Data Sequence Locality: a Generalization of Temporal Locality

Vincent Loechner, Benoît Meister, and Philippe Clauss

ICPS/LSIIT, Université Louis Pasteur, Strasbourg, France.
http://icps.u-strasbg.fr

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. Such objective classically translates into optimizing spatial and temporal data locality especially for nested loops. In this paper, we focus on temporal data locality. Unlike many existing methods, our approach pays special attention to TLB (Translation Lookaside Buffer) effectiveness since a TLB miss can take up to three times more cycles than a cache miss. We propose a generalization of the traditional approach for temporal locality improvement, called data sequence localization, which reduces the number of iterations that separates accesses to a given array element.
Keywords. Memory hierarchy, cache and TLB performance, temporal locality, loop nests, parameterized polyhedra, Ehrhart polynomials.

1 Introduction

Efficient use of memory resources is a significant source for enhancing application performance and for reducing power consumption for embedded processor applications [16]. Nowadays computers include several levels of memory hierarchy in which the lower levels are large but slow (disks, memory, etc.) while the higher levels are fast but small (caches, registers, etc.). 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. Spatial locality is achieved if several closely located memory locations are accessed before the cache line or page is 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. Taking advantage of spatial and temporal locality translates into minimizing cache misses, TLB (translation lookaside buffer) misses and page faults and thus increases performance.

The TLB holds only a limited number of pointers to recently referenced pages. On most computers, this number ranges from 8 to 256. Few microprocessors have a TLB reach larger than the secondary cache when using conventional page sizes. If the TLB reach is smaller than the cache, the processor will get TLB

misses while accessing data from the cache. 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 R12000 processor TLB supports 4K to 16M page sizes per TLB entry, which means the TLB reach ranges from 512K to 2G. Unfortunately, while many processors support multiple page sizes, few operating systems make full use of this feature [18].

Recent work has 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 [22]. These techniques consist either in unimodular [1] or non-unimodular [11] iteration space transformations as well as tiling [10, 20, 21]. More recently, Kandemir *et al.* [9] and O'Boyle and Knijnenburg [14] have proposed a unifying framework for loop and more general data transformations. In [14], the authors propose an extension to nonsingular data transformations.

Unfortunately, these approaches do not pay special attention to TLBs. Hence, a program that exhibits good cache locality can yield a very poor performance. This is mainly due to the fact that only the innermost loop level is considered in most of these methods. In the resulting programs, accesses with large strides can occur each time indices of the outer loops are incremented. This can yield many expensive TLB misses.

Spatial locality is classically improved by computing new memory data layouts yielding stride-one accesses in the innermost loop, and temporal locality is improved by transforming loops such that constant data are accessed in the innermost loop for some array references (blocking will also be discussed hereunder). We have proposed a different approach for spatial locality in [6], by determining new data layouts from the computation of Ehrhart polynomials [5].

In this paper, we only focus on temporal locality optimization by defining a more general and efficient approach: temporal locality can be improved by reducing the number of iterations that separates two successive accesses to a given reused data. Similar approaches of some other work are discussed in Sect. 4.

Let us consider the first example presented in Tab. 1. Observe in this example that reference $A(j,i)$ exhibits good temporal locality since any iteration between $(i,j,1)$ and (i,j,N) will access the same data. Although temporal reuse occurs for reference $B(k,j)$, there is no possible loop transformation that could result in a good temporal locality in the innermost loop for both references. Hence, any classical optimization method would consider such a loop nest as acceptable. Observe that all elements of array B are reused each time index i is incremented. If all these elements cannot be loaded simultaneously in the cache due to a too large value of N, any reused element is no longer present in the cache when it has to be accessed again. Moreover, accessing all elements of array B will generate TLB misses if B is stored in several memory pages. These unavoidable TLB misses are repeated each time i is incremented.

We made some performance mesures on a MIPS R12000 processor, with 2-way set-associative 32KB data L1 cache, 8MB L2 cache, and 64-entry TLB. We ran this loop nest for $N = 5000$ and measured data cache and TLB misses with

Table 1. Two versions of a loop nest.

first version	second version

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

# L1 cache misses: 15,638,299,806	# L1 cache misses: 91,642,448
# L2 cache misses: 3,906,794,992	# L2 cache misses: 1,899,471
# TLB misses: 16,120,451	# TLB misses: 16,290,785
computation time: 2712.14s	computation time: 425.21s

the performance tool $perfex$ using hardware counters. Computation time was measured using the system clock. We obtain the results presented in Tab. 1.

Let us now consider another semantically equivalent version of this loop nest, presented on righthand side of Tab. 1. Loops i and j have been interchanged in order to improve temporal locality for the smallest possible sequence of elements of B: $B(1,j), B(2,j), ..., B(N,j)$. The measures show important savings in the number of cache misses, due to the elimination of strides of size N^2.

This transformation seems quite similar to blocking: if the N elements of B can be loaded simultaneoulsly in the cache, then the same result could have been obtained by defining blocks of size $N \times 1 \times N$: If the N elements cannot be loaded entirely in the cache, blocking should result in better performance by adjusting the block size to a proper value. However, this is a difficult task greatly dependent on the cache parameters (size, associativity, hierarchy, etc.). Moreover, performance of blocking can often be reduced by cache conflicts effects, and blocking may break temporal locality. By allowing autoblocking and interchange of loops (-LNO:blocking=on:interchange=on) while compiling the previous initial example for $N = 5000$ with the SGI F90 compiler, loop k is divided into blocs of 1820 iterations. In the initial program and in our optimized version, each array element $A(j,i)$ is consecutively reused N times and is not further reused. Hence it is register-stored during the computation of these N iterations. In the blocked version, each $A(j,i)$ is consecutively reused 1820 times. Then all other array elements are reused in the same way before the same element $A(j,i)$ is reused again 1820 times and so on. Therefore, register locality is broken $N/1820$ times in this blocked version. For this example, lower performance is observed for the blocked version in particular in the number of TLB misses (48,567,268 versus 16,290,785 in our version).

Our proposed approach is independent of the target architecture and will improve performance in any situation. In any case it does not prevent additional blocking transformations. Although applying a blocking transformation on our optimized version for this example does not improve performance at all.

The example shows a simple loop interchange transformation, but our approach is not reduced to deduce opportune loop interchanges. All references in an imperfectly nested loop are examined, and eventually new iteration directions are computed with respect to data dependences. We propose a step by

step algorithm consisting in optimizing a maximum number of references at the most inner possible loop level. Our techniques use the polytope model [8] with an intensive use of its parametric extensions [13, 5] and of their implementations in the polyhedral library *PolyLib* [19].

Temporal locality can be fully optimized, *i.e.* in the innermost loop, only for a subset of occuring references. *Data sequence localization* consists in considering the remaining references at next loop levels, in order to minimize the number of occuring iterations between two successive reuses of the same data. It results in optimizing temporal locality for the sequence of data that is referenced between two such reuses. The smaller the reused data sequences are, the more successful is the opportunity that these sequences can be entirely loaded in the highest levels of the memory. Notice that the most favorable case corresponds to the smallest data sequence containing one single data. Hence we can observe that classical temporal optimization in the innermost loop is a particular case of data sequence localization.

Our temporal locality optimization method is presented in Sect. 2. Some prerequisites on unimodular transformations and data dependences are given first. Then the method is detailed step-by-step by presenting our temporal optimization algorithm for one reference, then for any number of references. Section 3 describes the positive effects of our method on parallel programs. After some comparisons with closer related work have been detailed in Sect. 4, conclusions and future objectives are given in Sect. 5.

2 Loop transformations for temporal locality

Loop nests and array references. We assume that loops are normalized such that their step is one. The iteration space of a loop nest of depth d is a d-dimensional convex polyhedron D_P. It is bounded by parameterized linear inequalities imposed by the affine loop bounds and hence is a parameterized convex polyhedron, where P is a p-vector of integer parameters: $P = (N_1, ..., N_p)$.

The references are affine combinations of the indices and the parameters. A reference to a d'-dimensional array element inside a d-dimensional loop nest is represented by a *homogeneous* integer reference matrix R, of size $(d' + p + 1) \times (d + p + 1)$. This matrix defines a reference in a space including the index variables, as well as the parameters and the constant. A reference is an affine mapping $f(I, P) = R\ (I, P, 1)^\top$, where $^\top$ denotes the transpose vector.

We consider one loop nest containing some references to optimize. Each reference i in the loop nest is associated to an accessed set of data $Data_i$, and all references are ordered by decreasing sizes of their data sets. The largest data sets are more likely to induce large strides resulting in many cache and TLB misses. Hence, our heuristics selects the different reuse directions associated with loops ordered from the innermost to the outermost loop, following the descending size of the associated sets of accessed data $\#Data_i$. 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 [4].

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, reversal and skewing [22].

Unimodular transformations. Let T be a unimodular $(d+p+1) \times (d+p+1)$ matrix. This matrix defines a *homogeneous* affine transformation t of the iteration domain as: $t : D_P \subset \mathbb{Z}^d \rightarrow D'_P = t(D_P) \subset \mathbb{Z}^d$

$$I \mapsto I' = t(I) \text{ such that } (I', P, 1)^\top = T\ (I, P, 1)^\top.$$

The transformed domain D'_P corresponds to a new scanning loop nest, obtained by applying the Fourier-Motzkin algorithm [2] to a perfect loop nest. This algorithm computes the lower and upper bounds of each iteration variable I'_k as a function of the parameters and the variables $I'_1 ... I'_{k-1}$ 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 geometrical space (same dimension and same variables). This can be done by using a variant of code sinking [12, 22]. Then, the transformation is applied to all these iteration domains. To get the resulting loop nest we apply Quilleré's algorithm [15], which constructs an optimized loop nest scanning several iteration domains.

Data dependences and validity of loop transformations. In order to be valid, the transformation has to respect the dependences of the loop. 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 [2]. We denote by $\preceq$ the lexicographical *lower or equal* operator.

Let $\mathcal{D}$ be the set of *distance vectors* related to data dependences occurring in the original loop nest: $\delta \in \mathcal{D}$ *iff* there exist two iterations $I \preceq J$, such that $J = I + \delta$, and there is a data dependence from iteration I to iteration J. 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 [2]: $t(\delta) \succ 0$ for each positive δ in $\mathcal{D}$.

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 $(d' + p + 1) \times (d + p + 1)$, where d' is the dimension of the array. 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 of matrix R has to be lower than $(d + p + 1)$.

The image defined by matrix R to 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 optimized inner loops. It is a polytope depending on parameter d_0. It is computed by applying function preimage R^{-1} to d_0, intersected with the

iteration domain D_P. Let $\mathcal{H} = Affine.Hull(R^{-1}d_0 \cap D_P)$. Let matrix B_T be the set of column-vectors generating $\mathcal{H}$. This matrix contains the scanning directions of the optimized inner loops.

Let $B = \left(\begin{array}{c|c} (B_D|B_T) & 0 \\ \hline 0 & Id \end{array}\right)$ in the homogeneous space. Matrix B_D is a basis of the space scanning each accessed data. It must be chosen in order for B to be unimodular and full rank (using the Hermite normal form theorem for example), and also to satisfy the dependences. As a result, B is a basis for the optimized scanning loops, the outermost loop corresponding to the leftmost column vector of B. Transformation matrix T is equal to B^{-1}.

Example 1. Consider the loop nest presented on the left of Tab. 2. There is an output dependence for array A, of distance vector $\delta = (0, 1, -1)^\top$. The reference matrix to variable A, is $R = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & -1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$.

Table 2. Optimizing a loop nest containing one reference.

original loop nest	optimized loop nest
`do i = 1, N`	`do u = 1, N`
`  do j = 1, N`	`  do v = 2, N+u`
`    do k = 1, i`	`    do w = max(1,v-N), min(u,v-1)`
`      A[i+N, j+k-1] = f(i,j,k)`	`      A[u+N, v-1] = f(u,v-w,w)`

Each point $d_0 = (x_0, y_0)^\top$ of the data space corresponds to the scanning of the following polytope: $R^{-1}d_0 \cap D_N = \{(x_0, y_0 - k, k)^\top \in D_N\}$, where D_N is the iteration domain. The vector supporting the affine hull of this polytope is $B_T = \begin{pmatrix} 0 \\ -1 \\ 1 \end{pmatrix}$. The dependence is satisfied with $B_D = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$.

The transformed loop nest is obtained by applying $T = B^{-1}$ to the iteration domain, and by transforming the references to I by $t^{-1}(I')$. The result is given Tab. 2, with $I' = (u, v, w)^\top$. Reference to variable A has been temporally optimized in the innermost loop: index w does not appear in the reference.

Multiple references optimization. The main objective of our approach is to reduce stride sizes generated from any reference. The largest stride that can be generated by a reference is the size of its associated data set. Hence, our heuristic states that references associated with the largest data sets should be optimized first. This introduces the concept of *data sequence localization*. Consider a loop nest containing n references to the same or to different arrays. The data set sizes are determined by computing the Ehrhart polynomials of the affine transformation of D_P by the reference matrices [4]. Then we compute the subspaces $\mathcal{H}_i$ where temporal reuse occurs for each reference i, as described below: $\mathcal{H}_i = Affine.Hull(R_i^{-1}d_0 \cap D_P)$.

The algorithm consists in 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 successively selects the 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:

1. Order the n references by decreasing data set sizes.
2. Compute the spaces $\mathcal{H}_i$ for each reference, $1 \leq i \leq n$.
3. for $col = d$ downto 1 do
   (a) 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 sets $\mathcal{H}_i$. If there are no more references to optimize choose a direction such that B is unimodular.
   (b) Put the vector $\mathcal{T}$ in the column col of matrix B. Check the dependences and the unimodularity of B; if they are not satisfied, go back to step 3a and choose another direction.

The unimodularity of B is checked using the Hermite normal form theorem ([17], corollary 4.1c): at each step of the algorithm, the column vectors of B should 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.

Example 2. Consider the 4-dimensional loop nest presented in Tab. 3. We choose first 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 one dependence on variable A, of distance vector $(0,\ 0,\ 0,\ 1)^\top$.

Table 3. A loop nest containing 3 references.

```
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+k,j] + B[i,l+k]
```

The first step of the algorithm consists in computing the linear spaces of temporal reuse $\mathcal{H}_i$, for $i = 1, 2, 3$. $\mathcal{H}_1$ is supported by vector $(0,\ 0,\ 0,\ 1)^\top$, $\mathcal{H}_2$ is supported by $(1,\ 0,\ 0,\ 0)^\top$ and $(0,\ 0,\ 1, -1)^\top$, and $\mathcal{H}_3$ is supported by $(0,\ 1,\ 0,\ 0)^\top$ and $(0,\ 0,\ 1, -1)^\top$. The algorithm then selects four new scanning vectors in this order:

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

Table 4. Optimized loop nest.

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

Table 5. Performance results

$N = 700$	Original loop	Loop blocking	Optimized loop
# L1 data cache misses	52,283,667,715	747,493,318	777,072,089
# L2 data cache misses	23,082,872	53,590,447	19,351,070
# TLB misses	1,345,342,838,661	127,516,826	124,344,788
computation time	10,873s	1,757s	1,688s

- vector $(1,\ 0,\ 0,\ 0)^\top$ optimizes reference R_2.
- vector $(0,\ 1,\ 0,\ 0)^\top$ optimizes reference R_3.

Finally, we get $B = \begin{pmatrix} 0&1&0&0&0&0 \\ 1&0&0&0&0&0 \\ 0&0&0&1&0&0 \\ 0&0&1&-1&0&0 \\ 0&0&0&0&1&0 \\ 0&0&0&0&0&1 \end{pmatrix}$ and $T = B^{-1} = \begin{pmatrix} 0&1&0&0&0&0 \\ 1&0&0&0&0&0 \\ 0&0&1&1&0&0 \\ 0&0&1&0&0&0 \\ 0&0&0&0&1&0 \\ 0&0&0&0&0&1 \end{pmatrix}$.

The dependence is satisfied and the resulting loop nest is presented in Tab. 4. Table 5 shows our experimental results for $N = 700$, on three version of the loop: the original loop nest without blocking nor interchange optimization, the SGI F90 compiler version allowing autoblocking and interchange of loops, and our version. These mesures show an important performance improvement from the original loop to the two optimized versions, and some more gain in the number of L2 data cache misses and TLB misses from the blocking version to our optimized version. These L2 and TLB misses have been transformed into L1 misses in our version, since the reused data sequences fit the lower memory hierarchy level. Getting even better results could be possible by now through our spatial locality optimization [6].

3 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 on massively parallel architectures: when a processor brings a data page into its local memory, it will reuse it as much as possible due to our temporal optimization. This yields significant reductions in page faults and hence in network traffic.

We can also say, as mentioned by Kandemir *et al.* in [9], 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, all the remaining accesses will be local.

Temporal optimizations presented in Sect. 2 often generate outer loops that carry no reuse and no data dependences. Hence, these outer loops are perfect candidates for parallelization since distinct iterations do not share any data.

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

4 Related work

In [7], Ding and Kennedy define the concept of *reuse distance* as being the number of distinctive data items appearing between two closest references to a same data. Although their objective of minimizing such distances seems quite similar to ours, they rather consider fusion of data-sharing loops. Reuse distance minimization inside a loop nest with multiple references is not considered at all.

Ciernak and Li in [3] introduce the concept of *reference distance* as being the number of different cache lines accessed between two references to the same cache line. However this item does not guide their optimization process, but is used as a metric of quality. They also define *stride vectors* as being vectors composed of the strides associated to each loop level. Their objective is to obtain such a vector with elements in decreasing order, whereas our objective is to minimize all elements of this vector, while breaking the largest strides into smaller ones. For example, in our method, stride vector (N, N) will be prefered to vector $(N^2, 1)$.

5 Conclusion

We have shown that significant performance improvements can be obtained in programs through data sequence localization. Our method, compared to blocking, is independent from the target architecture and hence does not need fine tuning relatively to the memory hierarchy specifications. We also have proposed an original approach for spatial locality optimization [6] which can be naturally unified with the temporal optimization method of this paper in order to get even better program performance.

All the geometric and arithmetic tools we used are implemented in the polyhedral library *PolyLib* (http://icps.u-strasbg.fr/Polylib). We are currently implementing the method presented in this paper in an environment for source to source transformations of Fortran programs. Once implemented, we are going to validate our method on some larger benchmarks.

We are now studying other important issues such as memory compression, since further performance improvements can be expected, and memory size minimization which is essential in embedded systems. We also investigate another approach, by considering architectural parameters characterizing the target architecture: cache size, cache associativity, cache line size, TLB reach, etc. In addition to data locality optimization, other important issues related to efficient memory use can be considered, such as array padding and cache set conflicts.

References

[1] U. Banerjee. Unimodular transformations of double loops. In *Advances in Languages and Compilers for Parallel Processing*. MIT Press, Cambridge, MA, 1991.

[2] U. Banerjee. *Loop Transformations for Restructuring Compilers - The Foundations*. Kluwer Academic Publishers, 1993. ISBN 0-7923-9318-X.

[3] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared-memory machines. In *Proc. Prog. Lang. Design and Implementation*, 1995.

[4] Ph. Clauss. Handling memory cache policy with integer points countings. In *Euro-Par'97*, pages 285–293, Passau, August 1997. Springer-Verlag, LNCS 1300.

[5] Ph. Clauss and V. Loechner. Parametric analysis of polyhedral iteration spaces. *Journal of VLSI Signal Processing*, 19(2):179–194, 1998. Kluwer Academic Pub.

[6] Ph. Clauss and B. Meister. Automatic memory layout transformations to optimize spatial locality in parameterized loop nests. *ACM SIGARCH Computer Architecture News*, 28(1):11–19, March 2000.

[7] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. In *Proc. of the 2001 International Parallel and Distributed Processing Symposium, San Francisco*, April 2001.

[8] P. Feautrier. *The Data Parallel Programming Model*, volume 1132 of *LNCS*, chapter Automatic Parallelization in the Polytope Model, pages 79–100. Springer-Verlag, 1996. G.-R. Perrin and A. Darte, Eds. ISBN 3-540-61736-1.

[9] 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.

[10] M. Lam, E. Rothberg, and M. Wolf. The cache performance of blocked algorithms. In *Int. Conf. ASPLOS*, April 1991.

[11] W. Li. *Compiling for NUMA parallel machines*. PhD thesis, Dept. Computer Science, Cornell University, Ithaca, NY, 1993.

[12] V. Loechner, B. Meister, and Ph. Clauss. Precise data locality optimization of nested loops. Technical report, ICPS, http://icps.u-strasbg.fr, 2001.

[13] V. Loechner and D. K. Wilde. Parameterized polyhedra and their vertices. *International Journal of Parallel Programming*, 25(6):525–549, December 1997.

[14] M. O'Boyle and P. Knijnenburg. Nonsingular data transformations: Definition, validity, and applications. *Int. J. of Parallel Programming*, 27(3):131–159, 1999.

[15] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *Int. J. of Parallel Programming*, 28(5):469–498, October 2000.

[16] J. M. Rabaey and M. Pedram. *Low Power Design Methodologies*. Kluwer Academic Publishers, 1995.

[17] A. Schrijver. *Theroy of Linear and Integer Programming*. John Wiley and Sons, New York, 1986. ISBN 0-471-90854-1.

[18] 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*, pages 204–213, June 1998.

[19] D.K. Wilde. A library for doing polyhedral operations. Master's thesis, Oregon State University, Corvallis, Oregon, 1993.

[20] M. Wolfe. More iteration space tiling. In *Proc. Supercomputing'89*, pages 655–664, November 1989.

[21] M. Wolfe. The tiny loop restructuring research tool. In *International Conference on Parallel Processing*, pages II. 46–53, 1991.

[22] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison Wesley, 1996. ISBN 0-8053-2730-4.