algorithm, we first define the notion of *split pattern* of a loop nest.

Let us consider imperfectly nested loops. Such a nest can be seen as a fusion of many perfectly nested loops. We consider here that these perfect loop nests L_i scan disjoint but contiguous convex iteration spaces D_i and that the union of these spaces D is convex.[4] Hence, the D_i's define a partition of the iteration space D. Let us now consider any other partition of the same iteration space D defined by convex, disjoint and contiguous subsets S_j. The mapping of this last partition onto the partition defined by the D_i's defines a new partition with smaller subsets. The scanning of these smaller subsets following the same initial iteration directions allows to generate a new imperfectly nested loop nest which results from the splitting of the original one following the *split pattern* defined by the S_j's (see Figure 9).

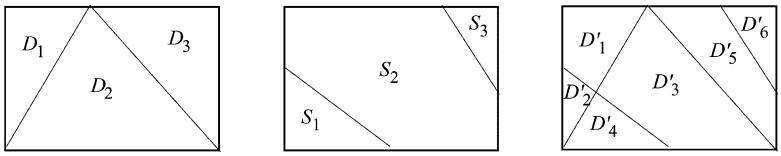

Figure 9. An initial iteration space $D_1 \cup D_2 \cup D_3$, a split pattern $S_1 \cup S_2 \cup S_3$ and the corresponding final iteration space $D'_1 \cup D'_2 \cup D'_3 \cup D'_4 \cup D'_5$.

In the following, we will denote by D_i the iteration space of reference i, that is, the set of iterations of the loop nest that enclose the ith reference. Moreover, if reference i has been temporally optimized as described in Section 4, innermost loops acessing the same data are not considered. For example in the following loop nest:

```
do i = 1, N
   do j = 1, N
     ...A(i)...
     do k = 1, N
       ...
     enddo
   enddo
enddo
```

the iteration space of reference $A(i)$ is $D_1 = \{1 \leq i \leq N\}$, since this loop nest would rather be written in the following way in order to improve register usage:

```
do i = 1, N
   r = A(i)
   do j = 1, N
     ...r...
     do k = 1, N
       ...
     enddo
   enddo
enddo
```

Using these notions, our algorithm consists in the following: the first part of the algorithm finds the best solution where the defavourable case occurs the least: each set S_{ij}^q defines iterations for which at least one of the references j ($j \neq i$) accesses data also accessed by the ith reference. These iterations represent the defavourable case where stride-one access would not occur for references j if the associated data layout and loop splitting would have been selected. In order to minimize the number of times this case occurs, we choose reference i associated with the smallest sets S_{ij}^q such that stride-one access will occur the most.

% Part I: **Finding the best data layouts**

1. let $E = \emptyset$, $q = r$ and $Q = \{1, 2, 3, \ldots, r\}$
2. compute all the sets S_{ij}^q, $(i, j) \in Q \times Q$, defined by $S_{ij}^q = T_j^{-1}(T_i(D_i) - E) \cap D_j$
3. select reference i_q associated to the smallest sets S_{ij}^q, that is, i_q such that:

$$\sum_{j \in Q, j \neq i_q} \#S_{i_q j}^q = \min_{i \in Q} \left(\sum_{j \in Q, j \neq i} \#S_{ij}^q \right)$$

4. let $E = E \cup Data_{i_q}$, $q = q - 1$ and $Q = Q - \{i_q\}$
5. if $q > 1$ go to step 2

% Part II: **Code generation**

6. let $q = r$, $Q = \{1, 2, 3, \ldots, r\}$, L denotes a set containing the original loop nest and D denote the iteration space of L
7. compute the new data layout defined by EP_q from L with the i_qth reference as described in the previous subsections
8. split L following the split pattern defined by the sets $S_{i_q j}^q$, $j \in Q$, $j \neq i_q$
9. for all the loops scanning subsets $s \subset \bigcup_{j \in Q, j \neq i_q} S_{i_q j}^q$, apply EP_q to all references j_k such that $k \leq q$ and $s \subset S_{i_q j_k}^q$
10. for all the remaining loops, apply EP_q to the i_qth references
11. let $q = q - 1, Q = Q - \{i_q\}, L$ denotes the loops scanning sets s such that $s \subset (D - S_{i_{q+1} i_q}^q)$ and D denotes the iteration space of L
12. if $q > 1$ go to step 7
13. compute the new data layout defined by EP_1 from L with the i_qth reference
14. apply EP_1 to the i_pth references in L

Let us look at an example with three references to a same array.

Example 9 Consider the following loop nest:

```
do i = 1, N
   do j = 1, N
     A(j,i)=A(i,i+j)+A(i,i)
   enddo
enddo
```

Let $D_N = \{(i, j) \in \mathbb{Z}^2 | 1 \leq i \leq N, 1 \leq j \leq N\}$, $T_1(i, j) = (j, i)$, $T_2(i, j) = (i, i + j)$ and $T_3(i, j) = (i, i)$. For $q = 3$, the above algorithm computes the following sets:

$$S_{21}^3 = \{(i, j) \in \mathbb{Z}^2 | 2 \leq i \leq N, 1 \leq j \leq i - 1\}$$

$$S_{12}^3 = \{(i, j) \in \mathbb{Z}^2 | 1 \leq i \leq N - 1, 1 \leq j \leq N - i\}$$

$$S_{31}^3 = \{(i, j) \in \mathbb{Z}^2 | 1 \leq i \leq N, i = j\}$$

$$S_{13}^3 = D$$

$$S_{32}^3 = \emptyset$$

$$S_{23}^3 = \emptyset$$

The size of each set is given by their respective Ehrhart polynomials:

$$\#S^3_{21} = N(N-1)/2$$
$$\#S^3_{12} = N(N-1)/2$$
$$\#S^3_{31} = N$$
$$\#S^3_{13} = N^2$$
$$\#S^3_{32} = \#S^3_{23} = 0$$

The third reference is selected since $(\#S^3_{31} + \#S^3_{32})$ is minimal: $Data_3 = \{(i, j) \in \mathbb{Z}^2 | 1 \leq i \leq N, i = j\}$ is added to E and $Q = \{1, 2\}$. For $q = 2$, the algorithm computes:

$$S^2_{12} = \{(i, j) \in \mathbb{Z}^2 | 1 \leq i \leq N-1, 1 \leq j \leq N-i\},$$
$$\#S^2_2 = N(N-1)/2,$$
$$S^2_{21} = \{(i, j) \in \mathbb{Z}^2 | 2 \leq i \leq N, 1 \leq j \leq i-1\},$$
$$\#S^2_1 = N(N-1)/2$$

Either the first or the second reference can be selected. Let us choose the second one. The first part of the algorithm ends.

The code generation starts with $q = 3$. Since the third reference has been selected, EP_3 is computed from this reference in the original loop: $EP_3(i0, j0) = i0$. The loop is split such that iterations where $i = j$ are scanned separately. EP_3 is applied to all third references and to the first reference in the loop where $i = j$:

```
do i = 1, N
   do j = 1,i-1
     A(j,i) = A(i,i+j) + A[i]
   enddo
   A[i] = A(i,i+i) + A[i]
   enddo
   do j = i + 1,N
     A(j,i) = A(i,i+j) + A[i]
   enddo
enddo
```

For $q = 2$ and since the second reference has been selected, EP_2 is computed from the second reference in all the loops: $EP_2 = N(i0 - 1) - i0 + j0 + N$. In order to allocate consecutively to memory the data accessed by EP_3 and the data accessed by EP_2, the biggest index of the data accessed by EP_3, that is, N, is added to EP_1. EP_2 is applied to all second references and to the first reference in the loop scanning

S^2_{21}, that is, the first loop:

```
do i = 1, N
   do j = 1, i-1
     A[N*(j-1)+i-j+N]=A[N*(i-1)+j+N]+A[i]
   enddo
   A[i]=A[N*(i-1)+i+N]+A[i]
   enddo
   do j = i+1, N
     A(j,i)=A[N*(i-1)+j+N]+A[i]
   enddo
enddo
```

Finally, the algorithm ends by computing EP_1 from the first reference in the last loop: $EP_1 = (2N - j0)(j0 - 1)/2 + i0 - j0 + N(N + 1)$. In order to allocate consecutively to memory data accessed by EP_2 and data accessed by EP_1, the biggest index of the data accessed by EP_2, that is, $N(N + 1)$, is added to EP_1. Hence, the final optimized loop is:

```
do i = 1, N
   do j = 1, i-1
     A[N*(j-1)+i-j+N]=A[N*(i-1)+j+N]+A[i]
   enddo
   A[i]=A[(N+1)*i]+A[i]
   do j = i+1, N
     A[(2*N-i)*(i-1)/2-i+j+N*(N+1)]=A[N*(i-1)+j+N]+A[i]
   enddo
enddo
```

Measurements have been made for $N = 5000$ giving the following results:

	Original loop	Transformed reference
# L1 data cache misses	28,272,270	17,371,446
# L2 data cache misses	1,677,582	1,484,270
# TLB misses	15,935,741	7,936,117
Computation time	5.20 s	2.95 s

A main disadvantage is the software overhead that can generate this loop splitting process. The number of resulting loops depends on the number of references to a same array that occur in the original loop. However, this process can be interrupted when the remaining references that have not yet been considered by our algorithm do not represent significant improvements: if, at step 11 of the algorithm, the weight of the remaining references defined by $(\#D \times \#Q - \sum_{j \in Q, j \neq i_q} \#S^q_{i_q j})$ is negligible as compared to the weight of all references defined by $(\#D_{orig} \times r)$. This is measured by the ratio:

$$\Pi(i_q) = \frac{\#D \times \#Q - \sum_{j \in Q, j \neq i_q} \#S^q_{i_q j}}{\#D_{orig} \times r}$$

where D_{orig} denotes the iteration space of the original loop nest. This ratio measures the weight of the optimized references relatively to the weight of all the references in the initial loop nest. In some of our experiments, we observed that it was still worth to split loops as $\Pi(i_q) > 1/100$.

The splitting process is stopped in the following way: for all remaining references, the convex hull of the accessed data is computed. For this data set, a data layout is determined by computing a new access function. This access function is computed by considering a loop scanning the data set with the same iteration directions as in the original loop, and by considering the same initial access function as one of these remaining references. Stride-one access is then obtained in the innermost loop for at least this remaining reference. The resulting program performance relatively to these references cannot be worse than the performance obtained from a classically optimized program.

All the cases presented in this section are relevant to the global optimization strategy given in Section 6. While considering a whole program with several loop nests accessing some common arrays, data layouts that were determined from a given loop nest will have to be propagated to some other loop nests that access the same arrays. This process is explained in next subsection.

5.4. Data layout propagation

Consider two loop nests $L1$ and $L2$ accessing an array Y. Consider also that a new data layout has been determined for Y while optimizing $L1$. Since optimizing $L1$ can result in splitting the original loop nest, several different data layouts could have been determined for Y. Let us consider only one of the innermost included loops as the same process would be repeated for any of them. This data layout has to be propagated to the reference in $L2$. In the same way, optimizing $L2$ can also result in splitting the original loop nest while improving references to some other arrays. So we also consider only one of the innermost included loops. Let $T_1(I)$, resp. $T_2(I)$, denote the original access function in $L1$, resp. $L2$. We denote by D, resp. D', the iteration space of $L1$, resp. $L2$.

The set of iterations of D' for which the data accessed by R_2 in $L2$ are also accessed by R_1 in $L1$ is defined by:

$$S_{12} = T_2^{-1}(T_1(D)) \cap D'$$

Hence, references to Y made while executing these iterations must use the same access function as the one used in $L1$: $L2$ has to be split according to the split pattern defined by S_{12} and the access function used in $L1$ has to be applied to the reference in the loop nest $L2$ scanning S_{12}.

In a general way, this new reference in $L2$ will not take advantage of the corresponding data layout as in $L1$. Unchanged references made in the loop scanning $D' - S_{12}$ can also be improved as presented in the previous subsections.

However, it can be possible to transform the loop scanning S_{12} such that the new reference will take advantage of the data layout: the new loop must scan S_{12} following the same iteration vectors as in $L1$. But such a transformation is not

always possible since it must not alter the data dependences and since it is closely related to temporal optimization (see subsection 4.1).

In this case, in order to prevent software overhead, the splitting process can be interrupted: the convex hull of $Data_1 \cup Data_2$ is computed. A new access function is computed by considering a loop scanning this data set with the same iteration directions as in $L1$ and with the access function T_1. Then the new access function is applied in both loops to both references.

6. Cost criteria driven optimizations of nested loops

While considering a whole program, our optimization process is based on precise indicators and on several benchmark experiments. Some of the main strategies are discussed in this section.

In general, temporal locality must be improved as much as possible, since it optimizes register use and no cache misses are generated in the innermost loops accessing the same data element. Moreover, while data layout transformation has an effect on all loop nests accessing array elements, temporal optimization only concerns one loop nest.

Optimization of a whole program is driven by the cost of the program's memory accesses. This cost can be evaluated on either a per referenced array basis, or per loop nest basis. An array driven approach would be the best approach if loop nests of a program often access different subarrays. But this is not the case in most programs. Therefore, a loop nest cost driven approach will generally give better results.

The cost of a loop nest L is the number of memory accesses occurring during the execution of L: $C(L) = \sum_{i=1}^{r} \#D_i$ where r denotes the number of different references occurring in L and D_i denotes the set of iterations of the loop nest enclosing the ith reference. All loop nests of a program are ordered according to this value. This cost function is similar to the one defined by Kandemir et al. in [12], except that our evaluation tools allow to compute exact symbolic values.

Loop nests are optimized from the costliest to the less costly nest. When considering one loop nest, the best combination of loop and data transformations depends on previous optimizations made on costlier nests and on the cost of the occurring references. In comparing several possible solutions, the best ones are characterized by the smallest strides generated by their references. For example, let us consider four different possible solutions for scanning the loop nest presented in Figure 10. For each loop nest and each reference, we compute the number of strides of any size occurring while incrementing the loop indices. The results are given in Figure 11. For example, in the first loop nest, strides of size $O(N^2)$ occur N^3 times while accessing memory by reference $B(i, l)$.

Since the objective is to minimize the number of large strides, the solution corresponding to loop nest 2 can be eliminated: N^2 strides of size $O(N^3)$ occur while accessing memory by reference $A(k, j, i)$. Looking at strides of size $O(N^2)$, solution 1 is eliminated since it yields more such strides than solutions 3 and 4. Comparing these latter solutions, solution 4 is selected since strides of size $O(N^2)$

Nest 1

```
do i = 1,N
 do j = 1,N
  do k = 1,N
   do l = 1,N
    A(k,j,i) = B(l,j)+B(i,l)
   enddo
  enddo
 enddo
enddo
```

Nest 2

```
do l = 1,N
 do j = 1,N
  do i = 1,N
   do k = 1,N
    A(k,j,i) = B(l,j)+B(i,l)
   enddo
  enddo
 enddo
enddo
```

Nest 3

```
do i = 1,N
 do l = 1,N
  do j = 1,N
   do k = 1,N
    A(k,j,i) = B(j,l)+B(l,i)
   enddo
  enddo
 enddo
enddo
```

Nest 4

```
do i = 1,N
 do j = 1,N
  do l = 1,N
   do k = 1,N
    A(k,j,i) = B(l,j)+B(i,l)
   enddo
  enddo
 enddo
enddo
```

Figure 10. Four different ways to scan the loop nest.

occur while accessing an array of size N^2, while in solution 3, strides of size $O(N^2)$ occur while accessing an array of larger size, that is, N^3. Hence the best solution is the one associated with loop nest 4. This result was validated by our performance measurements.

The best solution should be determined directly without enumerating all possible solutions. But such a direct approach can only be built using heuristics. The largest stride that a reference i may generate is the size of the set of accessed data $Data_i$. In general, strides larger than one are generated by reusing the same sequence of contiguous data several times. Such a reuse corresponds to a temporal reuse vector associated with the reference. In order to prevent these large strides, the innermost loop should scan data along this vector to maximize temporal reuse. Conversely, the further out (in the nesting order) the reuse loop, the larger the strides occurring while incrementing its index are, and the larger the reused data sequences are. Hence, minimizing stride sizes consists in scanning data in the reuse direction with the most inner possible loop, where temporal reuse will occur on the smallest possible data sequence. Notice that the most favourable case corresponds to the classical temporal optimization where the reused data sequence is reduced to one unique data. All these facts culminate in the general concept of *data sequence localization*.

Several references in a loop nest have to be simultaneously considered with several temporal reuse directions. As presented in Section 4, if there exist reuse directions that are common to several references, the best solutions are char-

loop nests		nest 1	nest 2	nest 3[a]	nest 4
stride sizes	references				
$O(N^3)$	$A(k,j,i)$	0	N^2	0	0
	Total	0	N^2	0	0
$O(N^2)$	$A(k,j,i)$	0	$N^3 - N^2$	$N^2 - N$	0
	$B(l,j)$	N	N	N	N
	$B(i,l)$	N^3	0	0	N^2
	Total	$N^3 + N$	$N^3 - N^2 + N$	N^2	$N^2 + N$
$O(N)$	$A(k,j,i)$	0	0	0	$N^3 - N^2$
	$B(l,j)$	$N^3 - N^2$	$N^2 - N$	0	0
	$B(i,l)$	$N^4 - N^3$	$N^2 - N$	0	$N^3 - N^2$
	Total	$N^4 + N^2$	$2(N^2 - N)$	0	$2(N^3 - N^2)$
1	$A(k,j,i)$	N^3	$N^4 - N^3$	$N^4 - N^2 + N$	$N^4 - N^3 + N^2$
	$B(l,j)$	$N^4 - N^3 + N^2 - N$	0	$N^3 - N$	$N^3 - N$
	$B(i,l)$	0	$N^3 - N^2 + N$	N^2	0
	Total	$N^4 + N^2 - N$	$N^4 - N^2 + N$	$N^4 + N^3$	$N^4 + N^2 - N$
0	$A(k,j,i)$	$N^4 - N^3$	0	0	0
	$B(l,j)$	0	$N^4 - N^2$	$N^4 - N^3$	$N^4 - N^2$
	$B(i,l)$	0	$N^4 - N^3$	$N^4 - N^2$	$N^4 - N^3$
	Total	$N^4 - N^3$	$2N^4 - N^3 - N^2$	$2N^4 - N^3 - N^2$	$2N^4 - N^3 - N^2$

[a] Notice that for nest 3, lines and colums of array B have been inverted

Figure 11. Number of occurring strides of any size for each loop nest and reference.

acterized by innermost loops scanning along these directions. The largest data sets are more likely to induce large strides resulting in the most cache and TLB misses. Hence, the best solutions are such that the different reuse directions are associated with loops ordered from the innermost to the outermost loop, following the descending size of the associated sets of accessed data $\#Data_i$. Our temporal optimization algorithm presented in Section 4 is based on these observations.

For any given reference i and its associated iteration space D_i, the size of its data set $\#Data_i$ is given by the Ehrhart polynomial of the affine transformation R_i of D_i, where R_i is the reference matrix [8].

Finally, stride-one access is obtained as often as possible by computing new data layouts as presented in Section 5. In the above example, this approach would have lead to the fourth nest which is the best solution.

The general algorithm can be presented as follows:

1. For all l loop nests L_i, compute the cost functions as presented at the beginning of this section: $C(L_i) = \sum_{i=1}^{r} \#D_i$.
2. Order them from the most to the least costly nest, that is, $L_1, L_2, \ldots$ such that $C(L_1) \geq C(L_2) \geq \cdots \geq C(L_l)$.
3. For $i = 1$ to l do

3a. For all references j occurring in L_i, compute the sizes of the data sets $\#Data_j$.

3b. Temporally optimize L_i using the algorithm of Section 4.

3c. For each reference accessing arrays that have already been accessed by some references occurring in costlier nests, propagate the data layouts as presented in subsection 5.4.
3d. For all remaining references, compute new data layouts for the accessed data in order to minimize stride sizes.

7. Consequences on parallel optimizations

Although our method is devoted to improving savings in cache and TLB miss rates, it also has an important impact on processor locality: when a processor brings a data page into its local memory, it will reuse it as much as possible due to our temporal and spatial optimizations. This yields significant reductions in page faults and hence in network traffic.

We can also say, as mentioned by Kandemir et al. in [12], that optimized programs do not need explicit data placement techniques on shared memory NUMA architectures: when a processor uses a data page frequently, the page is either replicated onto that processor's memory or migrated into it. In either cases, most of the remaining accesses will be local.

Unlike other methods, our data layout transformations also prevent processors from bringing pages containing useless data that could have been accessed by other processors. This is due to the fact that only active data are considered in our optimizations.

Temporal optimizations presented in Section 4 often generate outer loops that carry no reuse and no data dependences. Hence, these outer loops are perfect candidates for parallelization since they do not share any data.

All these facts allow the generation of data-parallel code with significant savings in interprocessor communication.

Our splitting process presented in subsections 5.3 and 5.4 allows us to extract more and different kinds of parallelism, that is, control parallelism, from a program. Consider two successive loops nests, $L1$ and $L2$, that share the same array in an initial sequential program, where the first nest reads some array elements and the second nest writes some other array elements. If the second nest accesses a significative set of array elements that are not accessed by the first nest, it is split into two loop nests $L21$ and $L22$ such that $L22$ accesses elements that are not accessed by the other nests. This latter nest $L22$ can then be computed in parallel with $L1$ since it does not share any of the same array elements.

Example 10 Consider the following two loop nests:

```
!$ Nest L1
do i = 1, N
   do j = 1, N
     Y = Y+A(j,i)
   enddo
enddo
```

```
!$ Nest L2
do i = 1, N
   do j = 1, N
     A(i+j, i) = B(i)+C(j)
   enddo
enddo
```

The updating of some array elements $A(i+j, i)$ in $L2$ prevents the parallelization of both nests $L1$ and $L2$. We compute the set of iterations in $L2$ for which the data accessed by $A(i+j, i)$ are also accessed by $A(j, i)$ in $L1$: let $T_1(i, j) = (j, i)$, $T_2(i, j) = (i+j, i)$ and $D_1 = D_2 = \{(i, j) \in \mathbb{Z}^2 | 1 \leq i \leq N, 1 \leq j \leq N\}$, $S_{12} = T_2^{-1}(T_1(D_1)) \cap D_2 = \{(i, j) \in \mathbb{Z}^2 | 1 \leq i \leq N, 1 \leq j \leq N - i\}$. Hence, iterations of $L2$ defined by $D_2 - S_{12}$ access elements that are not accessed by $L1$. We split $L2$ into two loop nests: $L21$ scanning S_{12} and $L22$ scanning $D_2 - S_{12}$. The final parallel program simultaneously computes $L1$ and $L22$. When computation of $L1$ has been completed, computation of $L21$ can start.

```
!$ PARALLEL REGION
!$ PARALLEL SECTION
!$ Nest L1
!$ PARALLEL DO
do i = 1, N
   do j = 1, N
     Y = Y+A(j,i)
   enddo
enddo
!$ END PARALLEL DO
!$ Nest L21
!$ PARALLEL DO
do i = 1, N
   do j = 1, N-i
     A(i+j, i) = B(i)+C(j)
   enddo
enddo
!$ END PARALLEL DO
!$ END PARALLEL SECTION
!$ PARALLEL SECTION
!$ Nest L22
!$ PARALLEL DO
do i = 1, N
   do j = N-i+1, N
     A(i+j,i) = B(i)+C(j)
   enddo
enddo
!$ END PARALLEL DO
```

```
!$ END PARALLEL SECTION
!$ END PARALLEL REGION
```

8. Conclusion

All the geometric and arithmetic tools used by our method are implemented in the polyhedral library *PolyLib*.[5] Some parts of our precise data locality optimization process have already been implemented, but some important program developments are still necessary. This paper shows that significant performance improvements can be obtained in programs, even if traditional optimizations have already been applied.

Systematically applying such precise optimizations can be seen as being too expensive a process for a general purpose compiler. However, several responses can be given to justify these optimizations:

— the continuously growing power of today's processors allows one to consider the feasibility of more complex compilers, since the associated computation times become more and more acceptable;
— the significant improvements brought by such precise optimizations can reduce computation times by several months for some essential and very complex problems that are considered nowadays, and whose computations are taking several years;
— in order to take full advantage of future processoring potential, compilers will need to generate executable codes explicitly stating the best way for computations to be performed (memory behaviour, instruction level parallelism, ...). This can only be obtained from a precise analysis of the program source;
— code generation for embedded processors requires longer analysis and compiling times. Optimized memory usage and computational efficiency are required for applications for several reasons: slower and less expensive processors are often used, memory significantly contributes to system cost, and power consumption must be minimized as more and more functionality is embedded.

Some further improvements can be expected by considering architectural parameters characterizing the target architecture: cache size, cache associativity, cache line size, TLB reach, etc. In addition to data locality optimization, some other important issues related to efficient memory use can be considered, such as array padding and cache set conflicts. We are currently investigating these ideas.

Acknowledgment

We are grateful to Doran K. Wilde for his helpful remarks and corrections on an early version of this paper.

Notes

1. In the homogeneous space including the index variables, the parameters, and the constant, of dimension $(d + q + 1)$. This is very useful in the later computations since only one matrix is needed to operate on variables as well as parameters and the constant.
2. Preimage is the inverse function of image. It is implemented in the PolyLib.
3. This is not the way temporal reuse is detected in our technique (see Section 4).
4. This is not a restriction in the scope of the paper. This situation characterizes the results of splitting loops in our algorithm presented in the following.
5. The PolyLib is freely available at http://icps.u-strasbg.fr/Polylib.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, Reading, Mass., 1987.
2. J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and computation transformations for multiprocessors. In A. Press, ed., *Proceedings, Principles and Practice of Parallel Programming*, 1995.
3. U. Banerjee. Unimodular transformations of double loops. In *Advances in Languages and Compilers for Parallel Processing*, 1991.
4. U. Banerjee. *Loop Transformations for Restructuring Compilers—The Foundations*. Kluwer Academic Publishers, Norwell, Mass., 1993.
5. S. Chatterjee, V. V. Jain, A. R. Lebeck, and S. Mundhra. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of the ACM International Conference on Supercomputing*, Rhodes, Greece, 1999.
6. M. Cierniak and W. Li. Unifying data and control transformations for distributed shared-memory machines. In *Proceedings Programming Language Design and Implementation*, 1995.
7. P. Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: applications to analyze and transform scientific programs. In *10th ACM International Conference on Supercomputing*, Philadelphia, 1996.
8. P. Clauss. Handling memory cache policy with integer points countings. *In Euro-Par'97*, Passau, pp. 285–293, 1997.
9. P. Clauss and V. Loechner. Parametric analysis of polyhedral iteration spaces. *Journal of VLSI Signal Processing*, 19: 179–194, 1998.
10. P. Feautrier. Automatic parallelization in the polytope model. *In* G. -R. Perrin and A. Darte, eds. *The Data Parallel Programming Model*, Vol. 1132 of *Lecture Notes in Computer Science*, pp. 79–100. Springer-Verlag, Berlin, 1996.
11. Y.-J. Ju. and H. Dietz. Reduction of cache coherence overhead by compiler data layout and loop transformations. In *Proceedings of the 4th International Workshop on Languages and Compilers for Parallel Computing*, 1992.
12. M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. A matrix-based approach to global locality optimization. *Journal of Parallel and Distributed Computing*, 58: 190–235, 1999.
13. M. Lam, E. Rothberg, and M. Wolf. The cache performance of blocked algorithms. In *International Conference on ASPLOS*, 1991.
14. S.-T. Leung and J. Zahorjan. Optimizing data locality by array restructuring. Technical Report 95-09-01, University of Washington, Department of Computer Science and Engineering, 1995.
15. W. Li. Compiling for NUMA parallel machines. Ph.D. thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1993.
16. V. Loechner and D. K. Wilde. Parameterized polyhedra and their Vertices. *International Journal of Parallel Programming*, 25: 525–549, 1997.
17. M. O'Boyle and P. Knijnenburg. Nonsingular data transformations: definition, validity, and applications. *International Journal of Parallel Programming*, 27: 131–159, 1999.
18. F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28: 469–498, 2000.

19. J. M. Rabaey and M. Pedram. *Low Power Design Methodologies*. Kluwer Academic Publishers, Norwell, Mass., 1995.
20. A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, New York, 1986.
21. M. R. Swanson, L. Stoller, and J. Carter. Increasing TLB reach using superpages backed by shadow memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 204–213, 1998.
22. D. Wilde. A library for doing polyhedral operations. Master's thesis, Oregon State University, Corvallis, 1993.
23. M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of ACM SIGPLAN 91 Conference Programming Language Design and Implementation*, Toronto, Ont., pp. 30–44, 1991.
24. M. Wolfe. More iteration space tiling. In *Proceedings of Supercomputing'89*, pp. 655–664, 1989.
25. M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison Wesley, Reading, Mass., 1996.