Invited Talks

Basis of Parallel Speculative Execution

Paul Feautrier

Laboratoire PRiSM
Université de Versailles
78035 VERSAILLES CEDEX
France

Abstract. The next generation of supercomputers will probably need large amounts of parallelism, both for generating the needed computing power and for masking memory latency. Furthermore, it is necessary to expand the use of parallelism to less regular programs than is usually found in numerical applications. The main obstacle to be overcome is the presence of control dependences, i.e. of situations in which the result of an operation is used to decide if another operation is going to be executed or not. This forbids parallelization, unless speculative execution is used: an operation is initiated before being sure that it must be executed.
Our aim here is to explore the logical constraints which must be satisfied by parallel speculative programs. This is best done in the framework of scheduling. For instance, one obtains speculative execution simply by ignoring some control dependences when constructing the schedule. If this is done, one has to insert compensating dependences to restore the correctness of the object program. One also has to keep enough information for being able to undo the effects of speculative operations. Lastly, one has to insure that everything stays finite in the object program, including the size of temporary memory and the amount of computation per logical time step. Preliminary solutions are given for some of these problems.

1 Speculative Execution, How and Why

The next generation of supercomputers will probably need large amounts of parallelism, both for generating the needed computing power and for masking memory latency by multithreading. To hide a memory access latency of L processor cycles, one need about L threads. At present, L is of the order of 50, which is manageable. If L grows to a few thousands, where are we to find that much parallelism?

In the case of regular programs (scientific computations and signal processing) this is not too difficult. One uses data parallelism, which will be enough provided the data sets are large. But regular programs are a small fraction of the workload of a processor. How do we find large amount of parallelism in irregular programs?

The main obstacle to be overcome is the presence of control dependences, i.e. of situations in which the result of an operation is used to decide if another operation is going to be executed or not. This forbids parallelization, unless speculative execution is used: an operation is initiated before being sure that it must be executed. To get a rough idea of the quantities involved, let T_1 be the sequential execution time. Let σ be the proportion of operations which are executed speculatively, and suppose that about one half of those are useful. Lastly, suppose that the removal of control dependences is enough to insure total parallelization. The elapsed time on P processors is then:

$$T_P = \frac{T_1(1+\sigma/2)}{P}$$

and the efficiency is:

$$\epsilon = \frac{P_1}{PT_P} = \frac{1}{1+\sigma/2}.$$

This is the most favorable case. It tell us that the amount of speculation must stay small if we want reasonable efficiency.

Our aim here is to explore the logical constraints which must be satisfied by parallel speculative programs. This is best done in the framework of scheduling. For instance, one obtains speculative execution simply by ignoring some control dependences when constructing the schedule. If this is done, one has to insert compensating dependences to restore the correctness of the object program. One also has to keep enough information for being able to undo the effects of speculative operations. Lastly, one has to insure that everything stays finite in the object program, including the size of temporary memory and the amount of computation per logical time step. Preliminary solutions are given for some of these problems.

2 Notations and Definitions

We will use here the notations and conventions of [4] and [2]. The program model we will restrict ourselves to is as follows:

- The only data structures are scalars, and arrays of scalars.
- Expressions do not include any pointer or pointer-like mechanism such as aliasing, `EQUIVALENCE`, etc.
- Basic statements are assignments to scalars or array elements.
- The only control structures are the sequence, the `do` loop, the `while` or `repeat` loop, and the conditional construct `if..then..else`, without restriction on stopping conditions of `while` loops, nor on predicates in `if`'s. `goto`s and procedure calls are prohibited.
- `while` loops have explicit counters.
- Array subscripts must be affine functions of the counters of surrounding `do`, `while` or `repeat` loops and of structure parameters. The input program is supposed to be correct, thus subscripts stay within array bounds, and all `while` loops terminate after a finite number of iterations.

The emphasis will be on operations, i.e. executions of data processing statements. An operation is named by giving the name of the executed statement and its iteration vector, i.e. the values of the surrounding loop counters. Each operation has a set of governing predicates, which is obtained in the following way:

- If the operation is in the true branch of a conditional `if p ...`, then p is among its governing predicates.
- If the operation is in the false branch of a conditional, then $\neg p$ is among its governing predicates.
- If the operation is in the body of a `do i = 1 while p` loop, then p is among its governing predicates.

The fact that `while` loops terminate cannot be checked at compile time. As a consequence, the fact that array subscripts stay within array bounds cannot be checked at compile time when subscripts are expressions involving `while`-loop counters. Our position is that such checks are the responsibility of the original programmer.

2.1 Iteration domains

The *iteration domain* $\mathcal{D}(S)$ of statement `S` is the set of values that the iteration vector takes in the course of execution of `S`. Unfortunately, iteration domains for dynamic control programs cannot be predicted at compile time. In the particular case where there is only one outermost `while` loop, we know at compile time that the iteration domain is built from the integral points inside a convex polyhedron; this polyhedron is bounded if the loop terminates, but this bound cannot be known statically [3]. In more general cases, the iteration domain has no particular shape and looks like a (possibly multidimensional) "comb" [7].

In dynamic control programs, we have to consider the calculation of `if` and `while` predicates as full fledged operations. The corresponding statement is given a name, and the name of the operation is obtained, as usual, by concatenating this name and its iteration vector.

2.2 Approximate iteration domains

Definition 1. The approximate iteration domain $\widehat{\mathcal{D}}(S)$ of a statement S is the set of all instances of S which would be obtained if all the governing predicates of S always evaluated to true.

The approximate domain of S is a superset of the (actual) iteration domain.

Observe that, in a static control program, the approximate domain of any statement S is equal to the actual iteration domain, i.e. $\widehat{\mathcal{D}}(S) = \mathcal{D}(S)$ for any S, and there is no need for handling control dependences since they are already taken into account in the expression of $\mathcal{D}(S)$.

3 Dependence Relations

3.1 Memory- and value-based dependences

Two operations u and v can execute in parallel if they are independent, i.e. if they do not modify any of their shared variables. If this condition is not verified, and if u is executed before v (written $u \prec v$), then u and v are said to be data dependent, denoted by $u \, \delta \, v$. Dependences are usually classified into flow, output and anti-dependences. Two operations u and v can execute in parallel if u is not comparable to v in the transitive closure of δ . We say that a dependence from u to v is *satisfied* if u executes before v. All dependences should be satisfied, thus limiting parallelism. Note that if u produces a value for a variable a and if v is the computation of an `if` or `while` predicate in which a occurs, then $u \, \delta \, v$. For simplicity, we will suppose here that predicates have no side effect.

These dependences, however, are *memory-based dependences.* They are language- and program-dependent, and are not semantically related to the algorithm. On the contrary, *value-based dependences* or *data flow dependences* capture the production and uses of computed values. Dataflow analysis for static control programs in the presence of arrays is now well understood [4, 9, 8, 10]. In the case of dynamic control programs, a fuzzy array data flow analysis (FADA) has been proposed in [2]. The result of fuzzy array dataflow analysis is a multi-level conditional called a *quast*. Each leaf is a *set* of potential dataflow sources. Notice that these sets may possibly be infinite. Each quast leaf is governed by a *context* given by the conjunction of predicates appearing on the unique path from the quast's root to the leaf.

3.2 Control dependences

Definition There is a control dependence from operation u to operation v if the execution of v depends on the outcome of u. u is called the *governing operation.* Such a dependence is denoted by $u \, \delta^c \, v$. In particular, the evaluation of a `while` loop predicate is control dependent on the outcome of the evaluation at the previous iterations, if any.

Description of control dependences

The case of the `if` *construct* Let us consider the following program piece:

```
G : if .... then
      ...
S : ...
    end if
```

where S is some statement in the `then` or `else` arm, perhaps surrounded by loops. Let c be the depth of the `if` construct, i.e. the number of loops surrounding

G. Let $\mathbf{a}$ (resp. $\mathbf{b}$) be the iteration vector of G (resp. S). Then, there is a control dependence from $\langle G, \mathbf{a}\rangle$ to $\langle S, \mathbf{b}\rangle$ iff

$$\mathbf{b}[1..c] = \mathbf{a}. \tag{1}$$

(if $c = 0$, then $\mathbf{a}$ and $\mathbf{b}[1..c]$ are equal to the vector of dimension 0 and the condition (1) is trivially true.)

The case of `while` *loops* Let us consider the following program piece:

```
G : do x = 1 while ...
        ...
S :     ...
        ...
    end do
```

where S is some statement in the `while` loop body, perhaps surrounded by loops within the body. Let c be the depth of the `while` construct, i.e. the number of loops surrounding G. Let $\mathbf{a}$ (resp. $\mathbf{b}$) be the iteration vector of G (resp. S). Then, there is a control dependence from $\langle G, \mathbf{a}\rangle$ to $\langle S, \mathbf{b}\rangle$ iff

$$\mathbf{a}[1..c] = \mathbf{b}[1..c] \ \wedge \ \mathbf{a}[c+1] \leq \mathbf{b}[c+1] \tag{2}$$

4 Basic Methods for Speculative Program Construction

4.1 Scheduling

Let us suppose that our target parallel compiler is a global memory machine. When constructing parallel programs for such a machine, one may use a *schedule*, i.e. a function θ from the set of all operations, E to the set of positive integers, $\mathbb{N}$. The execution order of the parallel program is:

$$u \prec_{//} v \equiv \theta(u) < \theta(v).$$

Operation which are scheduled at the same date are non comparable for this order and are executed in parallel. As is well known, a sufficient condition for a parallel program to be equivalent to a sequential one is that 1) they have the same set of operations, and 2) dependent operation are executed in the same order in the sequential and parallel program. We deduce that the schedule must satisfy the following constraint:

$$u \,\delta\, v \wedge u \prec v \Rightarrow \theta(u) < \theta(v),$$

which, since θ is integer valued, is equivalent to:

$$u \,\delta\, v \wedge u \prec v \Rightarrow \theta(u) + 1 \leq \theta(v). \tag{3}$$

This is a system of functional inequalities, which must be solved for the unknown function θ. As is often true for system of inequalities, it may have many different solutions. For the sake of expediency, one selects beforehand the shape of θ, thus restricting the range of possible solutions. In this case, recall that the operations u and v are instance of statements $\mathtt{R}$ and $\mathtt{S}$, and that their name is of the form $\langle \mathtt{R}, \mathbf{a}\rangle$, (resp. $\langle \mathtt{S}, \mathbf{b}\rangle$), where $\mathbf{a}$ and $\mathbf{b}$ are iteration vectors. In the constraint (3) above, $u\,\delta\,v$ and $u \prec v$ both translate into affine constraints on $\mathbf{a}$ and $\mathbf{b}$. The translation is exact for $\prec$, but may be conservative for δ in the presence of dynamic control. It seems natural to select affine schedules: the consequent in (3) becomes an affine constraint too.

Schematically, our problem takes the following form: let x be the variables of the problem ($\mathbf{a}, \mathbf{b}$ and the structure parameters). Let $\phi_i(x) \geq 0, i = 1, M$ be the affine constraints in the antecedent of (3), and let $\psi(x) \geq 0$ be its consequent. By Farkas lemma, we know that (3) is satisfied iff there exists positive numbers $\lambda_i, i = 0, M$ such that:

$$\psi(x) \equiv \lambda_0 + \sum_{i=1}^{M} \lambda_i \phi_i(x).$$

This relation is valid for all values of x. Hence, one can equate the constant term and the coefficient of each variable in each side of the identity, to get a set of linear equations where the unknowns are the coefficients of the schedules and the Farkas multipliers, λ_i. Since the latter are constrained to be positive, the system must be solved by linear programming. Most of the time the system has many solutions. One can minimize various objective functions, as e.g., the number of synchronization points.

Unfortunately, some loop nests do not have affine schedules. The reason is that when a loop nest has an affine schedule, it has *ipso facto* a large degree of parallelism. However, it is clear that some loop nests have few or even no parallelism, hence no affine schedule. The solution in this case is to use a *multidimensional schedule*, whose domain is $\mathbb{N}^d, d > 1$ ordered according to lexicographic order. Such a schedule can have as low a degree of parallelism as necessary, and can even represent sequential programs.

The selection of a multidimensional schedule can be automated by using algorithms from [6, 5]. It can be proved that any loop nest in an imperative program has a multidimensional schedule. Multidimensional schedules can be particularly useful in the case of dynamic control programs, since we have in that case to overestimate the dependences and hence to underestimate the degree of parallelism.

4.2 How to write speculative programs

In this context, our proposal for speculative execution is very simple: it consists in ignoring some of the control dependences in the source program. Let us first explore informally some of the consequences of this idea.

Firstly, why does this proposal have a chance of increasing the degree of parallelism of the object program? This is simply because each dependence is an obstacle for parallel execution, and that removing one of them may give a better schedule. Why is the resulting program speculative? Suppose that operation u is in control dependence with v, and that the dependence is kept when computing a schedule. We thus have $\theta(u) < \theta(v)$. If this dependence is discarded for speculative execution, then the above constraint may not hold. In this case, v may have to be started before its governing predicate, hence speculatively.

A speculative operation must not modify the program memory. Its result must be held in temporary storage until its governing predicates are evaluated. If the outcome is **true** , then the result is committed by moving it to permanent storage. In the opposite case, the result is discarded from temporary storage.

Our objective now is to give rules for the correctness of a speculative program. One of the questions to be faced first is the problem of *spurious exceptions*. A speculative operation may not be executed in the source program. Hence, there is no guarantee that it will not raise an error. Such an error may come from the original program, or from the boldness of our speculation. Software solutions to this problem are too slow. A hardware solution is to take advantage of the exceptional values which are provided by some number systems, notably the Not-a-Number value (NaN) of the IEEE 754 floating point norm. If the floating point unit detects an error, it should not raise an exception, but rather generate a NaN. The exception should be raised, either if the NaN result is committed, or when it is next used as an argument. Note the dire effect this deferred exception scheme has on debugging. This simply reinforce the familiar motto: "Never debug an algorithm on a parallel machine".

4.3 Partial and total correctness of a speculative program

A program is partially correct if, whenever the program terminates, its results are correct. A program is totally correct if, besides being partially correct, it always terminates provided its initial data is within its input range.

In the present case, we fortunately do not have to specify what are correct results, or what is the input range of our program. Our basic hypothesis is that the original sequential program is totally correct. What we have to guarantee is that, whenever the original program terminates, the speculative program also terminates giving identical results.

Partial correctness We have *partial correctness* iff, whenever a program terminates, its results are correct. We are going to prove a stronger result: in the speculative program, each non speculative operation has the same result as in the original program.

In the non speculative case, this means only that data dependences are to be satisfied, and this is one of the validity condition of the schedule. A problem arises when one of the arguments is the result of a speculative operation. This result cannot be used until it has been committed, i.e. until its governing predicate has terminated. This can be enforced by the introduction of *compensating dependences*, which are the composition[1] of a control dependence and a

[1] in the sense of the composition of relations.

flow dependence. Compensating dependences are to be taken into account for scheduling.

Theorem 2. *A speculative program in which compensating dependences are satisfied is partially correct.*

Proof. Suppose *a contrario* that there is an operation u which is executed in the original program and whose result is not the same in the original program and in the speculative version. We may suppose that this operation is the earliest one in the execution order of the parallel program. Since operations are deterministic, this implies that one of the arguments of the operation is not correct. This argument may be the result of a non speculative operation, which is necessarily in flow dependence with the distinguished operation. By the definition of a dependence, the source operation is executed earlier in the parallel program than u and gives an incorrect result, a contradiction.

If the incorrect argument may be the result of several operations, it may be that we have chosen the wrong one. This implies that one of the governing operation has given an incorrect result. Since, thanks to the compensating dependences, the governing operations are executed before u in the parallel program, we have found another contradiction.

4.4 Total correctness

Intuitively, not taking a control dependence into account may yield a nonterminating behavior. In the case of dynamic control programs where the only `while` loop is the outermost loop, a necessary and sufficient condition for correctness is that fronts must be finite [3]. Our aim here is to give a termination criterion for more general speculative programs. Together with the preceding theorem, it will entail the total correctness of the object program.

Firstly, we must have some idea of the structure of the target code. As is the case for synchronous parallel programs, we will suppose that the outer loops are sequential, their function being the enumeration of the successive values of time. Contrary to the static control case, these loops are `while` loops. Their number is equal to the dimension of the schedule. The operations which are scheduled at time t constitute the *front* at time t. The definition of a front is more complicated than in the static control case. An operation which is scheduled at time t is to be executed provided that its governing operations which have been executed before t have evaluated to true.

The result of operation u will be written $\rho(u)$. With this notation, the speculative front at time t is:

$$\mathcal{F}_S(t) = \{u \in \widehat{\mathcal{D}}(S) \mid \theta(u) = t, \forall v : (\theta(v) \ll t \wedge v\, \delta^c\, u \Rightarrow \rho(v) = \textbf{true})\}, \quad (4)$$

$$\mathcal{F}(t) = \bigcup_S \mathcal{F}_S(t). \quad (5)$$

To write the speculative program, we need the following auxiliary functions:

$$\text{first} = \min\{t \mid \mathcal{F}(t) \neq \emptyset\} \tag{6}$$

$$\text{next}(\tau) = \min\{t \mid \mathcal{F}(t) \neq \emptyset \wedge t \gg \tau\} \tag{7}$$

with the proviso that these functions take the undefined value $\perp$ if the set over which the minimum is computed is empty. "first" gives the first clock tick at which there is work to be done, and "next(τ)" is the first clock tick after τ at which there is work to be done.

With these notations, the abstract speculative program is:

```
Program A
do $t$ = first while $t \neq \perp$
  doall $\mathcal{F}(t)$
  $t$ = next($t$)
end do
```

Let us introduce:

$$\mathcal{B}(t) = \bigcup_{\tau \ll t} \mathcal{F}(\tau).$$

To be correct, the abstract program A above has to satisfy several conditions:

1. An uncommitted value has to be held in temporary storage until the results of all governing predicates are known. The size of the temporary storage has to be finite.
2. Each operation has to be executed in finite *real* time. Since the execution time of a parallel program is bounded from below by the number of its operations divided by the number of processors, this means that the total number of operations before any logical instant t has to be finite:
$$\text{Card } \mathcal{B}(t) < \infty \tag{8}$$
3. Lastly, the speculative program must terminate whenever the original program does.

The set of uncommitted results is a subset of the set of all results. Hence, condition 2 implies 1. It is more difficult to prove that:

Theorem 3. *Program A terminates, provided the source programs terminates and the schedule is such that all sets $\mathcal{B}(t)$ are finite.*

Proof. The proof is only sketched here. If the original program terminates, it means that no `while` predicate evaluates to true more than a finite number of times. Since the speculative program is partially correct, it has the same property. Since all fronts are finite, there must be an infinite number of fronts for non termination. One can show that this implies an infinite number of true `while` predicates, a contradiction.

The conjunction of theorems 2 and 3 gives a sufficient condition for the total correctness of the target speculative program.

Testing the legality of a schedule According to theorem 3, a schedule is legal if, for all t, Card $\mathcal{B}(t) < \infty$. From the definition (4), one obtains:

$$\begin{aligned}\mathcal{B}_S(t) = \{\mathbf{a} \in \widehat{\mathcal{D}}(S) \mid \theta(\langle \mathtt{S}, \mathbf{a}\rangle) \ll t, \\ \forall v : \theta(v) \ll \theta(\langle \mathtt{S}, \mathbf{a}\rangle) \wedge v\, \delta^c \,\langle \mathtt{S}, \mathbf{a}\rangle \Rightarrow \rho(v) = \textbf{true}\ \}, \quad (9)\\ \mathcal{B}(t) = \bigcup_S \mathcal{B}_S(t). \quad (10)\end{aligned}$$

Since there are only a finite number of statements, it is sufficient to check that each set $\mathcal{B}_S(t)$ is finite. We may obtain a superset of $\mathcal{B}_S(t)$ by selecting several values of v, $v_1, \ldots, v_n$ and replacing the universal quantifier on v by a finite conjunction on $v_1, \ldots, v_n$. To select interesting values for v, let us recall that in the original program, all `while` loops terminate. Let p_i be the nesting level of one of the `while` loops surrounding `S`, and let G be its predicate. The termination condition may be written:

$$\forall \mathbf{a} \exists y : \rho(\langle G, \mathbf{a}[1..p_i - 1].y\rangle) = \textbf{false}.$$

Equivalently, by introducing a Skölem function, we have:

$$\forall \mathbf{a} : \rho(\langle G, \mathbf{a}[1..p_i - 1].N(\mathbf{a}[1..p_i - 1])\rangle) = \textbf{false}.$$

Let us chose:

$$v_i = \langle G, \mathbf{a}[1..p_i - 1].N(\mathbf{a}[1..p_i - 1])\rangle.$$

An easy computation gives:

$$\begin{aligned}\mathcal{B}_S(t) \subseteq \overline{\mathcal{B}_S(t)} = \{\langle \mathtt{S}, \mathbf{a}\rangle \in \widehat{\mathcal{D}}(S) \mid \theta(\langle \mathtt{S}, \mathbf{a}\rangle) \ll t, \\ \bigwedge_{i=1}^{n} \theta(\langle \mathtt{S}, \mathbf{a}\rangle) \ll \theta(v_i) \vee \mathbf{a}[p_i] < N(\mathbf{a}[1..p_i - 1])\}. \quad (11)\end{aligned}$$

Since $\theta(\langle \mathtt{S}, \mathbf{a}\rangle) \ll t$ implies $\theta(\langle \mathtt{S}, \mathbf{a}\rangle)[1] \leq t[1]$, and since the coefficients of `while` counters in a schedule are necessarily positive, this gives an upper bound for all `while` counters which have a nonzero coefficient in $\theta(\langle \mathtt{S}, \mathbf{a}\rangle)[1]$. Let us now consider a counter $\mathbf{a}[p_i]$ which does not occur in $\theta(\langle \mathtt{S}, \mathbf{a}\rangle)[1]$. In the corresponding term of (13), the second disjunct gives an upper bound, but to be sure that $\mathbf{a}[p_i]$ is bounded we need another bound from the first disjunct. Let d_i be the index of the first coordinate of θ in which $\mathbf{a}[p_i]$ occurs. We will obtain a bound provided that:

$$\theta(\langle \mathtt{S}, \mathbf{a}\rangle)[1..d_i - 1] = \theta(v_i)[1..d_i - 1].$$

This suggests the following legality check:

- For all statement and all surrounding `while` loops:
  1. Let p be the depth of the distinguished `while` loop, and let `G` be its predicate. Check whether $\theta(\langle G, \mathbf{a}\rangle) \ll \theta(\langle S, \mathbf{b}\rangle)$ for all $\mathbf{a}, \mathbf{b}$ such that $\mathbf{a}[1..p] = \mathbf{b}[1..p]$ and $\mathbf{a}[p+1] < \mathbf{b}[p+1]$. This situation corresponds to non speculative execution.

2. If this test fails, check that the corresponding loop counter occurs in the first row of the schedule.
3. If this test fails, find the first row of the schedule in which the counter occurs. If there is no such row, the schedule is illegal.
4. Let d be the rank of the selected row. Check whether $\theta(\langle G, \mathbf{a}\rangle)[1..d] = \theta(\langle S, \mathbf{b}\rangle)[1..d]$. If this test fails the schedule is illegal.

Converting this test into a scheduling algorithm will be the subject of future research.

5 Conclusion : Open Problems, Extensions

It should be clear now that this paper is just a preliminary investigation of the problem of constructing parallel speculative programs. We have given a very general prescription for speculation: ignore some control dependences. This prescription may impair the data flow of the program, which has to be reinstated by inserting *compensating* dependences. One then compute a schedule by familiar methods. The result does not necessarily corresponds to a realistic program: the schedule must pass a legality test. Is there a way of directly constructing a legal schedule? Enforcing that `while` loop counters have non zero coefficients in a schedule is easy, but what are we to do if the resulting linear program is unfeasible?

The problem of code generation for a speculative program is still largely open. We need, for instance, an efficient way of computing the "first" and "next" functions. Scanning a front is another problem. If the schedule satisfies the first or second legality criterion (1 and 2 above), the front is a polyhedron or a disjoint union of polyhedra that can be scanned by familiar techniques [1]. In the last legal case the situation is more complicated. Lastly, we have to design a technique for holding uncommitted values and for executing the commit/discard process.

Another question is the selection of the control dependences which are to be ignored. Like all methods based on schedules, this one is at its best when applied to loop nests, not to isolated loops. A simple heuristic is to check that, if all dependences are ignored but the data dependences, the nest has parallelism.

When the data dependences enforce sequential execution by themselves, it may be that expanding arrays may result in a program with more parallelism. The problem of expanding arrays in a dynamic control environment is open and seems quite difficult. However, if we could construct a single assignment program, the problem of commitment would be greatly simplified.

Lastly, in some important cases it may be possible to use speculative results in place of the exact result. This happens, for instance, when a `while` loop implements a stable convergent process. Doing a few extra iterations may not seriously damage the final results. This allows to dispense with the commit/discard mechanism, but there is no simple way to detect this situation. A directive from the programmer is needed.

References

1. Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. In *Proc. third SIGPLAN Symp. on Principles and Practice of Parallel Programming,* pages 39–50, ACM Press, April 1991.
2. Denis Barthou, Jean-François Collard, and Paul Feautrier. Fuzzy array dataflow analysis. *Journal of Parallel and Distributed Computing,* 40:210–226, 1997.
3. J.-F. Collard. Space-time transformation of while-loops using speculative execution. In *Proc. of the 1994 Scalable High Performance Computing Conf.*, pages 429–436, IEEE, Knoxville, TN, May 1994.
4. Paul Feautrier. Dataflow analysis of scalar and array references. *Int. J. of Parallel Programming,* 20(1):23–53, February 1991.
5. Paul Feautrier. Some efficient solutions to the affine scheduling problem, II, multidimensional time. *Int. J. of Parallel Programming,* 21(6):389–420, December 1992.
6. Paul Feautrier. Some efficient solutions to the affine scheduling problem, part I, one dimensional time. *Int. J. of Parallel Programming,* 21(5):313–348, October 1992.
7. M. Griebl and C. Lengauer. On the space-time mapping of while-loops. *Parallel Processing Letters,* 1994. To appear. Also available as Report MIP-9304, Fakultät für Mathematik und Informatik, Universität Passau, Germany.
8. Vadim Maslov. Lazy array dataflow dependence analysis. In *Proc. 21st ACM SIGPLAN-SIGACT Symp. POPL,* pages 2–15, January 1994.
9. Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array dataflow analysis and its use in array privatization. In *Proc. of ACM Conf. on Principles of Programming Languages,* pages 2–15, January 1993.
10. William Pugh and D. Wonnacott. *An Exact Method for the Analysis of Value-Based Data Dependences.* Technical Report CS-TR-3196, U. of Maryland, December 1993.