Loop Parallelization

Generation of Synchronous Code for Automatic Parallelization of while Loops

Martin Griebl[1] and Jean-François Collard[2]

[1] Universität Passau, FMI, Innstraße 33, D-94032 Passau, Germany.
Martin.Griebl@fmi.uni-passau.de
[2] ENS Lyon, LIP, 46 Allée d'Italie, F-69364 Lyon Cedex 07, France.
Jean-Francois.Collard@lip.ens-lyon.fr

Abstract. Automatic parallelization of imperative programs has focused on nests of do loops with affine bounds and affine dependences, because in this case execution domains and dependences can be precisely known at compile-time. When dynamic control structures, such as `while` loops, are used, existing methods for conversion to single-assignment form and domain scanning are inapplicable. This paper gives an algorithm to automatically generate parallel code, together with an algorithm to possibly convert the program to single-assignment form.

1 Introduction

Automatic parallelization of imperative programs has focused on nests of `do` loops with affine bounds and affine dependences [10], mainly because dependences can then precisely be known at compile-time. Data or "memory-based" dependences are due to reuse of memory cells, and thus are language- and program-dependent, whereas dataflows or "value-based dependences" denote transmissions of values and thus are algorithm-dependent. Memory-based dependences can be eliminated if a memory cell is associated with each program operation (the program is then in *single-assignment form*). Intuitively, cancelling memory-based dependences allows to extract more parallelism, hence the interest in automatic parallelization for algorithms to convert programs automatically to equivalent single-assignment form. Then, parallelization through space-time mapping boils down to finding a new coordinate system where some dimensions correspond to time and the others to (virtual) processor coordinates. Code generation then consists of producing a program which scans the execution domain in the new coordinate system.

However, using `while` loops and/or `if`s introduces two main problems:

1. The flow of data is not precisely known at compile-time and must generally be approximated, the ambiguity being resolved at run time. Thus, existing algorithms for automatic conversion to single-assignment form fail.
2. The lack of regularity in execution domains of dynamic control programs forbids scanning schemes to be entirely static. One needs a way of scanning a conservative superset of the execution domain, and of checking on the fly whether a given point corresponds to an actual execution.

This paper proposes solutions to both problems. Our assumed target machine is some abstract shared-memory machine. Section 2 first gives some necessary definitions. Section 3 gives the algorithms for code generation, which are the topic of this paper.

2 Definitions

Mathematical Definitions. The k-th element of a given vector $\boldsymbol{x}$ is denoted by $\boldsymbol{x}[k]$. Furthermore, $\ll$ ($\lll$) denotes the (strict) lexicographical order on such vectors. "max" denotes the maximum operator according to order $\ll$. The modulo operation is denoted by %, and the true and false boolean values by *tt* and *ff*, respectively. A *$\mathbb{Z}$-polyhedron* is the intersection of an integer lattice and a convex real polyhedron [1].

Program Model. We shall restrict ourselves to the following program model:

- The only data structures are arrays of basic types, where array subscripts are affine functions of the counters of surrounding loops and parameters.
- Basic statements are assignments to scalars or array elements.
- The only control structures for a *static control program (SCP)* are the sequence and the `do` loop; *dynamic control programs (DCP)* include, in addition, `while` or `repeat` loop, and conditional `if..then..else` constructs, without restriction on predicates of `while` loops and `if`s.

Statements and Their Instances. An *operation* is a *dynamic instance* of a (syntactic) statement. The instance of a statement in a `do` loop nest is identified by the statement's name and the corresponding loop counter's values. The vector of these values is called *iteration vector*.

While Loops. Since we also want to identify the operations specified by `while` loops, we simply add an artificial counter to every `while` loop. (Note that some variables may be used implicitly as counters, and that there exist algorithms to detect such variables.) The initial value of every such artificial counter is some arbitrary value *lb* (often 0), its step is 1, and its upper bound is not known. Hereafter, we shall write `while` loops as: `do` $w := lb$ `while` (*cond*) S.

A nest of `while` loops that fits our program model is declared in program **WW**:

```
     program WW
G1 : do w1 := 0 while ( P1(w1) )
G2 :       do w2 := 0 while ( P2(w1, w2) )
S  :           a[w1+w2] := a[w1+w2-1]
```

The iteration vector is (w_1, w_2), thus an operation is identified by $\langle S, w_1, w_2\rangle$. The *execution domain* is the set of values that the iteration vector takes in the course of the execution. If the surrounding loops are only `do` loops, then the execution domain is a finite $\mathbb{Z}$-polyhedron. Since we cannot predict statically the flow of control of programs containing `while` loops, their execution domains have to be approximated. The smallest possible *approximate execution domain* of S in program `WW` is the $\mathbb{Z}$-polyhedron $\mathbf{D}(S) = \{(w_1, w_2) | (w_1, w_2) \in \mathbb{N}^2, w_1 \geq 0, w_2 \geq 0\}$. Dimensions of the approximate execution domain that correspond to `while` loops are infinite, but only a finite subset of the infinite polyhedron is executed at run time.

If there is at least one `while` loop not at the outermost level, the execution domain is not convex. Together with its control dependences, it looks more like a (possibly, multi-dimensional) comb (Figure 5a). The sequence of points along a line of arrows in the execution comb—we call it a *tooth*—corresponds to the execution of one entire `while` loop.

The maximal value that a `while` loop counter takes during program execution will be stored in a variable called a *placeholder*; its value is calculated dynamically. For instance, let δ_1 and δ_2 be the placeholders for w_1 and w_2 in program `WW`. Then, we can approximate the execution domain by $\{(w_1, w_2) | 0 \leq w_1 \leq \delta_1, 0 \leq w_2 \leq \delta_2\}$ after the execution terminates.

Finally, note that unpredictable execution domains imply that data dependences have to be approximated as well.

For now, we suppose that predicates P_1 and P_2 in program `WW` do not depend on array a, but only on w_1, and w_1 and w_2, respectively. This simplification allows us to concentrate on code generation (the focus of this paper) without having to deal with parallelization methods such as speculative execution [3].

Parallelization in the Polyhedron Model. A parallelization is a relaxation of the execution order of the instances of S while preserving the dependences together with the termination properties of the input program. Actually, the execution order of `while` loop nests, such as program `WW`, is over-constrained. To show this, we proceed in several steps.

First, we apply a preprocessing step to the source program, in which we (1) explicitely guard the loop body with a predicate *executed* and (2) add a boolean variable *terminated* that stores the current global state of the execution. Then, data dependence analysis and, optionally, conversion to single-assignment form are applied. Based on these results, we derive a *space-time mapping*, i.e., we calculate for every iteration when and where it shall be executed. Finally we *scan* the space-time mapped index domains to generate the (parallel) target loop nest. In the following subsections, we describe each of these steps in more detail and demonstrate the problems that occur.

3 Parallelization Process

3.1 Control Flow in `while` Loops

The flow of control in nests of `while` loops is less constrained than it may appear. For instance, if $P_1(0)$ evaluates to *tt*, then the program's semantics is not changed if the operation $P_1(1)$ immediately follows $P_1(0)$—provided the input program is correct, i.e. all `while` loops terminate and no fatal exception occurs.

More formally, program WW is equivalent to the program in Figure 1.

$$
\begin{array}{ll}
 & (\forall\ (w_1, w_2)\ :\ w_1 \geq 0, w_2 \geq 0\ : \textbf{do begin} \\
E: & \quad \texttt{if}\ \textit{executed}(w_1, w_2) \\
S: & \quad\quad \textbf{then}\ a[w_1+w_2] := a[w_1+w_2-1]\ \textbf{endif}\ ; \\
 & \quad \texttt{if}\ \textit{terminated}\ \textbf{then}\ \texttt{STOP}\ \textbf{endif} \\
 & \textbf{end})
\end{array}
$$

Fig. 1. Program equivalent to program WW

In this recurrent form, the execution order is not over-specified anymore. STOP should be understood as a global immediate program stop. *executed*(w_1, w_2) tests on the fly whether the current instance (operation) should execute or not; *executed*(w_1, w_2) depends on some "previous" instances of this predicate, thus implicitly giving some constraints on the execution order. As mentioned earlier, *terminated* is a global, shared boolean scalar variable that stores the current global execution status.

Predicate executed. The body of a nest of `while` loops (e.g., S in WW) is executed at point x with level r, iff, for all points x' at level $r' \leq r$ whose coordinates $x'_1, \cdots, x'_{r'}$ are identical to $x_1, \cdots, x_{r'}$, respectively, all predicates of `while` loops surrounding x' evaluate to *tt*. Formally, *executed* is defined recursively, where *executed at some level* r means that the body of the loop at level r must be executed—no matter whether it is a statement or another loop:

$\textit{executed}(x_1, \cdots, x_d) = \textit{executed}_d(x_1, \cdots, x_d)$ where $(\forall\ r : 1 \leq r \leq d$:

$\textit{executed}_r(x_1, \cdots, x_r, lb_{r+1}, \cdots, lb_d) =$

$$
\begin{array}{lll}
\texttt{if}\ x_r > lb_r & \rightarrow & \textit{executed}_r(x_1, \cdots, x_r - 1, lb_{r+1}, \cdots, lb_d) \wedge \textit{cond}_r(x_1, \cdots, x_r) \\
[]\ \ x_r = lb_r \wedge r > 1 & \rightarrow & \textit{executed}_{r-1}(x_1, \cdots, x_{r-1}, lb_r, \cdots, lb_d) \wedge \textit{cond}_r(x_1, \cdots, x_r) \\
[]\ \ x_r = lb_r \wedge r = 1 & \rightarrow & \textit{cond}_1(x_1) \\
[]\ \ x_r < lb_r & \rightarrow & \textit{ff} \\
\textbf{endif}\ ) & &
\end{array}
$$

A more detailed explanation is given in [8]. Note that the recursive definition of predicate *executed*$_r$ follows the `while` dependences. If the space-time mapping respects these dependences we can be sure that, during scanning, predicate *executed* never is evaluated at any point x before it was evaluated at x's predecessor.

Termination Problem. A subtle communication scheme for detecting termination in a distributed-memory model where only local communications exist was proposed in [8]. In this paper, shared memory is assumed and detecting termination is simpler.

The execution of a `while` loop nest terminates when the outermost `while` loop has terminated and all instances of inner `while` loops have terminated, too. To implement this, we use a shared global counter that is incremented by every tooth in any dimension that started its execution, and that is decremented by every terminating tooth in any dimension. Thus, the whole program terminates iff there are no active teeth at all, i.e., the counter has been reset to 0.

A formalization of this idea can be added to an imperative specification of *executed* such that the calculation of *terminated* is hidden as a side effect of the masking function *executed* in the target program ($exec_r$ is an r-dimensional persistent array that stores the value of $executed_r(x_1, \cdots, x_r, lb_{r+1}, \cdots, lb_d)$). Function *executed* is called for each scanned point in the approximate execution domain.

$executed(x_1, \cdots, x_d) \equiv$
$r := level(x_1, \cdots, x_d)$;
if $exec_r[x_1, \cdots, x_{r-1}, x_r - 1] \wedge \neg cond_r(x_1, \cdots, x_r)$ **then** $decr(count)$ **endif** ;
$exec_r[x_1, \cdots, x_r] := \ exec_r[x_1, \cdots, x_r] \wedge cond_r(x_1, \cdots, x_r)$;
do $k := 1 + level(x_1, \cdots, x_d)$ **to** d
    $exec_k[x_1, \cdots, x_k] := exec_{k-1}[x_1, \cdots, x_{k-1}] \wedge cond_k(x_1, \cdots, x_k)$;
    if $exec_k[x_1, \cdots, x_k]$ **then** $incr(count)$ **endif**
enddo ;
barrier ;
$terminated := (count = 0)$;
return ($exec_d[x_1, \cdots, x_d]$)

where functions $incr(count)$ and $decr(count)$ increment and decrement *count* by 1, respectively. $cond_0()$ and $executed_0()$ must be initialized to *tt*. The *level* of a point is defined as d minus the number of trailing *lb* coordinates.

Case distinction by calculating the *level* above yields the code generation scheme for *executed* in Fig. 2. The generated code for *executed* in the case of program **WW** is as follows:

function $executed(w_1, w_2)$ **: boolean**
if $w_2 > 0$ **then**
    if $exec_2[w_1, w_2 - 1]$ **and not** $P_2(w_1, w_2)$ **then** $decr(count)$ **endif** ;
    $exec_2[w_1, w_2] := exec_2[w_1, w_2 - 1]$ **and** $P_2(w_1, w_2)$;
else if $w_1 > 0$ **then**
    if $exec_1[w_1 - 1]$ **and not** $P_1(w_1)$ **then** $decr(count)$ **endif** ;
    $exec_1[w_1] := exec_1[w_1 - 1]$ **and** $P_1(w_1)$;
    $exec_2[w_1, w_2] := exec_1[w_1]$ **and** $P_2(w_1, w_2)$;
    if $exec_2[w_1, w_2]$ **then** $incr(count)$ **endif**
else /* $w_1 = w_2 = 0$ */
    $exec_1[w_1] := P_1(w_1)$;
    if $exec_1[w_1]$ **then** $incr(count)$ **endif** ;
    $exec_2[w_1, w_2] := exec_1[w_1]$ **and** $P_2(w_1, w_2)$;
    if $exec_2[w_1, w_2]$ **then** $incr(count)$ **endif**
endif ;
barrier ;
$terminated := (count = 0)$;
return ($exec_2[w_1, w_2]$)

```
Algorithm executed_generator
Input:
    • The d while loop conditions.
    • The d loop counters (x_1, ···, x_d) (become the arguments to executed).
Output: Code implementing function executed

generate( function executed(x_1, ···, x_d)  : boolean )
for r:=d downto 0
    if r ≥ 1 then
        generate( if x_r > 0 then )
        generate( if exec_r[x_1, ···, x_{r-1}, x_r - 1] and not cond_r(x_1, ···, x_r)
                  then decr(count) endif )
        generate( exec_r[x_1, ···, x_r] := exec_r[x_1, ···, x_{r-1}, x_r - 1] and
                  cond_r(x_1, ···, x_r) )
    end if
    for k := r+1 to d
        generate( exec_k[x_1, ···, x_k] := exec_{k-1}[x_1, ···, x_{k-1}] and
                  cond_k(x_1, ···, x_k) )
        generate( if exec_k[x_1, ···, x_k] then incr(count) endif )
    end for
    if r ≥ 1 then generate ( else ) else generate ( endif )
end for
generate( barrier )
generate( terminated := (count = 0) )
generate( return ( exec_d[x_1, ···, x_d] ) )
```

Fig. 2. Algorithm *executed_generator* for automatic generation of the code for *executed*.

Lemma 1. The implementation of terminated via the counters is correct.

Sketch of the Proof. The following properties ensure that, at a given time step t, *terminated* is not set to *tt* if some `while` loop iteration has not terminated in the execution domain:

- For every tooth in every dimension *count* is incremented once (at its root) and decremented once (at its tip)—in this order. During execution every tooth contributes 1 to the global value of *count*, whereas before the start and after termination there is no contribution to *count*.
- If there is at least one processor evaluating some $executed_r(x_1, \cdots, x_d)$ $(1 \leq r \leq d)$ to *tt* at time t then the tooth τ at level r through $(x_1, \cdots, x_r, lb_{r+1}, \cdots, lb_d)$ has started but not yet finished. Thus, at this point in time, τ is contributing 1 to *count*.
- The barrier synchronisation ensures that all updates to *count* occurred before the processors read the value of *count*. Since the order in which increments and decrements take place is not relevant to the final value, all processors see the same value.
- Since every tooth with some executing point on it contributes 1 to *count* and since there could not have been more decrements than increments *count* must at least have the value 1, thus preventing termination.

Remark. In data-parallel languages with the construct *whilesomewhere*, termination detection by the counter scheme can be replaced by formulating the outermost loop on time as *whilesomewhere*($executed_{level(x_1,\cdots,x_d)}$).

3.2 Dependence Analysis

Let E be the statement calling *executed* in Fig 1. Dependences due to access to arrays $exec_1$ and $exec_2$ are one-to-one, corresponding to edges e_1 and e_2 in Figure 3. In contrast, dependences on S have to be approximated by sets because we cannot predict at compile-time which operations execute and which do not. Hence, in the case of dynamic control programs, elaborate dependence analyses have to be applied [5]. For instance, an analysis of program **WW** tells that the source of the datum read by $\langle S, w_1, w_2\rangle$ is

$$\left|\begin{array}{l}\textbf{if}\ w_2 \geq 1\\ \textbf{then}\ \{\langle S, w_1, w_2-1\rangle\}\\ \textbf{else}\left|\begin{array}{l}\textbf{if}\ w_1 \geq 1\\ \textbf{then}\ \{\langle S, \alpha, \beta\rangle \mid \alpha+\beta = w_1-1, \alpha \geq 0, \beta \geq 0, \alpha < w_1\}\\ \textbf{else}\ \{\bot\}\end{array}\right.\end{array}\right. \tag{1}$$

The first leaf of (1) is a singleton, meaning that if $w_2 \geq 1$, only one operation can be the source of the flow of $a[w_1, w_2-1]$ to $\langle S, w_1 w_2\rangle$. In contrast, the second leaf is a non-singleton set of possible sources. The last leaf only contains $\bot$, the "undefined" value, meaning that the read has no source in the given program. The two non-bottom leaves yield edges e_3 and e_4 in Figure 3. Similarly, edges e_5 and e_6 correspond to memory-based, output and anti dependences respectively.

To eliminate memory-based dependences, the input program may be converted into single-assignment form by applying the following rules:

- Replace lhs expressions by an array subscripted by iteration vectors.
- Replace rhs expressions by the result of the dataflow analysis (such as (1)):
  - replace singleton leaves by references to the array cells written by the corresponding operation, or by initial references if the leaf is $\{\bot\}$,
  - replace non-singleton leaves by a call to a function *last* (defined later).

Example 1. A single-assignment version of program **WW** is:

```
    (∀ (w1, w2) : w1 ≥ 0 ∧ w2 ≥ 0 : do
    begin
          if executed(w1, w2)
S :       then A[w1, w2] :=if w2 ≥ 1 then A[w1, w2-1]
                           else if w1 ≥ 1 then last_{A,a}(w1, w2)
                                else a[w1+w2-1]
          if terminated then STOP
    end)
```

Edges	Description	Conditions
e_1	$\langle E, w_1, w_2-1\rangle \to \langle E, w_1, w_2\rangle$	$w_2 \geq 1$
e_2	$\langle E, w_1-1, 0\rangle \to \langle E, w_1, 0\rangle$	$w_1 \geq 1$
e_3	$\langle S, w_1, w_2-1\rangle \to \langle S, w_1, w_2\rangle$	$w_2 \geq 1$
e_4	$\{\langle S, \alpha, \beta\rangle \mid \alpha+\beta = w_1+w_2-1, \alpha \geq 0, \beta \geq 0, \alpha < w_1\} \to \langle S, w_1, 0\rangle$	$w_1 \geq 1, w_2 = 0$
e_5	$\{\langle S, \alpha, \beta\rangle \mid \alpha+\beta = w_1+w_2, \alpha \geq 0, \beta \geq 0, \alpha < w_1\} \to \langle S, w_1, w_2\rangle$	$w_1 \geq 1$
e_6	$\{\langle S, \alpha, \beta\rangle \mid \alpha+\beta-1 = w_1+w_2, \alpha \geq 0, \beta \geq 0, \alpha < w_1\} \to \langle S, w_1, w_2\rangle$	$w_1 \geq 1$

Fig. 3. Dependences in program **WW**.

Predicate *executed* restores the flow of control [8, 9], and function *last* dynamically restores the flow of data. In other words, predicate *executed* checks whether the current loop iteration corresponds to an actual execution of statement S. Function *last* returns the value produced by the operation executed last (according to order $\ll$), which wrote into memory cell $a(w_1+w_2-1)$. Function *last* is similar to the ϕ-function proposed by Cytron et al. [6], and implements the result of an array dataflow analysis since the returned value is the one produced by the last possible source that was executed, or by the initial element of array a if no possible source was executed. An implementation for *last* is:

```
function last_{A,a}(w1, w2)  : datum
do α := w1-1 to 0 step -1
      β := w1+w2-1-α
      if exec2[α, β] then return ( A[α, β] )
enddo
return ( a[w1+w2-1] )
```

Automatic Generation of Function last. In Figure 4, we propose an algorithm *last_generator* to generate automatically the code for function *last*. This algorithm scans a given $\mathbb{Z}$-polyhedron $\mathcal{D}$ in opposite lexicographical order. u_k (l_k) stores the upper (lower) bound on the kth coordinate of scanned operations and is equal to the floor (ceiling) of the first component of the projection of $\mathcal{D}$ on the $n-k+1$ first dimensions. u_k and l_k can be computed by thanks to software such as PIP [7]. If the upper bound is undefined, then u_k is set equal to the corresponding placeholder (Section 2).

When the current point corresponds to an actual execution, then *last* returns the corresponding cell in array A (passed as a second argument to *last_generator*). If no scanned point corresponds to an executed operation, then a read from the original cell of array a is returned.

Example 2. For program **WW**, the first argument to *last_generator* is the non-bottom part of the second leaf of (1), i.e. $\{\alpha, \beta \mid \alpha+\beta = w_1-1, \alpha \geq 0, \beta \geq 0, \alpha < w_1\}$; so, $n = 2$. The remaining arguments are array A, the initial array expression

Algorithm *last_generator*
Input:

- A $\mathbb{Z}$-polyhedron $\mathcal{D}$ given by a system of affine constraints.
- An array A.
- An array expression e.
- The loop counters w (become the arguments to *last*).

Output: Code implementing Function *last*.

generate(**function** $last_{A,a}$ (w) : **datum**)
let n be the dimension of $\mathcal{D}$
for $k := 1$ to n
  compute $l_k := \lceil \min_{\ll} \{\alpha_k, \ldots, \alpha_n | \, \alpha \in \mathcal{D}\}[1] \rceil$
  compute $u_k := \lfloor \max_{\ll} \{\alpha_k, \ldots, \alpha_n | \, \alpha \in \mathcal{D}\}[1] \rfloor$
  if $u_k = \infty$ then $u_k := \delta_k$
  if $l_k = u_k$ then
    generate(α_k **:=** l_k)
  else
    generate(**do** α_k **:=** u_k **to** l_k **step -1**)
  if $k = n-1$ then
    generate(**if** $exec_n[\alpha]$ **then return** (A [α]))
  if $l_k \neq u_k$ then
    generate(**enddo**)
end for
generate(**return** (e))

Fig. 4. Algorithm *last_generator* for automatic generation of the code for *last*.

$e = a[w_1+w_2-1]$, and the counters of the loops surrounding the call to *last*, i.e. $w = (w_1, w_2)$. For $k = 0$, $\min_{\ll} \{(\alpha,\beta) | \, (\alpha,\beta) \in \mathcal{D}\} = (0, w_1+w_2-1)$, hence $l_0 = 0$. Symmetrically, $\max_{\ll} \{(\alpha,\beta) | \, (\alpha,\beta) \in \mathcal{D}\} = (w_1-1, w_2)$, thus $u_0 = w_1-1$. Hence the bounds of the outermost **do** loop. For $k = 1$,

$$l_1 = \min_{\ll} \{(\beta) | \, (\alpha,\beta) \in \mathcal{D}\} = (w_1+w_2-\alpha-1)$$
$$u_1 = \max_{\ll} \{(\beta) | \, (\alpha,\beta) \in \mathcal{D}\} = (w_1+w_2-\alpha-1)$$

Since $u_1 = l_1$, a simple assignment to β is generated instead of an inner loop.

3.3 Finding Space-Time Mappings

Finding space-time mappings, i.e. schedules and processor mappings, is beyond the scope of this paper. For more details on the subject, the reader is referred to [10].

If program **WW** is not converted to single-assignment form, then a possible schedule for both S and *executed*, is: $\theta(\langle S, w_1, w_2 \rangle) = 3w_1 + w_2 + C$. ($C$ is some arbitrary additive constant.) A possible processor mapping is w_1, yielding a unimodular space-time mapping.

In the case of single-assignment form, getting rid of dependences e_5 and e_6 allows a faster schedule. The method proposed in [4] derives the following scheduling function (for both S and *executed*, as before): $\theta(\langle S, w_1, w_2\rangle) = w_1{+}w_2$. A possible processor mapping is also w_1.

In both cases, the target code will have to scan all operations that are really executed and avoid "holes". Figure 5 shows, on the left, a possible execution of program **WW**; black dots represent real executions and grey dots denote approximated operations. When mapping $t = w_1+w_2, p = w_1$ is applied (right), only three operations should be spawned at time step 3 and one (operation $(3,2)$) should be skipped. Code generation is responsible for ensuring this [8].

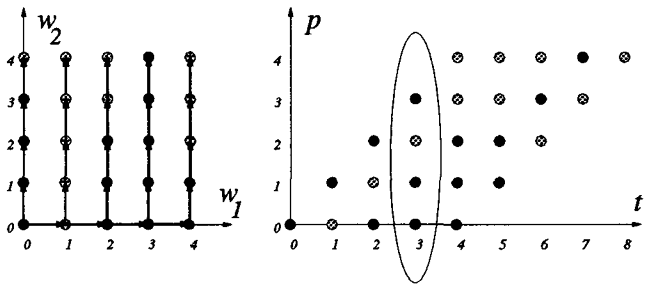

Fig. 5. A given execution of Program **WW** (left) and the target execution domain (right) with mapping $t = w_1+w_2, p = w_1$.

3.4 Code Generation

After applying the space-time mapping we must re-construct a target loop nest. For that purpose, we shall first present the target loops themselves according to standard techniques [10], then show how the body and the auxiliary functions *executed* and possibly *last* are reindexed according to [2], and finally solve implementational problems for the arrays *exec*.

Without Single-Assignment. With the space-time mapping shown above, the scanning of the target domain yields:

do $t := 0$ **while** (**not** *terminated*)
  doall $p := 0$ **to** $\lfloor \frac{t}{3} \rfloor$
    if $executed(w_1, w_2)$ **then** $a[w_1{+}w_2] := a[w_1{+}w_2{-}1]$

Reindexing. Let T be the space-time transformation. It is defined by: $t = 3w_1+w_2$, $p = w_1$, so its inverse T^{-1} is: $w_1 = p$, $w_2 = t{-}3p$. Let L be the subscripting function; subcript $L(w_1, w_2)$ is replaced by $T(L(T^{-1}(t,p)))$. For instance, when $L(w_1, w_2) = (w_1, w_2{-}1)$, then $L(T^{-1}(t,p)) = (p, t{-}3p{-}1)$ and eventually the new subscript is $(3p + (t{-}3p{-}1), p) = (t{-}1, p)$.

Thus, *executed*(t,p) is derived from *executed*(w_1,w_2) in Section 3.1 by replacing the array subscripts for $exec_r$ by the result of $T(L(T^{-1}(t,p)))$, and by replacing all other occurrences of w_1, w_2 by $p, t-3p$, respectively.

Memory Allocation. Since the size of the arrays $exec_r$ (for $1 \leq r \leq d$) can grow dynamically, we must use dynamic data structures instead of arrays. In general, for reducing memory requirements, we fold the time dimension of the arrays for $exec_r$ according to the delay of accesses on them [2], and we bound the space dimensions by expressions w.r.t. time.

In our concrete example, the longest delay of accesses on array $exec_2$ is from time t to time $t-1$, i.e., 1. Thus, we may fold the first dimension of array $exec_2$ by modulo 2. The second dimension is bounded by $p = w_1 = (t-w_2)/3 \leq \lceil t/3 \rceil$. Therefore, we have to insert the memory allocation statements $exec_2[t$ % $2]$:=**malloc** $(\lceil t/3 \rceil)$ and **free** $((t-1)$ % $2)$ as the first and the penultimate statement of the body of *executed*, respectively.

With Single-Assigment. The mapping presented in Section 3.3 is invertible $(w_1 = p, w_2 = t-p)$, so program **WW** becomes:

```
program WW
do t := 1 while ( not  terminated )
  doall p := 0 to t-1
     if exec2[t,p] then
       A( t,p ):= if t-p-1 ≥ 1  then A( t-1,p )
                  else if p ≥ 1 then last(p,t-p) else a(t-2)
```

Functions *executed* and *last* have to be reindexed according to the space-time transformation:

```
function last(w1,w2)
do α := w1-1 to 0 step -1
  β := w1+w2-1-α
  if exec2[α,β] then return ( A( α+β,β ) )
return ( a[w1+w2-1] )
```

executed also has to be reindexed as explained in 3.4. Similarly, dynamic allocation is used as described in Paragraph 3.4.

4 Conclusions

Automatic parallelization of dynamic control programs, e.g. including **while** loops, requires not only appropriate dependence analysis and scheduler, but an appropriate code generator, too. This paper proposed algorithms for this purpose.

To eliminate memory-based dependences while coping with unpredictable data flows, a new mechanism (function *last*) has been introduced and an algorithm to generate the code implementing *last* has been proposed. Several other implementation schemes for *last* can be imagined, and thorough experiments are necessary to select the most effective one; the scheme presented in this paper is the simplest, most abstract one.

Scanning irregular non-dense execution domains requires some run-time tests, and thus incurs an execution overhead; it may also yield unbalanced workloads on processors. Both properties mainly depend on the application, and we expect that parallelizing nests of `while` loops will prove to be efficient only for some types of algorithms.

On the other hand, we believe that one of the main drawbacks of current automatic parallelizers is their severe syntactical restrictions on input programs. The methods proposed in this paper allow code generators in automatic parallelizers to accept a much wider range of programs than current implementations do.

Acknowledgments

Thanks for the support by the German-French research exchange program PROCOPE, the DFG project RecuR, the CNRS program PRS, PRC/MRE contract ParaDigme, and the DRET contract 91/1180.
Many thanks to Nils Ellmenreich for a careful reading and to Chris Lengauer and Gil Utard for the perusal and fruitful comments on this paper.

References

1. C. Ancourt. *Génération automatique de codes de transfert pour multiprocesseurs à mémoires locales.* PhD thesis, Univ. of Paris 6, Paris. March 1990.
2. J.-F. Collard. Code generation in automatic parallelizers. In C. Girault, editor, *Proc. of the Int. Conf. on Applications in Parallel and Distributed Comp., IFIP W.G 10.3*, pages 185–194, Caracas, Venezuela. North Holland, April 1994.
3. J.-F. Collard. Automatic parallelization of `while`-loops using speculative execution. *Int. J. Parallel Programming*, 1995. To appear. Earlier version: *Proc. 1994 Scalable High Performance Comp. Conf.*, pages 429–436. IEEE, May 1994.
4. J.-F. Collard and P. Feautrier. A method for static scheduling of dynamic control programs. Tech. Report 94-34, LIP, Ecole N. S. de Lyon. December 1994.
5. J.-F. Collard, D. Barthou and P. Feautrier. Fuzzy array dataflow analysis. In *Proc. of 5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Prog.*. Santa Barbara, CA. July 1995.
6. R. Cytron et al. An Efficient Method of Computing Static Single Assignment Form. In *Proc. of 16th ACM Symp. on Principles of Programming Languages*, pages 25–35. January 1989.
7. P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, September 1988.
8. M. Griebl and C. Lengauer. On scanning space-time mapped `while` loops. In B. Buchberger and J. Volkert, editors, *Parallel Processing: CONPAR 94 – VAPP VI*, LNCS 854, pages 677–688. Springer-Verlag, 1994.
9. M. Griebl and C. Lengauer. On the parallelization of loop nests containing `while` loops. In N. Mirenkov, editor, *Proc. Aizu Int. Symp. on Parallel Algorithm/Architecture Synthesis (pAs'95)*, pages 10–18, Aizu-Wakamatsu, Japan. IEEE, March 1995.
10. C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, LNCS 715, pages 398–416. Springer-Verlag, 1993.