Automated Generation of Polyhedral Process Networks from Affine Nested-Loop Programs with Dynamic Loop Bounds

DMITRY NADEZHKIN, HRISTO NIKOLOV, and TODOR STEFANOV,
Leiden Institute of Advanced Computer Science

The Process Networks (PNs) is a suitable parallel model of computation (MoC) used to specify embedded streaming applications in a parallel form facilitating the efficient mapping onto embedded parallel execution platforms. Unfortunately, specifying an application using a parallel MoC is a very difficult and highly error-prone task. To overcome the associated difficulties, we have developed the *pn* compiler, which derives specific Polyhedral Process Networks (PPN) parallel specifications from sequential static affine nested loop programs (SANLPs). However, there are many applications, for example, multimedia applications (MPEG coders/decoders, smart cameras, etc.) that have adaptive and dynamic behavior which cannot be expressed as SANLPs. Therefore, in order to handle dynamic multimedia applications, in this article we address the important question whether we can relax some of the restrictions of the SANLPs while keeping the ability to perform compile-time analysis and to derive PPNs. Achieving this would significantly extend the range of applications that can be parallelized in an automated way.

The main contribution of this article is a first approach for automated translation of affine nested loop programs with dynamic loop bounds into input-output equivalent Polyhedral Process Networks. In addition, we present a method for analyzing the execution overhead introduced in the PPNs derived from programs with dynamic loop bounds. The presented automated translation approach has been evaluated by deriving a PPN parallel specification from a real-life application called Low Speed Obstacle Detection (LSOD) used in the smart cameras domain. By executing the derived PPN, we have obtained results which indicate that the approach we present in this article facilitates efficient parallel implementations of sequential nested loop programs with dynamic loop bounds. That is, our approach reveals the possible parallelism available in such applications, which allows for the utilization of multiple cores in an efficient way.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms: Design, Theory, Algorithms, Performance

Additional Key Words and Phrases: Models of Computation, polyhedral process networks, compiler techniques for MPSoCs, parallel programing

ACM Reference Format:
Nadezhkin, D., Stefanov, T., and Nikolov, H. 2013. Automated generation of polyhedral process networks from affine nested-loop programs with dynamic loop bounds. ACM Trans. Embedd. Comput. Syst. 13, 1s, Article 28 (November 2013), 24 pages.
DOI: http://dx.doi.org/10.1145/2536747.2536750

1. INTRODUCTION

Moving from sequential computing to parallel computing has become necessary nowadays because single-processor embedded systems cannot cope anymore with applications complexity, throughput, and power consumption constraints that are inherent to

Author's addresses: D. Nadezhkin, H. Nikolov, and T. Stefanov, Leiden Institute of Advanced Computer Science, Leiden University, Niels Bohrweg 1, 2333CA, Leiden, The Netherlands. Correspondence email: {dmitryn, stefanov, nikilov}@liacs.nl.

DOI: http://dx.doi.org/10.1145/2536747.2536750

```
parameter N 10 100;

 for j = 1 to 6*N-3,
   A[j] = Func1()
 endfor

 for j = 0 to N,
   for i = j to 3*j-2,
     if( i+j < 4*N-6 )
       A[i] = Func2( A[2*i-1], A[2*i+1] )
     endif
     Func3( A[i] )
   endfor
endfor
```

Fig. 1. Pseudocode of a SANLP.

so many embedded applications. Although, we are witnessing the emergence of parallel (multicore and multiprocessor) systems in all markets: from general-purpose computing to embedded systems, for example, multimedia systems, game consoles and all sorts of mobile devices, the transition from sequential to parallel computing is far from trivial. To satisfy emerging applications requirements, the multiprocessor embedded systems must be programmed in a way that the available parallelism is revealed and exploited efficiently. However, programming of a multiprocessor system is a challenging, error-prone, and time consuming task as it involves the partitioning of programs, and consequently, synchronization of different program partitions. In recent years, a lot of attention has been paid to the design of parallel systems. However, insufficient attention has been paid to the development of concepts, methodologies, and tools for efficient programming of such systems. Therefore, the programming still remains a major difficulty and challenge [Martin 2006]. Today, system designers experience significant difficulties in programming parallel systems because the way an application is specified by an application developer, typically as a sequential program using a sequential *Model of Computation* (MoC), does not match the way multiprocessor systems operate, that is, multiple cores run (possibly) in parallel.

If an application is specified using a parallel MoC, then the mapping of this application onto a multiprocessor system can be done in a systematic and transparent way by using a disciplined approach [Mihal and Keutzer 2003]. Using a parallel MoC facilitates the programming of parallel multiprocessor systems because a parallel MoC makes the parallelism available in an application and the communication between the application tasks explicit. Unfortunately, specifying an application using a parallel MoC is very difficult as the application developers (i) have to be familiar with a particular parallel MoC; (ii) have to study the application in order to identify possible parallelism that is available and to reveal it by using the parallel model.

To relieve the designer from all these difficulties, the *pn* compiler [Verdoolaege et al. 2007] was introduced. It implements techniques for automated parallelization of *Static Affine Nested-Loop Programs* (SANLP) written in *C* into input-output equivalent *Polyhedral Process Network* (PPN) descriptions. In the *pn* partitioning strategy, a process is created for every statement and function call found in the top-level of the program. In this way, the designers have control over the granularity of the created partitions.

An example of a SANLP is given in Figure 1. A SANLP consists of a set of statements and function calls, each possibly enclosed in loops and/or guarded by conditions. The loops do not have to be perfectly nested. All lower and upper bounds of the loops as well as all expressions in conditions and array accesses have to be affine functions of enclosing loop iterators and static parameters. The parameters are symbolic constants, that is, their values cannot change during the execution of the program. Rather, parameter values determine different program instances. In addition, data communication between function calls must be explicit. For example, see function `Func2()` at line 8 which accepts 2 elements of array `A[]` as input arguments. Providing just a pointer to array `A[]` in this case is not allowed. The above restrictions allow a compact mathematical representation of a SANLP using the well-known polyhedral model [Feautrier 1996]. The SANLPs can be converted in an automated way into Polyhedral Process Networks [Verdoolaege et al. 2007].

The PPN model of computation is a special (static) case of the *Kahn Process Networks* (KPN) [Kahn 1974] model of computation. A PPN consists of concurrent autonomous processes that communicate data in a point-to-point fashion over bounded FIFO channels using a blocking read/write on an empty/full FIFO as synchronization mechanism. In addition, everything about the execution of a PPN is known at compile-time. The latter enables techniques for modeling, analysis, and SW/HW synthesis in a systematic and automated way, and allows the calculation of buffer sizes that guarantee deadlock-free execution. In comparison, computing buffer sizes is not possible for the more general KPN model. We are interested in the process network model because it provides a sound formalism, well suited for capturing and modeling of data-flow dominated applications in the realm of multimedia, imaging, and signal processing, that naturally contain tasks communicating via streams of data. Moreover, it has been already shown that process networks allow effective and efficient mappings of streaming applications to certain parallel execution platforms [Stefanov et al. 2004; de Kock 2002; Goossens et al. 2003; Dwivedi et al. 2004; Castrillon et al. 2010; Haid et al. 2009].

Many scientific, matrix computation, and signal processing applications can be specified as static affine nested loop programs, and therefore, the *pn* compiler [Verdoolaege et al. 2007] can be used to derive equivalent parallel PPN specifications. However, many multimedia applications such as MPEG coders/decoders, smart cameras, etc. have adaptive and dynamic behavior which cannot be expressed as SANLPs. In order to handle dynamic applications, in this article, we address the important question whether we can relax some of the restrictions of the SANLPs while keeping the ability to perform compile-time analysis and to derive PPNs in an automated way. Achieving this will significantly extend the range of applications that can be parallelized in an automated way. The main contribution of this article is a first approach for automated translation of affine nested loop programs with dynamic loop bounds (`Dynloop`) into input-output equivalent Polyhedral Process Networks. In addition, we propose an analysis which estimates the execution overhead introduced in the PPNs derived from programs with dynamic loop bounds. The presented automated PPN derivation approach has been evaluated by deriving a PPN parallel specification from a real-life application called *Low Speed Obstacle Detection* (LSOD) used in the smart cameras domain. The obtained results indicate that the approach we present in this article facilitates efficient parallel implementations of sequential nested loop programs with dynamic loop bounds. That is, our approach reveals the possible parallelism available in such applications, which allows for the utilization of multiple cores in an efficient way.

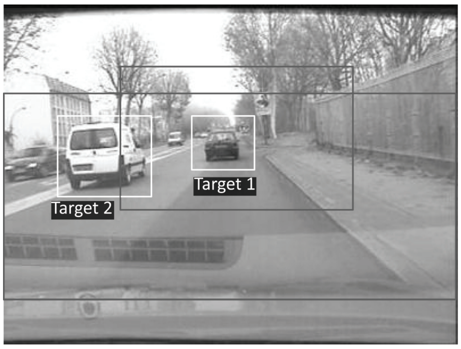
1.1. Motivating Example

As a motivating example, we use an application from the smart cameras domain called low speed obstacle detection (LSOD). With the LSOD description in this section, we illustrate a program that has the specific dynamic behavior we consider in this article and we outline the problems introduced by this behavior.

The LSOD application is intended to detect and to track objects in front of a car in traffic. The output of the system presents spatial positions for targets—cars, pedestrians, etc. Applying several general image processing algorithms helps to find new targets, and to track existing targets. The algorithms implement shadow detection, symmetry detection, lights detection, motion segmentation, and vertical edge detection. The

```
0 vsum[] = 0
1 for k = 1 to Targets,
2   [Height,Width,X,Y] = getLSODTarget(k)
3   for j = 0 to Height+1,
4     for i = 0 to Width+1,
5       img[j,i] = readTarget(X,Y)
6     endfor
7   endfor
8   for j = 1 to Height,
9     for i = 1 to Width,
10      img_out[j,i] = edgeDetection(
                     img[j-1,i-1],img[j-1,i+1],
                     img[j  ,i-1],img[j  ,i+1],
                     img[j+1,i-1],img[j+1,i+1])
11      img_out[j,i] = absVal( img_out[j,i] )
12    endfor
13  endfor
14  for j = 1 to Height,
15    for i = 1 to Width,
16      vsum[i] = vertSum( vsum[i], img_out[j,i] )
17    endfor
18  endfor
19 endfor
```

(a) Pseudocode of the edge detection part of the motivating example.Target size is specified by variables `Height` and `Width`.



(b) LSOD applied on real data. The vehicles in front of the camera are detected and tracked. The dark rectangles depict the area of the image that is processed.

Fig. 2. Pseudocode of the edge detection part of the LSOD application and its application on real data.

output of each algorithm is collected by a particle filter component [Arulampalam and Maskell 2002] for analysis.

The first step in the LSOD application is to obtain two images from a given camera picture. They are named high- and low-resolution images and are depicted by the two dark rectangles in Figure 2(b). Applying different image processing algorithms on these images, hypotheses whether cars exist are computed. Possible targets are defined as coordinates and dimensions of rectangles belonging either to the high- or low-resolution image. In Figure 2(b), two possible targets are presented by the white rectangles, surrounding the cars. Then, for every identified target, the image gradient in vertical direction of the area of the target is computed. The result is finally analyzed in order to support or decline a target.

The edge detection part of the LSOD application, shown in Figure 2(a), is an example of a program which is not a static affine nested loop program. This program is affine nested loop program but it has dynamic control as function `getLSODTarget()` at line 2 initializes variables `Height` and `Width` used as loop bounds. These variables define the size of a target, that is, the amount of data to be processed, and change values for every target at run-time. Since targets are moving in front of a camera (which is also moving), the identified positions stored in variables `(X,Y)` and dimensions `(Height,Width)` will differ for different targets in the frame and for one and the same target in different frames. That is why, the values of variables `Height` and `Width` (as well as the number of targets) are unknown at compile-time, and therefore, the *pn* compiler [Verdoolaege et al. 2007] cannot handle the program shown in Figure 2(a). In this article, we propose a solution approach to this problem by introducing a novel procedure for automated translation of affine nested loops programs with dynamic loop bounds into input-output equivalent polyhedral process networks.

The remaining part of this article is organized as follows. In the following section, we cover the related work. In Section 3, we introduce some notations and present two techniques currently used to analyze sequential programs. This is needed for better understanding of the solution approach we propose and discuss in Section 4. Then, in Section 5, we present an analysis to estimate the execution overhead introduced in the

PPNs derived from programs with dynamic loop bounds. An application of our solution approach to the LSOD in Figure 2(a), as well as, performance and overhead evaluation of the generated PPN are presented in Section 6. Finally, Section 7 concludes this article.

2. RELATED WORK

The work presented in this article is an extension to previous works on systematic and automated derivation of process networks from affine nested loops programs. That is, Turjan et al. [2004] proposed an automated derivation of process networks from *static* affine nested loop programs. In SANLPs, the memory array subscripts, loop bounds and conditional control structures are affine constructs of surrounding loop iterators, program parameters and constants. Stefanov [2004] further developed a procedure of process network derivation from more relaxed class of affine nested loop programs called *Weakly Dynamic Programs* (WDPs). In this class of affine nested loops programs, the conditions in control structures might be dependent on some information that is unknown at compile-time and may change at run-time. In contrast, our approach presented in this article deals with affine nested loop programs with loop bounds (`Dynloop`) unknown at compile-time and determined at run-time.

There are a number of efforts which address the problem of parallelization of nested-loop programs with dynamic structures. Raman et al. [2008] devise the *Parallel-Stage Decoupled Software Pipelining* (PS-DSWP) multithreading technique to extract pipeline parallelism from codes with irregular, pointer-based memory accesses and arbitrary control flow, which generally include while-loops. A parallel-stage allows to obtain parallelism from pipeline stages in which all iterations can execute concurrently. In contrast, besides the pipeline- and iteration-level parallelism, our approach supports also task- and data-level parallelism. Moreover, we can generate parallel code for multiprocessor systems with distributed memory.

Knobe and Sarkar [1998] proposed a procedure for converting nested loop programs into a single assignment form that they called *Array Static Single Assignment* (ASSA). Their procedure accepts as an input a more general class of nested loop programs than the programs considered in this article (`Dynloop`). Because of this, and due to the fact that most of the analysis presented in Knobe and Sarkar [1998] is done at run-time, their approach produces a large overhead in terms of memory usage and execution time. The LooPo compiler [Griebl and Lengauer 1996] deals with parallelization of more general class of nested loop program than the class we consider in this article. It includes nested loop programs with unscannable execution spaces which boundaries are determined at run-time. The proposed parallelization procedure is based on run-time detection of executed statements as well as detection of program termination [Geigl et al. 1999]. In contrast to Knobe and Sarkar [1998] and Griebl and Lengauer [1996], we use Fuzzy *Array Dataflow Analysis* (FADA) algorithm in order to perform approximated dependence analysis at compile-time. Moreover, we do as much as possible analysis at compile-time, thereby reducing the run-time overhead significantly.

A different approach is taken by Benabderrahmane et al. [2010] where they embed the `control` and `exit` predicates to the general data-dependent control-flow programs. These predicates are used instead of data dependent control structures and while-loops as first-class citizens of the algebraic representation. Subsequently, a polyhedral representation is derived and code generation is performed from static program analysis. In this approach, hiding all dynamism in algebraic representations also diminishes the parallelism available in the initial program as less information is visible for analysis. By contrast, our technique exposes and utilizes all available parallelism.

```
1  parameter M 1 10
2  parameter N 1 10

3  for k = 1 to M,
S1:  y[k] = F1()
5  endfor
6  for i = 1 to N,
7    for j = i to M,
8      if j <= 2,
S2:      y[j] = F2()
10     endif
S3:    [] = F3(y[j])
12   endfor
13 endfor
```

(a) Static Affine Nested Loop Program.

```
1  parameter M 1 10
2  parameter N 1 10

3  for k = 1 to M,
S1:  y[k] = F1()
5  endfor
6  for i = 1 to N,
7    for j = i to M,
8      if y[j] <= 2,
S2:      y[j] = F2()
10     endif
S3:    [] = F3(y[j])
12   endfor
13 endfor
```

(b) Weakly Dynamic Program.

Fig. 3. Examples of SANLP and WDP programs. The only difference is that in WDP, the conditional statement in line 8 is data-dependent.

3. BACKGROUND

In this section, we introduce some notations used throughout the article. Also, for better understanding of the solution approach, we briefly present two state-of-the-art techniques used to analyze sequential programs. The first one, called *Exact Array Dataflow Analysis* (EADA) [Feautrier 1991], is used to analyze static programs, namely SANLPs. EADA is implemented in the *pn* compiler for the translation of SANLPs to polyhedral process networks. We formally describe EADA in Section 3.2. The second technique, which we present in Section 3.3, allows for the analysis of programs with more relaxed constraints than SANLPs. That is, we consider the *Fuzzy Array Dataflow Analysis* (FADA) introduced in Collard et al. [1995]. FADA is an enhanced version of EADA and it is used in Stefanov [2004] to analyze Weakly Dynamic Programs (WDP). WDPs are class of affine nested loop programs which may have *if*-conditions dependent on data which is unknown at compile-time and which may change at run-time. In Stefanov [2004], FADA is used for translation of WDPs to equivalent PPNs. Similarly to SANLPs, in WDPs loop bounds have to be affine functions of enclosing loop iterators and static parameters. In this article, we further relax these restrictions by considering sequential affine nested loop programs with dynamic loop bounds. In this section, we introduce FADA because an important part of the solution approach presented in Section 4 is based on this technique.

3.1. Notations

An iteration vector x of a statement is built of iterators of surrounding loops. The set of values of an iteration vector for which a statement is executed represents an iteration domain, denoted by $\mathbf{D}()$. For example, the iteration domain of statement $S2$ in Figure 3(a) is: $\mathbf{D}(S2) = \{1 \leq i \leq N \wedge i \leq j \leq M \wedge j \leq 2\}$. An evaluation of a single statement W on iteration x is called an operation and denoted as $\langle W, x\rangle$. By "$\prec$" we denote ordering of operations. An operation $\langle W, x\rangle$ is evaluated before an operation $\langle R, y\rangle$, that is, $\langle W, x\rangle \prec \langle R, y\rangle$, according to the program sequence if: (1) x lexicographically precedes y; or (2) if $x = y$ and statement W precedes statement R in the program text. As described in Feautrier [1991], order "$\prec$" can be expanded to a system of linear inequalities. With "max", we denote the lexicographical maximum operator. In this article, we use `Dynloop` to designate affine nested loop programs with dynamic loop bounds.

3.2. Exact Array Dataflow Analysis

In this section, we formally describe the EADA algorithm, which is used to perform the dependence analysis on static programs. The goal of the dependence analysis is to

Table I. Examples of System (1) for S1S3 and S2S3 Statements

$\mathbf{Q}_{S1S3}((i_3, j_3))$	$\mathbf{Q}_{S2S3}((i_3, j_3))$	
$1 \le k \le M$	$1 \le i_2 \le N \wedge i_2 \le j_2 \le M \wedge$ $j_2 \le 2$	(c1)
$k = j_3$	$j_2 = j_3$	(c2)
`true`	$\langle S2, (i_2, j_2)\rangle \prec \langle S3, (i_3, j_3)\rangle$	(c3)

determine if evaluation of a statement depends on evaluation of other statements and to find these evaluations. For example, in the SANLP program depicted in Figure 3(a), the purpose of the dependence analysis is to find whether statement $S3$ depends on statements $S1$ or $S2$ via array `y[]` and at which iterations. Or in other words, for every element of array `y[]` read at a given iteration of statement S3, the dependence analysis finds which statement, $S1$ or $S2$, and at which iteration it writes data to the given array element. The result of the analysis forms the dependency relations between iterations of statements writing/reading to/from the array.

Consider two statements W and R, and operations $\langle W, x\rangle$ and $\langle R, y\rangle$, where the first operation writes to an array and the second operation reads from it. The operation $\langle W, x\rangle$ is a source for operation $\langle R, y\rangle$ if it satisfies the system of linear (in)equalities:

$$\begin{aligned}\mathbf{Q}_{WR}(y) = \{x \mid\ & x \in \mathbf{D}(W), && \text{(c1)}\\ & \mathcal{I}_W(x) = \mathcal{I}_R(y), && \text{(c2)}\\ & \langle W, x\rangle \prec \langle R, y\rangle.\} && \text{(c3)}.\end{aligned} \tag{1}$$

The first constraint (c1) states that the source iteration x has to exist, that is, it has to belong to the iteration domain of a W statement. The constraint (c2) specifies that if there is a dependency between two operations, both have to access the same array element. To access an array element, operation $\langle W, x\rangle$ uses an affine indexing function $\mathcal{I}_W()$ and operation $\langle R, y\rangle$ uses an affine indexing function $\mathcal{I}_R()$. The (c3) constraint determines an order of operations, that is, source operation $\langle W, x\rangle$ has to be evaluated *before* operation $\langle R, y\rangle$.

There might be many operations of a single statement satisfying system (1), that is, writing to the same array element. However, we are interested in the last write operation before reading by $\langle R, y\rangle$ from the same element occurs. Therefore, the source operation is the lexicographical maximum between all operations satisfying system $\mathbf{Q}_{WR}(y)$:

$$\mathbf{K}_{WR}(y) = \max\ \ \mathbf{Q}_{WR}(y). \tag{2}$$

The maximum of system (2) can be found using the method presented in Feautrier [1988].

So far, operations of only a single statement have been considered, while there might be several statements $W_1, \ldots, W_m$ writing to the same array element. In this case, we have to consider all pairs $W_1/R, \ldots, W_m/R$. The actual source is the "last" operation between all operations of all statements:

$$\sigma(\langle R, y\rangle) = \max\ \ \{\langle W_k, \mathbf{K}_{W_kR}(y)\rangle \mid k \in [1, m]\}. \tag{3}$$

For example, consider the program in Figure 3(a). There are two statements, S1 and S2 writing to array `y[]` and one statement $S3$ reading from that array. Therefore, we consider two pairs S1S3 and S2S3. For each pair we build the system of linear inequalities (1) as depicted in Table I (see $\mathbf{Q}_{S1S3}((i_3, j_3))$ and $\mathbf{Q}_{S2S3}((i_3, j_3))$). With (i_3, j_3), we denote the iteration vector (i, j) of statement $S3$.

After finding the "max" according to Eq. (3) for the systems shown in Table I, the source operation $\sigma(\langle \mathrm{S3}, (i_3, j_3)\rangle)$ for the data read by statement S3 is:

$$\begin{array}{r l} \textbf{if } j_3 \leq 2 \textbf{ then} & \langle S2, (i_3, j_3)\rangle \\ \textbf{else} & \langle S1, (j_3)\rangle. \end{array} \tag{4}$$

Both branches of the *if*-statement in the solution shown above represent solutions of the parametric integer linear programming (PILP) problems formulated in Table I. The *if*-condition is derived by finding the lexicographical maximum by Eq. (3). Solution (4) can be interpreted as follows: the source of the data for statement $S3$ of the program in Figure 3(a) can be two statements – the source is statement $S1$ when the iterator j of $S3$ is greater than 2, otherwise, the source is statement $S2$.

3.3. Fuzzy Array Dataflow Analysis

In this section, we formally describe the Fuzzy Array Dataflow Analysis (FADA). The FADA algorithm is used to perform dependence analysis on Weakly Dynamic Programs (WDP) which have data-dependent *if*-conditions [Stefanov 2004]. We introduce FADA because it is an important part of the solution we present in Section 4.

Consider two statements W and R of a *weakly dynamic* program. Operation $\langle W, x\rangle$ writes to and operation $\langle R, y\rangle$ reads from the same array. Moreover, let statement W be surrounded by a data-dependent *if*-condition. As a running example, consider Figure 3(b): statements $S2$ and $S3$ are W and R, respectively, and the *if*-condition in line 8 surrounding statement $S2$ is a data-dependent condition.

In Section 3.2, we showed that in order to have two operations $\langle W, x\rangle$ and $\langle R, y\rangle$ of a *static* program dependent, they have to comply to the system of linear inequalities (1). In the same way, to find whether operation $\langle W, x\rangle$ is a source for operation $\langle R, y\rangle$ in a *dynamic* program, we need to build a system of linear inequalities:

$$\begin{array}{r l l} \mathbf{Q}_{WR}(y,\alpha) = \{x \mid & x \in \mathbf{D}(W), x = \alpha, & (c1) \\ & \mathcal{I}_W(x) = \mathcal{I}_R(y), & (c2) \\ & \langle W, x\rangle \prec \langle R, y\rangle.\} & (c3) \end{array} \tag{5}$$

The meaning of constraints ($c2$) and ($c3$) are the same as in system (1): operations should access the same array element and the writing operation should occur before the reading operation. We will explain the meaning of constraint ($c1$). As statement W is surrounded by data-dependent *if*-condition, exact operations of W cannot be determined at compile-time. Thus, for any reading operation $\langle R, y\rangle$, it is impossible to determine the *exact* source operation. The idea of the FADA algorithm is to introduce a parameter which would hide unknown information, that is, a parameter is used to indicate at which iteration a writing operation $\langle W, x\rangle$ may occur. We do not know exactly at which iteration points $x \in \mathbf{D}(W)$ writing to the array occurs, but we assume that this happens for iterations $x = \alpha$, where α is a free parameter vector whose values have to be determined at run-time. Because source operations satisfying system (5) are not exact, we call them *approximated* sources.

Similarly to the EADA algorithm, we are interested in the last write operation before reading by $\langle R, y\rangle$ from the same element occurs, that is, the lexicographical maximum between all operations satisfying system $\mathbf{Q}_{WR}(y,\alpha)$:

$$\mathbf{K}_{WR}(y,\alpha) = \max \mathbf{Q}_{WR}(y,\alpha). \tag{6}$$

Finally, we need to consider all statements $W_1, \ldots, W_m$ writing to the same array element. For each W_k, $k = [1..m]$, we find approximated source. To find the source, we combine all approximated sources as described in Collard et al. [1995]:

$$\sigma(\langle R, y\rangle, \alpha) = \max\{\langle W_k, \mathbf{K}_{W_k R}(y)\rangle \mid\ k \in [1, m]\}. \tag{7}$$

Table II. Examples of System (5) for Statements S1S3 and S2S3

$\mathbf{Q}_{S1S3}((i_3, j_3))$	$\mathbf{Q}_{S2S3}((i_3, j_3), (\alpha_i, \alpha_j))$	
$1 \le k \le M$	$1 \le i_2 \le N \wedge i_2 \le j_2 \le M \wedge$ $i_2 = \alpha_i \wedge j_2 = \alpha_j$	(c1)
$k = j_3$	$j_2 = j_3$	(c2)
`true`	$\langle S2, (i_2, j_2)\rangle \prec \langle S3, (i_3, j_3)\rangle$	(c3)

For example, consider the WDP depicted in Figure 3(b). There are two statements $S1$ and $S2$ writing to array `y[]` and one statement $S3$ which reads from it. For every pair $S1S3$ and $S2S3$, we build the systems of linear inequalities (5) which is depicted in Table II. For pair $S1S3$, all operations of statement $S1$ are known, and thus, a parameter is not introduced, see system $\mathbf{Q}_{S1S3}((i_3, j_3))$ in Table II. However, for pair $S2S3$, see system $\mathbf{Q}_{S2S3}((i_3, j_3), (\alpha_i, \alpha_j))$, we introduce vector of parameters (α_i, α_j) as shown in System (5), because statement $S2$ is surrounded by the dynamic *if*-condition at line 8 in Figure 3(b), and thus, exact operations of $S2$ cannot be determined at compile-time. These parameters are used to designate at which iteration of $S2$ a writing to the array `y[]` may occur. Values of the parameters are determined at run-time.

Approximated sources in S1S3 and S2S3 pairs are found by solving the PILP problems formulated in Table II. The "max" source defined in Eq. (7) is determined by recurrent algorithm of combining direct dependencies described in Section 5.2 of Collard et al. [1995]. Thus, the source operation for statement S3: $\sigma(\langle S3, (i_3, j_3)\rangle, (\alpha_i, \alpha_j))$ is:

$$\begin{aligned} \textbf{if } i_3 \ge \alpha_i \wedge j_3 == \alpha_j \textbf{ then } & \langle S2, (\alpha_i, \alpha_j)\rangle \\ \textbf{else } & \langle S1, (j_3)\rangle. \end{aligned} \tag{8}$$

From Solution (8), we see that, for any read operation $\langle S3, (i_3, j_3)\rangle$, there are two data sources: statements $S1$ or $S2$. When for a given iteration (i_3, j_3) of statement $S3$, at least one of the previous evaluations of the condition at line 8 in Figure 3(b) was `true`, then parameter $\alpha_i \le i_3$ and, parameter $\alpha_j = j_3$, thus, the source is statement $S2$. Otherwise, the source is statement $S1$. In contrast to Solution (4), Solution (8) is approximated, because it depends on parameter vector (α_i, α_j) which value is determined at run-time.

4. SOLUTION APPROACH

In this section, we present the compile-time procedure we have devised for translating affine nested loop programs with dynamic loop bounds (`Dynloop`) into input-output equivalent polyhedral process networks. We have found out that a `Dynloop` program can be formally represented as a *Weakly Dynamic Program* (WDP). Therefore, in the proposed solution approach we can employ the FADA dependence analysis technique, described in Section 3.3.

The procedure for translating `Dynloop` programs to polyhedral process networks consists of 5 steps. First, the initial `Dynloop` program is represented as a WDP. Second, we find all data dependencies in the corresponding WDP program by applying the FADA analysis on it. Recall that the result of the analysis is approximated, that is, it depends on some parameters whose values are determined at run-time. Third, based on the results of the analysis, we create a *dynamic Single Assignment Code* (dSAC) representation of the WDP program. The dSAC was proposed in Stefanov [2004] as an extension of the SAC [Feautrier 1991]. A dSAC program is input-output equivalent to the corresponding WDP and it has the property that every data variable or an array element is written *at most once*. This implies that some variables may not be written at all. We derive a dSAC program using the FADA algorithm, therefore, parameters introduced by FADA are present in the dSAC as well. The values of these parameters in

```
1   parameter N 1 10;

2   for j = 1 to N,
3     for i = 1 to f(...),
S1:     y[ i ] = F1()
5     endfor
6   endfor

S2: [...] = F2( y[5] )
```

Fig. 4. An example of a `Dynloop` program.

dSAC are assigned using control arrays. The generation of the control arrays has been studied in Stefanov [2004], whereas, in this section, we present an extension to this procedure. In the fourth step, the topology of the corresponding PPN is derived, as well as the code executed in each process. In the last fifth step, we compute the buffer sizes of FIFO channels that guarantee a deadlock-free execution of a PPN. In the remaining part of this section, we describe the four steps in more detail. Computing buffer sizes (step 5) is out of the scope of this article. We illustrate the proposed solution approach using the example shown in Figure 4.

Step 1 *(Dynloop-to-WDP).* Consider the `Dynloop` program in Figure 4. In this program, the upper bound of the *for*-loop at line 3 is determined by an arbitrary function $f(\cdots)$. The upper bound of the inner loop i may change at every iteration of the outer loop j but cannot be changed on iterations of i. More importantly, the values of the upper bound are unknown at compile-time as they are determined at run-time by $f()$.

In order to be able to apply our solution approach, we assume that the range of the values that function $f()$ may have is finite. This is particularly true for all programs that execute in finite memory, that is, the programs we are interested in.

Then, without altering the functionality, we modify the initial `Dynloop` program to the program shown in Figure 5(a). Such modification is general and applicable to any `Dynloop` program. First, we substitute the upper bound of the loop at line 3 in Figure 4 with a constant equal to the maximum value of $f()$, denoted by `max_f`, see line 4 in Figure 5(a). For example, for the program in Figure 5(a), in order for the 5th element of array `y[]` to be read at line 10, the value of `max_f` should be greater than 5. We will use `max_f` ≥ 5 in the rest of this article.

In general, the value of `max_f` can be determined in four different ways:

(1) provided by the application/program developers (e.g., by using `pragmas` in the code);
(2) calculated by analyzing the arrays' capacity and indexing functions;
(3) deduced by studying the ranges of function $f()$;
(4) by taking the maximum of the data type used to declare the loop iterator.

For example, consider method (2). Assume that the capacity of the array `y[]` is 100 elements. Then, by taking into account the array indexing function at line 4 in Figure 4 and that the program is correct, we can calculate that the maximum value of iterator i and, consequently the `max_f` equals to 100.

Second, we introduce an array `X[]` used to capture the values of the dynamic upper bound at run-time. That is, the elements of `X[]` are written by function $f()$ at line 3 in Figure 5(a), just before the *for*-loop. The same array elements are used in evaluating the *if*-condition at line 5 in Figure 5(a), which preserves the original program behavior. This newly created program belongs to the class of the *weakly dynamic* programs. Since the loop bounds of the program in Figure 5(a) are fixed and known at compile-time, we can apply the FADA algorithm on this program to perform dependence analysis.

```
1  parameter N 1 10;

2  for j = 1 to N,
3    X[j] = f(...)
4    for i = 1 to  max_f,
5      if i <=  X[j],
S1:      y[i] = F1()
7      endif
8    endfor
9  endfor

S2:[] = F2( y[5] )
```

(a) Newly created WDP program

```
1  parameter N 1 10;

2  for j = 1 to N,
3    X[j] = f(...)
4    for i = 1 to max_f,
5      if i <= X[j],
S1:       y_1[j,i] = F1();
7      endif
8    endfor
9   endfor

10  if c1 <= N && c2 == 5,
11    in_0 = y_1[c1,c2]
12  else
13    in_0 = 0
14  endif
S2:[] = F2( in_0 )
```

(b) Initial dSAC

Fig. 5. A WDP program equivalent to the Dynloop program in Figure 4 and its corresponding dSAC.

Table III. An Example of System (5) for S1S2 Pair

$\mathbf{Q}_{S1S2}((\alpha_j, \alpha_i))$	
$1 \leq j_1 \leq N \wedge 1 \leq i_1 \leq \texttt{max\_f}$	(c1)
$j_1 = \alpha_j \wedge i_1 = \alpha_i$	
$i_1 = 5$	(c2)
true	(c3)

The formal description of the FADA algorithm has been given in Section 3.3. In the following section, we demonstrate only the application of FADA on our running example.

Step 2 *(FADA analysis).* The WDP program in Figure 5(a) has two statements $S1$ and $S2$ which communicate through array y[]. According to FADA, for pair $S1S2$, we build the system of linear inequalities shown in Table III which corresponds to Eq. (5). Constraint $c1$ in Table III describes all possible source iterations of statement $S1$, i.e., its iteration domain. Parameters (α_j, α_i) store the iteration point (j_1, i_1) of statement $S1$ where writing to array y[] may occur.

After solving the PILP problem defined in Table III, the approximated source operation defined in Eq. (7) for statement $S2$: $\sigma(\langle S2, ()\rangle, (\alpha_j, \alpha_i))$ is:

$$\begin{array}{l}\textbf{if}\ \ N \geq \alpha_j \wedge 5 == \alpha_i\ \ \textbf{then}\ \ \langle S1, (\alpha_j, \alpha_i)\rangle \\ \qquad\qquad\qquad\qquad\quad \textbf{else}\ \ \perp . \end{array} \tag{9}$$

From Solution (9), we see that for read operation $\langle S2, ()\rangle$ there is one data source. If, for at least one iteration $(j_1, 5)$ of statement $S1$, the condition at line 5 in Figure 5(a) is evaluated to true, then the source is statement $S1$. Otherwise, the source for y[5] is undefined and statement $S2$ will use the initial value of y[5]. For the sake of brevity, the initialization of array y[] is omitted in the example.

Figure 6 illustrates Solution 9 graphically. In this figure, the iteration domain (j, i) of statement S1 listed at line 6 in Figure 5(a) is shown. It is assumed that $N = 10$ and $\texttt{max\_f} = 10$. Black dots represent the iterations when statement S1 is executed at run-time, that is, the *if*-condition at line 5 evaluated to true. The vector of parameters (α_j, α_i) points at the last operation of the source statement $\langle S1, (j_1, i_1)\rangle$ which will be needed by the read operation $\langle S2, ()\rangle$. For the example in Figure 6, the last writing to y[5] occurred when $j = 8$ and $i = 5$. Therefore, $(\alpha_j, \alpha_i) = (8, 5)$.

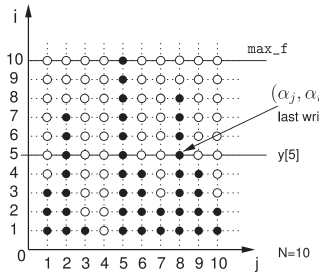


Fig. 6. Representation of Solution 9 for a given execution of the program in Figure 5(a).

Initial dSAC. The solution provided by FADA is used to modify the WDP program in order to capture the identified dependencies in an explicit way. The result for our running example is shown in Figure 5(b), which is in a dynamic single-assignment-code (dSAC) form. The dSAC is an extension of the SAC [Feautrier 1991]. In contrast to SAC where every variable is written exactly once, in dSAC every variable is written *at most once*. This implies that some of the variables may not be written at all.

Based on Solution (9), we modify the WDP in Figure 5(a) and generate the dSAC in Figure 5(b) by inserting the code lines 10-14 shown in Figure 5(b). This code is needed to implement array element accesses such that the dependencies identified by FADA are respected. For example, the *if*-condition at line 10 implements Solution (9). Recall that when the *if*-condition evaluates to true, then the source of the data is statement $S1$. This is captured at line 11. Otherwise, statement $S2$ will use the initial value of `y[5]`. Assume that in our example, `y[5]` has been initialized to zero. Therefore, at line 13, the input argument for statement $S2$ has been set to zero as well.

Recall that to deal with a dynamic *if*-condition, for every pair of statements the FADA algorithm introduces vector of parameters that corresponds to the iteration vector. In our example, there are two parameters (see line 10 in Figure 5(b)) which are reflected in the following way. Parameter $c1$ corresponds to α_j. It is related to iterator j and may have values $c1 \in [1..\mathtt{N}]$. Parameter $c2$ corresponds to α_i. It is related to iterator i and may have values $c2 \in [1..\mathtt{max\_f}]$. The meaning of the parameter values in this program is to indicate the last iteration of j when function $F1()$ has been executed at the fifth iteration of i. The values of parameters $c1$ and $c2$ are unknown at compile-time. They are determined at run-time, during the execution of the program. Therefore, we need a mechanism to generate and propagate the values at run-time in a way that keeps the correct program behavior.

Step 3 *(Control Arrays).* In order to keep the functionality of the dSAC equivalent to the functionality of the initial WDP, we introduce *local* and *global* control arrays that are used to initialize and propagate values of parameters at run-time.

Local Control Arrays. A local control array is added for the set of parameters introduced by FADA and is used to store values of the set of parameters for every iteration. We illustrate the idea of local control arrays on the example in Figure 6.

Figure 6 depicts the iteration domain (j, i) of statement S1 shown at line 6 in Figure 5(a). Black dots are iterations when statement S1 is executed at run-time, that is, the *if*-condition at line 5 evaluated to `true`. Parameters introduced by FADA in the previous step happen to take up the values of iteration vectors when the last writing needed by a read operation occurred. It is not possible to determine such iterations at compile-time. Therefore, we use a local control array to store the values of all iterations when a source statement is executed (black dots).

```
1  parameter N 1 10;

2  for j = 1 to N,
3    X[j] = f()
4    for i = 1 to max_f,
5      if i <= X[j],
6        y_1[j,i] = F1()
7        lcl_c1c2[i] = (j,i)
8      endif
9    endfor
10 endfor

11  (c1,c2) = lcl_c1c2[5]
12 if c1 <= N && c2 == 5,
13   in_0 = y_1[c1,c2]
14 else
15   in_0 = 0
16 endif
17 [] = F2( in_0 )
```

(a) Initial dSAC shown in Figure 5(b) with local control array

```
1  parameter N 1 10;

2  for j = 1 to N,
3    X[j] = f()
4    for i = 1 to max_f,
5      if i <= X[j],
6        y_1[j,i] = F1()
7        lcl_c1c2[i] = (j,i)
8      endif
S1:      ctrl_c1c2[i] = lcl_c1c2[i]
10   endfor
11 endfor

S2: (c1,c2) = ctrl_c1c2[5]
13 if c1 <= N && c2 == 5,
14   in_0 = y_1[c1,c2]
15 else
16   in_0 = 0
17 endif
18 [] = F2( in_0 )
```

(b) Modified dSAC code with new global control array

```
1  parameter N 1 10;

2  for j = 1 to N,
3    X[j] = f()
4    for i = 1 to max_f,
5      if i <= X[j],
6        y_1[j,i] = F1()
7        lcl_c1c2[i] = (j,i)
8      endif
9      ctrl_c1c2_1[j,i] = lcl_c1c2[i]
10   endfor
11 endfor

12  (c1,c2) = ctrl_c1c2_1[N, 5]
13 if c1 <= N && c2 == 5,
14   in_0 = y_1[c1,c2]
15 else
16   in_0 = 0
17 endif
18 [] = F2( in_0 )
```

(c) Final dSAC

Fig. 7. Examples of the initial dSAC with a local control array, the modified dSAC with a global control array, and the final dSAC.

For our example in Figure 5(b), a new local control array of vectors `lcl_c1c2[]` is introduced to the program as shown in Figure 7(a). The components of each vector correspond to parameters $c1$ and $c2$ derived by the FADA analysis of pair S1S2. We use the original index function used with the data variable `y`, that is, `y[i]`, to perform the access to the local control arrays, that is, `lcl_c1c2[i]`. In order to distinguish iterations where parameters values have been stored, the elements of the control arrays must be initialized with values that are greater than the maximum value of the corresponding parameters. Recall that, for our example, parameter $c1 \in [1..\mathtt{N}]$ and $c2 \in [1..\mathtt{max\_f}]$. Therefore, the corresponding local control array is initialized as follows:

$$\forall \mathtt{i} : 1 \leq \mathtt{i} \leq \mathtt{max\_f} : \mathtt{lcl\_c1c2[i]} = (\mathtt{N}+1, \mathtt{max\_f}+1). \quad (10)$$

For the sake of brevity, this initialization is not shown in Figure 7(a). Writing to the local control array is performed just after function $F1()$, see line 7 in Figure 7(a). This guarantees that when the function is executed, the current iteration vector is stored in the control array.

The values of the local control array are propagated and assigned to the parameters $c1$ and $c2$ at line 11. These parameters are used to evaluate the conditions at line 12 which determine the source of the data for function $F2()$. With the introduction of the local control array to the program shown in Figure 7(a), this program is input-output equivalent to the program in Figure 5(a).

Global Control Arrays. Unfortunately, introducing *local* control arrays to the dSAC code violates the property that "every variable is written *at most once*". For example, local control array `lcl_c1c2[i]` that initializes parameters $c1$ and $c2$ at line 11 in Figure 7(a) is not in a single assignment form, that is, elements of `lcl_c1c2[i]` may be written more than once (see line 7). Therefore, the program in Figure 7(a) is not in a dSAC form. In order to be able to create a process network, as discussed later in Step 4, and most importantly, to create the FIFO channels used for transferring data, the corresponding data variables/arrays must be in a single assignment form. A novel contribution of our work is a procedure that extends the control arrays generation described in the previous section. Our extension solves the problem that the local control array in Figure 7(a) is not in the single assignment form. Here, we explain how such control array is transformed into a single assignment form.

Table IV. ILP System (5) for the Control Arrays at Lines 9 and 12

$\mathbf{Q}_{S1S2}()$	
$1 \leq j_1 \leq N \wedge 1 \leq i_1 \leq \texttt{max\_f}$	(c1)
$i_1 = 5$	(c2)
`true`	(c3)

In order to represent the program in Figure 7(a) as dSAC, we need to identify the relation between writing to and reading from the control array. Thus, we need to perform dataflow analysis for the local control array, where the writings to the control array occur inside a block surrounded by a dynamic *if*-condition. We achieve this in the following way. While keeping the same functionality, we modify the program by introducing and additional *global* control array (`ctrl_c1c2[]`) *outside* the block surrounded by the dynamic *if*-condition, see lines 9 and 12 in Figure 7(b). This program is input-output equivalent to the program in Figure 7(a). The new control array is written (line 9) at every iteration of the *for*-loops and takes the same values as the local control array `lcl_c1c2[]`. Consequently, we can perform the *static* exact array dataflow analysis (presented in Section 3.2) on control array `ctrl_c1c2[]`. We can always do this, because the introduced new array is not surrounded by the dynamic *if*-condition.

For EADA analysis, we need to build a system of linear inequalities as it has been shown in Section 3.2. The system for pair $S1S2$ at lines 9 and 12 from Figure 7(b) is built in Table IV. Recall, that `max_f` is a scalar and, in this example, we assume that `max_f` $\geq$ 5. After finding the maximum of the system according to Eq. (3), the final solution and the source operation is $\langle S1, (N, 5)\rangle$.

Based on this solution, we replace the original one-dimensional array `ctrl_c1c2[]`, see lines 9 and 12 in Figure 7(b), with two-dimensional array `ctrl_c1c2_1[]` shown at lines 9 and 12 in Figure 7(c). The program in Figure 7(c) is in a dSAC form because the new global control array `ctrl_c1c2_1[]` used to initialize parameters $c1$ and $c2$ is in a single assignment form. This dSAC is the final input-output equivalent representation of our running example which is the `Dynloop` program in Figure 4. We use this final dSAC to generate a process network which is explained in the next section.

Step 4 (PPN Generation). In this step of our solution approach, we describe how the processes and FIFO channels are created from the corresponding final dSAC program.

Recall that a PPN consists of autonomous processes that communicate data in a point-to-point fashion over bounded FIFO channels. A process of a PPN consists of a target *function*, *input ports*, *output ports*, and *control*. The target function specifies how data tokens from input streams are transformed to data tokens to output streams. The input and output ports are used to connect a process to FIFO channels. Data read from the input ports is used to initialize the function arguments. Data produced as a result of the function execution is written to the output ports. The control specifies how many times the function is executed, which input ports to read and which output ports to write every time the function is executed. The control of a process can be compactly represented mathematically (using the polytope model [Feautrier 1996]) in terms of linearly bounded sets of iteration vectors.

The procedure for PPN generation from the final dSAC consists of three substeps. First, based on the final dSAC representation of a `Dynloop` program derived in the previous step, the topology of the PPN is created. The topology is created by instantiating processes and communication channels. Second, the internal code structure of each process is derived from the final dSAC specification. It is important to note, that in this substep, the created communication channels are not FIFOs but multidimensional arrays. Third, the multidimensional arrays that are used for data communication

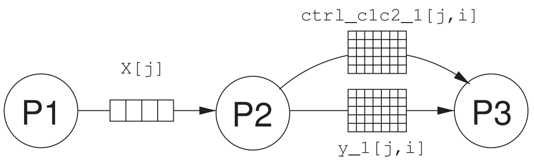
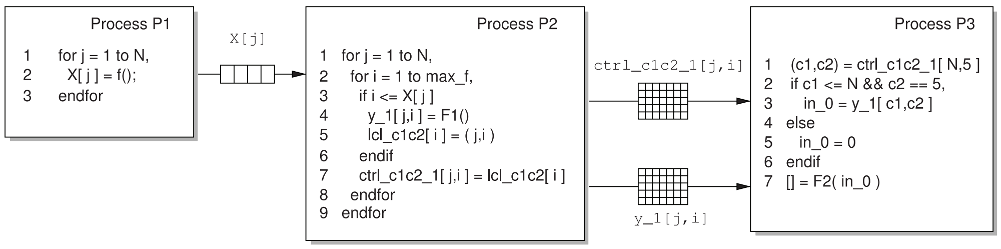


Fig. 8. The topology of the PPN derived from the dSAC in Figure 7(c).



Process P1

```
for j = 1 to N,
  X[ j ] = f();
endfor
```

Process P2

```
for j = 1 to N,
  for i = 1 to max_f,
    if i <= X[ j ]
      y_1[ j,i ] = F1()
      lcl_c1c2[ i ] = ( j,i )
    endif
    ctrl_c1c2_1[ j,i ] = lcl_c1c2[ i ]
  endfor
endfor
```

Process P3

```
(c1,c2) = ctrl_c1c2_1[ N,5 ]
if c1 <= N && c2 == 5,
   in_0 = y_1[ c1,c2 ]
else
   in_0 = 0
endif
[] = F2( in_0 )
```

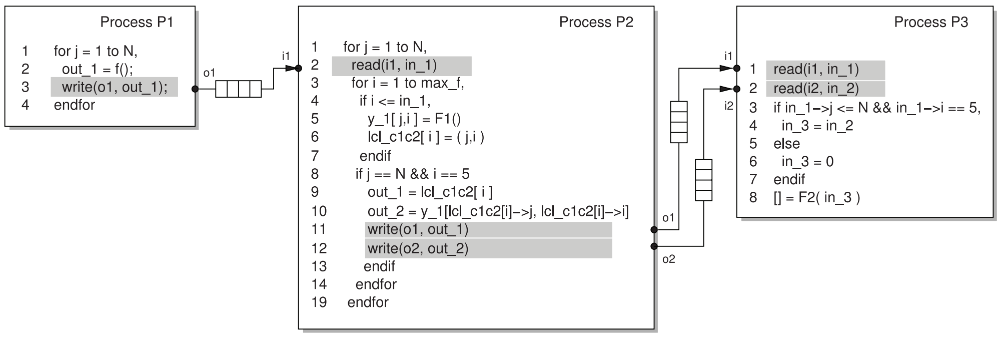
Fig. 9. The internal code structure of each process in the PPN derived from the dSAC in Figure 7(c).

between function statements in the final dSAC are replaced by FIFO channels. In other words, we replace the multidimensional array accesses in the code of each process with a read/write primitives to implement synchronization through blocking read/write on FIFO channels. Here, we explain the three substeps in more detail using the final dSAC in Figure 7(c).

Topology Creation of a PPN (Substep 1). The PPN corresponding to the dSAC in Figure 7(c) is shown in Figure 8. This PPN consists of three processes and three communication channels. We explain how these processes and communication channels are created. In our approach, a process is created for every function statement in the dSAC program. Therefore, the PPN in Figure 8 has three processes: process $P1$ corresponds to function $f()$ at line 3 in Figure 7(c), process $P2$ corresponds to function $F1()$ at line 6, and process $P3$ corresponds to $F2()$ at line 18 in the same figure. The three communication channels correspond to arrays which are in a single assignment form in the dSAC in Figure 7(c). These arrays are: one-dimensional array `X[j]` at line 3 and 5 in Figure 7(c), two-dimensional data array `y_1[j,i]` at lines 6 and 14, and one two-dimensional control array `ctrl_c1c2_1[j,i]` at lines 9 and 12 in the same figure. Recall that array `X[j]` is in a single assignment form because of the way we introduced this array in Step 1 of our solution approach. Array `y_1[j,i]` is the single assignment form of array `y[i]` derived by applying the FADA analysis on the WDP program in Figure 5(a) as described in Step 2 of our solution approach. The control array `ctrl_c1c2_1[j,i]` is introduced and transformed in a single assignment form in Step 3 of our solution approach. In the following substep, we describe how the internal code structure of each process is created.

Internal Code Structure Generation (Substep 2). Consider Figure 9 where the internal code structures of processes $P1$, $P2$, and $P3$ of the PPN in Figure 8 are shown. Here, we explain how these code structures are derived from the corresponding dSAC specification depicted in Figure 7(c).

Every process of a PPN executes a sequential nested loop program. The internal code structure of each process is formed by code pieces of the dSAC program. The iteration domain of a process is the iteration domain of a statement in the dSAC program. For example, the iteration domain of process $P2$ is formed by the iteration domain of function $F1$ defined by lines 2, 4, and 5 in Figure 7(c). Additionally, the lines of dSAC where data and control are read and written are added to the process code. For



```
Process P1
1   for j = 1 to N,
2     out_1 = f();
3     write(o1, out_1);
4   endfor
```

```
Process P2
1   for j = 1 to N,
2     read(i1, in_1)
3     for i = 1 to max_f,
4       if i <= in_1,
5         y_1[ j,i ] = F1()
6         lcl_c1c2[ i ] = ( j,i )
7       endif
8       if j == N && i == 5
9         out_1 = lcl_c1c2[ i ]
10        out_2 = y_1[lcl_c1c2[i]–>j, lcl_c1c2[i]–>i]
11        write(o1, out_1)
12        write(o2, out_2)
13      endif
14    endfor
19  endfor
```

```
Process P3
1   read(i1, in_1)
2   read(i2, in_2)
3   if in_1–>j <= N && in_1–>i == 5,
4     in_3 = in_2
5   else
6     in_3 = 0
7   endif
8   [] = F2( in_3 )
```

Fig. 10. The final PPN derived from the program in Figure 4.

example, lines 6–11 are added to the internal code structure of process $P2$ shown in Figure 9. Similarly, the internal code structure of processes P1 and P3 are formed by lines 2–3 and 12–18, respectively, from the dSAC shown in Figure 7(c).

Linearization (Substep 3). At this point, the processes of the PPN communicate data via multidimensional arrays. In this substep, we explain how the multidimensional arrays are replaced with FIFO channels. This process is called *linearization*.

In the PPN depicted in Figure 9, processes are connected with communication channels which are the multidimensional arrays used in the dSAC shown in Figure 7(c). However, as explained in Section 1, the processes in a PPN communicate via FIFOs and synchronize using a blocking read/write on an empty/full FIFO channel, that is, an execution of a process is suspended if it tries to read from an empty FIFO channel, or tries to write to a full channel, respectively. Therefore, in order to generate a PPN, the multidimensional array accesses have to be replaced with corresponding *write* and *read* operations on FIFO channels.

To implement the linearization, we adapted the approaches proposed in Turjan et al. [2002] and Nadezhkin and Stefanov [2010]. In these works, the communication characteristics are identified when exchanging data between pair of statements. Based on this information, the multidimensional array accesses are replaced with one-dimensional FIFO accesses. The result of the linearization applied on the multidimensional arrays in Figure 9 is shown in Figure 10. In each process, the multidimensional arrays accesses are substituted by read/write primitives from/to FIFO channels. Internally, these read/write primitives realize the blocking synchronization between processes. For example, writing to the global control array at line 7 of process $P2$ in Figure 9 is substituted by writing to the FIFO at line 11 in process $P2$ in Figure 10.

The communication read/write primitives access the FIFO channels through ports. That is, every process has a set of input ports and a set of output ports connected to FIFO channels. For example, process $P2$ reads from a single channel via port $i1$ at line 2 and writes data to 2 channels via ports $o1$ and $o2$ at lines 11 and 12, respectively. Additionally, we apply the iteration domain reconstruction of ports described in Turjan [2007] to avoid transferring more data tokens than needed. For details about domain reconstruction, we refer to Turjan [2007].

5. OVERHEAD ANALYSIS

In this section, we discuss the overhead in the generated process networks, which results from the proposed approach for systematic parallelization of sequential programs with dynamic loop bounds. There are two types of overhead in the generated process networks, that is, memory and execution time overhead. The memory overhead is due to the introduced control arrays, as well as, the created dataflow and control FIFO channels. It highly depends on the characteristics of the application being parallelized, see Section 6 and Table VI. Therefore, it is very difficult to be analyzed systematically. However, we can systematically analyze the execution time overhead which is introduced by the approach we propose in this article. This overhead is caused by the execution of some "dummy" iterations not present in the initial sequential program. Here, we discuss this overhead in details. Recall that in our approach, we substitute a dynamic upper loop bound with the maximum value (max_f) that the bound may have during the execution of the program. Then, at run-time, the actual number of iterations at which a function executes is determined by the behavior of the application and the current value of the dynamic loop bound. This means that if the actual number of executions (x) is smaller than the maximum number, then the corresponding process performs ($max\_f - x$) "dummy" iterations. The overhead we consider is the time spent in performing these iterations.

It is important to note that it is difficult to determine the exact amount of the overhead because it depends on values which are determined and change at run-time. Here, we define the overhead and determine how it varies for particular range of its terms. Assume that max_f is the maximum value of a dynamic loop bound and x represents the actual number of iterations in which a process executes its associated function. When a function executes, it takes W_x time units. Performing a "dummy" iteration takes W time units, respectively. This is the time spent in one iteration but not executing the corresponding function. Then, for any given values of max_f, x, W_x, and W, the total execution time (T_{ex}) is:

$$T_{ex} = x(W_x + W) + (max\_f - x)W, \tag{11}$$

where $x(W_x + W)$ is the time spent on real computation (T_{real}) and $(max\_f - x)W$ is the extra time spent performing "dummy" iterations. Consequently, we can compute the introduced execution overhead as follows:

$$\frac{T_{ex}}{T_{real}} = \frac{x(W_x + W) + (max\_f - x)W}{x(W_x + W)} = 1 + \frac{(max\_f - x)W}{x(W_x + W)},$$

where the percentage of the execution overhead ($Ovhd$) is:

$$Ovhd = \frac{(max\_f - x)W}{x(W_x + W)} \cdot 100 = \frac{(max\_f - x)}{x} \cdot \frac{W}{(W_x + W)} \cdot 100\ [\%]. \tag{12}$$

Equation (12) shows that the overhead depends on two ratios. The first one, $\frac{(max\_f - x)}{x}$, depends on (i) the application characteristics, which determine max_f, and (ii) the execution behavior, which determines the values of x at run-time. The second ratio is related to the computation performed by a process (executed on a particular processor) as it represents the ratio between the time to perform a "dummy" iteration and the time spent on actual computing. Figure 11 illustrates the amount of overhead for the following ranges of the two ratios in Eq. (12):

(1) $0 \leq \frac{max\_f - x}{x} \leq 2 \Rightarrow$ for any value of max_f, $\frac{max\_f}{3} \leq x \leq max\_f$;
(2) $0.01 \leq \frac{W}{W_x + W} \leq 0.5 \Rightarrow$ for any value of W, $W \leq W_x \leq 99 \cdot W$.

These ranges capture the characteristics for a wide spectrum of applications and their behavior. Moreover, our experience shows that if a particular application has sufficient inherited parallelism, then the approach we propose to parallelize sequential

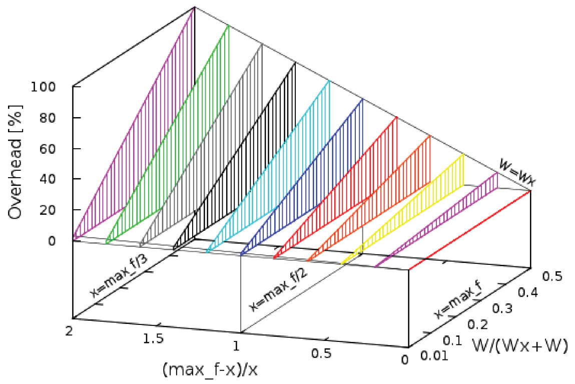


Fig. 11. The amount of introduced overhead.

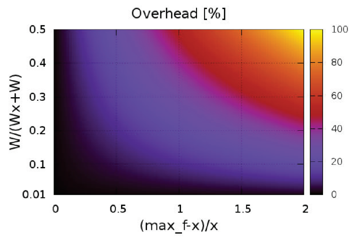


Fig. 12. Overhead's color map.

programs with dynamic loop bounds can lead to performance speed-up if the two ratios stay within those specified ranges.

In case $x = max\_f$, there is no overhead (see the right part of Figure 11) because there are no "dummy" iterations to be executed ($max\_f - x = 0$). Then, by decreasing the value of x, the overhead increases. The rate of the increase is determined also by the value of $W/(W_x + W)$. The values of this ratio used in the figure capture functions with low and high workload. The lowest workload we consider is $W_x = W$, that is, the time to execute the corresponding function is equal to the time of a "dummy" iteration (see the back plane of the figure). We use such a low workload to illustrate some extreme values of the overhead. For example, when $W_x = W$ and $x = max\_f/2$ the maximum overhead is 50%. The combined effect of both ratios leads to 100% overhead when $W_x = W$ and $x = max\_f/3$, see the left part of Figure 11. In contrast, functions with high workload, that is, $50 \cdot W \leq W_x \leq 99 \cdot W$, lead to very low overhead. For example, even if $x = max\_f/3$, the introduced overhead is around 5–10% as it can be observed at the bottom-left part in the figure. This indicates that the approach we propose is not sensitive to functions with high workloads.

For easier evaluation of the overhead values, we plot the percentage overhead as a color map in Figure 12. From this figure, it is seen that overhead above 35% is present only in 1/4 of the cases. In addition, 1/16 of the cases, see the area with overhead ≥80%, correspond to functions with very low workload and a large number of 'dummy' iterations. For the other 3/4 of the cases, we would like to emphasize on the following two areas. First, if the ratio $(max\_f - x)/x$ is smaller than 0.25, then the granularity of the executed functions does not affect the overhead, which is below 10%, see the dark vertical strip on the left part of the figure. This indicates also that for lightweight functions, the overhead will be small if the executed iterations are close to the max_f value. Similarly, in case of functions with high workload ($50 \cdot W \leq W_x \leq 99 \cdot W$), the number of "dummy" iterations that are executed does not affect the overhead, which again is below 10%—see the dark horizontal strip at the bottom of Figure 12. The second area covers almost half of the plot, see the arc-shape stripe in the middle of Figure 12. This area shows that even with a large variety of the values of both ratios, the overhead is kept below 35%, which is relatively low. This area also shows that such overhead can be achieved even if one of the ratios goes to its extreme value. For example, 35% overhead is achieved if $W/(W_x + W)$ reaches its maximum value of 0.5 and $(max\_f - x)/x = 0.7$.

```
for k = 1 to Targets,
  [Height,Width] = getLSODTarget()
  X_j[k] = Height
  X_i[k] = Width
  for j = 0 to maxHeight + 1,
    for i = 0 to maxWidth + 1,
      if (j <= X_j[k] + 1 && i <= X_i[k] + 1)
        img_1[k,j,i] = readTarget(X,Y)
        lcl_1[j,i] = (j,i)
      endif
      ctrl_1[k,j,i] = lcl_1[j,i]
    endfor
  endfor
  for j = 1 to maxHeight,
    for i = 1 to maxWidth,
      (c11,c12) = ctrl_1[k,j-1,i-1]
      if (c11 == j-1 && c12 == i-1)
        in_0 = img_1[k,j,i]
      endif
      (c21,c22) = ctrl_1[k,j-1,i+1]
      if (c21 == j-1 && c22 == i+1)
        in_1 = img_1[k,j,i]
      endif
      (c31,c32) = ctrl_1[k,j,i-1]
      if (c31 == j && c32 == i-1)
        in_1 = img_1[k,j,i]
      endif
      (c41,c42) = ctrl_1[k,j,i+1]
      if (c41 == j && c42 == i+1)
        in_1 = img_1[k,j,i]
      endif
      (c51,c52) = ctrl_1[k,j+1,i-1]
      if (c51 == j+1 && c52 == i-1)
        in_1 = img_1[k,j,i]
      endif
      (c61,c62) = ctrl_1[k,j+1,i+1]
      if (c61 == j+1 && c62 == i+1)
        in_5 = img_1[k,j,i]
      endif
      if (j <= X_j[k] && i <= X_i[k])
        img_out_1[k,j,i] = edgeDetection(
                               in_0, in_1,
                               in_2, in_3,
                               in_4, in_5)
        img_out_2[k,j,i] = absVal( img_out_1[k,j,i] )
        lcl_2[j,i] = (j,i)
      endif
      ctrl_2[k,j,i] = lcl_2[j,i]
    endfor
  endfor
  for j = 1 to maxHeight,
    for i = 1 to maxWidth,
      (c71,c72) = ctrl_2[k,j,i]
      if (j == c71 && i == c72)
        in_0 = img_out_2[k,j,i]
      endif
      if (j <= X_j[k] && i <= X_i[k])
        if( j >= 1 )
          in_1 = vsum[i]
        else
          in_1 = 0
        endif
        vsum[i] = vertSum( in_1, in_0 )
      endif
    endfor
  endfor
endfor
```

Fig. 13. Final dSAC.

6. EXPERIMENT AND RESULTS

In this section, we demonstrate our solution approach presented in Section 4 applied on the motivating example, that is, the LSOD application depicted in Figure 2(a). We present some of the results we have obtained by implementing and executing the derived parallel PPN specification of the LSOD application on a shared memory multiprocessor system. The main objective of this experiment is to show the practical applicability of our approach on a real-life application and to show the benefits of our parallelization approach. For the implementation, we derive the PPN specification of the LSOD application following the approach presented in Section 4. Then, we use the ESPAM [Nikolov et al. 2008] tool and the HDPC [Farago 2009] back-end library to generate C++ code for the processes and the FIFO channels.

The result of Steps 1 to 3 of our solution approach applied on the LSOD program in Figure 2(a) is illustrated in Figure 13. It is the final dSAC specification. We use the final dSAC to derive the PPN topology, shown in Figure 14, as well as to derive the internal code structure of all processes. Examples of the internal code structures of processes `readTarget` and `edgeDetection` are depicted in Figure 15. Here, we emphasize on some of the important moments of the PPN derivation, we describe the PPN topology, show how code for processes is generated, and comment on the overhead introduced in the generated PPN.

According to Step 1 of our solution approach we substitute the dynamic upper bound functions with constants equal to the maximum values these functions can have. The initial LSOD program in Figure 2(a) has three loop nests with dynamic upper bound functions: `Height+1`, `Width+1`, `Height` and `Width` at lines 3–4, 8–9 and 14–15, respectively. These functions take their maximum when variables `Height` and `Width` are maximum, that is, equal to some constants `maxHeight` and `maxWidth`, respectively. In the LSOD program, the maximum values of `Height` and `Width` are the maximum

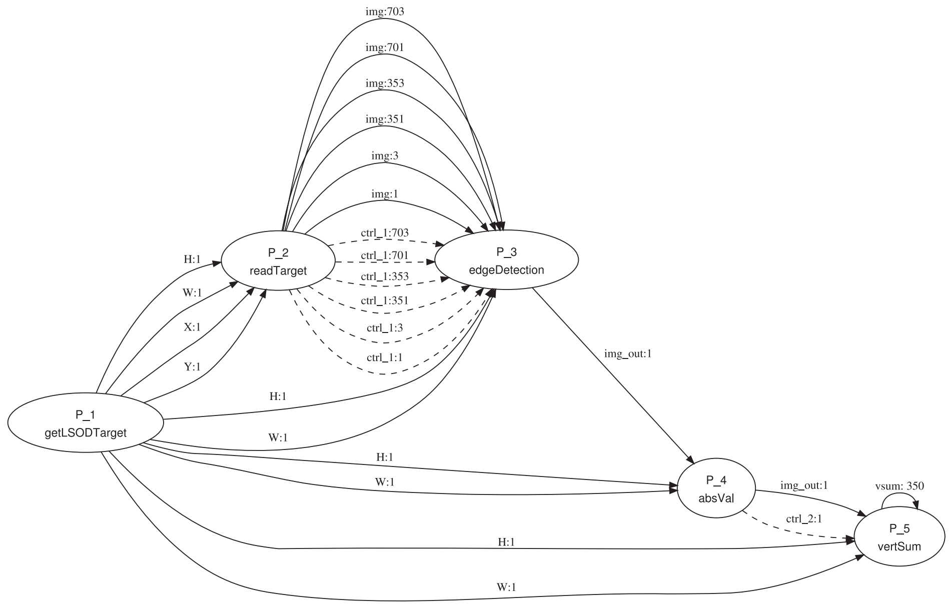


Fig. 14. The PPN derived from the LSOD program.

dimensions that a target may have. Therefore, we substitute the dynamic upper bound functions with constants equal to the maximum of these functions as depicted in Figure 13 at lines 5–6, 14–15 and 48–49.

The result of applying the FADA analysis which constitutes Step 2 of our solution approach is illustrated at lines 17–19, 21–23, 25–27, 29–31, 33-35, 37–39 and 51-53 in Figure 13. In total, 6 two-dimensional vectors of parameters $(c11, c12), \ldots, (c61, c62)$ were introduced by the FADA algorithm analyzing the data-dependencies between functions `readTarget()` and `edgeDetection()` shown in Figure 2(a). Also, one two-dimensional vector of parameters $(c71, c72)$ was introduced after analyzing the data-dependencies between functions `absVal()` and `vertSum()`.

At Step 3 of our solution approach, we introduce local and global control arrays in order to initialize and propagate the values of the parameters at run-time. For the pair of functions `readTarget()` and `edgeDetection()`, 6 vectors of parameters were introduced by FADA. All these parameter vectors correspond to the single iteration vector (j_2, i_2) of the source function `readTarget()`. Therefore, at lines 9 and 11 only one local and one global control arrays are generated for this pair of functions. Similarly, at lines 43 and 45 one local and one global control arrays are added for the pair of functions `absVal()` and `vertSum()`.

By applying Step 4 of our approach to the final dSAC of the LSOD application depicted in Figure 13, we derive the topology of the PPN depicted in Figure 14. The topology consists of 5 processes, 19 *data channels* shown as solid lines that are used to exchange data between processes and 7 *control channels* shown as dashed lines used to propagate values of global control arrays. The edges of the PPN in Figure 14 are annotated with the buffer sizes calculated for the LSOD program considering maximum dimensions of the targets to be 350x300 pixels. This means that we set `maxWidth` $= 350$ and `maxHeight` $= 300$.

Finally, as an example of the internal structure of the PPN processes, in Figure 15, we present the pseudocode for `readTarget` and `edgeDetection` processes derived after the linearization step. These processes exhibit the most intensive data communication

```
for k = 1 to Targets,
  for j = 0 to maxHeight + 1,
    for i = 0 to maxWidth + 1,
      if (j == 0 && i == 0)
        read(0, H)
        read(1, W)
        read(2, X)
        read(3, Y)
      endif
      if (j <= H + 1 && i <= W + 1)
        img_1 = readTarget(X,Y)
        lcl_1 = (j,i)
        if( j <= H-1 && i <= W-1 )
          write( 1, img_1 )
        if( j <= H-1 && i >= 2 )
          write( 3, img_1 )
      ...
      endif
      ctrl_1 = lcl_1
      if( j <= H-1 && i <= W-1 )
        write( 0, ctrl_1 )
      if( j <= H-1  && i >= 2 )
        write( 1, ctrl_1 )
      ...
    endfor
  endfor
endfor
```

(a) Process `readTarget`

```
for k = 1 to Targets,
  for j = 1 to maxHeight,
    for i = 1 to maxWidth,
      if (j == 0 && i == 0)
        read(0, H)
        read(1, W)
      endif
      if (j <= H && i <= W)
        read(0, ctrl_1)
        if (ctrl_1.j == j-1 && ctrl_1.i == i-1)
          read(1, in_0)
        read(2, ctrl_1)
        if (ctrl_1.j == j-1 && ctrl_1.i == i+1)
          read(3, in_1)
        ...
        img_out = edgeDetection(
                    in_0, in_1,
                    in_2, in_3,
                    in_4, in_5)
        write(0, img_out)
     endif
    endfor
  endfor
endfor
```

(b) Process `edgeDetection`

Fig. 15. Internal code structures of processes `readTarget` and `edgeDetection` of the PPN derived from the LSOD application.

in the PPN. The internal code structures of these processes are generated as it has been explained in Step 4 of our solution approach. Note, that the input/output ports used to access FIFO channels (see the read/write primitives in Figure 15) are automatically mapped to physical addresses generated by the Espam tool (in a separate header file).

Performance Evaluation of the LSOD PPN. We evaluate our approach by executing the parallel LSOD application specification on an `Intel®` `Xeon®` quad-core machine running a Linux operating system. We used the ESPAM [Nikolov et al. 2008] tool and the HDPC [Farago 2009] library to map the processes of the generated PPN onto cores and to generate C++ code for these cores. We used the GCC compiler to generate the final binary code. The HDPC library employs the `boost-thread` framework that enables the use of multithreaded implementations. That is, the derived PPN has been translated to a multithreaded program realizing the LSOD application, in which every process of the PPN is a separate thread.

In this experiment, we implemented and executed the parallel PPN specification of the LSOD application considering 5 different mapping configurations. The obtained results are shown in Figure 16. The horizontal axis depicts the number of cores used in each configuration, that is, we mapped the 5 processes of the PPN onto 1, 2, 3, 4, and 5 cores, respectively. Note that because the `Intel®` `Xeon®` processors support hyper-threading, the operating systems “sees” 8 different cores. Therefore, we could map 5 processes on 5 different cores exploiting maximum concurrency. Obviously, when using less than 5 cores, some processes have to share the same core. In this case, we let the operating system to schedule the execution of these processes (i.e., threads) on a particular core. We experimented with grouping different processes together, that is, mapping several processes on a single core. Figure 16 presents the best results that we have obtained. In addition, every configuration has been executed multiple times and the bars show the obtained average speed-up. The first configuration (see the leftmost bar in Figure 16) represents our reference configuration, in which, we mapped all 5 processes of the PPN onto a single core. We consider the speed-up of

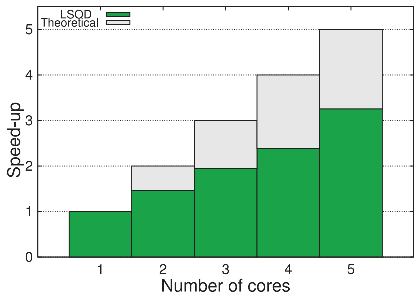


Fig. 16. Evaluation of LSOD PPN on several CPUs.

Table V. Statistics LSOD PPN

Process	P_1	P_2	P_3	P_4	P_5	PPN
$W/(W_x + W)$	0.53	0.38	0.26	0.47	0.3	0.39
$(max\_f - x)/x$	0	1.09	1.09	1.09	1.09	0.87

this configuration to be 1. Also, we have normalized the performance of the other configurations with respect to the performance of this reference configuration. Looking at the performance of the other 4 configurations, we see that by increasing the number of cores, the speed-up increases below the theoretical maximum shown as gray bars in Figure 16. We found that because of the data dependencies in the LSOD application and the imbalanced workload of the functions executed by different processes, the theoretical maximum cannot be achieved. In addition, in all configurations except the one using 5 cores (see the rightmost bar in Figure 16), there is an overhead introduced by the operating system for scheduling different threads on one core. As a result, the rightmost configuration exhibits a slightly larger speed-up increase compared to the other configurations. Finally, there is the execution time overhead caused by the extra "dummy" iterations, which we discussed in Section 5. Here, we present some numbers with regards to this execution time overhead, as well as, the memory overhead in the LSOD process network.

Execution Time Overhead. From the execution statistics obtained by profiling of the LSOD application and its PPN, we computed the two ratios in Eq. (12). Table V shows the ratios for each process and the whole PPN. Note that for computing $(max\_f - x)/x$, we need to consider that the targets are 2-dimensional. Therefore, we used the term:

$$\frac{maxWidth \cdot maxHeight - x \cdot y}{x \cdot y} = \frac{350 \cdot 300 - x \cdot y}{x \cdot y}.$$

The terms x and y represent the average target size, which we computed from the targets used when executing the program. Based on the computed values in the last column of Table V and applying Eq. (12), the overhead due to the execution of "dummy" iterations of the LSOD PPN is 33.93%.

Memory Overhead. In order to evaluate the memory overhead, we measured the memory requirements for the sequential LSOD program and compared this with the memory requirements for executing the corresponding PPN. The sequential program requires in total 13650 Bytes of memory, which includes both the code and the data. In order to make a fair comparison, it is important to note that this number (13650 Bytes) does not include the data array used to buffer the largest possible target, that is, variable `img[TH][TW]` which requires $350 \times 300 = 105000$ Bytes. Although, we use such a variable in the sequential program, the program can be written more efficiently

Table VI. Memory overhead of the LSOD PPN

Process	P_1	P_2	P_3	P_4	P_5
Code (bytes)	1626	2302	2510	1742	1978
Data (bytes)	1420	1360	1360	1360	1360
Total (bytes)	3046	3662	3870	3102	3338

	PPN	Sequential
Memory (bytes)	17018	13650
FIFOs (bytes)	18384	–
Overhead	2.6x	–

in a way that we do not need to buffer the whole (largest possible) target. The left part of Table VI shows the memory requirements for every process in the generated process network. In addition, we need to consider also the memory used to implement the FIFO channels. In total, the PPN requires 17018 Bytes for implementing the processes and 18384 Bytes for the FIFO channels, see the right part of Figure VI. Then, if we compare these numbers with the number of the sequential program, we see that the memory overhead is 2.6x, which is reasonable provided that this is the memory requirement for the implementation of 5 processes and 26 FIFO channels.

Overall, the average efficiency of the 4 parallel implementations of the LSOD process network is around 70%. The efficiency (*Eff*) is defined as:

$$Eff = \frac{SP}{C},$$

where SP is the speed-up and C represent the number of cores used to achieve this speed-up. The obtained results clearly indicate that the approach we presented in this article facilitates efficient parallel implementations of sequential nested loop programs with dynamic loop bounds. That is, our approach reveals the possible parallelism available in such applications, which allows for the utilization of multiple cores in an efficient way.

7. CONCLUSIONS

In this article, we presented a first approach for automated translation of affine nested loops programs with dynamic loop bounds (`Dynloop`) into input-output equivalent polyhedral process networks (PPN). This approach can be implemented efficiently in a compiler that will help to reduce significantly the time for parallelizing sequential programs. The approach presented in this article includes only basic techniques that have to be applied in order to derive a PPN automatically from a `Dynloop` program. The obtained results of the case study with the LSOD application indicate that the approach we presented in this article facilitates efficient parallel implementations of `Dynloop` programs. That is, our approach reveals the parallelism available in such applications, which allows for the utilization of multiple cores in an efficient way.

REFERENCES

ARULAMPALAM, S. AND MASKELL, S. 2002. A tutorial of partical filter for on-line non-linear/non-Gaussian Bayesian tracking. *IEEE Trans. Sig. Process.* 68–73.

BENABDERRAHMANE, M.-W., POUCHET, L.-N., COHEN, A., AND BASTOUL, C. 2010. The polyhedral model is more widely applicable than you think. In *Proceedings of ETAPS CC'10*.

CASTRILLON, J., ET AL. 2010. Trace-based KPN composability analysis for mapping simultaneous applications to MPsoc platforms. In *Proceedings of DATE'10*.

COLLARD, J.-F., BARTHOU, D., AND FEAUTRIER, P. 1995. Fuzzy array dataflow analysis. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, 92–101.

DE KOCK, E. 2002. Multiprocessor mapping of process networks: A JPEG decoding case study. In *Proceedings of the 15th International Symposium on System Synthesis (ISSS'02)*, 68–73.

DWIVEDI, B., ET AL. 2004. Automatic synthesis of system on chip multiprocessor architectures for process networks. In *Proceedings of the CODES+ISSS*.

FARAGO, T. 2009. A framework for heterogeneous desktop parallel computing. M.S. thesis, LERC, LIACS.

Feautrier, P. 1988. Parametric integer programming. *RAIRO Recherche Opérationnelle 22,* 3, 243–268.

Feautrier, P. 1991. Dataflow analysis of scalar and array references. *Para. Prog. 20,* 1, 23–53.

Feautrier, P. 1996. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*. Lecture Notes in Computer Science, vol. 1132, 79–103.

Geigl, M., Griebl, M., and Lengauer, C. 1999. Termination detection in parallel loop nests with while loops. *Paral. Comput. 25,* 12, 1489–1510.

Goossens K., et al. 2003. Guaranteeing the quality of services in networks on chip. In *Networks on Chip*. Kluwer Publishers, 61–82.

Griebl, M. and Lengauer, C. 1996. The loop parallelizer loopo. In *Proceedings of the 6th Workshop on Compilers for Parallel Computers*, vol. 21. Forschungszentrum, 311–320.

Haid, W., et al. 2009. Efficient execution of Kahn process networks on multi-processor systems using protothreads and windowed FIFOS. In *Proceedings of ESTIMedia*. IEEE, 35–44.

Kahn, G. 1974. The Semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 74*. North-Holland Publishing Co.

Knobe, K. and Sarkar, V. 1998. Array SSA form and its use in Parallelization. In *Proceedings of the ACM Symposium on Principles of Programming Languages (PoPL)*. CA, 107–120.

Martin, G. 2006. Overview of the MPSoC design challenge. In *Proceedings of DAC*.

Mihal, A. and Keutzer, K. 2003. Mapping concurrent applications onto architectural platforms. In *Networks on Chips*, A. Jantsch and H. Tenhunen, Eds., Kluwer Academic Publishers, 39–59.

Nadezhkin, D. and Stefanov, T. 2010. Identifying communication models in process networks derived from weakly dynamic programs. In *Proceedings of SAMOS X*. 372–379.

Nikolov, H., Stefanov, T., and Deprettere, E. F. 2008. Systematic and automated multiprocessor system design, programming, and implementation. *IEEE Trans. CAD 27,* 3, 542–555.

Raman, E., Ottoni, G., Raman, A., Bridges, M. J., and August, D. I. 2008. Parallel-stage decoupled software pipelining. In *Proceedings of the 6th CGO*, 114–123.

Stefanov, T. 2004. Converting weakly dynamic programs to equivalent process network specifications. Ph.D. thesis. Leiden University, The Netherlands, ISBN: 90-9018629-8.

Stefanov T., et al. 2004. System design using Kahn process networks: The Compaan/Laura approach. In *Proceedings of DATE*. 340–345.

Turjan, A. 2007. Compiling nested loop programs to process networks. Ph.D. thesis. Leiden University, The Netherlands.

Turjan, A., Kienhuis, B., and Deprettere, E. 2002. Realizations of the extended linearization model in the Compaan tool chain. In *Proceedings of the 2nd Samos Workshop*.

Turjan, A., Kienhuis, B., and Deprettere, E. 2004. Translating affine nested-loop programs to process networks. In *Proceedings of CASES'04*, DC.

Verdoolaege, S., Nikolov, H., and Stefanov, T. 2007. PN: A tool for improved derivation of process networks. *EURASIP J. Embed. Syst. 2007,* 1, 19–19.

Received August 2011; revised February 2012; accepted June 2012