Compiler Support for Parallel Code Generation through Kernel Recognition

Manuel Arenaz, Juan Touriño, and Ramón Doallo
Department of Electronics and Systems, University of A Coruña, Spain
{arenaz,juan,doallo}@udc.es

Abstract

The automatic parallelization of loops that contain complex computations is still a challenge for current parallelizing compilers. The main limitations are related to the analysis of expressions that contain subscripted subscripts, and the analysis of conditional statements that introduce complex control flows at run-time. We use the term complex loop to designate loops with such characteristics. In this paper, we focus on the generation of parallel code for sequential complex loop nests using a generic compiler framework (proposed in an earlier paper [3]) that accomplishes kernel recognition through the analysis of the Gated Single Assignment program representation. Specifically, we present an extension of this framework that enables its use as a powerful tool for gathering source code information that is relevant for the parallelization of each computational kernel. A set of example codes are analyzed in detail to illustrate the potential of our approach. Experimental results using a benchmark suite of complex loop nests are also presented.

1. Introduction

The automatic parallelization of sequential codes requires knowledge about how to reorder the execution of the statements of the code while preserving the sequential semantics. Current parallelizing compilers are mainly based on dependence analysis [13, 18], which provides not only information about what code sections (usually loops) can be executed in parallel, but also useful information for the generation of efficient parallel code. Classical dependence analysis was shown to be effective for the parallelization of regular loop nests, which contain array references whose subscript expressions can be rewritten as affine or linear functions of the index variables of the enclosing loops.

The classical approach mentioned above often fails to parallelize complex loop nests because, in general, they rest on information-gathering techniques that cannot extract the necessary source code information at compile-time. The main sources of uncertainty are the presence of array references with subscripted subscripts (i.e. the subscripts expressions contain array references) and complex control constructs. The parallelization of complex loops has been addressed through the design of efficient parallelizing transformations that exploit the characteristics of well-known computational kernels [2, 6, 7, 9, 11, 20]. In general, these techniques are described assuming that all the necessary information is available to the compiler. However, it is difficult to perform this complex task efficiently.

In this paper, the automatic generation of parallel code for complex loops is addressed using the generic compiler infrastructure proposed in [3]. The recognition of computational kernels is carried out by means of two classification algorithms that perform an exhaustive analysis of the code. This analysis mainly consists of searching the expressions that compose the statements of the code for occurrences of variables that introduce loop-carried dependences. We present extensions of these algorithms that enable the use of the infrastructure as a powerful information-gathering tool that provides efficient support for the generation of parallel code. The detailed description of the compiler framework can be found in [3]. In this paper we present the parts of the framework that are needed to generate parallel code for a collection of loop nests extracted from real codes.

The paper is organized as follows. Section 2 gives an overview of both the kernel recognition framework and the subsequent generation of parallel code. Concepts and terms that will be used throughout the paper are also introduced. Section 3 presents some case studies to explain our extensions of the framework. These examples focus on complex loops that contain computational kernels frequently found in real codes, namely, irregular assignment and consecutively written array. Section 4 presents experimental results that show the efficacy of our approach. Finally, Section 5 discusses related work, and Section 6 concludes the paper.

2. Framework Overview

2.1. Kernel Recognition

The compiler framework proposed in [3] consists of an extensible generic infrastructure for the recognition of a

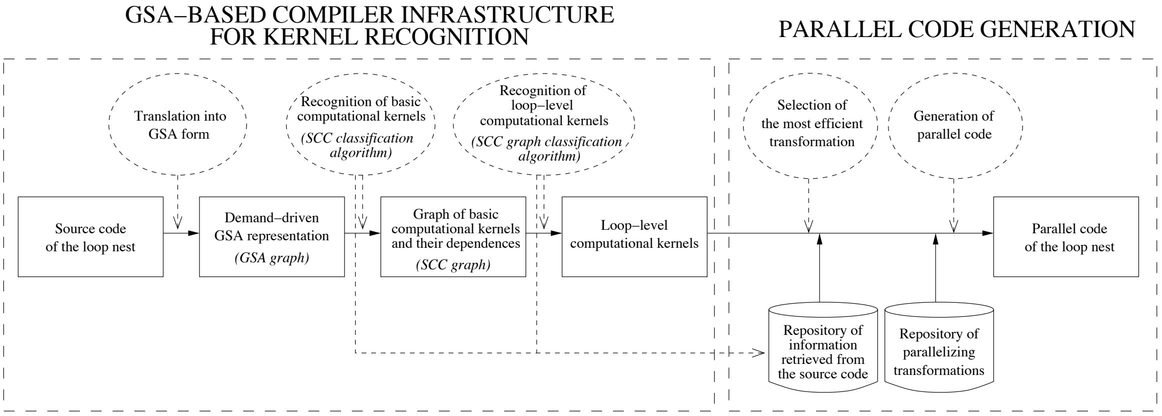

GSA–BASED COMPILER INFRASTRUCTURE
FOR KERNEL RECOGNITION
PARALLEL CODE GENERATION
Translation into GSA form
Recognition of basic computational kernels (SCC classification algorithm)
Recognition of loop–level computational kernels (SCC graph classification algorithm)
Selection of the most efficient transformation
Generation of parallel code
Source code of the loop nest
Demand–driven GSA representation (GSA graph)
Graph of basic computational kernels and their dependences (SCC graph)
Loop–level computational kernels
Parallel code of the loop nest
Repository of information retrieved from the source code
Repository of parallelizing transformations

Figure 1. Block diagram of the parallelization framework.

wide variety of computational kernels that are frequently found both in regular and complex loop nests. The internals of the framework are shown in Fig. 1. Next, the different stages, depicted as dashed ovals in the figure, are described.

The first stage is the translation of the source code into a demand-driven implementation of the Gated Single Assignment (GSA) form [17]. GSA is an extension of the well-known Static Single Assignment (SSA) form that captures the flow of values of scalar and array variables, even in loops with complex control flow. This task is accomplished by inserting a set of special operators, ϕ, right after the points of the program where the control flow merges, and by renaming the variables of the program so that they are assigned unique names in the definition statements. Three types of ϕ's are distinguished in GSA: $\mu(x_{out}, x_{in})$, which appears at loop headers and selects the initial x_{out} and loop-carried x_{in} values of a variable; $\gamma(c, x_{true}, x_{false})$, which is located at the confluence node associated with a branch and captures the condition c for each definition to reach the confluence node: x_{true} or x_{false}, if c is true/false; and $\alpha(a_{prev}, s, rhs)$, which replaces the right-hand side of an array assignment statement $a(s) = rhs$, and represents that the s-th entry of a is assigned the value rhs while the other entries take the values of the previous definition of the array, denoted as a_{prev}. The GSA intermediate representation has some properties that ease the development of tools for automatic program analysis: the elimination of false dependences for scalar and array definitions (not for array element references), the syntactical representation of reaching definition information, and the capture of the conditional expressions that determine the control flow of the program.

The second and third stages address the recognition of the kernels computed in the loop nest at two different levels. In the second stage, the strongly connected components (SCCs) that appear in the data dependence graph of the GSA form (GSA graph from now on) are analyzed. This intra-SCC analysis consists of a recursive algorithm that determines the class of kernel computed during the execution of the statements of the SCC (SCC class from now on), for instance, irregular assignment, conditional or non-conditional induction variable, etc. A detailed description of the kernels can be found in [1]. The SCC classification algorithm reduces the computation of a SCC class to classifying the statements that compose the SCC. For each statement, a post-order traversal of the corresponding syntax tree is performed. The nodes of the tree represent operators. The children of a node correspond to the arguments of the operator. At each node, the class of the operator is calculated by a *transfer function* that merges the classes associated with the operators of the child nodes. In [1] we present a detailed description of transfer functions for the most common operators, for instance, sum (T_+), product (T_*), scalar reference (T_y) or array reference ($T_{x(s)}$). These transfer functions check whether the statements of a SCC fulfill the characteristics of one of the computational kernels recognized by the framework. The SCC classification algorithm provides the compiler with the set of basic kernels calculated in the SCCs of the loop, and with the data and control dependences between these basic kernels. This information is summarized in a data structure called the *SCC use-def chain graph* (SCC graph from now on).

The third stage recognizes more complex kernels that result from a combination of a set of basic kernels, for instance, a consecutively written array, a minimum/maximum with location kernel, etc. This inter-SCC analysis is carried out by a classification algorithm that processes the SCC graph. The SCC graph is an intermediate program representation that exhibits the minimal set of properties (scenarios) that characterize the computation of a loop-level kernel. The algorithm uses these scenarios as a guide for the execution of additional tests (see for instance [10, 19]) that enable the recognition of loop-level kernels.

2.2. Generation of Parallel Code through Kernel Recognition

In the literature a great variety of parallelizing transformations targeted for specific computational kernels have been proposed. In the scope of irregular codes, well-known examples are irregular reductions [6, 7, 10, 21], irregular assignments [2, 9], DOACROSS loops [11, 20], and other classes of kernels [10]. As shown in Fig. 1, our approach rests on a repository of such parallelizing transformations, including new techniques we have developed. In addition, a second repository stores the information retrieved from the source code during the execution of the SCC classification algorithm and the SCC graph classification algorithm.

Once the set of loop-level kernels computed in the loop nest have been recognized, the most efficient code transformations are selected from the repository in order to maximize the performance of the parallel code. The selection criteria should consider not only the characteristics of the target parallel architecture (shared/distributed memory, interconnection network...), but also the parameters of the application (e.g. sparsity, degree of contention..., defined in [21] for irregular reductions). If there is no technique for a given kernel in the repository, then a generic approach could be applied. An example technique based on the speculative parallel execution of irregular loops is proposed in [14]. First, the code is executed in parallel and, later, a fully parallel data dependence test is applied to determine if it had any cross-iteration dependence; if the test fails, the code is reexecuted serially. Generic methods can be applied to any loop with complex computations. However, efficiency usually drops with regard to code transformations that are tuned for the efficient execution of a specific kernel on a given target architecture.

After the selection stage, the generation of parallel code is carried out. If the loop contains a set of independent computational kernels, loop fission is applied and each kernel is parallelized according to the corresponding code transformation. If there are dependences between the kernels, the compiler analyzes such dependences and derives a set of constraints that the parallel code must fulfill in order to preserve the sequential semantics. These constraints will usually impose a specific mapping of loop iterations to processors, and will result in additional pre-processing and/or post-processing stages in the parallel code.

3. Compiler Support to Parallelize Complex Loops: Case Studies

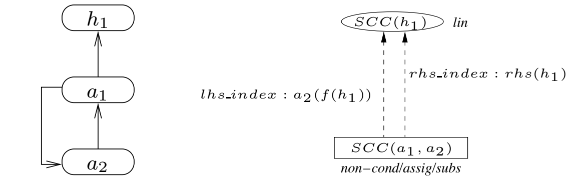
In this section, the potential of our extended compiler framework is shown by analyzing several complex loop nests that contain a set of computational kernels that are frequently found in real codes. Sections 3.1 and 3.2 describe the support needed to implement two techniques for the parallelization of irregular assignments. An illustrative simple example is used for this purpose. Section 3.3 focuses on two loop nests, extracted from the library of sparse matrix operations *SparsKit-II* [15], that contain different variants of the consecutively written array kernel.

```
a(...) = ...
DO h = 1, f_size
   a(f(h)) = rhs(h)
END DO
... = ...a(...)...
```

(a) Source code.

```
a_0(...) = ...
DO h_1 = 1, f_size, 1
   a_1 = μ(a_0, a_2)
   a_2 = α(a_1, f(h_1), rhs(h_1))
END DO ... = ...a_1(...)...
```

(b) GSA form.



(c) Data dependence GSA graph.

(d) SCC use-def chain graph.

Figure 2. Irregular assignment computations.

3.1. Irregular Assignments: Inspector-Executor

An *irregular assignment* (see Fig. 2(a)) consists of a loop where, at each iteration h, the array entry $a(f(h))$ is assigned a value denoted as $rhs(h)$, f being the subscript array. The expression $rhs(h)$ does not contain occurrences of a, thus the code is free of loop-carried true data dependences. Nevertheless, as the subscript expression $f(h)$ is loop-variant, loop-carried output data dependences may be present at run-time (unless f is a permutation array).

We proposed in [2] a strategy for the parallelization of irregular assignments using the inspector-executor model. The key idea consists of mapping loop iterations to processors so that each processor carries out conflict-free computations that exploit data write locality and preserve load-balancing. The implementation of the inspector-executor approach in a parallelizing compiler requires the extraction of the following information from the source code at compile-time: the array of results (a in Fig. 2(a)), the subscript array that defines the write access pattern (f), the most efficient location for the insertion of the inspector code, the size of the arrays a and f (a_{size} and f_{size}), and the number of processors (P). This information is summarized in Table 1. For each parallelizing transformation applicable to a computational kernel, the mechanism needed to retrieve each piece of information is shown, namely, the com-

Table 1. Information-gathering requirements for the generation of parallel code.

Computational kernel	Parallelizing transformation	Relevant information for the generation of parallel code	Gathering mechanism		
			Fwk	Ext	Other
Irregular Assignment	Inspector-executor (*see Section 3.1*)	a	$\surd$		
		Arrays of the access pattern		$\surd$	
		Location of the inspector code		$\surd$	
		P, a_{size}, f_{size}			$\surd$
	Array expansion (*see Section 3.2*)	a, α-statements	$\surd$		
		P, a_{size}			$\surd$
		Mapping of sequential iterations			$\surd$
		Mapping of array entries			$\surd$
Irregular Reduction	Inspector-executor	a	$\surd$		
		Arrays of the access pattern		$\surd$	
		Location of the inspector code		$\surd$	
		P, a_{size}, f_{size}			$\surd$
	Array expansion	a	$\surd$		
		P, a_{size}			$\surd$
		Mapping of sequential iterations			$\surd$
		Mapping of array entries			$\surd$
Semantic reduction	Parallel reduction	Reduction variable	$\surd$		
		P			$\surd$
		Mapping of sequential iterations			$\surd$
Consecutively written array	Splitting and merging (*see Section 3.3.1*)	a	$\surd$		
		P			$\surd$
		a_{size}			$\surd$
	DOALL loop with run-time test (*see Section 3.3.2*)	a	$\surd$		
		Ψ, Δ		$\surd$	
		P			$\surd$

piler framework (*Fwk*), the extensions proposed in this paper (*Ext*), or other mechanism (*Other*) such as user-supplied parameters, compiler directives or lexical/syntactical analysis. Within our framework, the array of results a is retrieved straightforwardly as the unique source code variable represented by a SCC that enables the recognition of the irregular assignment kernel (see Section 3.1.1 for more details). The parameters a_{size}, f_{size} and P are obtained by means of the lexical/syntactical analysis of the source code, user supplied parameters, or default values determined by the compiler. The remaining pieces of information need extensions of the framework and, thus, are the focus of the following sections. Section 3.1.1 describes the recognition of the irregular assignment of Fig. 2(a) and explains our extensions to gather the array variable f that defines the write access pattern. Section 3.1.2 outlines an algorithm to determine the location of the inspector code.

3.1.1 Array Variables of the Write Access Pattern

As shown in Fig. 1, the first step of the framework is the translation of the loop do_h into the GSA form of Fig. 2(b). In this step special operators, μ and α, that capture the run-time flow of values of the array variable a are inserted in the code. Furthermore, each definition of a is assigned a unique name (a_1 and a_2) in order to represent reaching definition information syntactically.

The second step is the construction of the SCC graph of do_h through the execution of the SCC classification algorithm. The data dependence graph of the GSA form is depicted in Fig. 2(c). The nodes and the edges represent statements and use-def chains between statements, respectively. The nodes are labeled with the left-hand side symbol of the statement in the GSA form. For the sake of clarity, the use-def chains whose target is a definition located outside the loop (a_0, f and rhs) are not depicted. The GSA graph contains two SCCs: $SCC(h_1)$, which consists of $h_1 = 1, f_{size}, 1$ and represents the computation of the linear index variable h; and $SCC(a_1, a_2)$, which is composed of $a_1 = \mu(a_0, a_2)$ and $a_2 = \alpha(a_1, f(h_1), rhs(h_1))$ and captures the computation of the array variable a. As a result, the SCC graph of Fig. 2(d) contains two nodes labeled as $SCC(h_1)$ (the oval) and $SCC(a_1, a_2)$ (the rectangle). The class of kernel computed at run-time during the execution of the statements of each SCC is printed next to the corresponding node. Thus, the SCC class of $SCC(h_1)$

is *lin*, which means that the loop index variable h is a linear induction variable. The notation *non-cond/assig/subs* corresponding to the class of $SCC(a_1, a_2)$ is as follows: *non-cond* indicates that the execution of $a(f(h)) = rhs(h)$ does not depend on any condition; *assig* is the abbreviation for assignment operation (in contrast to reduction and recurrence operation); finally, *subs* captures the loop-variant nature of the left-hand side subscript expression $f(h)$ of the statement $a(f(h)) = rhs(h)$. The interpretation of the SCC graph is completed with the description of the edges, which represent the use-def chains between statements of different SCCs. The labels show the expression that contains the occurrence of the variable defined in the target SCC, as well as the location of the expression within the statement of the source SCC: left-hand side subscript (*lhs_index*) or right-hand side subscript expression (*rhs_index*). The relevance of this information for kernel recognition will be pointed out throughout the paper.

The SCC classification algorithm performs a post-order traversal of the syntax trees that represent the statements of $SCC(h_1)$ and $SCC(a_1, a_2)$. Let us focus on the subtree that represents the left-hand side subscript expression $f(h_1)$ of $a_2 = \alpha(a_1, f(h_1), rhs(h_1))$. The goal is to calculate the class $[f(h_1)]^{a_2(f(h_1))}_{\blacktriangleleft:1,1,a_2(f(h_1))}$. The notation $[e]^{e_{ref}}_{\beta:l,sl,E}$ is as follows: e is the expression target for classification; e_{ref} is the left-hand side expression of the statement being analyzed; E is the left-hand side, the right-hand side or the conditional expression of the statement; l is the level of e within the tree representation of E (see the concept *level of an expression* [18, Chapter 3]); *sl* is the indirection level of e within E; and β indicates the position of E within the statement where it is included: the left-hand side (denoted as ◀), the right-hand side (▶), or the conditional expression of an if–endif statement (denoted as ?). The value of l and *sl* is initialized to zero and later incremented as the post-order traversal advances.

The root node of the subtree of $f(h_1)$ corresponds to an array reference whose child nodes are a reference to the array f and a reference to the scalar h_1. First, the array f is classified as an invariant expression (i.e. $[f]^{a_2(f(h_1))}_{\blacktriangleleft:2,1,a_2(f(h_1))}$=*inv*) as its value is not modified during the execution of do_{h_1}. Second, the subscript expression h_1 is classified as a linear expression ($[h_1]^{a_2(f(h_1))}_{\blacktriangleleft:2,2,a_2(f(h_1))}$=*lin*) because, in each iteration, it takes the values of the linear induction variable h_1 represented by $SCC(h_1)$. Finally, the transfer function of array references, $T_{x(s)}$, is applied:

$$[x(s)]^{y(r)}_{p:l,sl,E} = \begin{cases} \textit{unk} & \text{if } x \neq y,\ [s]^{y(r)}_{p:(l+1),(sl+1),E}{=}\textit{unk} \\ \textit{subs} & \text{if } x \neq y,\ l > 0,\ [x]^{y(r)}_{p:(l+1),sl,E}{=}\textit{inv}, \\ & \text{and } [s]^{y(r)}_{p:(l+1),(sl+1),E}{=}\textit{lin} \\ ... & \end{cases} \tag{1}$$

where *unk* denotes an unrecognized computational kernel. According to the second entry of Eq. (1), the compiler concludes that $[f(h_1)]^{a_2(f(h_1))}_{\blacktriangleleft:1,1,a_2(f(h_1))}$=*subs*.

Other functionalities can be incorporated in the transfer functions in order to widen the scope of application of the framework, for instance, symbolic analysis for the computation of the closed form expression of an induction variable (see [5]). Next, we extend $T_{x(s)}$ so that the array f is retrieved from the source code, and stored in the corresponding repository (see Fig. 1). First, $[x(s)]^{y(r)}_{p:l,sl,E}$ is computed by Eq. (1). After that $T_{x(s)}$ applies the following rule:

$$\text{retrieve } x \qquad \text{if } [x(s)]^{y(r)}_{p:l,sl,E}{=}\textit{subs},\ p = \blacktriangleleft,\ sl \geq 1 \tag{2}$$

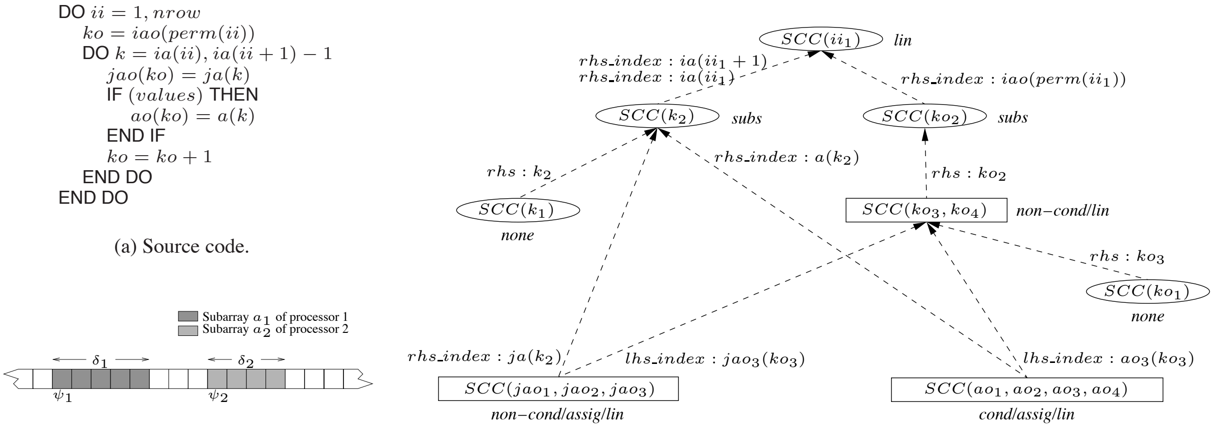
In our case study, $[f(h_1)]^{a_2(f(h_1))}_{\blacktriangleleft:1,1,a_2(f(h_1))}$=*subs*, the position p =◀ and the subscript level $sl = 1$, which indicates that $f(h_1)$ is a loop-variant subscript expression that appears in the left-hand side of the statement $a_2 = \alpha(a_1, f(h_1), rhs(h_1))$. Thus, the compiler concludes that the array variable f defines the write access pattern of the irregular assignment. Note that the condition $sl \geq 1$ also enables the identification of the set of arrays involved in array references with multiple indirection levels (e.g. f and g in the statement $a(f(g(h))) = rhs(h)$).

3.1.2 Location of the Inspector Code

The performance of the parallelizing techniques based on the inspector-executor model usually depends on the reuse of the inspector throughout the execution of the program. Therefore, the point of the program where the inspector code is inserted determines the performance. Next, we briefly outline an algorithm that takes advantage of the demand-driven implementation of the GSA form. Let $v_1, ..., v_n$ be the set of variables that define the pattern of indirect write operations ($v_1, ..., v_n$ are extracted as explained in Section 3.1.1). Let B_{DO} be the basic block of the control flow graph (CFG) that contains the header of the loop. Let $B_1, ..., B_n$ represent the basic blocks that contain the definition statements of $v_1, ..., v_n$. The most appropriate location for the inspector code is the first basic block that is a successor of $B_1, ..., B_n$ in the CFG, and that dominates B_{DO}. The demand-driven implementation of the GSA form provides an efficient solution to the statement-level reaching definition problem. Thus, the identification of the $B_1, ..., B_n$ from $v_1, ..., v_n$ is straightforward. Next, the target basic block is determined through the analysis of the *dominance tree* [12], which is constructed during the translation of the source code into GSA form.

3.2. Irregular Assignments: Array Expansion

A different parallelization strategy based on array expansion is described in [9]. Each processor computes the ir-



```
DO $ii = 1, nrow$
   $ko = iao(perm(ii))$
   DO $k = ia(ii), ia(ii+1)-1$
      $jao(ko) = ja(k)$
      IF $(values)$ THEN
         $ao(ko) = a(k)$
      END IF
      $ko = ko + 1$
   END DO
END DO
```

(a) Source code.

(b) Write access pattern of arrays jao and ao.

(c) SCC graph.

Figure 3. Permutation of the rows of a sparse matrix.

regular assignment corresponding to a set of loop iterations preserving the order of the sequential execution. The partial results are stored in expanded arrays, $a(1:a_{size}, 1:P)$ and $@a(1:a_{size}, 1:P)$, that allow distinct processors to write in different memory locations concurrently. The @-array stores the last loop iteration at which the elements of a were modified. Next, the processors are synchronized. Finally, the processors apply a reduction operation that obtains the partial results with highest iteration numbers.

The implementation of the array expansion approach requires to extract the array variable that stores the result (a), the source code statements that perform write operations on that array ($a(f(h)) = rhs(h)$), the size of the array (a_{size}), the number of processors (P), and the mapping of computations to processors to calculate the partial results and to carry out the final reduction operation. Within our framework, a, a_{size} and P are retrieved as explained in Section 3.1. Furthermore, the source code statement that modifies a corresponds to the α-statement of $SCC(a_1, a_2)$, that is, $a_2 = \alpha(a_1, f(h_1), rhs(h_1))$ in Fig. 2(b). Finally, the two mappings are obtained as user supplied parameters, or as default values determined by the compiler. This information is summarized in Table 1. It should be noted that some transformations do not require extensions of the framework: the array expansion approaches of irregular assignment and irregular reduction, the parallel execution of semantic reduction operations (e.g. minimum/maximum and minimum/maximum with location), and the array splitting and merging method to parallelize consecutively written arrays. These techniques rest on the privatization of variables in order to allow different processors to write in different memory locations concurrently. The methods execute a final stage where the private results are combined through an appropriate function that preserves the semantics of the sequential program.

3.3. Consecutively Written Arrays

A *consecutively written array* (*CWA*) is a kernel that consists of writing consecutive entries of an array in consecutive locations during the execution of a loop. For illustrative purposes consider only the innermost loop do_k of the level-2 loop nest of Fig. 3(a). The code was extracted from the routine *rperm* (module *unary*) of the library of sparse matrix operations *SparsKit-II* [15]. In general, CWAs are implemented by means of a monotonic induction variable [5] of step one (ko in do_k of Fig. 3(a)) that determines the array entries to be written, and an assignment statement that sets the value of an array entry using as the left-hand side subscript expression a monotonic function of the induction variable. The figure shows a *non-conditional CWA* ($jao(ko)$), whose distinguishing characteristic is that the array variable is modified in all the loop iterations. When the array is computed only in those iterations where a condition is fulfilled, the kernel is called *conditional CWA* (e.g. $ao(ko)$ in Fig. 3(a)). In the rest of this section, the generation of parallel code for three complex loop nests that contain CWAs is analyzed.

3.3.1 Array Splitting and Merging

Current parallelizing compilers usually transform non-conditional CWAs into parallel code in two phases: first, a closed form expression for the corresponding linear in-

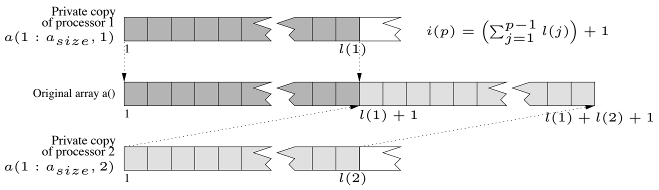


Figure 4. Graphic depiction of array splitting and merging.

duction variable is computed; and second, the references to the variable are replaced with such expression. The SCC classification algorithm used in the framework (second stage in Fig. 1) is a generalization of the classification scheme proposed in [5]. Thus, as described in that work, the transfer functions can be extended with capabilities for the symbolic computation of closed form expressions. The approach described above cannot be applied to conditional CWAs, except if the compiler can determine that the conditions are loop-invariant. The *array splitting and merging* transformation described in [10] enables the parallelization of conditional CWAs. The iterations of the sequential loop are mapped to processors according to a block distribution. Furthermore, the array is expanded to allow the processors to work in parallel on a private copy of the array from the first position. Finally, the original array is constructed by concatenating the private copies in increasing order of processor number (see Fig. 4).

Within our compiler framework, CWAs are recognized in two steps. First, the SCC classification algorithm performs an intra-SCC analysis that detects two basic kernels: a monotonic induction variable ($ko = ko + 1$ in do_k of Fig. 3(a)) and an array assignment operation ($jao(ko) = ja(k)$). Next, the SCC graph classification algorithm examines the SCC graph of Fig. 3(c). Let us focus on the use-def chain between $SCC(ko_3, ko_4)$ and $SCC(jao_1, jao_2, jao_3)$. The label $lhs\_index : jao_3(ko_3)$ indicates that the left-hand side subscript expression of $jao(ko) = ja(k)$ contains an occurrence of ko. In the scope of this scenario, the compiler checks whether the subscript ko is a monotonic function of ko, and then executes a monotonicity test [10, 19] to assure that every time $jao(ko)$ is computed, the monotonic variable ko is updated. As the test is successful, the variable jao is recognized as a non-conditional CWA during the execution of the loop. A similar analysis is applied to $ao(ko)$. Note that $values$ is a loop-invariant expression, and thus $ao(ko) = a(k)$ is executed in every (or none) loop iteration.

Regarding the generation of parallel code, the implementation of the array splitting and merging transformation requires no extension of the framework (see Table 1). The CWA is identified straightforwardly during the execution of the SCC graph classification algorithm. The remaining information (namely, the size of the array and number of processors) is retrieved as described in Section 3.2. In the following sections, some variants of CWAs that require the implementation of framework extensions are studied in detail.

3.3.2 Segmented CWAs

An interesting case is the level-2 loop nest from *SparsKit-II* shown in Fig. 3(a). It carries out a permutation of the rows of a sparse matrix (ao, jao, iao). The remarkable characteristic is the computation of a linear induction variable of step one, ko, that is set to the value of a loop-variant expression, $iao(perm(ii))$, at the beginning of each iteration of the outermost loop do_{ii}. As a result, each do_{ii} iteration performs write operations on a subarray a_{ii} ($ii = 1, ..., nrow$) of consecutive entries of ao and jao. The pattern of write operations is depicted in Fig. 3(b). Without loss of generality, assume that each loop iteration is mapped to a different processor. The subarray written by each processor p is defined by a starting position ψ_p and a length δ_p. The loop do_{ii} can be executed in a fully parallel manner (*DOALL* loop) if the subarrays do not overlap. The overlapping cannot be checked at compile-time because, in general, the value of $iao(perm(ii))$ is known at run-time only. We propose a solution that consists of inserting the following run-time test in the source code of the program. Let $\Psi = \{\psi_1, \psi_2, ..., \psi_{nrow}\}$ and $\Delta = \{\delta_1, \delta_2, ..., \delta_{nrow}\}$ be, respectively, the sets of starting positions and lengths of the subarrays, $a_1, a_2, ..., a_{nrow}$, defined in each loop iteration. The subarrays a_{ii} do not overlap if there is no subarray whose starting position corresponds to an entry of another subarray:

$$\nexists a_{ii} \;/\; \psi_k \leq \psi_{ii} \leq \psi_k + \delta_k + 1,\ \forall k \in \{1, \ldots, nrow\}, k \neq ii \tag{3}$$

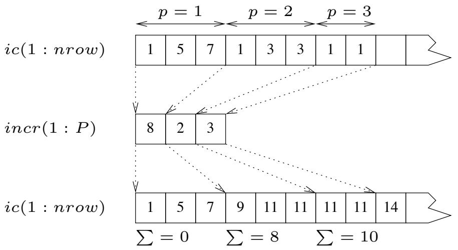
For this test to be implemented, the compiler needs to determine the sets Ψ and Δ, and an appropriate point of the program to insert the code of the test proposed above. In the example code of Fig. 3(a), the starting positions are

$$\Psi = \{iao(perm(ii)) \;\; ; \; ii = 1 \ldots nrow\} \tag{4}$$

Within the framework, this expression is extracted from the source code during the execution of the SCC graph classification algorithm. In particular, the algorithm analyzes a use-def chain between the following two SCCs: $SCC(ko_2)$ of class *subs* composed of $ko = iao(perm(ii))$, and $SCC(ko_3, ko_4)$ of class *non-cond/lin* corresponding to $ko = ko + 1$. As both SCCs define the same variable ko, and the statements belong to loop-bodies of different nesting levels, the reinitialized induction variable ko is recognized successfully. We extend the transfer functions

```
DO $ii = 1, nrow$
   DO $k = imask(ii), imask(ii + 1) - 1$
      $iw(jmask(k)) = true$
   END DO
   $k1 = ia(ii)$
   $k2 = ia(ii + 1) - 1$
   $ic(ii) = len + 1$
   DO $k = k1, k2$
      $j = ja(k)$
      IF $(iw(j))$ THEN
         $len = len + 1$
         $jc(len) = j$
         $c(len) = a(k)$
      END IF
   END DO
   DO $k = imask(ii), imask(ii + 1) - 1$
      $iw(jmask(k)) = false$
   END DO
END DO
```

(a) Source code.



(b) Post-processing stage for the parallelization of the computation of array ic.

Figure 5. Filter of the contents of a sparse matrix using a mask matrix.

to gather the right-hand side expression of the statement $ko = iao(perm(ii))$ during the recognition. Regarding the set of lengths, it is computed as the number of iterations of the innermost loop do_k for each do_{ii} loop iteration:

$$\Delta = \{ia(ii + 1) - ia(ii) \;\; ; \; ii = 1 \ldots nrow\} \qquad (5)$$

The expression above is symbolically computed at the end of the classification of the SCC that represents the index variable of do_k, that is, $SCC(k_2)$.

Finally, the compiler must insert the run-time test so that its overhead is minimized. The arrays iao, $perm$ and ia involved in the computation of Ψ and Δ are invariant with respect to do_{ii}. Thus, the most efficient location is the point of the program where the results of the test are reused a higher number of executions of do_{ii}. The extensions needed to gather these arrays and the optimal location are similar to those described in Sections 3.1.1 and 3.1.2.

3.3.3 Combination of CWAs with other Computational Kernels

In Fig. 5(a), a level-2 loop nest do_{ii} extracted from the routine *amask* of the module *unary* of *SparsKit-II* is presented. The code builds a sparse matrix in compressed row storage format (c, jc, ic) from an input matrix (a, ja, ia) by extracting only the elements that are stored in the positions pointed by a sparse mask matrix ($imask$, $jmask$). From the kernel recognition point of view, the loop consists of two non-independent kernels: a conditional CWA (variables jc and c) and an array assignment operation (variable ic). The computations associated with the array iw do not introduce loop-carried dependences that prevent the parallelization of do_{ii}. This characteristic can be detected using the SCC graph corresponding to the source code of Fig. 5(a). The details can be consulted in [3].

The loop nest of Fig. 5(a) can be parallelized according to the array splitting and merging technique described in Section 3.3.1. However, there is an important issue that must be considered in order to preserve the sequential semantics. In each do_{ii} iteration, the array entry $ic(ii)$ is set to the value $len + 1$, len being the monotonic induction variable used to compute the CWAs jc and c. This dependence between the two kernels represents a constraint that must be considered in order to preserve the sequential semantics. Next, we show the translation of such constraint into parallel code.

In the parallel code of the array splitting and merging transformation, each processor p uses a private copy of the scalar len. As a result, the values stored in ic at the end of the parallel execution of do_{ii} do not match the values of the sequential execution. Within our framework, the SCC graph contains a use-def chain between the SCCs that represent the computation of the array ic and the monotonic induction variable len, which is referenced in the right-hand side of the statement $ic(ii) = len + 1$. At this moment of the analysis, our extended transfer functions annotate the loop do_{ii} to indicate that during the parallel code generation stage the iterations of the loop do_{ii} must be mapped to processors according to a block distribution in order to be able to recover the global monotonic sequence of len from the private monotonic sequences computed by the processors. This recovery task is performed in a post-processing stage inserted just after do_{ii} in the parallel code, and that operates as follows. Consider the graphic depiction for three processors shown in Fig. 5(b). Each processor modifies the entries $ic(ii)$ corresponding to its loop iterations. After the parallel execution, the value of len is equal to its increment during the execution of the iterations assigned to the processor (array $incr(1 : P)$ in the figure). Finally, each processor p corrects the value of its $ic(ii)$ entries by adding to each entry the sum of the total number of elements computed by processors $1, \ldots, (p - 1)$, i.e. $\sum_{i=1}^{p-1} incr(i)$.

Table 2. Effectiveness of our extended framework for the SparsKit-II library.

	Parallel	Sequential
Simple loops	136	22
Level-1	109	22
Level-2	26	0
Level-4	1	0
Independent compound loops	8	2
Level-1	7	2
Level-2	1	0
Dependent compound loops	7	0
Level-1	1	0
Level-2	6	0

4. Experimental Results

We have developed a prototype of our extended framework of approximately $30,000$ lines of C++ code using the support given by the internal representation of the Polaris compiler [4]. Polaris also provides the GSA form and the CFG of Fortran77 source code. Our benchmark suite is the *SparsKit-II* library [15], which consists of a set of costly routines to perform operations with sparse matrices. We do not present experimental results in terms of the efficiency of the parallelizing techniques as this work was accomplished by their respective authors. Thus, Table 2 shows results in terms of the number of loops that can be executed in parallel using our framework. The results are organized in three categories: *simple loops*, which contain only one loop-level kernel; *independent compound loops*, which present a set of independent kernels; and *dependent compound loops*, with dependences between the kernels. Statistics are presented for each nesting level. Loop nests with kernels that are not recognized by the framework were not considered.

SparsKit-II contains 136 simple loops that can be translated into parallel code straightforwardly because they compute array assignments with regular (72) and irregular (17) access patterns, irregular reductions (24), scalar reductions (13), CWAs (6), conditional linear induction variables (1) and semantic kernels (2 find-and-set kernels and 1 minimum with location). The 22 sequential simple loops correspond to array recurrences with regular access patterns, whose analysis is not implemented in this version of our prototype. The kernels mentioned above are described in detail in [1]. As a comparison, we have checked that the Polaris compiler fails to parallelize those loops containing irregular assignments (21) and irregular reductions (2), loops with induction variables whose closed form expression cannot be computed (10 conditional CWAs), and some loops that compute semantic kernels (1 minimum with location).

Regarding independent compound loops, the 10 loop nests detected in *SparsKit-II* compute a combination of the following kernels: regular/irregular assignment, regular/irregular reduction, CWA, find-and-set kernel, regular array recurrence, and linear induction variable. Our extended framework also enables the automatic parallelization of dependent compound loop nests. The 7 loops that appear in *SparsKit-II* consist of a combination of two dependent kernels: a CWA and a regular/irregular array assignment. The dependence is due to the use of a linear induction variable as described in the case study of Section 3.3.3. We have shown that the framework provides efficient support for the insertion of an appropriate post-processing stage that preserves the sequential semantics.

5. Related Work

In the literature, the automatic generation of loop-level parallel code is addressed from different viewpoints. Keßler [8] proposes a speculative program comprehension method for the recognition of syntactical variants of computational kernels that involve sparse vectors and matrices (e.g. product, linear system solution, etc.). Aggressive parallel code generation is achieved by replacing the loop with an equivalent parallel algorithm or machine-specific library routines. Unlike our approach, in automatic program comprehension the semantics of the code is considered. However, the transfer functions of our framework can be extended to take semantics into account.

A different viewpoint takes advantage of classical data dependence analysis [13, 18] to disprove the existence of dependences that prevent parallelization. Code transformations that remove/handle such dependences [2, 5, 6, 10, 21, 22] are used to enable parallelization. The extended framework described in this paper is targeted for the analysis of complex loops. However, it also provides a generic platform for the analysis of regular loops by applying both data dependence techniques and code transformations on demand.

Suganuma et al. [16] address the generation of parallel code for a set of complex loop nests that only contain scalar reductions. Unlike our approach, the main limitation is the narrow scope of application. Nevertheless, they optimize the corresponding parallel code by grouping interprocessor communications. Currently, we do not perform an inter-loop analysis (i.e., involving different loop nests) to accomplish this kind of optimization.

6. Conclusions

This paper has addressed the automatic generation of parallel code in the scope of complex loop nests where today's parallelizing compilers fail. We have proposed some extensions of a generic extensible compiler framework for kernel recognition that make it a powerful information-gathering technique. In particular, we have shown the potential of our approach by describing the efficient support supplied for the parallelization of a set of complex loop nests extracted from real codes. In addition, we have presented our framework as a unified platform for the integration of both existing and new techniques for the automatic recognition of loop-level parallelism and the generation of parallel code.

As future work, we intend to measure the effectiveness of our automatic parallelization strategy in the analysis of complex loop nests included in other representative benchmark suites. The parallelization of regular loops through the integration of classical dependence tests within our framework will be studied. Finally, we intend to extend the compiler framework with inter-loop analysis to generate efficient parallel code targeted for specific architectures. For instance, this analysis could improve locality exploitation and reduce interprocessor communication.

References

[1] M. Arenaz. *Compiler Framework for the Automatic Detection of Loop-Level Parallelism.* PhD thesis, Departament of Electronics and Systems, University of A Coruña, Mar. 2003. Available at http://www.des.udc.es/~arenaz.

[2] M. Arenaz, J. Touriño, and R. Doallo. Run-time support for parallel irregular assignments. In *6th Int'l Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers, LCR'02*, Washington DC, Mar. 2002.

[3] M. Arenaz, J. Touriño, and R. Doallo. A GSA-based compiler infrastructure to extract parallelism from complex loops. In *17th ACM Int'l Conference on Supercomputing, ICS'03*, pages 193–204, San Francisco, CA, June 2003.

[4] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, W. M. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, Dec. 1996.

[5] M. Gerlek, E. Stoltz, and M. Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA. *ACM Transactions on Programming Languages and Systems*, 17(1):85–122, 1995.

[6] E. Gutiérrez, O. Plata, and E. Zapata. Balanced, locality-based parallel irregular reductions. In *14th Int'l Workshop on Languages and Compilers for Parallel Computing, LCPC 2001*, Cumberland Falls, KY, Aug. 2001.

[7] H. Han and C.-W. Tseng. Efficient compiler and run-time support for parallel irregular reductions. *Parallel Computing*, 26(13-14):1861–1887, 2000.

[8] C. Keßler. Applicability of program comprehension to sparse matrix computations. In *3rd Int'l European Conference on Parallel Processing, Euro-Par'97*, pages 347–351, Passau, Germany, Aug. 1997.

[9] K. Knobe and V. Sarkar. Array SSA form and its use in parallelization. In *25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1998*, pages 107–120, San Diego, CA, Jan. 1998.

[10] Y. Lin and D. Padua. On the automatic parallelization of sparse and irregular Fortran programs. In *4th Int'l Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers, LCR'98*, pages 41–56, Pittsburgh, PA, May 1998.

[11] M. Martín, D. Singh, J. Touriño, and F. Rivera. Exploiting locality in the run-time parallelization of irregular loops. In *31st Int'l Conference on Parallel Processing, ICPP 2002*, pages 27–34, Vancouver, Canada, Aug. 2002.

[12] S. Muchnick. *Compiler Design and Implementation.* Morgan Kaufmann Publishers, 1997.

[13] K. Psarris and K. Kyriakopoulos. Data dependence testing in practice. In *1999 Int'l Conference on Parallel Architectures and Compilation Techniques, PACT'99*, pages 264–273, Newport Beach, CA, Oct. 1999.

[14] L. Rauchwerger and D. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):160–180, Feb. 1999.

[15] Y. Saad. *SPARSKIT: A basic tool kit for sparse matrix computations.* Available at http://www-users.cs.umn.edu/-/~saad/software/SPARSKIT/sparskit.html.

[16] T. Suganuma, H. Komatsu, and T. Nakatani. Detection and global optimization of reduction operations for distributed parallel machines. In *10th ACM Int'l Conference on Supercomputing, ICS'96*, pages 18–25, Philadelphia, PA, May 1996.

[17] P. Tu and D. Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *9th ACM Int'l Conference on Supercomputing, ICS'95*, pages 414–423, Barcelona, Spain, July 1995.

[18] M. Wolfe. *High performance compilers for parallel computing.* Addison-Wesley, 1996.

[19] P. Wu, A. Cohen, J. Hoeflinger, and D. Padua. Monotonic evolution: An alternative to induction variable substitution for dependence analysis. In *15th ACM Int'l Conference on Supercomputing, ICS'01*, pages 78–91, Sorrento, Italy, June 2001.

[20] C.-Z. Xu and V. Chaudhary. Time stamp algorithms for runtime parallelization of DOACROSS loops with dynamic dependences. *IEEE Transactions on Parallel and Distributed Systems*, 12(5):433–450, 2001.

[21] H. Yu and L. Rauchwerger. Adaptive reduction parallelization techniques. In *14th ACM Int'l Conference on Supercomputing, ICS'00*, pages 66–77, Santa Fe, NM, May 2000.

[22] F. Zhang and E. D'Hollander. Enhancing parallelism by removing cyclic data dependencies. In *6th Int'l PARLE Conference, Parallel Architectures and Languages Europe, PARLE'94*, pages 387–397, Athens, Greece, July 1994.