Automatic Data Mapping of Signal Processing Applications*

Corinne Ancourt[2] & Denis Barthou[2]& Christophe Guettier[1,2] & François Irigoin[2] & Bertrand Jeannet[1] & Jean Jourdan[1] & Juliette Mattioli[1]

[1] LCR, Thomson-CSF, Domaine de Corbeville, F-91404 ORSAY CEDEX, France.
[2] CRI, ENSMP, 35 rue Saint-Honoré, F-77305 Fontainebleau CEDEX, France.

Abstract. This paper presents a technique to map automatically a complete digital signal processing (DSP) application onto a parallel machine with distributed memory. Unlike other applications where coarse or medium grain scheduling techniques can be used, DSP applications integrate several thousand of tasks and hence necessitate fine grain considerations. Moreover finding an effective mapping imperatively require to take into account both architectural resources constraints and real time constraints. The main contribution of this paper is to show how it is possible to handle and to solve data partitioning, and fine-grain scheduling under the above operational constraints using Concurrent Constraints Logic Programming languages (CCLP). Our concurrent resolution technique undertaking linear and non linear constraints takes advantage of the special features of signal processing applications and provides a solution equivalent to a manual solution for the representative *Panoramic Analysis (PA)* application.

Keywords: parallelizing compiler, scheduling, constraint logic programming

Introduction

The post World War II era has resulted in the trend of using Digital Signal Processing (DSP) technologies for both military and civilian applications. The growing requirements for sophisticated algorithms, especially those used for 3-D applicative domains, lead to process in real time large multi-dimensional arrays of data. These applications are executed on parallel computers, that offer enough computing power [25].

The mapping of DSP applications onto parallel machines raises new problems. The real time and target machine constraints are imperative. The solution must fit the available hardware: the local memory, the number of processors, the processor communications. The application latency must meet the real time requirements. This necessitates fine-grain optimizations. Combining both kinds of constraints is still out of the scope of automation and requires deep human skills.

* Submitted to ASAP'97

This paper presents a new technique to map automatically DSP application, represented by a sequence of loop nests, onto a SPMD distributed memory machine. This technique is based on formalizations of the architectural, applicative and mapping models by constraints. The result is (1) a fine grain affine schedule of computations, (2) their distribution onto processors and (3) a memory allocation. Computations are distributed in a block-cyclic way on processors. Communications are overlapped with computations when possible. The memory model is precise: Only the amount of memory useful to the computations is allocated.

The general mapping problem (data and computation onto processors) has been proved to be NP-complete [36, 37]. While data dependence constraints can be translated into linear inequations and then solved by classical linear programming algorithms, resource constraints require non linear expressions. Solving directly both constraints is still out of the scope of any general algorithms and necessitates the combination of integer programming and search [24]. Following the same idea of combining constraints solving and nondeterminism, our technique uses a CCLP [19, 53] approach. Unlike conventional constraint solvers based on black box algorithms, CCLP languages use an incomplete constraint solvers over a finite domains algebra. The two main advantages of using such algorithm are first to enhance compositionality features [52, 31] and secondly to offer basic control structures for expressing new constraints [52].

Our approach takes as input the specification of different models such as: the target machine, the communication cost, the application, the partitioning, the data alignment, the memory allocation and the scheduling models. Then, the CCLP assets enable to handle linear and non linear expressions and to yield, through the concurrent propagation of the constraints over all the models to solutions, satisfying the global problem. The solution outlook depends on multiple criteria as memory allocation or latency.

The article is organized as follows. Firstly, the characteristics of the target machine and DSP applications are presented. Secondly, our constraint formalization of the problem is exposed: Especially, the partitioning, scheduling and memory models are detailed. Thirdly, the concurrent resolution programming technique is presented, followed by our prototype results. Finally, a comparison with other approaches is described before concluding.

1 Architectural and Applicative features

This section presents an overview of the architectural and application features that characterize our general mapping problem formulation.

1.1 Architectural features

The target machine is an abstract SPMD distributed memory machine. The mapping is constrained by machine resources:

Number of processors. The application is mapped on all processors. However, criteria like memory allocation or communication minimization may enforce the use of fewer processors.

Local memory size. Because there is no global memory, the amount of memory necessary to execute a set of computations, mapped onto a processor at a given moment, must fit the available processor memory.

Processor rate. The latency criteria (amount of time between one input and the corresponding output) can be fixed to a maximum value.

Overlap of computations and communications. The partitioning model takes advantage of this property to overlap communications with computations.

1.2 Applicative features

In this section the DSP applicative features are described. These features have been investigated for several years at Thomson-CSF by A. Demeure.

The application is a sequence of loop nests in a single-assignment form It describes an acyclic graph of tasks. Each loop nest includes a procedure call that reads one or several multidimensional data arrays and updates one different array. Array accesses are affine functions of loop indices with eventual modulo. Figure 1 presents a global view of PA application [8]. Figure 2 details the first PA loop nest.

Parallelism. Since the application is in a single-assignment form, each loop nest is full-parallel. Furthermore, the loops are perfectly nested.

Procedures can be seen as black boxes where computational dependencies are encapsulated. Procedures are DSP library calls, such as *Fast Fourier Transform.* Our approach schedules the application at this procedural level.

Arrays have one infinite dimension, due to the real time constraint. The computational recurrence extraction from the application puts forward a cyclic schedule of a finite amount of computations. Then, classical parallelization techniques can be used.

DSP applications manipulate array references that can be represented by `Read` and `Write` regions [50, 12]. `Read` and `Write` regions represent, with affine constraints [13], the set of array elements read and written by the procedure. Figure 2 gives the FFT read and write regions. As procedures are generally DSP library calls, these regions should be allocated fully in the local memory during the procedure computation.

2 Constraint Formalization

Our technique uses a multi-model approach [32] to describe the general mapping problem. Due to space limitation, only the partitioning, scheduling and memory models are presented here. But the communication, latency, architectural and applicative models obviously influence the resolution.

```
doall r,c
  call FFT(r,c)
enddo
doall r,f,v
  call BeamForming(r,f,v)
enddo
doall r,f,v
  call Energy(r,f,v)
enddo
doall r,v
  call ShortIntegration(r,v)
enddo
doall r,v
  call AzimutStabilization(r,v)
enddo
doall r,v
  call LongIntegration(r,v)
enddo
```

Fig. 1. *Panoramic Analysis* application

```
do r=0,infinity
 do c=0,511

  c   Read Region:
  c        SENSOR(c,512*r:512*r+511)
  c   Write Region:
  c        TABFFT(c,0:255,r)

  call FFTDbl(SENSOR(c,512*r:512*r+511),
            TABFFT(c,0:255,r))

 enddo
enddo
```

Fig. 2. FFT Loop nest

2.1 Partitioning Model

The partitioning model is designed to map computations onto the target machine. Since DSP applications are sequences of parallel loop nests, the partitioning problem results in loop nest by loop nest partitioning.
The multidimensional iteration domain (I) is partitioned, and computations are not replicated:

$$I = \bigcup_{i=1}^{n} Part_i, \quad \forall j,\ 1 \leq j \neq i < n,\ Part_i \bigcap Part_j = \emptyset$$

The application parallelism degree, memory location requirement and time scheduling parameters are controlled by the partitioning. The iteration domain is decomposed over 3 vector parameters: x, y, z. Block, cyclic and block-cyclic distributions are possible.

$$\forall i \in I, \quad \begin{cases} i = LPx + Ly + z \\ \forall z, 0 \leq L^{-1}z < 1, \quad \forall y, 0 \leq P^{-1}y < 1 \\ det(L) \neq 0,\ det(P) \neq 0 \end{cases}$$

P and L are diagonal square integer matrices. Except for the infinite dimension, the 3 parameters can be assigned independently to Processor p, Cyclic recurrence c or Local memory l. The finite resource constraints imply: $x = c$ for the infinite dimension. The case where $(x, y, z) = (c, p, l)$ implies that $\max(l) = \prod_i L_{ii}$ is the number of local iterations executed by one processor at each cycle c (each local iteration execute a procedure call). $\max(p) = \prod_i P_{ii}$ gives the maximum number of processors and $\max(c)$ the maximum number of synchronizations (cycles) necessary for the loop nest completion.

Due to DSP application features, the array access functions use at most per array dimension one external loop index and one internal loop index which scans the read or write region. Since read and write regions are not partitionable, only the external loop nest is partitioned. So, partitioning matrices are diagonal (with an eventual permutation). Figure 4 presents the PA loop nest partitioning. It expresses that 1 iteration `r` (L_{11}) and 64 (L_{22}) iterations `h` (see Figures 1,2) are mapped on each of the 8 ($P_{11} * P_{22}$) processors.

2.2 Scheduling Model

The scheduling model is designed to associate to each computational block a logical execution event on a processor. The resulting schedule can be viewed as a succession of loop transformations. In general, it is not possible to find *automatically* the transformation set to apply such that the final schedule is optimal. So, the affine scheduling approach, used in systolic arrays and parallelization techniques [23, 22, 15, 16, 17], is chosen and applied to our context.

The partitioning model states that elementary computations having to be scheduled (called *block* hereafter) are the set of L pipelined local iterations mapped onto p at cycle c. Since the programming model is SPMD, p does not need to appear in the schedule formulation. Thus, it only depends on vector c which fully describes the block of l iterations to perform. We choose the affine schedule class of events to search as:

$$d^k(c^k) = N(\alpha^k \cdot c^k + \beta^k) + k$$

Variables are indexed by the loop nest number k. d^k is the scheduling function of the k^{th}-loop nest. α^k and β^k are the scheduling affine parameters. α^k is a line vector, and β^k is scalar. N is the number of loop nests. It is used in the formulae with the offset $+k$ in order to avoid the execution at the same date of two computations belonging to different loop nests.

In the same way, two computational blocks of a single loop nest cannot be executed at the same date. Let c_i^k and c_j^k with $i < j$ be two cyclic components of the partitioned loop nest N^k. Then, the execution period of Cycle c_i^k must be greater than the execution time of all cycles c_j^k. Hence, Constraints: $\alpha_i^k > \sum_{j>i} \alpha_j^k \max(c_j^k)$ with $\alpha_n^k \geq 1$ must be verified.

As an example of additional constraints that link the partitioning and scheduling models, the data flow dependencies express that a piece of data of loop nest N^r cannot be read before being updated by N^w. These dependencies between two cycles c^w of loop nest N^w and c^r of N^r imply that:

$$\forall(c^w, c^r)\ Dependence(c^w, c^r) \Rightarrow d^w(c^w) + 1 \leq d^r(c^r) \qquad (1)$$

d^w (respectively d^r) is the scheduling associated to N^r (resp. N^w).

Note that these dependencies are computed between iterations of *different* loop nests. Data flow dependencies are approximated by their convex hull representation. However, this approximation lets us to obtain the same set of valid schedules as with the exact representation without any loss. Due to DSP application characteristics, this representation can remain symbolic. This improves the constraints propagation, since no costly algorithm is needed to solve the dependence test.

2.3 Memory model

The memory model ensures the application executability under a memory constraint. A capacitive memory model is used. It evaluates the memory required for each computational block mapped onto a processor by analyzing the data dependencies. An allocation function can be extracted straightforwardly from the memory allocation result when the schedule is known after the optimization phase.

The number of data blocks needed to execute the computational block is computed. Due to the partitioning model all computational blocks have the same simple structure and the same size. Data dependencies are used to determine the data block life time. A data block is *alive* from its creation date (corresponding to its allocation) to its last use date. For each computational block, the schedule and data dependencies give the maximum life time of a data block and the number of data block creations during one cycle. This gives the required memory capacity per computational block and cycle. The addition of the different computational block memory requirements give the amount of memory necessary to the complete application execution.

The memory is organized in segments of identical data blocks, one per loop nest. This eliminates the problems of memory fragmentation and the eventual need of block relocation. Data duplications due to input sets of references overlap between successive iterations are eliminated by using partial data block decompositions. Only new partial data blocks are kept and fused to others. This refinement is powerful enough to handle any multidimensional read overlaps and proved very efficient on the studied DSP applications.

3 Resolution

Constraint logic programming is a generalization of logic programming where unification is replaced by constraint solving over several computation domains. These domains include linear rational arithmetics, boolean algebra, Presburger arithmetics and finite domains [20]. More recently the introduction of the notion of constraint entailment, stemming from the *Ask & Tell* paradigm of concurrent programming [46], enhanced the CCLP framework with synchronism mechanisms. This new class of CCLP (see fig. 3) languages [42, 52] offers control structures enabling in one hand a better interleaving of the goals of several models and on another hand a new way to define non-primitive constraints.

Tell: Satisfaction mechanism

$$P, \mathcal{T}h(\mathcal{S}) \models (\exists)c$$

Ask: Entailment mechanism

$$P, \mathcal{T}h(\mathcal{S}) \models (\forall)(\sigma \rightarrow c)$$

where P is a CCLP program, $\mathcal{T}h(\mathcal{S})$ a theory of the S algebra, σ a guard and c a constraint.
CCLP program: Set of logic rules of the form $\{A \leftarrow a \;\; c \mid A_1, ..., A_k\}$
where a, c et $\{A_i\}$ represent respectively a set of constraints of type **ask**, of constraints of type **tell** and logical atoms.

Fig. 3. CCLP programs and its two basic mechanisms

The cardinality operator $\#(l, u, [c_1, ..., c_n)$ [51], the constructive disjunction operator $\bigtriangledown(c_1, ..., c_n)$ [31], the entailment and the conditional propagation operators are some examples of new connectives of CCLP languages. From an operational standpoint, they are based on constraint solving, constraint entailment and arithmetic reasoning. Going in deeper details on CCLP is out of the scope of this paper but we have used these new capabilities to extend our CCLP languages Meta(F) [11] in order to solve efficiently polynomial constraints over finite domain variables.

Thanks to their unique combination of constraint-solving, nondeterminism, and relational form, CCLP languages have been shown to be very effective in solving complex and heterogeneous problems [53, 30] comparable in efficiency to specialized algorithms. In the two next sections we show how we can handle and solve our mapping problem using such kind of new language.

3.1 How does CCLP handle our global mapping problem ?

The mapping models such as partitioning and scheduling are represented with mathematical variables and affine constraints. Non-linear constraints link the different models and generally are composed with complex and polynomial terms. For example, constraint (2) links partitioning and architecture models. The number of processors required by the partitioning must be smaller than the number of processors available.

$$NumberOfProcessors \geq max_k(\Pi_{i=1}^{n}(P_{i,i}^{k})) \tag{2}$$

The latency, resources and data-flow dependencies constraints (1) are global constraints.

The effective CCLP expressions of the global mapping problem has required an in-depth collaboration between CCLP and Parallelism specialists. The fine grain models, issued from parallelization techniques, induce a CCLP model mostly based on the expression of sets of procedure calls, data blocks and dependency relationships. Those sets are represented as intension rather than extension models.

In some cases, this task was impossible to perform directly and the proposed models have to be recasted in a set of expressible constraints representing an approximation of the model. For instance, the dependency relationships between blocks of computation cannot be stated in the original constraint (1) due to the $\forall(c^w, c^r)$. The constraint has been implemented as constraint (3):

$$\forall(c^w_s, c^r_s) \quad d^w(c^w_s) + 1 \leq d^r(c^r_s) \tag{3}$$

where (c^w_s, c^r_s) are the vertex components of the convex hull of the dependencies, that have been computed symbolically. Hence, the scope of this $\forall$ is restricted to the number of vertices.

3.2 How does CCLP solve our global mapping problem ?

While storing the different constraints, the CCLP system builds a solution-space on a model-per-model basis. Each model solution space is pruned when constraints are propagated from other models. Once all models have been built into the system, non-linear constraints linking the different models still have to be met. Solutions must be looked for in a resulting overall search space using a specific global search.

This search relies (1) on the semantic of the variables of each model and their importance w.r.t. other models and (2) the goal to achieve (i.e. resource minimization under latency constraint, latency minimization under resource constraint).Each variable takes part in a global cross-model composite solving, such that only relevant information is exchanged between models. The global search looks for partial solutions in the different concurrent models. For instance, the set of scheduling variables $(\boldsymbol{\alpha}_i, \beta_i)$ and partitioning matrices P_i, L_i are partially instantiated by inter-model constraints during the resolution. Model-specific or more global heuristics are used to improve the resolution:e.g. schedule choices are driven by computing the shortest path in the data-flow graph.

Based over models semantic and specific heuristics, the global mapping problems is solved through CCLP using complex composition schemes.

If dedicated algorithms are used, the composition of the different functions only is possible by sequential solving according to the functional programming paradigm. It restricts the composition facilities and has a too high complexity. Traditional generic solvers, as Simplex, are designed to solve only linear constraints in a convex rational context. The Simplex category algorithms does not support models cooperation.

Integer programming allows to recast complex non-linear constraints using boolean variables. Therefore, links between models are represented using boolean variables which restricts partial information exchanges between models.

4 Results

This section illustrates our prototype results. The user specifies the target machine and the option criteria. In this example, the optimizing cost function is the *memory size minimization*. The target machine has 8 processors. The latency constraint is set to 4.10^8 processor clock cycles and the memory is unbounded. Figure 4 describes the partitioning and schedule of PA. The loop nest parallelism and locality are expressed with the diagonal matrices `P` and `L` .

4.1 Partitioning

The partitioning characteristics follow. (1) Only finite dimensions are mapped onto the 8 processors. This solution satisfies the latency constraints. (2) The write region of the second loop nest is identical to the read region of the third loop nest. So the system fuses these loop nests in order to reduce memory allocation. (3) The access analysis of the second and third loop nests presents read region overlaps between successive iteration execution. This overlap is detected. The system parallelizes according to another dimension to avoid data replication.

Partitionning	*FFT*	*Beam Forming, Energy*	*BroadBand*	*Sht Integ*	*Azimut*	*Long Integ*
Parallelism, $P =$	$\begin{pmatrix}1&0\\0&8\end{pmatrix}$	$\begin{pmatrix}1&0&0\\0&1&0\\0&0&8\end{pmatrix}$	$\begin{pmatrix}1&0\\0&8\end{pmatrix}$	$\begin{pmatrix}1&0\\0&8\end{pmatrix}$	$\begin{pmatrix}1&0\\0&8\end{pmatrix}$	$\begin{pmatrix}1&0\\0&8\end{pmatrix}$
Locality, $L =$	$\begin{pmatrix}1&0\\0&64\end{pmatrix}$	$\begin{pmatrix}1&0&0\\0&128&0\\0&0&25\end{pmatrix}$	$\begin{pmatrix}1&0\\0&16\end{pmatrix}$	$\begin{pmatrix}1&0\\0&16\end{pmatrix}$	$\begin{pmatrix}1&0\\0&16\end{pmatrix}$	$\begin{pmatrix}1&0\\0&16\end{pmatrix}$

Scheduling	*FFT*	*Beam Forming, Energy*	*BroadBand*	*Sht Integ*	*Azimut*	*Long Integ*
α	$\begin{pmatrix}6\\1\end{pmatrix}$	$\begin{pmatrix}6\\1\\1\end{pmatrix}$	$\begin{pmatrix}6\\1\end{pmatrix}$	$\begin{pmatrix}48\\1\end{pmatrix}$	$\begin{pmatrix}48\\1\end{pmatrix}$	$\begin{pmatrix}384\\1\end{pmatrix}$
β	0	1	2	45	46	383

Fig. 4. Partitioning and Scheduling matrices for Panoramic Analysis

4.2 Schedule

According to the different partitions, only the time dimension is globally scheduled. From the α and β scheduling parameters in Figure 4, the schedule can be expressed using the regular expression:

$$(((FFT, [BF, E], BB)^8, SI, SA)^8, LI)^\infty$$

Computational dependencies between iterations are satisfied. The system provides a fine grain schedule at the procedural level using the dependence graph shortest-path. This enables the use of data as soon as possible, avoids buffer allocations, and produces output results at the earliest. On the right hand side, the corresponding loop nest is represented.

```
do ili=0,infinity
  do isa=0,7
    do ibb=0,7
      FFT(ibb)
      BeamFormingEnergy(ibb)
      BroadBand(ibb)
    enddo
    ShortInteg(isa)
    StabAzimut(isa)
  enddo
  LongInteg(ili)
enddo
```

Eight iterations of Tasks `FFT,BF-E,BB` (executed every $\alpha_1^1 = 6$ steps) are performed before one iteration of `SI,SA` (executed every 48 = 6*8 steps). The last task `LongInteg` cannot be executed before 8 iterations of the precedent ones. So it is executed every 384 (=8*48) steps.

4.3 Comparison with manual mappings

Manual mappings of DSP applications are performed in different ways. In general, user-friendly interfaces provided by manufacturers offer some help for *coarse grain parallelism.* The application is scheduled at the task level and not at the procedural level. Thus, load balancing is more difficult to obtain.

While it is hard for a human being to instantiate the different models satisfying all constraints, we have compared our solution to two different manual solutions. The first one is based on loop transformation techniques. The second one uses the maximization of the processor usage as only economic function. Our result is equivalent to the one suggested by parallelization techniques. It is better than the second one which requires more memory allocation.

4.4 Towards global optimization

Between two successive solutions, the system takes important decisions to optimize the mapping. The optimization trace is shown in Figure 5.

The first solution is obtained in a few minutes while this optimization is completed in ten minutes on a SPARC-10 Workstation. These times have to be compared with human being inquiries to comprehend and map the application.

Row 0 represents the original set of constraints: a large initial memory size, 8 processors, and a quite restricted latency. Solution 1 gives a bad partitioning of the fused loop nests `Beam Forming-Energy`, and produces an allocation with data replication. Solutions 2 and 3 are mixed: parallelism is set on different dimensions. Solution 4 maps parallelism on the appropriated dimension, thus minimizes data replication. Finally, the system finds that taking 4 processors still satisfy the latency constraint and reduces memory cost.

Memory Optimization			
sol. numb.	Nb proc.	Memory Kwords	Latency Mcycl.
0	8	10000.0	400
1	8	1358.5	175
2	8	1057.7	175
3	8	907.3	175
4	8	832.1	175
5	4	832.0	350

Fig. 5. Optimizing the memory size

5 Related Work

Mapping applications onto parallel machines addresses issues such as scheduling [10], parallelization [1], loop transformations [29, 4, 38], parallel languages [35, 28, 3], integer linear programming and machine architecture. A lot of work has been done to optimize a few criteria such as data and/or computation distribution [40, 21, 34, 7, 43], parallelism detection, minimization of communications [18, 2, 41, 5], processor network usage. This section focuses on the most relevant work.

Although manual loop transformation techniques are attractive and give good results, it is not possible to find automatically the transformation set to apply for obtaining the optimal schedule [33, 9]. However restructuring the application such that the parallelism and data locality are maximized is yet a relevant objective. Many studies [9, 41, 48] present interesting approaches. Thereafter, the compiler is in charge of mapping physically the optimized application of the target machine. Compared to our approach, there is no real time and architectural constraints (number of processors and memory resources) to take into account during the parallelization phase.

Similar techniques are used in systolic arrays [16, 17, 14] and parallelization [23, 22, 26] communities to compute affine schedules. In the systolic community, these techniques are applied on a single loop nest with complex internal dependencies. The other approaches dealing with complete applications, do not have the same architectural and application constraints. The parallelism grain is at the instruction level, there is no real time constraint and the target machine is generally virtual.

DSP application features are taken into account in [45]. This approach is based on task fusion, but for a sequential result. Mapping statically DSP application with specific signal requirements [27, 49] have been widely investigated. The representative Ptolemy framework [39, 47, 44] brings some solution but at a coarse grain level. Most of the resolution schemes are based on dedicated algorithms [6].

Our approach is the first one to propose an optimal *affine* schedule of a *complete* application with a *fine grain* parallelism (at the procedural level) and its mapping onto a architecture under resource and real time constraints.

Conclusion

A technique to map automatically DSP applications onto distributed memory machines has been introduced in this paper. It uses a multi-model approach to describe the general mapping problem and a concurrent resolution framework based on the Constraint Logic Programming. Even if the presented model constraints are linear, our system comes to terms with non-linear constraints.

Our experiences on DSP benchmark show that our prototype takes into account all architectural and applicative parameters. Sequential, pipelined and parallel schedules are generated depending on the applications. Comparisons with manual solutions proves that our approach may provide interesting, indeed better, solutions.

Future work focuses on developing strategies to speed-up the solution enumeration and on extending the set of applications automatically proceed.

Acknowledgments

We wish to give special thanks to F. Coelho for his constructive remarks and critical reading of this paper. We also thanks T. Brizard, P. Legal and B. Marchand for their continuous support.

References

1. J.R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.
2. J.M. Anderson and M.S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *SIGPLAN Conf on Programming Language Design and Implementation*, pages 112–125, Albuquerque, NM, June 1993. ACM Press.
3. Françoise André, J.-L. Pazat, and Henry Thomas. Pandore: a system to manage data distribution. In *Int. Conf. on Supercomputing*, pages 380–388, June 1990.
4. U. Banerjee. Unimodular transformations of do loops. Technical Report CSRD Rpt. No. 1036, University of Illinois, August 1990.
5. D. Bau, I. Kodukula, K. Pingali, and P. Stodghill. Solving alignment using elementary linear algebra. In *Proc. of the seventh Annual Workshop on Languages and Compilers for Parallelism*, pages 4.1–4.15, August 1994.
6. S. S. Bhattacharyya, S. Sriram, and E. A. Lee. Latency-constrained resynchronisation for multiprocessor dsp implementation. In *Proceedings of ASAP'96*, 1996.
7. E. Bixby, K. Kennedy, and U. Kremer. Automatic data layout using 0-1 integer programming. In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, August 1994.
8. M. Bouvet. *Traitements des Signaux Pour les Systèmes Sonars.* Masson.
9. D. Callahan. *A Global Approach to Detection of Parallelism.* PhD thesis, Rice University, March 1987.
10. P. Clauss, C. Mongenet, and G.-R. Perrin. Synthesis of size-optimal toroïdal arrays for the algebraic path problem: A new contribution. *Parallel Computing, North-Holand*, 18:185–194, 1992.

11. P. Codognet, F. Fages, J.Jourdan, R. Lissajoux, and T. Sola. On the design of `meta(f)` and its application to air traffic control. In *Proc. ICLP'92*, Washington DC, USA, 1992.
12. Béatrice Creusillet. *Array Region Analyses and Applications.* PhD thesis, École des Mines de Paris, December 1996.
13. Béatrice Creusillet and François Irigoin. Interprocedural array region analyses. *International Journal of Parallel Programming (special issue on LCPC)*, 24(6):513–546, 1996.
14. A. Darte and Y. Robert. Constructive methods for scheduling uniform loop nests. *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, 5(8):814, August 1994.
15. Alain Darte, Leonid Khachiyan, and Yves Ropbert. Linear scheduling is nearly optimal. In *Parallel Processing Letters*, pages 73–81, 1991.
16. Alain Darte and Yves Robert. Affine-by-statement scheduling of uniform loop nests over parametric domains. Technical Report 92-16, LIP-IMAG, April 1992.
17. Alain Darte and Yves Robert. Mapping uniform loop nests onto distributed memory architectures. *Parallel Computing*, 20:679–710, 1994.
18. C. G. Diderich and M. Gengler. Solving the constant-degree parallelism alignment problem. In *Europar'96.* Laboratoire d'Informatique du Parallélisme, August 96.
19. M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T.Graf, and F. Berthier. The constraint logic programming language chip. In *International Conference on Fifth Generation Computer System*, Tokyo, Japan, December 1988.
20. M. Dincbas, H. Simonis, P. Van Hentenryck, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language chip. In *fifth Generation Computer Systems conference*, Tokyo, Japan, Dec. 1988.
21. P. Feautrier. Toward automatic distribution. *Parallel Processing Letters*, 4(3):233–244, 1994.
22. Paul Feautrier. Some efficient solution to the affine scheduling problem, II, multidimensional time. *Int. J. of Parallel Programming*, 21(6):389–420, December 1992.
23. Paul Feautrier. Some efficient solutions to the affine scheduling problem, I, one dimensional time. *Int. J. of Parallel Programming*, 21(5):313–348, October 1992.
24. Paul Feautrier. Fine-grain scheduling under resource constraints. In *7th Workshop on Language and Compiler for Parallel Computers*, August 1994.
25. David Foxwell and Mark Hewish. High-performance asw at an affordable price. *Jane' IDR Review*, pages 39–43, July 1996.
26. R. Govindarajan, E. R. Altman, and G. R. Gao. A framework for ressource-constrained rate-optimal software pipelining. *IEEE Transactions On Parallel And Distributed Systems*, 7(11):1133–1149, Nov 1996.
27. Ching-Chih Han, Kwei-Jay Lin, and Chao-Ju Hou. Distance constrained scheduling and its applications to real-time systems. *IEEE Transactions On Computers*, 45(7):814–825, Jul 1996.
28. S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the fortran d programming system. In *Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.
29. F. Irigoin. *Partitionnement de boucles imbriquées, une technique d'optimisation pour les programmes scientifiques.* PhD thesis, Université Pierre et Marie Curie, juin 1987.
30. J. Jourdan and R. Lissajoux. Plc et séquencement des vols à l'arrivée. In *Proc. Transportation and Constraint Programming*, Montpellier, France, 1995.

31. J. Jourdan and T. Sola. The versatility of handling disjunctions as constraints. Technical Report LACS-92-8, Thomson-CSF Central Research Lab, December 1992.
32. Jean Jourdan. *Concurrence et coopération de modèles multiples dans les langages de contraintes CLP et CC : Vers une méthodologie de programmation par modélisation*. PhD thesis, Université Denis Diderot, Paris VII, 1995.
33. K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing*, Portland, Or., August 1993.
34. K. Knobe, J. D. Lukas, and G. L. Steele. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *J. of Parallel and Distributed Computing*, 8, 1990.
35. C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr. Zosel, and M. Zosel. *The Hight Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.
36. U. Kremer. NP–completeness of dynamic remapping. In *Workshop on Compilers for Parallel Computers, Delft*, pages 135–141, December 1993.
37. Ulrich Kremer. *Automatic Data Layout for Distributed Memory Machines*. PhD thesis, Rice University, Houston, Texas, October 1995. Available as CRPC-TR95-559-S.
38. K.G. Kumar, D. Kulkarni, and A. Basu. Deriving good transformations for mapping nested loops on hierarchical parallel machines. In *International Conference on Supercomputing*, pages 82–92, July 1992.
39. E. A. Lee and D. G. Messerschmitt. Synchronous dataflow. In *Proceedings of the IEEE*, September 1987.
40. J. Li and M. Chen. The data alignment phase in compiling programs for distributed memory machines. *Journal of Parallel and Distributed Computing*, 13:213–221, 1991.
41. A. W. Lim and M. S. Lam. Communication-free parallelization via affine transformations. In *Procs of the 7^{th} Languages and Compilers for Parallel Computing, LNCS (to appear)*, August 1994.
42. M.J. Maher. Logic semantics for a class of committed-choice programs. In Jean-Louis Lassez, editor, *ICLP'87: Proceedings 4th International Conference on Logic Programming*, pages 858–876, Melbourne, 1987. MIT.
43. Dion Michèle. *Alignement et distribution en parallélisation automatique*. Thèse informatique, ENS,LYON, 1996. 136 P.
44. P. Murthy, S. S. Bhattacharyya, and E. A. Lee. Minimising memory requirements for chain-structured synchronous dataflow programs. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing, April 1994.*, 1996.
45. T. A. Proebsting and S. A. Watterson. Filter fusion. In *Symposium on Principles of Programming Language*, 1996.
46. V. Saraswat. The concurrent logic programming language cp: Denotational and operational semantics. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany*, pages 49–62, January 1987.
47. Gilbert C. Sih and Edward A. Lee. Declustering: A new multiprocessor scheduling technique. *IEEE Trans. on Parallel and Distributed Systems*, 4(6):625–637, June 1993.
48. S. Singhai and K. McKinley. Loop fusion for data locality and parallelism. In *Proceedings of the Mid-Atlantic Student Workshop on Programming Languages and Systems*, New Paltz, April 1996.

49. J. Subhlok and Gary Vondran. Optimal latency-troughput tradeoffs for data parallel pipelines. In *Proc. SPAA'96*, Padua, Italy, June 1996.
50. Rémi Triolet. *Contribution à la Parallélisation Automatique de Programme Fortran Comportant des appels de Procédures.* PhD thesis, Université Paris VI, 1984.
51. P. Van Hentenryck and Y. Deville. The cardinality operator: A new logical connective for constraint logic programming. In Koichi Furukawa, editor, *ICLP'91 Proceedings 8th International Conference on Logic Programming*, pages 745–759. MIT Press, 1991.
52. P. Van Hentenryck, V. Saraswat, and Y. Deville. Constraint processing in CC(FD). Technical report, Brown University, 1992.
53. P. Van Hentenryck, H. Simonis, and M. Dincbas. Constraint satisfaction using constraint logic programming. *Artificial Intelligence Journal*, 58:113–159, 1992.