# On the Optimality of Allen and Kennedy's Algorithm for Parallelism Extraction in Nested Loops

Alain Darte and Frédéric Vivien*

LIP, URA CNRS 1398, ENS-Lyon, F - 69364 LYON Cedex 07, France
e-mail: [Alain.Darte,Frederic.Vivien]@lip.ens-lyon.fr

**Abstract.** We explore the link between dependence abstractions and maximal parallelism extraction in nested loops. Our goal is to find, for each dependence abstraction, the minimal transformations needed for maximal parallelism extraction. The result of this paper is that Allen and Kennedy's algorithm is optimal when dependences are approximated by dependence levels. This means that even the most sophisticated algorithm cannot detect more parallelism than found by Allen and Kennedy's algorithm, as long as dependence level is the only information available.

## 1 Introduction

Many automatic loop parallelization techniques have been introduced over the last 30 years, starting from the early work of Karp, Miller and Winograd [12] in 1967 who studied the structure of computations in repetitive codes called systems of uniform recurrence equations. This work defined the foundation of today's loop compilation techniques. It has been widely exploited and extended in the systolic array community, as well as in the compiler-parallelizer community: Lamport [14] proposed a parallel scheme - the hyperplane method - in 1974, then several loop transformations were introduced (loop distribution/fusion, loop skewing, loop reversal, loop interchange, ...) for vectorizing computations, maximizing parallelism, maximizing locality and/or minimizing synchronizations. These techniques have been used as basic tools for optimizing algorithms, the most two famous being certainly Allen and Kennedy's algorithm [1], designed at Rice in the Fortran D compiler, and Wolf and Lam's algorithm [18], designed at Stanford in the SUIF compiler.

At the same time, dependence analysis has been developed so as to provide sufficient information for checking the legality of these loop transformations, in the sense that they do not change the final result of the program. Different abstractions of dependences have been defined (among others dependence distance [16], dependence level [1], dependence direction vector [19], dependence polyhedron or cone [11], ...), and more and more accurate tests for dependence analysis have been designed (among others Banerjee's tests [2], I test [13], $\Delta$ test [9], $\lambda$ test [15], PIP test [7], PIPS test [10], Omega test [17], ...).

* Supported by the CNRS-INRIA project *ReMaP*.

In general, dependence abstractions and dependence tests have been introduced with some particular loop transformations in mind. For example, the dependence level was designed for Allen and Kennedy's algorithm, whereas the PIP test is the main tool for Feautrier's method for array expansion [7] and parallelism extraction by affine schedules [8]. However, very few authors have studied, in a general manner, the links between both theories, dependence analysis and loop restructurations, and have tried to answer the following two dual questions:

- What is the minimal dependence abstraction needed for checking the legality of a given transformation?
- What is the simplest algorithm that exploits all information provided by a given dependence abstraction at best?

With the answer to the first question, we can adapt the dependence analysis to the parallelization algorithm, and avoid implementing an expensive dependence test if it is not needed. This question has been deeply studied in Yang's thesis [21], and summarized in Yang, Ancourt and Irigoin's paper [20].

Conversely, with the answer to the second question, we can adapt the parallelization algorithm to the dependence analysis, and avoid using an expensive parallelization algorithm, if a simpler algorithm extracts the same degree of parallelism. This question has been addressed by Darte and Vivien in [6] for dependence abstractions based on a polyhedral approximation of distance vectors.

Completing this previous work, we propose a more precise study of the link between *dependence abstractions* and *parallelism extraction* in the particular case of *dependence levels*. Our main result is that, in this context, Allen and Kennedy's parallelization algorithm is optimal for parallelism extraction, which means that even the most sophisticated algorithm cannot detect more parallel loops than Allen and Kennedy's algorithm does, as long as dependence level is the only information available. In other words, loop distribution is sufficient for detecting maximal parallelism in dependence graphs with dependence levels. There is no need to use more complicated transformations such as loop interchange, loop skewing, or any other transformations that could be invented, because there is an intrinsic limitation in the dependence level abstraction that prevents detecting more parallelism.

The paper is organized as follows. In Section 2, we explain what we call maximal parallelism extraction for a given dependence abstraction. In Section 3, we recall the definition of dependence levels and we present Allen and Kennedy's algorithm in its simplest form, which is sufficient for what we want to prove. In Section 4, we build a set of loops that are equivalent to the loops to be parallelized, in the sense that they have the same dependence graph. These loops contain exactly the degree of parallelism found by Allen and Kennedy's algorithm which proves the optimality (however, as the proof is quite long, we refer to [5] for an extended version). Finally, Section 5 summarizes the paper.

## 2 Theoretical framework

### 2.1 Notations

The notations used in the following sections are:

- $f(N) = O(N)$ if $\exists\, k > 0$ such that $f(N) \leq kN$ for all sufficiently large $N$.
- $f(N) = \Omega(N)$ if $\exists\, k > 0$ such that $f(N) \geq kN$ for all sufficiently large $N$.
- $f(N) = \Theta(N)$ if $f(N) = O(N)$ and $f(N) = \Omega(N)$.
- If $X$ is a finite set, $|X|$ denotes the number of elements in $X$.
- $G = (V, E)$ denotes a directed graph with vertices $V$ and edges $E$.
- $e = (x, y)$ denotes an edge from vertex $x$ to vertex $y$.

### 2.2 Dependence graphs

We restrict to the case of perfectly nested loops for making the discussion simpler. The structure of perfectly nested loops can be captured by an ordered set of statements $S_1, \ldots, S_s$ (with $S_i$ textually before $S_j$ if $i < j$) and an iteration domain $\mathcal{D} \subset \mathbb{Z}^n$ that describes the values of the loops counters ($n$ is the number of nested loops). Given a statement $S$, to each $n$-dimensional vector $I \in \mathcal{D}$ corresponds a particular execution (called instance) of $S$, denoted by $S(I)$.

Dependences between instances of statements define the **expanded dependence graph (EDG)**. The vertices of the EDG are all possible instances $\{(S_i, I) \mid 1 \leq i \leq s \text{ and } I \in \mathcal{D}\}$. There is an edge from $(S_i, I)$ to $(S_j, J)$ (denoted by $S_i(I) \Longrightarrow S_j(J)$) if executing instance $S_j(J)$ before instance $S_i(I)$ may change the result of the program. For all $1 \leq i, j \leq s$, one defines the **distance set** $E_{i,j}$ as follows:

$$E_{i,j} = \{(J - I) \mid S_i(I) \Longrightarrow S_j(J)\} \qquad (E_{i,j} \subset \mathbb{Z}^n) \tag{1}$$

In general, the EDG (and the distance sets) cannot be computed at compile-time, either because some information is missing (such as the values of size parameters or the exact accesses to memory), or because generating the whole graph is too expensive. Instead, dependences are captured through a smaller, (in general) cyclic directed (multi) graph, with $s$ vertices, called the **reduced dependence graph (RDG)**. Each edge $e$ has a label $w(e)$. This label has a different meaning depending upon the dependence abstraction that is used: it represents [2] a set $D_e \subset \mathbb{Z}^n$ such that:

$$\forall i, j,\ 1 \leq i, j \leq s,\ E_{i,j} \subset \left(\cup_{e=(S_i,S_j)} D_e\right) \tag{2}$$

In other words, the RDG describes, in a condensed manner, a superset of the EDG, called the **apparent dependence graph (ADG)**. The ADG and the EDG have the same vertices, but the ADG has more edges, defined by:

$$(S_i, I) \Longrightarrow (S_j, J) \text{ in the ADG} \Leftrightarrow \exists\, e = (S_i, S_j) \text{ in the RDG} \mid (J - I) \in D_e \tag{3}$$

Equations 1 and 2 ensure that the EDG is a subset of the ADG.

[2] except for exact dependence analysis where it defines a subset of $\mathbb{Z}^n \times \mathbb{Z}^n$.

## 2.3 Maximal degree of parallelism

We now define what we call maximal parallelism extraction in reduced dependence graphs. We consider that the only information available for extracting parallelism in a set of loops $L$ is the RDG associated to $L$. Any parallelization algorithm that transforms $L$ into an equivalent code $L_t$ has to preserve all dependences summarized in the RDG, i.e. all dependences described in the ADG (and not in the EDG which is not known!). If $(S_i, I) \Longrightarrow (S_j, J)$ in the ADG then $(S_i, I)$ must be computed before $(S_j, J)$ in the transformed code $L_t$.

**Definition 1.** For a given statement $S$ in $L$, we define the **$S$-latency** of the transformed code $L_t$ as the minimal number of clock cycles needed to execute $L_t$ if: 1) an unbounded number of processors is available; 2) executing an instance of $S$ requires one clock cycle; 3) any other operation requires zero clock cycle. The latency of $L_t$ is defined as the sum of all $S$-latencies.

Since two instances linked by an edge in the ADG cannot be computed at the same clock cycle in the transformed code $L_t$, the latency of $L_t$, whatever the parallelization algorithm, is larger than the length of the longest path in the ADG. With this simple remark, we can define a theoretical framework in which the optimality of parallelization algorithms, with respect to a given dependence abstraction, can be discussed. In the following definitions, the RDGs are supposed to be labeled with a fixed dependence abstraction and the optimality is defined with respect to this abstraction.

**Definition 2.** Let $G$ be a RDG and $\mathcal{D}$ be the $n$-dimensional cube of size $N$. Let $d$ be the smallest non negative integer such that the length of the longest path in the ADG, is $O(N^d)$. Then, we say that the **degree of (intrinsic) parallelism** in $G$ is $(n-d)$ or that $G$ contains $(n-d)$ degrees of parallelism.

**Definition 3.** Let $L$ be a set of $n$ nested loops and $G$ its RDG. Apply a parallelization algorithm $A$ to $G$ and suppose that $\mathcal{D}$ is the $n$-dimensional cube of size $N$. The **degree of parallelism extraction** for $A$ is $(n-d)$ if $d$ is the smallest non negative integer such that the latency of the transformed code is $O(N^d)$.

**Definition 4.** An algorithm $A$ performs maximal parallelism extraction (or is said **optimal for parallelism extraction**) if for each RDG $G$, the degree of parallelism extraction is equal to the degree of intrinsic parallelism.

Then the optimality of a parallelization algorithm $A$ can be proved as follows. Let $L$ be a set of $n$ nested loops, $G$ its RDG, and $G_a$ the corresponding ADG. Let $(n-d)$ be the degree of parallelism extraction for $A$ in $G$. Then, we have at least two ways for proving the optimality of $A$:

1. Build in $G_a$ a dependence path whose length is not $O(N^{d-1})$.
2. Build a set of loops $L'$ whose RDG is also $G$ and whose EDG contains a dependence path whose length is not $O(N^{d-1})$.

Note that (2) implies (1) since the EDG of $L'$ is included in $G_a$ ($L$ and $L'$ have the same RDG). Therefore, proving (2) is more powerful. It reveals the intrinsic limitations due to the dependence abstraction itself: even if their EDGs may be different, $L$ and $L'$ cannot be distinguished by the parallelization algorithm, since they have the same RDG. Therefore, since $L'$ is parallelized optimally, the algorithm is considered optimal with respect to the dependence abstraction that is used. Figure 1 recalls the links between $L$, $L'$ and their EDG, RDG and ADG. The loops $L'$ are called **apparent nested loops**.

$$\begin{tikzcd}
\text{Initial loops L} \arrow[d] & & & & \text{Apparent loops L'} \arrow[d] & \\
\text{EDG} \arrow[drr] & \subset & \text{ADG} & \supset & \text{EDG} \arrow[dll] & \\
& & \text{RDG} \arrow[d] & & & \text{contains a dependence path whose length is not } O(N^{d-1}) \arrow[ul] \\
& & \text{Transformed loops with (n-d) degrees of parallelism extracted} & & &
\end{tikzcd}$$

**Fig. 1.** Links between $L$, $L'$ and their EDG, ADG and RDG

One can define a more precise notion of optimality by using the notion of $S$-latency instead of latency. The $S$-latency is related to the $S$-length of the longest path in the ADG, where the $S$-length of a path $P$ is the number of vertices in $P$ that are instances of $S$. Similarly, we define the $S$-degree of parallelism extraction in $L_t$, and the $S$-degree of intrinsic parallelism in a RDG. Finally, we have the following notion of optimality:

**Definition 5.** An algorithm $A$ performs maximal parallelism extraction (or is said **optimal for parallelism extraction**) if for each RDG $G$ and for each statement $S$ of $G$, the $S$-degree of parallelism extraction for $A$ in $G$ is equal to the $S$-degree of intrinsic parallelism in $G$.

With this definition, we can discuss the quality of parallelizing algorithms even for statements that do not belong to the most sequential part of the code. Note that this definition of optimality is more precise than Definition 4 since the degree of intrinsic parallelism (resp. of parallelism extraction) in $G$ is the minimal $S$-degree of intrinsic parallelism (resp. of parallelism extraction) in $G$.

One could argue that the latency and the $S$-latency of a transformed code are not easy to compute. Indeed, in the general case, the latency can be computed only by executing the transformed code with a fixed value of $N$. However, for most known parallelizing algorithms, the degree of parallelism extraction (but not necessarily the latency) can be computed simply by examining the structure of the transformed code, as shown by Lemma 6.

*Lemma 6. In addition to the hypotheses of Definition 3, assume that each statement $S$ of the initial code $L$ appears only once in the transformed code $L_t$ and is surrounded by exactly $n$ loops. Furthermore, assume that the iteration domain $\mathcal{D}_t$ described by these $n$ loops contains a $n$-cube of size $\Omega(N)$ and is contained in a $n$-cube of size $O(N)$. Then, the number of parallel loops that surrounds $S$ is the $S$-degree of parallelism extraction and the minimal $S$-degree of parallelism extraction is the degree of parallelism extraction.*

We now discuss, within this theoretical framework, the optimality of Allen and Kennedy's algorithm with respect to the dependence level abstraction. A similar study for other dependence abstractions can be found in [4].

## 3 Allen and Kennedy's algorithm

Allen and Kennedy's algorithm has first been designed for vectorizing loops. Then, it has been extended so as to maximize the number of parallel loops and to minimize the number of synchronizations in the transformed code. It has been shown (see details in [3, 22]) that for each statement of the initial code, as many surrounding loops as possible are detected as parallel loops. Therefore, one could think that what we want to prove in this paper has been already proved!

However, looking precisely into the details of Allen and Kennedy's proof reveals that what has actually been proved is the following: consider a statement $S$ of the initial code and $L_i$ one of the surrounding loops. Then $L_i$ is marked as parallel if and only if there is no dependence at level $i$ between two instances of $S$. This result proves that the algorithm is optimal among all parallelization algorithms that describe, in the transformed code, the instances of $S$ with exactly the same loops as in the initial code. This does not prove a general optimality property. In particular, this does not prove that it is not possible to detect more parallelism with more sophisticated techniques than loop distribution and loop fusion. This paper gives an answer to this question.

We first recall the definition of dependence level, the dependence approximation used by Allen and Kennedy's algorithm. Then, we recall Allen and Kennedy's algorithm, in its simplest form, which is sufficient for discussing the optimality for parallelism extraction.

The **dependence level** associated to a dependence distance $J - I$ where $S_i(I) \Longrightarrow S_j(J)$ is either $\infty$ if $J - I = 0$ or the smallest integer $l$, $1 \leq l \leq n$, such that the $l$-th component of $J - I$ is non zero (and thus positive). A **reduced leveled dependence graph (RLDG)** is a reduced dependence graph whose edges are labeled by dependence levels. The **level** $l(G)$ of a RLDG $G$ is the minimal level of an edge of $G$: $l(G) = \min\{l(e) \mid e \in G\}$.

We need to recall some graphs definitions: a **strongly connected component** of a directed graph $G$ is a maximal subgraph of $G$ in which for any vertices $p$ and $q$ ($p \neq q$) there is a path from $p$ to $q$. The **acyclic condensation** of a graph $G$ is the acyclic graph whose nodes are the strongly connected components $\mathcal{V}_1, \ldots, \mathcal{V}_c$ of $G$ and there is an edge from $\mathcal{V}_i$ to $\mathcal{V}_j$ if there is an edge $e = (x_i, y_j)$ in $G$ such that $x_i \in \mathcal{V}_i$ and $y_j \in \mathcal{V}_j$.

To apply Allen and Kennedy's algorithm to a RLDG $G$, call AK($G$, 1) where:

```
AK(H,l)
  – $H' = H \setminus \{e \mid l(e) < l\}$
  – Build $H''$ the acyclic condensation of $H'$, and number its vertices $\mathcal{V}_1, \ldots, \mathcal{V}_c$ in a
    topological sort order.
  – For $i = 1$ to $c$ do
      1. If $\mathcal{V}_i$ is reduced to a single statement $S$, with no edge, then generate parallel
         DO loops (DOALL) in all remaining dimensions (i.e. for levels $l$ to $n$) and
         generate code for $S$.
      2. Otherwise, let $k = l(\mathcal{V}_i)$. Generate parallel DO loops (DOALL) for levels from
         $l$ to $k-1$, and a sequential DO loop (DOSEQ) for level $k$. Call AK($\mathcal{V}_i, k+1$).
```

## 4 Loop nest generation algorithm

In this section, we present a systematic procedure, called LGA (for Loops Generation Algorithm), that builds, from a RLDG $G$, a perfect loop nest $L'$ whose RLDG is exactly $G$. $L'$ are the desired apparent loops (see Section 2) that we use to prove the optimality of Algorithm AK (see Theorem 7). The construction of $L'$ is based on the notion of critical edges. These edges are built in Section 4.1. The exact formulation of Procedure LGA is given in Section 4.2. Finally, in Section 4.3, we show that the RLDG associated to $L'$ is $G$, as desired.

### 4.1 Critical edges

The procedure Critical given below defines a set of edges $E_c$, called **critical** edges, that we need for defining the apparent loops $L'$. We call Critical($G_i$) for each strongly connected component $G_i$ of the RLDG $G$ which contains at least one edge.

```
Critical(H)
  1. $l \leftarrow l(H)$.
  2. Select an edge $f$ of $H$ with level $l$. Call $f$ the critical edge of $H$. $E_c \leftarrow E_c \cup \{f\}$.
  3. $H' = H \setminus \{e \mid l(e) \leq l\}$.
  4. Let $H_1, \ldots, H_c$ be the strongly connected components of $H'$.
  5. Call Critical($H_i$) for each $H_i$ that has at least one edge.
```

### 4.2 Generation of apparent nested loops

Let $G = (E, V)$ be a RLDG. We assume that $G$ has been built in a consistent way from some nested loops. Therefore, vertices can be numbered according to the topological order defined by the edges whose level is $\infty$: $v_i \xrightarrow{\infty} v_j \Rightarrow i < j$. We denote by $d$ the **dimension** of $G$: $d = \max\{l(e) \mid e \in E \text{ and } l(e) < \infty\}$. The apparent loops $L'$ corresponding to $G$ will consist of $d$ perfectly nested loops, with $|V|$ statements. Each statement will be of the form $a_i[I] = \text{rhs}(i)$ where $a_i$ is a $d$-dimensional array and rhs($i$) is the right-hand side that defines array $a_i$. In the following, $E_c$ is the set of critical edges of $G$ (defined in Section 4.1) and "@" denotes the operator of expression concatenation.

```
LGA(G)

  Initialization:
    For $i = 1$ to $|V|$ do rhs($i$) $\leftarrow$ "1"

  Computation of the statements of $L'$:
    For each $e = (v_i, v_j) \in E$ do
      if $l(e) = \infty$ then rhs($j$) $\leftarrow$ rhs($j$) @ "$+a_i[I_1, \dots, I_d]$"
      if $l(e) < \infty$ and $e \notin E_c$
                then rhs($j$) $\leftarrow$ rhs($j$) @ "$+a_i[I_1, \dots, I_{l(e)-1}, I_{l(e)} - 1, I_{l(e)+1}, \dots, I_d]$"
      if $e \in E_c$   then rhs($j$) $\leftarrow$ rhs($j$) @ "$+a_i[I_1, \dots, I_{l(e)-1}, I_{l(e)} - 1, \underbrace{N, \dots, N}_{d-l(e)}]$"

  Code generation for $L'$:
    For $i = 1$ to $d$ do generate ("For $I_i = 1$ to $N$ do")
    For $i = 1$ to $|V|$ do generate ("$a_i[I_1, \dots, I_d]$ :=" @ rhs($i$))
```

## 4.3 Optimality result

The following results are detailed in [5]. Let $L$ be a set of $n$ perfectly nested loops whose RLDG is $G$. Use Algorithm LGA to generate the apparent loops $L'$. The RLDG of $L'$ is $G$ by construction. Let $d_S$ be the number of calls to Procedure Critical that concern Statement $S$: $n - d_S$ is also equal to the degree of parallelism extraction for Allen and Kennedy's algorithm. Furthermore:

***Theorem 7.*** *For each strongly connected component $G_i$ of $G$, there is a path in the EDG of $L'$ which visits, $\Omega(N^{d_s})$ times, each statement $S$ in $G_i$.*

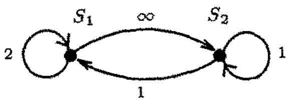
The proof is long, technical and painful. The fundamental corollary is:

***Corollary 8.*** *Allen and Kennedy's algorithm is optimal for parallelism extraction in reduced leveled dependence graphs (optimal in the sense of Definition 5).*

We illustrate the optimality theorem through the following example:

```
For $i = 1$ to $N$ do
  For $j = 1$ to $N$ do
    $S_1 : a(i,j) := 1 + a(i,j-1) + b(i-1,j)$
    $S_2 : b(i,j) := 1 + a(i,j) + b(i-1,j)$
```

*(a) Original loop nest L*



*(b) RLDG*

The RLDG is strongly connected and contains one edge at level 1. Thus, the first loop is marked sequential. At level 2, each statement is in a separate strongly connected component and, after loop distribution, the second loop is marked sequential for $S_1$ and parallel for $S_2$. Indeed, one can check that $L$ and the apparent loops $L'$ have the same RLDG and that $L'$ contains an exact dependence path including $\Theta(N^2)$ instances of $S_1$ and $\Theta(N)$ instances of $S_2$. This proves the optimality of Algorithm AK in our example even if the original program contains only uniform dependences, and, therefore, can be scheduled with a single sequential loop using the hyperplane method.

```
For $i = 1$ to $N$ do
    For $j = 1$ to $N$ do
        $S_1 : \mathrm{a}(i,j) := 1 + \mathrm{a}(i,j-1) + \mathrm{b}(i-1,N)$
        $S_2 : \mathrm{b}(i,j) := 1 + \mathrm{a}(i,j) + \mathrm{b}(i-1,j)$
```

*Apparent loops $L'$*

## 5 Conclusion

We have introduced a theoretical framework in which the optimality of algorithms that detect parallelism in nested loops can be discussed. We have formalized the notions of degree of parallelism extraction (with respect to a given dependence abstraction) and of degree of intrinsic parallelism (contained in a reduced dependence graph). This study explains the impact of a given dependence abstraction on the maximal parallelism that can be detected: it determines whether the limitations of a parallelization algorithm are due to the algorithm itself or are due to the weaknesses of the dependence abstraction.

In this framework, we have studied more precisely the link between dependence abstractions and parallelism extraction in the particular case of *dependence level*. Our main result is the optimality of Allen and Kennedy's algorithm for parallelism extraction in reduced leveled dependence graphs. This means that even the most sophisticated algorithm cannot detect more parallelism, as long as dependence level is the only information available. In other words, loop distribution is sufficient for detecting maximal parallelism in dependence graphs with levels.

The proof is based on the following fact: given a set of loops $L$ whose dependences are specified by levels, we are able to systematically build a set of loops $L'$ that cannot be distinguished from $L$ (i.e. they have the same reduced dependence graph) and that contains exactly the degree of parallelism found by Allen and Kennedy's algorithm. We call these loops the apparent loops. We believe this construction is of interest because it better explains why some loops appear sequential when considering the reduced dependence graph while they actually may contain some parallelism.

## References

1. J.R. Allen and K. Kennedy. Automatic translations of Fortran programs to vector form. *ACM Toplas*, 9:491–542, 1987.
2. U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Norwell, MA, 1988.
3. D. Callahan. *A Global Approach to Detection of Parallelism.* PhD thesis, Dept. of Computer Science, Rice University, Houston, TX, 1987.
4. Alain Darte and Frédéric Vivien. A classification of nested loops parallelization algorithms. In *INRIA-IEEE Symposium on Emerging Technologies and Factory Automation*, pages 217–224. IEEE Computer Society Press, 1995.

5. Alain Darte and Frédéric Vivien. On the optimality of Allen and Kennedy's algorithm for parallelism extraction in nested loops. Technical Report 96-05, LIP, ENS-Lyon, France, February 1996. Extended version of Europar'96.
6. Alain Darte and Frédéric Vivien. Optimal fine and medium grain parallelism in polyhedral reduced dependence graphs. In *Proceedings of PACT'96*, Boston, MA, October 1996. IEEE Computer Society Press. To appear.
7. Paul Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–51, 1991.
8. Paul Feautrier. Some efficient solutions to the affine scheduling problem, part II, multi-dimensional time. *Int. J. Parallel Programming*, 21(6):389–420, December 1992.
9. G. Goff, K. Kennedy, and C.W. Tseng. Practical dependence testing. In *Proceedings of ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.
10. F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: an overview of the PIPS project. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.
11. F. Irigoin and R. Triolet. Computing dependence direction vectors and dependence cones with linear systems. Technical Report ENSMP-CAI-87-E94, Ecole des Mines de Paris, Fontainebleau (France), 1987.
12. R.M. Karp, R.E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, July 1967.
13. X.Y. Kong, D. Klappholz, and K. Psarris. The I test: a new test for subscript data dependence. In Padua, editor, *Proceedings of 1990 International Conference of Parallel Processing*, August 1990.
14. Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.
15. Z.Y. Li, P.-C. Yew, and C.Q. Zhu. Data dependence analysis on multi-dimensional array references. In *Proceedings of the 1989 ACM International Conference on Supercomputing*, pages 215–224, Crete, Greece, June 1989.
16. Y. Muraoka. *Parallelism exposure and exploitation in programs*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, February 1971.
17. William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, August 1992.
18. Michael E. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distributed Systems*, 2(4):452–471, October 1991.
19. Michael Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge MA, 1989.
20. Y.-Q. Yang, C. Ancourt, and F. Irigoin. Minimal data dependence abstractions for loop transformations. *International Journal of Parallel Programming*, 23(4):359–388, August 1995.
21. Yi-Qing Yang. *Tests des dépendances et transformations de programme*. PhD thesis, Ecole Nationale Supérieure des Mines de Paris, Fontainebleau, France, 1993.
22. Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1990.