A GSA-Based Compiler Infrastructure to Extract Parallelism from Complex Loops[*]

Manuel Arenaz, Juan Touriño, and Ramón Doallo
Computer Architecture Group
Department of Electronics and Systems
University of A Coruña
15071 A Coruña, Spain
{arenaz,juan,doallo}@udc.es

ABSTRACT

This paper presents a new approach for the detection of coarse-grain parallelism in loop nests that contain complex computations, including subscripted subscripts as well as conditional statements that introduce complex control flows at run-time. The approach is based on the recognition of the computational kernels calculated in a loop without considering the semantics of the code. The detection is carried out on top of the Gated Single Assignment (GSA) program representation at two different levels. First, the use-def chains between the statements that compose the strongly connected components (SCCs) of the GSA use-def chain graph are analyzed (intra-SCC analysis). As a result, the kernel computed in each SCC is recognized. Second, the use-def chains between statements of different SCCs are examined (inter-SCC analysis). This second abstraction level enables the detection of more complex computational kernels by the compiler. A prototype was implemented using the infrastructure provided by the Polaris compiler. Experimental results that show the effectiveness of our approach for the detection of coarse-grain parallelism in a suite of real codes are presented.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processor—*compilers, optimization*

General Terms

Algorithms, Languages

Keywords

Parallelizing compilers, loop-level kernel recognition, GSA, strongly connected components

[*]This work was funded by the Ministry of Science and Technology of Spain and FEDER funds of the European Union (Project TIC2001-3694-C02).

1. INTRODUCTION

The automatic detection of parallelism in loops that contain complex computations is still a challenge for current parallelizing compilers mainly due to the presence of subscripted subscripts and/or complex control constructs. We use the term *complex loop* to designate loops with such characteristics. In classical dependence analysis [13, 18], the potential parallelism of complex loops usually cannot be uncovered because the necessary information for dependence testing is not available at compile-time. In other occasions, the limitations of compiler technology arise from the use of information-gathering techniques that are not sufficiently sophisticated. An approach to the problem is the combination of dependence testing with source code pattern-matching techniques that recognize specific, isolated patterns [12]. These methods have two major drawbacks: dependence on the source code quality and difficulty in analyzing complex control constructs.

A different approach for the detection issue is automatic program comprehension. This kind of technique [8] recognizes syntactical variations of operations with vectors and matrices by taking into account the semantics of the source code. For this reason, the scope of application is mainly limited to numerical programs.

In this paper, we present a compiler infrastructure that enables the recognition of a wide range of *computational kernels* independently of the semantics and the quality of the code. It is based on the analysis of the strongly connected components (SCCs) that appear in the use-def chain graph of the *Gated Single Assignment (GSA)* program representation [16], which supplies efficient support for the examination of the control constructs that appear in the loop body. Our infrastructure provides a unified framework for the detection of parallelism in loop nests that contain complex scalar and array kernels. We use the terms *scalar kernel* and *array kernel* to designate kernels whose result is stored in a scalar and an array variable, respectively. A complete list of the kernels currently detected is detailed in [1]. Some examples are complex forms of induction variables, linked-list traversals, masked operations with scalars and arrays, array operations with subscripted subscripts (e.g. irregular assignments, irregular reductions, array recurrences)... In this paper, we only introduce the kernels needed in our case studies.

The rest of the paper is organized as follows. Section 2 gives the reader a general overview of our proposal. Con-

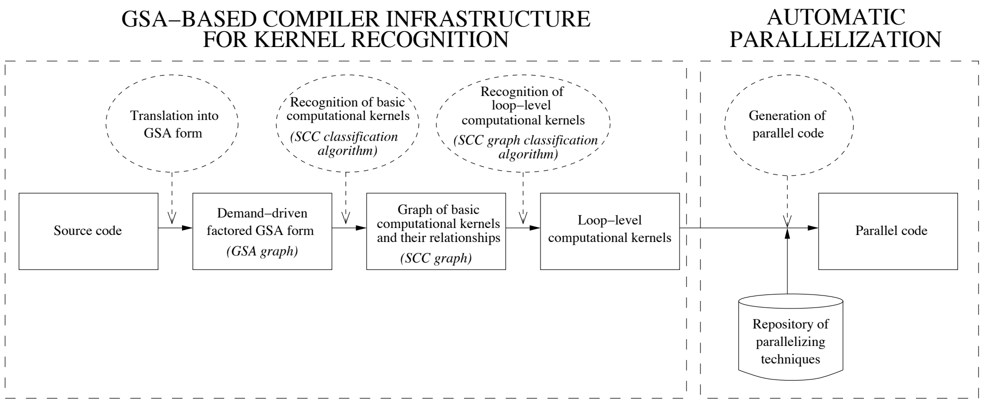

GSA–BASED COMPILER INFRASTRUCTURE FOR KERNEL RECOGNITION
AUTOMATIC PARALLELIZATION
Translation into GSA form
Recognition of basic computational kernels
(SCC classification algorithm)
Recognition of loop–level computational kernels
(SCC graph classification algorithm)
Generation of parallel code
Source code
Demand–driven factored GSA form
(GSA graph)
Graph of basic computational kernels and their relationships
(SCC graph)
Loop–level computational kernels
Parallel code
Repository of parallelizing techniques

Figure 1: Block diagram of the automatic detection technique.

cepts and terms that will be used in the explanations are also introduced. Sections 3 and 4 describe our algorithms for the recognition of the computational kernels represented by the SCCs of the loop body (intra-SCC analysis), and by the combination of a set of SCCs (inter-SCC analysis). Section 5 presents a complex case study that shows the potential of our infrastructure for the extraction of coarse-grain parallelism. Section 6 is devoted to experimental results that compare our approach with the Polaris parallelizing compiler. Finally, Section 7 discusses related work, and Section 8 concludes the paper.

2. FRAMEWORK OVERVIEW

Our strategy for the automatic detection (and further parallelization) of complex loops is based on a compiler infrastructure that aims at the recognition of the kernels computed in a loop. The block diagram of Fig. 1 shows a scheme of the strategy. The stages in the operation of the compiler infrastructure are depicted as dashed ovals. The sequence of intermediate program forms that enable the extraction of parallelism from the source code is represented by the chain of solid rectangular boxes.

The different stages are described in the following subsections using as a guide the *consecutively written array* kernel [10] presented in Fig. 2(a). This kernel consists of writing consecutive entries of an array, a, in consecutive locations during the execution of a loop, do_h. The kernel implementation usually consists of a combination of a linear induction variable of step one (variable i) that determines the array entries to be written, and an assignment operation that defines the value of an array entry, $a(i)$, using a linear function of the induction variable as the left-hand side subscript expression. The complexity of this loop comes from the fact that i is incremented in those iterations where the condition $c(h)$ is fulfilled. In general, the condition is not loop-invariant, so the sequence of values of i cannot be expressed as a function of the loop index variable h. The scalar tmp represents temporary computations that do not introduce loop-carried dependences at run-time.

2.1 Translation into GSA Form

The first step is the translation of the source code into the *Gated Single Assignment (GSA)* program representation [16]. We perform kernel recognition on top of GSA because it has some properties that ease the development

```
$i = 1$
DO $h = 1, n$
   IF $(c(h))$ THEN
      $tmp = f(h)$
      $a(i) = tmp + 2$
      $i = i + 1$
   END IF
END DO
```

(a) Source code.

```
$i_1 = 1$
DO $h_1 = 1, n, 1$
   $i_2 = \mu(i_1, i_4)$
   $a_1 = \mu(a_0, a_3)$
   $tmp_1 = \mu(tmp_0, tmp_3)$
   IF $(c(h_1))$ THEN
      $tmp_2 = f(h_1)$
      $a_2 = \alpha(a_1, i_2, tmp_2 + 2)$
      $i_3 = i_2 + 1$
   END IF
   $i_4 = \gamma(c(h_1), i_3, i_2)$
   $a_3 = \gamma(c(h_1), a_2, a_1)$
   $tmp_3 = \gamma(c(h_1), tmp_2, tmp_1)$
END DO
```

(b) GSA form.

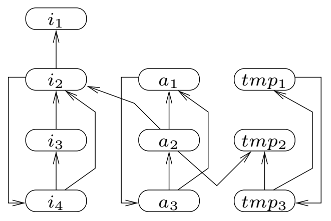
Figure 2: Consecutively written array computational kernel.

of tools for automatic program analysis. Some of the most relevant are the elimination of false dependences for scalar and array definitions (not for array element references), the representation of reaching definition information in a syntactical manner, and the capture of the conditional expressions that determine the control flow of the program.

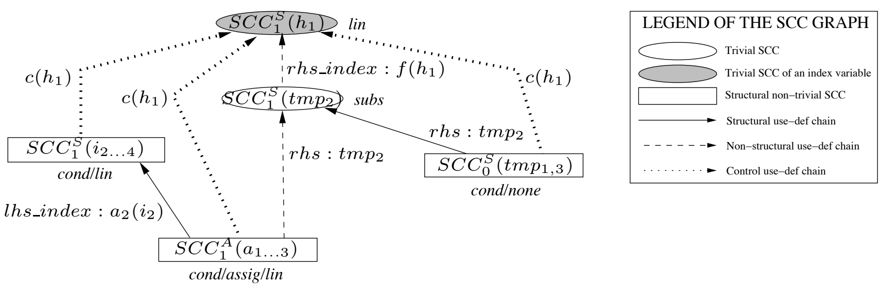
The GSA form is an extension of *Static Single Assignment (SSA)* [5] that captures the flow of values of scalar and array variables, even in loops with complex control constructs. This task is accomplished by inserting a set of special statements right after the points of the program where the control flow merges, and by renaming the variables of the program so that they are assigned unique names in the definition statements. The GSA form corresponding to the loop of Fig. 2(a) is presented in Fig. 2(b). Special statements are inserted for each variable defined within the loop body, that is, the temporary scalar *tmp*, the linear induction variable i, and the array variable a. Three types are distinguished on the basis of the location within the loop body: μ-statements, associated with the header of the loop do_{h_1}; γ-statements, inserted after the if-endif construct; and α-statements, which replace array assignment statements.

2.2 Recognition of Basic Kernels

In an early stage of the work, a suite of real codes that contain complex loops was carefully analyzed by hand. As a result, it was found that the body of most of the loop



(a) GSA graph.



(b) SCC graph.

Figure 3: Graph representations of the kernel shown in Fig. 2.

nests can be represented as a combination of simple kernels, which will be referred to as *basic computational kernels*. The recognition of basic kernels is carried out through the analysis of the strongly connected components (SCCs) that appear in the *GSA graph* of the loop, that is, the graph of factored use-def chains corresponding to the GSA form. We call *SCC class* to the type of basic kernel, and *SCC classification algorithm* to the procedure that derives the SCC class from the GSA graph. The SCC class is denoted as $[SCC(x_{1...n})]$, $x_1, \ldots, x_n$ being the variables defined in the GSA statements of the SCC.

The GSA graph associated with the loop of Fig. 2 is depicted in Fig. 3(a). The nodes represent the assignment statements, and the labels are the left-hand side symbols. The edges correspond to chains between pairs of statements. For clarity, the information related to the index variable of the loop, h_1, was omitted from the graph. The consecutively written array computations contain two basic kernels: a conditional linear induction variable (variable i) and an array assignment operation (variable a). In the former kernel, the statements that define different instances of i ($i_2 = \mu(i_1, i_4)$, $i_3 = i_2 + 1$ and $i_4 = \gamma(c(h_1), i_3, i_2)$) compose a SCC in the GSA graph. In the latter kernel, the SCC consists of $a_1 = \mu(a_0, a_3)$, $a_2 = \alpha(a_1, i_2, tmp_2 + 2)$ and $a_3 = \gamma(c(h_1), a_2, a_1)$. We use the notations $SCC_1^S(i_{2...4})$ and $SCC_1^A(a_{1...3})$ for the SCCs of i and a, respectively. The subscript denotes the *cardinality of the SCC*, which is the number of different variables of the source code that are defined in the statements of the SCC. When the cardinality is zero or one, the superscript shows whether the variable of the source code is a scalar (S for scalar SCCs) or an array (A for array SCCs). The GSA graph of Fig. 3(a) contains two additional SCCs that represent the flow of values of the temporary variable *tmp*: $SCC_1^S(tmp_2)$ and $SCC_0^S(tmp_{1,3})$. SCCs with cardinality greater than one are not considered in this work as they represent a small percentage of the SCCs that appear in our benchmark suite.

2.3 Recognition of Loop-Level Kernels

The goal of our compiler infrastructure is the recognition of higher-level computational kernels that result from a combination of a set of basic kernels. In order to illustrate this fact, consider the loop of Fig. 2(a). The isolated detection of the linear induction variable i, and the array assignment operation a, does not provide enough information to recognize the consecutively written array loop-level kernel. For the compiler to have success, the relationship between both basic kernels has to be analyzed.

The information extracted from the source code during the execution of the SCC classification algorithm is represented in the *SCC graph*. In Fig. 3(b) the SCC graph of the consecutively written array computations of Fig. 2 is depicted. Nodes of different shapes represent SCCs with different properties. Oval nodes highlight the presence of *trivial SCCs* (e.g. $SCC_1^S(tmp_2)$), which consist of only one GSA statement. Shaded ovals represent the trivial SCCs corresponding to loop index variables (e.g. $SCC_1^S(h_1)$). Regarding non-trivial SCCs, two subtypes are distinguished: *structural non-trivial SCCs* (rectangular nodes), where the variable of the kernel is not tested in the conditional expression of any γ-statement of the SCC (e.g. $SCC_1^A(a_{1...3})$); and *semantic non-trivial SCCs*, whose variable is tested (not used in our example). The class of each SCC is printed next to the corresponding node. The notations for the SCC classes are introduced using the examples of Fig. 3(b). The class of the scalar component $SCC_1^S(i_{2...4})$ is denoted by the pair *cond/lin*, where the term *lin* means that the kernel consists of a linear induction variable, and the term *cond* indicates that it is conditionally computed during the execution of the loop. The value *none* is used for the initialization of the classes before the execution of the SCC classification algorithm. As a result, when the SCC contains only μ or γ statements, the class inherits that value (e.g. $SCC_1^S(tmp_{1,3})$). The notation used for trivial SCCs consists of one term that shows the scalar kernel computed in the statements of the SCC. Thus, the value *subs* (abbreviation of subscripted) of $[SCC_1^S(tmp_2)]$ indicates the loop-variant nature of the expression $f(h_1)$ assigned to *tmp* in each iteration of do_{h_1}. Finally, the notation *cond/assig/lin* corresponding to the class of the array component $SCC_1^A(a_{1...3})$ is as follows. The first element (*cond*) is the conditionality of the kernel, the second element is related to the structure of the assignment statements (*assig* for assignment operations), and the third element captures the class of the left-hand side subscript expression of the statements (*lin* for a linear access to the entries of the array). The interpretation of the SCC graph is completed with the description of the edges. The edges represent the use-def chains (and thus relationships) between statements of different SCCs. In order to identify the use-def chains that are relevant for the recognition of loop-level kernels, three types are distinguished: *structural use-def chains* (solid edges), *non-structural use-def chains* (dashed edges) and *control use-def chains* (dotted edges). The labels of the edges show the expression that contains the occurrence of the variable of the target SCC. For structural and non-structural use-def chains, the location of the expression within the statement of the source SCC is also presented: left-hand side subscript (*lhs_index*), right-hand side subscript (*rhs_index*), or right-hand side non-subscript expression (*rhs*). The relevance of this information for kernel recognition will be pointed out throughout the paper.

The SCC graph is an intermediate program representation that exhibits the minimal set of properties that characterize the computation of a loop-level kernel. We have designed a *SCC graph classification algorithm* that identifies these typical scenarios in the SCC graph, providing a compiler environment that guides the execution of additional checks that actually lead to the recognition of loop-level kernels. We illustrate this key idea with the example of Fig. 3(b). Consider the structural chain denoted as $SCC_1^A(a_{1\ldots3}) \Rightarrow SCC_1^S(i_{2\ldots4})$, and depicted as a solid edge in the figure. It indicates that the induction variable i is referenced in the index expression of the left-hand side array reference (the label of the use-def chain contains *lhs_index*) of the source code assignment statement $a(i) = tmp + 2$. This information points out the existence of a potential consecutively written array kernel. As will be shown in Section 4, the kernel is recognized after carrying out some additional checks.

2.4 Generation of Parallel Code

The last stage of our automatic parallelization approach is the generation of parallel code. This task, which is outside the scope of this paper, basically consists of applying a parallelizing technique to each loop-level kernel that, according to our framework, can be executed in parallel. Techniques that cover the parallelization of doacross loops [11, 20], irregular reductions [7, 21], irregular assignments [3, 9] or even several types of kernels [10] can be found in the literature. The kernel recognition algorithms are a powerful information-gathering infrastructure that supplies the information that the implementation of the parallelizing transformations require. For instance, some techniques based on the inspector-executor model reorder the iterations of a loop by analyzing the contents of a set of index arrays. During the operation of our algorithms these arrays are identified straightforwardly. Another example consists of determining the optimal point of a program for the insertion of a run-time test or an inspector. This issue can be addressed efficiently by using the reaching definition information of the GSA form.

3. RECOGNITION OF BASIC KERNELS

The recognition of basic kernels is addressed through the classification of the SCCs of the GSA graph according to the taxonomy of SCC classes presented in [2]. We have designed a recursive algorithm that reduces the computation of a SCC class to determining the class of the statements and the expressions that compose the SCC. The core of the procedure is a *demand-driven contextual classification scheme* that computes the class of an expression by means of a *transfer function* that merges the classes of the corresponding subexpressions. The objective of this analysis is searching the occurrences of the variable that introduces loop-carried dependences, i.e. the variable associated with the SCC. When the classification of a different statement or SCC is needed for the analysis, the recursive step is executed. The SCC classification algorithm provides the compiler with the classes of the SCCs and the use-def chains between pairs of SCCs. This information is represented in the SCC graph of the loop nest.

```
DO h = 1, g_size
   ...
   i = g(i)
END DO
```

(a) Loop that traverses a linked-list implemented with the array g.

```
DO h = 1, g_size
   ...
   i = h + 1
   r = r + g(i)
END DO
```

(b) Loop that computes a scalar reduction using the variable r.

Figure 4: Source codes to illustrate the contextual classification of expressions.

The term contextual classification refers to the fact that an expression is not always assigned the same class. The notation $[e]_{p:l,E}^{e_{ref}}$ is used to represent the *contextual class* of an expression e. The parameters that define the context are: e_{ref}, the reference expression whose occurrences are searched; l, the level of e within the tree representation of E, e being a subexpression of another expression E (see the concept *level of an expression* [18, Chapter 3]); and p, which indicates if E is a subexpression of the left-hand side of a statement (denoted as $\blacktriangleleft$), the right-hand side ($\blacktriangleright$), or the conditional expression of a γ-statement (denoted as ?). Consider the loops presented in Fig. 4. Let us focus on the expression $g(i)$ to illustrate why contextual classification is needed. In Fig. 4(a), the loop-carried dependence introduced by the scalar variable i is represented by a SCC in the GSA graph. The right-hand side of the statement $i = g(i)$ is the expression $g(i)$. Thus, the class $[g(i)]_{\blacktriangleright:0,g(i)}^{i}$ is said to be a linked-list traversal (denoted by the class *list*) because the right-hand side expression consists of a reference to an array variable g whose subscript expression is an occurrence of e_{ref}, which is the left-hand side variable i. On the other hand, the loop of Fig. 4(b) calculates a scalar reduction using the variable r. In this case, $g(i)$ is an operand of the sum operator that appears in the statement $r = r + g(i)$, i being a scalar variable that takes a different value in each loop iteration. Thus, $[g(i)]_{\blacktriangleright:1,r+g(i)}^{r}$ represents the computation of a loop-variant expression (class *subs*) because the subscript i does not match the reference expression r.

The rest of this section is organized as follows. Section 3.1 describes the SCC classification algorithm. Section 3.2 explains this algorithm in detail using the example loop of Fig. 2 as a guide.

3.1 SCC Classification Algorithm

The goals of this algorithm are the classification of the set of SCCs of the GSA graph, and the classification of the use-def chains between pairs of SCCs. It proceeds as follows. For each non-classified component, $SCC(x_{1\ldots n})$, it is pushed onto a stack of SCCs and the computation of its class, $[SCC(x_{1\ldots n})]$, is started. If the SCC is trivial, then the class is $[SCC(x_1)]$=$[x_1 = e_1]$, where $x_1 = e_1$ is the unique statement of the SCC. If it is non-trivial, then $[SCC(x_{1\ldots n})]$ is calculated as $[x_1 = \mu(x_0, x_n)]$, where $x_1 = \mu(x_0, x_n)$ is the μ-statement inserted after the header of the outermost loop of the loop nest. As shown above, the determination of $[SCC(x_{1\ldots n})]$ is simplified to the computation of the class of a statement. Let $x = e$ be a GSA statement. The class $[x = e]$ is calculated as the contextual class of the right-hand side expression $[e]^{x}_{\blacktriangleright:0,e}$. From this moment, the transfer functions perform a systematic traversal of e that aims at the recognition of occurrences of the reference expression x. The contextual classification of these occurrences will lead to the recognition of the different classes of SCCs. Finally, if the SCC is semantic (see Section 2.3), the algorithm determines the appropriate semantic class by checking the properties of the corresponding conditional expressions.

The base case of our recursive algorithm is the classification of independent SCCs as they contain no occurrence of variables defined in statements of other SCCs. Once $[SCC(x_{1\ldots n})]$ is determined, $SCC(x_{1\ldots n})$ is popped from the stack, and the classification process of the remaining SCCs continues.

The recursive step is executed when a non-independent component, $SCC(x_{1\ldots n})$, is found. During the analysis of the statements of the SCC, the occurrences of the variables defined in other strongly connected components are found. Each occurrence y enables the detection of a use-def chain $SCC(x_{1\ldots n}) \rightarrow SCC(y_{1\ldots m})$. At this moment, the classification process of the source component $SCC(x_{1\ldots n})$ is deferred, $SCC(x_{1\ldots n})$ is pushed onto the stack of SCCs, and the classification of the target component $SCC(y_{1\ldots m})$ is started. When $[SCC(y_{1\ldots m})]$ is computed, $SCC(y_{1\ldots m})$ is popped from the stack, and the classification of $SCC(x_{1\ldots n})$ continues at the same point where it had been deferred. Once all the occurrences have been processed, $[SCC(x_{1\ldots n})]$ is successfully determined. The final stage of the algorithm is the classification of the use-def chains whose source component is $SCC(x_{1\ldots n})$ as structural, non-structural or control chains.

The algorithm described above reaches a deadlock state when the GSA graph contains mutually dependent SCCs. These situations arise because, in order to enable a proper kernel recognition, the SCCs of the GSA graph are constructed by ignoring the control use-def chains. The detection of mutually dependent SCCs is performed by using the stack of SCCs in the following manner. If a use-def chain $SCC(x_{1\ldots n}) \rightarrow SCC(y_{1\ldots m})$ is found, the contents of the stack are checked before starting the classification of $SCC(y_{1\ldots m})$. If $SCC(y_{1\ldots m})$ is already in the stack, it means that there is a set of use-def chains from $SCC(y_{1\ldots m})$ to $SCC(x_{1\ldots n})$. Consequently, $SCC(x_{1\ldots n})$ and $SCC(y_{1\ldots m})$ are mutually dependent. The transfer functions of the algorithm have been designed so that mutually dependent SCCs are assigned the class *unknown*.

3.2 Case Study

The graphs of Fig. 3 consist of five components. Two non-trivial SCCs, $SCC^{S}_{1}(i_{2\ldots4})$ and $SCC^{A}_{1}(a_{1\ldots3})$, represent the conditional induction variable i, and the conditional array assignment operation a, respectively. There are also two trivial SCCs, $SCC^{S}_{1}(h_1)$ and $SCC^{S}_{1}(tmp_2)$, which capture the loop index h, and the temporary variable *tmp*. As *tmp* is computed inside an if-endif construct, an additional component, $SCC^{S}_{0}(tmp_{1,3})$, appears in the GSA graph in order to represent the flow of values. Without loss of generality, let us assume that the SCCs are processed in the following order: $SCC^{S}_{1}(i_{2\ldots4})$, $SCC^{S}_{0}(tmp_{1,3})$, $SCC^{A}_{1}(a_{1\ldots3})$, $SCC^{S}_{1}(h_1)$ and $SCC^{S}_{1}(tmp_2)$.

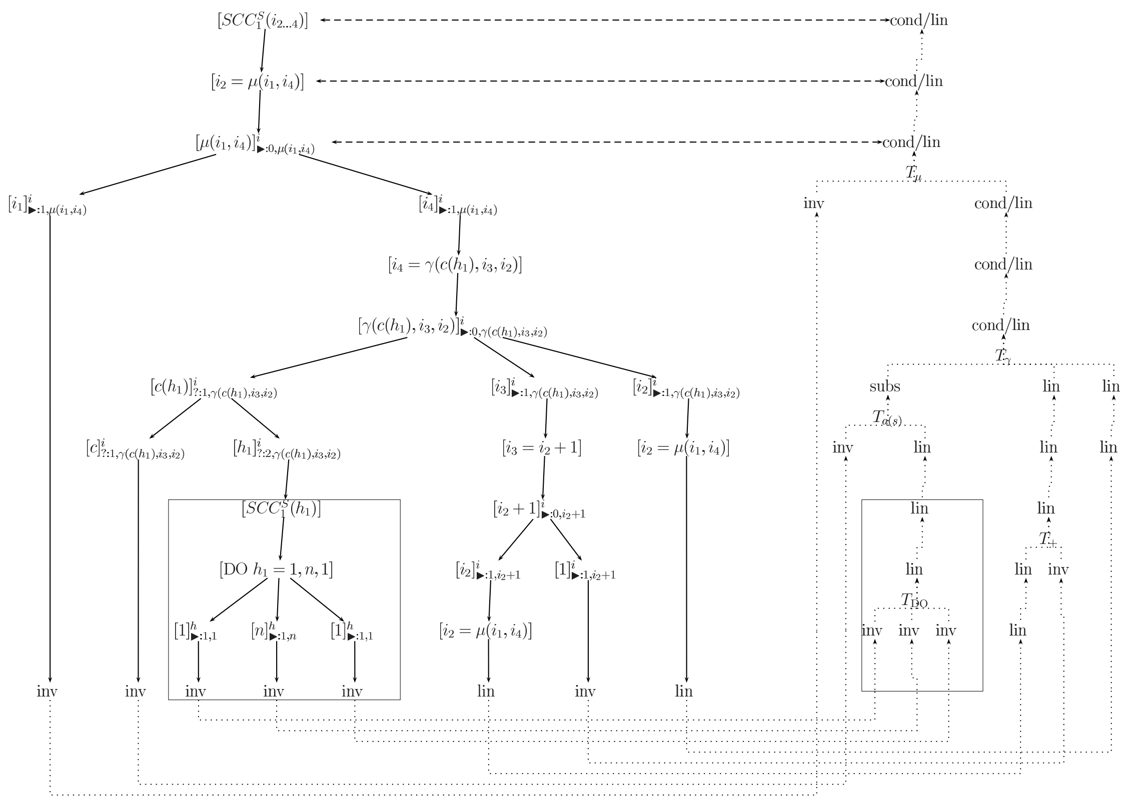
The class $[SCC^{S}_{1}(i_{2\ldots4})]$ is determined first. A tree representation of the classification process is depicted in Fig. 5. The picture consists of two trees. The tree on the left illustrates the decomposition of $[SCC^{S}_{1}(i_{2\ldots4})]$ into the classification of the statements and the expressions included in the component. The labels of the child nodes represent the classes that have to be determined in order to compute the class shown in the label of the parent node. The solid edges highlight this top-down process. The tree on the right shows the class derived for each node of the left-hand side tree. The dashed edges depicted in the first three levels emphasize this correspondence. The class of each expression, statement or SCC is calculated by means of an appropriate transfer function that merges the classes associated with the child nodes. In [1] we have defined a set of transfer functions that check the characteristics of a wide range of kernels. Note that by making little changes in the transfer functions, the infrastructure can be easily extended to recognize new kernels. Due to space limitations, partial definitions of the transfer functions that cover the relevant situations for the analysis of our example loop are presented in this paper. For clarity, the name of the transfer function is not shown in the figure when a parent node inherits the class of its unique child. The dotted edges highlight this bottom-up process.

The execution of the SCC classification algorithm corresponds to the depth-first traversal of the tree on the left. As stated in Section 3.1, $[SCC^{S}_{1}(i_{2\ldots4})]$ is reduced to determining the class of the statement associated with the header of do_{h_1}, i.e. $[i_2 = \mu(i_1, i_4)]$. The class $[i_2 = \mu(i_1, i_4)]$ inherits the contextual class of its right-hand side expression, $[\mu(i_1, i_4)]^{i}_{\blacktriangleright:0,\mu(i_1,i_4)}$, which will be obtained after the classification of the arguments i_1 and i_4. The occurrence i_1 corresponds to the initialization of the induction variable before the execution of the loop. As the statement $i_1 = 1$ is located outside the loop body, the class $[i_1]^{i}_{\blacktriangleright:1,\mu(i_1,i_4)}$ is determined by applying the transfer function of loop-invariant expressions, K:

$$T_K \quad : \quad [K]^{e_{ref}}_{p:l,E} = \\ = \begin{cases} inv & \text{if } e_{ref} \text{ is scalar} \\ assig/[s]^{a(s)}_{\blacktriangleleft:1,a(s)} & \text{if } e_{ref} \text{ matches } a(s),\ l = 0 \\ inv & \text{if } e_{ref} \text{ matches } a(s),\ l > 0 \end{cases} \tag{1}$$

where the class *inv* represents a loop-invariant expression. As explained in Section 2.3, the notation for the classes of scalar and array SCCs consists of tuples whose first element represents the conditionality of the SCC. As the conditionality refers to the presence of γ-statements in the SCC, it will only be modified in the transfer function of the γ special operator, T_γ. For the sake of clarity, this information (i.e., the term *cond*) is not shown in the transfer functions. Regarding our example, as the reference expression is a scalar variable i, the first entry of T_K is applied and $[i_1]^{i}_{\blacktriangleright:1,\mu(i_1,i_4)}$ is set to *inv*.

The second argument of $\mu(i_1, i_4)$ is an occurrence of the variable defined in the γ-statement, $i_4 = \gamma(c(h_1), i_3, i_2)$ of $SCC^{S}_{1}(i_{2\ldots4})$. The class $[i_4]^{i}_{\blacktriangleright:1,\mu(i_1,i_4)}$ will be the result of



$[SCC_1^S(i_{2...4})]$
cond/lin
$[i_2 = \mu(i_1, i_4)]$
cond/lin
$[\mu(i_1, i_4)]^i_{\blacktriangleright:0,\mu(i_1,i_4)}$
cond/lin
T_μ
$[i_1]^i_{\blacktriangleright:1,\mu(i_1,i_4)}$
$[i_4]^i_{\blacktriangleright:1,\mu(i_1,i_4)}$
inv
cond/lin
$[i_4 = \gamma(c(h_1), i_3, i_2)]$
cond/lin
$[\gamma(c(h_1), i_3, i_2)]^i_{\blacktriangleright:0,\gamma(c(h_1),i_3,i_2)}$
cond/lin
T_γ
$[c(h_1)]^i_{?:1,\gamma(c(h_1),i_3,i_2)}$
$[i_3]^i_{\blacktriangleright:1,\gamma(c(h_1),i_3,i_2)}$
$[i_2]^i_{\blacktriangleright:1,\gamma(c(h_1),i_3,i_2)}$
subs
lin
lin
$T_{a(s)}$
$[c]^i_{?:1,\gamma(c(h_1),i_3,i_2)}$
$[h_1]^i_{?:2,\gamma(c(h_1),i_3,i_2)}$
$[i_3 = i_2 + 1]$
$[i_2 = \mu(i_1, i_4)]$
inv
lin
lin
lin
$[SCC_1^S(h_1)]$
$[i_2 + 1]^i_{\blacktriangleright:0,i_2+1}$
lin
lin
$[\mathrm{DO}\ h_1 = 1, n, 1]$
$[i_2]^i_{\blacktriangleright:1,i_2+1}$
$[1]^i_{\blacktriangleright:1,i_2+1}$
T_+
lin
lin
inv
T_{DO}
$[1]^h_{\blacktriangleright:1,1}$
$[n]^h_{\blacktriangleright:1,n}$
$[1]^h_{\blacktriangleright:1,1}$
$[i_2 = \mu(i_1, i_4)]$
inv
inv
inv
lin
inv
inv
inv
inv
inv
lin
inv
lin

Figure 5: Classification of the components $SCC_1^S(i_{2...4})$ and $SCC_1^S(h_1)$ of the SCC graph shown in Fig. 3(b).

the transfer function of the identifier of a scalar/array variable (generically denoted as y):

$$T_y \quad : \quad [y]^{e_{ref}}_{p:l,E} =$$

$$= \begin{cases} [stm_{def}] & \text{if } stm_{def} \in SCC(x_{1...n}) \\ & \text{and } stm_{def} \notin stack_{stm} \\ lin & \text{if } stm_{def} \in SCC(x_{1...n}) \\ & \text{and } stm_{def} \in stack_{stm} \\ & \text{and } stm_{def} \text{ is a } \mu\text{-statement} \\ & \text{and } y \text{ is a scalar variable} \\ & \text{and } stm_{use} \text{ is a scalar} \\ & \quad \text{or an } \alpha\text{-statement} \\ [SCC(y_{1...m})] & \text{if } stm_{def} \notin SCC(x_{1...n}) \\ & \text{and } stm_{def} \in SCC(y_{1...m}) \\ & \text{and } stm_{def} \text{ is a scalar, } \alpha \\ & \quad \text{or } \gamma\text{-statement} \\ & \text{and } [SCC(y_{1...m})] \neq unk \\ & \text{and } SCC(y_{1...m}) \text{ is scalar} \\ & \text{and } l > 0 \end{cases} \qquad (2)$$

where $SCC(x_{1...n})$ is the component whose classification is in progress (i.e. $SCC_1^S(i_{2...4})$); stm_{def} and stm_{use} are the definition and use statements of the variable within the loop body (in our case study, $i_4 = \gamma(c(h_1), i_3, i_2)$ and $i_2 = \mu(i_1, i_4)$, respectively); $SCC(y_{1...m})$ represents the SCC where y is defined; and $stack_{stm}$ is the stack of statements already visited in $SCC(x_{1...n})$. As $i_4 = \gamma(c(h_1), i_3, i_2)$ is a statement of $SCC_1^S(i_{2...4})$ that has not been visited yet, it is not included in $stack_{stm}$. Thus, according to the first entry of Eq. (2), the computation of $[SCC_1^S(i_{2...4})]$ is reduced to determining the class of the definition statement $[i_4 = \gamma(c(h_1), i_3, i_2)]$.

The classification of $i_4 = \gamma(c(h_1), i_3, i_2)$ continues with the analysis of the right-hand side γ expression and, later, the analysis of the subexpressions $c(h_1)$, i_3 and i_2. Let us focus on the classification of $c(h_1)$. The array reference contains two subexpressions c and h_1 that are classified before applying the transfer function $T_{a(s)}$:

$$T_{a(s)} \quad : \quad [a(s)]^y_{p:l,E} =$$

$$= \begin{cases} unk & \text{if } [s]^y_{p:(l+1),E} = unk \\ inv & \text{if } [a]^y_{p:l,E} = [s]^y_{p:(l+1),E} = inv \\ list & \text{if } [a]^y_{p:l,E} = inv, \text{ and } [s]^y_{p:(l+1),E} = list \\ subs & \text{otherwise} \end{cases} \qquad (3)$$

where $a(s)$ is the array reference target for classification, y represents a reference expression that consists of a scalar variable, and the class *unk* (abbreviation of *unknown*) denotes a non-recognized scalar kernel. First, $[c]^i_{?:1,\gamma(c(h_1),i_3,i_2)}$ is set to *inv* by applying the first entry of T_K in Eq. (1).

The computation of $[h_1]^i_{?:2,\gamma(c(h_1),i_3,i_2)}$ deserves special mention because, as h_1 is defined in a different component $SCC_1^S(h_1)$, the recursive step of the SCC classification algorithm is applied (see the third entry of T_y in Eq. (2)). Thus, the classification of $SCC_1^S(i_{2...4})$ is deferred, it is pushed onto the stack of SCCs, and the classification of $SCC_1^S(h_1)$

is started on demand. The details of the computation of $[SCC_1^S(h_1)]$ are depicted inside boxes in the trees of Fig. 5. As $SCC_1^S(h_1)$ is independent (DO $h_1 = 1, n, 1$ contains no occurrences of the variables associated with other SCCs), $[SCC_1^S(h_1)]$ is assigned the value *lin* according to the third entry of the transfer function for loop header statements, which is presented below:

$$T_{\mathrm{DO}} \quad : \quad [do\, v = e_{init}, e_{limit}, e_{step}] =$$
$$= \begin{cases} unk & \text{if } [e_{init}]^{v}_{\blacktriangleright:1,e_{init}} = unk \text{ or} \\ & [e_{limit}]^{v}_{\blacktriangleright:1,e_{limit}} = unk \text{ or} \\ & [e_{step}]^{v}_{\blacktriangleright:1,e_{step}} = unk \\ subs & \text{if } [e_{init}]^{v}_{\blacktriangleright:1,e_{init}} = subs \text{ or} \\ & [e_{limit}]^{v}_{\blacktriangleright:1,e_{limit}} = subs \text{ or} \\ & [e_{step}]^{v}_{\blacktriangleright:1,e_{step}} = subs \\ lin & \text{otherwise} \end{cases} \tag{4}$$

Next, $SCC_1^S(i_{2\ldots4})$ is popped from the stack and its classification process continues at the same point where it had been deferred. Thus, the fourth entry of Eq. (3) is applied and $[c(h_1)]^{i}_{?:1,\gamma(c(h_1),i_3,i_2)}$ is set to the class *subs*.

The execution of the SCC classification algorithm continues in a similar manner. At the end, after applying more transfer functions (see T_+, T_γ or T_μ in Fig. 5), the class *cond/lin* that represents the *conditional linear induction variable* kernel is derived for $[SCC_1^S(i_{2\ldots4})]$. When the classification of $SCC_1^S(i_{2\ldots4})$ finishes, two components of the loop have been successfully classified: $SCC_1^S(i_{2\ldots4})$ and $SCC_1^S(h_1)$. Due to space limitations, the details about the classification of the remaining components, $[SCC_0^S(tmp_{1,3})]$, $[SCC_1^A(a_{1\ldots3})]$ and $[SCC_1^S(tmp_2)]$, are not presented. The results are shown in the SCC graph of Fig. 3(b).

The information gathered from the source code during the execution of the SCC classification algorithm is represented in the SCC graph. The nodes show the SCCs and the SCC classes. The edges are the use-def chains between SCCs, which are found when the classification process of a new SCC is launched on demand. The label of a use-def chain is determined as follows:

$$label = \begin{cases} E' & \text{if } p = ? \\ lhs\_index : E' & \text{if } p = \blacktriangleleft,\ \textit{subscript-level} > 0 \\ rhs\_index : E' & \text{if } p = \blacktriangleright,\ \textit{subscript-level} > 0 \\ rhs : E' & \text{if } p = \blacktriangleright,\ \textit{subscript-level} = 0 \end{cases} \tag{5}$$

where E' is the scalar/array reference of minimum level within E that contains the occurrence; the term *subscript-level* represents the indirection level of the occurrence within E, i.e., the number of array references that there are in the path from the root node of E. Furthermore, the different classes of use-def chains are distinguished at this moment. The chain $SCC_0^S(tmp_{1,3}) \Rightarrow SCC_1^S(tmp_2)$ is structural (solid edges in Fig. 3(b)) because it captures a dependence between two scalar SCCs that are associated with the same variable in the source code, namely, *tmp*. Furthermore, the components $SCC_1^A(a_{1\ldots3})$ and $SCC_1^S(i_{2\ldots4})$ belong to the classes *cond/assig/lin* and *cond/lin*, respectively. As the class of the index expression and that of the scalar variable coincide (*lin*), the use-def chain $SCC_1^A(a_{1\ldots3}) \Rightarrow SCC_1^S(i_{2\ldots4})$ is classified as structural. The loop contains three SCCs that include a γ-statement: $SCC_1^S(i_{2\ldots4})$, $SCC_0^S(tmp_{1,3})$ and $SCC_1^A(a_{1\ldots3})$. In all the cases, the conditional expression is $c(h_1)$. Thus, the SCC graph contains three control chains (dotted edges) whose target SCC is $SCC_1^S(h_1)$: $SCC_1^S(i_{2\ldots4}) \rightsquigarrow SCC_1^S(h_1)$, $SCC_0^S(tmp_{1,3}) \rightsquigarrow SCC_1^S(h_1)$, and $SCC_1^A(a_{1\ldots3}) \rightsquigarrow SCC_1^S(h_1)$. Finally, the use-def chains represented by the notation $SCC_1^S(tmp_2) \not\Rightarrow SCC_1^S(h_1)$ and $SCC_1^A(a_{1\ldots3}) \not\Rightarrow SCC_1^S(tmp_2)$ are determined to be non-structural (dashed edges in Fig. 3(b)) because they do not fulfill the conditions stated above for structural and control chains. In the following section, an algorithm for the analysis of the SCC graph is presented.

4. RECOGNITION OF LOOP-LEVEL KERNELS

The recognition of loop-level kernels involves two main tasks. On the one hand, the analysis of the SCC graph in order to isolate the set of kernels computed in a loop nest. And, on the other hand, the actual recognition of the different kernels. Section 4.1 presents our SCC graph classification algorithm, which carries out both tasks in a unique traversal of the graph. Section 4.2 describes the algorithm in the scope of our case study.

4.1 SCC Graph Classification Algorithm

The analysis of the SCC graph is performed as follows. For each connected subgraph that appears in the SCC graph, a demand-driven classification algorithm starts from each *non-wrap-around source node* (abbreviated as *NWSN*). The wrap-around source nodes are not considered because they are associated with SCCs that consist of μ-statements only, and thus they do not correspond with any statement of the source code of the loop. The procedure for the classification of NWSN subgraphs (i.e. the subgraph composed of the set of nodes and edges that are reachable from a NWSN) is the core of the SCC graph classification algorithm. It basically consists of a post-order traversal of the NWSN subgraph. When a node $SCC(x_{1\ldots n})$ is visited, the successors in the SCC graph that are reached through structural, control and non-structural use-def chains are classified in that order. The structural chains are analyzed first because they capture the typical scenarios for the computation of the kernels, and thus they serve as a guide for the execution of additional checks. When a loop-level kernel is detected, the corresponding class is added to the NWSN subgraph class, i.e. the set of loop-level kernel classes associated with the NWSN subgraph. Next, control and non-structural chains are analyzed in order to gather additional information from the source code of the loop nest. When the classification of a successor finishes, a transfer function determines the loop-level kernel represented by the target and source SCCs of the corresponding use-def chain. The classification of the NWSN subgraph fails if the class of the target SCC or the class of the source SCC are *unknown*. In that case, the NWSN subgraph class is set to *unknown*, and the classification process of inner loops starts. It should be noted that the nodes are visited only once. If a target SCC has already been visited, the transfer function is applied directly. At the end of the NWSN classification algorithm, the class of the SCC of the NWSN is added to the NWSN subgraph class.

The final goal is to determine the set of loop-level kernels computed in the loop nest. It is accomplished through a transfer function that basically makes the union of the classes of the NWSN subgraphs included in the SCC graph.

4.2 Case Study

The SCC graph of our example, shown in Fig. 3(b), contains two NWSNs: $SCC_1^A(a_{1...3})$ and $SCC_0^S(tmp_{1,3})$. Assume that the NWSN subgraph associated with $SCC_1^A(a_{1...3})$ is classified first. The demand-driven algorithm visits the node associated with $SCC_1^S(i_{2...4})$ through the structural chain $SCC_1^A(a_{1...3}) \Rightarrow SCC_1^S(i_{2...4})$, and begins with the classification of the control chain $SCC_1^S(i_{2...4}) \rightsquigarrow SCC_1^S(h_1)$. As the target component $SCC_1^S(h_1)$ is independent, the demand-driven algorithm addresses the analysis of the control chain. The corresponding transfer function does not find a loop-level kernel because $SCC_1^S(h_1)$ represents the computation of the temporary variable h_1 associated with the loop index. Consequently, the analysis of the structural use-def chain $SCC_1^A(a_{1...3}) \Rightarrow SCC_1^S(i_{2...4})$ is accomplished. As the class of the source array SCC is *cond/assig/lin*, the class of the target scalar SCC is *cond/lin*, and the occurrence of i appears in the left-hand side subscript of the statement $a(i) = tmp + 2$ (which is represented as *lhs_index* in the label of the edge), the transfer function of structural chains identifies the typical scenario for the computation of a consecutively written array kernel. Next, further checks are performed in order to confirm the existence of the kernel. For this purpose, we have used the approach of [10]:

1. All the operations on i are increments (or decrements) of one unit.

2. Every time an array entry $a(i)$ is written, the corresponding induction variable i is updated. This task is accomplished through the analysis of the control flow graph (CFG) of the loop.

At this moment, the compiler recognizes the consecutively written array kernel and adds the class *cond/cwa* to the NWSN subgraph class. The demand-driven algorithm continues the analysis of the remaining edges and nodes of the NWSN subgraph. However, no more loop-level kernels are detected because the target SCCs of the use-def chains represent the computation of temporary variables.

The SCC graph contains another NWSN subgraph that starts from $SCC_0^S(tmp_{1,3})$. However, no loop-level kernel is detected (i.e. the NWSN subgraph class is empty) because the NWSN $SCC_0^S(tmp_{1,3})$ consists only of μ and γ statements that capture the flow of values of the temporary scalar *tmp*, which does not provide the compiler with relevant information from the point of view of kernel recognition. Finally, the union of the NWSN subgraph classes (the class *cond/cwa* and the empty class) determines that the loop computes a consecutively written array.

5. ANALYSIS OF COMPLEX LOOP NESTS

More complex computational kernels can be found in full-scale programs. In order to show the potential of our GSA-based compiler infrastructure, we have selected the source code of Fig. 6. The example corresponds to a routine that 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 mask matrix in sparse format ($imask$, $jmask$). In this section, the detection of the loop-level kernels is described with no previous assumption and independently of the semantics of this code.

Consider the GSA form presented in Fig. 7. The code consists of two outer loops. In the first loop, do_{j_1}, the initialization of the temporary array iw to the logical constant value $false$ is performed. The second loop nest, do_{ii_1}, is mainly devoted to the computation of two conditional consecutively written arrays c and jc, the scalar len being the corresponding conditional induction variable. The complexity of this example comes from the use of the variable iw to control what loop iterations of do_{k_3} actually compute the array entries $jc(len)$ and $c(len)$. This mechanism is implemented with two inner loops, do_{k_2} and do_{k_4}, which are executed, respectively, at the beginning and at the end of each do_{ii_1} iteration. In the first loop do_{k_2}, the array elements to be processed are marked. The value of iw is tested in do_{k_3}. If the mark is set, the corresponding elements $c(len)$ and $jc(len)$ are calculated. Otherwise, no processing is performed. Finally, the marks are unset in the second loop do_{k_4}.

Let us focus on the recognition of loop-level kernels. Consider the SCC graph of the level-2 loop do_{ii_1} depicted in Fig. 8. It contains three NWSNs: $SCC_1^A(jc_{1...4})$, $SCC_1^A(c_{1...4})$ and $SCC_1^A(ic_{1,2})$. As the classification of the NWSN subgraphs associated with the NWSN nodes $SCC_1^A(jc_{1...4})$ and $SCC_1^A(c_{1...4})$ are very similar, we will focus on the detection of the consecutively written array jc. The section concludes with the classification of the NWSN subgraph corresponding to $SCC_1^A(ic_{1,2})$.

The demand-driven classification of the NWSN subgraph of $SCC_1^A(jc_{1...4})$ begins with the non-structural use-def chains $SCC_1^S(k_2) \nRightarrow SCC_1^S(ii_1)$ and $SCC_1^S(k_4) \nRightarrow SCC_1^S(ii_1)$. As $SCC_1^S(ii_1)$ is associated with the temporary loop index variable, the algorithm continues with the analysis of the structural chains whose source component is $SCC_1^A(iw_{3...7})$. This SCC captures the flow of values of iw and, thus, enables the recognition of the masking mechanism by the compiler. It is accomplished by checking the following properties:

1. The class $[SCC_1^A(iw_{3...7})]$ is *non-cond/assig/subs*, with two structural chains $SCC_1^A(iw_{3...7}) \Rightarrow SCC_1^S(k_2)$ and $SCC_1^A(iw_{3...7}) \Rightarrow SCC_1^S(k_4)$.

2. $SCC_1^A(iw_{3...7})$ contains two α-statements that belong to the body of the loops do_{k_2} and do_{k_4}.

3. The same elements of the auxiliary array iw are written in do_{k_2} and do_{k_4}. This constraint is tested as follows:

   (a) Check that the iteration spaces are equal. This can be assured by proving that the init, limit and step expressions of do_{k_2} and do_{k_4} are pair-wise syntactically identical, so that both loop indices take the same values. In the GSA form of Fig. 7, the init, limit and step expressions are, respectively, $imask(ii_1)$, $imask(ii_1 + 1) - 1$ and 1 in both loops.

   (b) Check that the left-hand side subscript expressions of the α-statements also take the same value during the execution of do_{ii_1}. This condition is fulfilled because both expressions, $jmask(k_2)$ and $jmask(k_4)$, are syntactically identical except for the occurrences of the index variables k_2 and k_4. However, as shown above, k_2 and k_4 also take the same values.

```
DO $j = 1, ncol$
   $iw(j) = false$
END DO

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
```

Figure 6: Source code that filters the contents of a sparse matrix using a mask matrix (extracted from the *SparsKit-II* library, module *unary*, subroutine *amask*).

```
DO $j_1 = 1, ncol, 1$
   $iw_1 = \mu(iw_0, iw_2)$
   $iw_2 = \alpha(iw_1, j_1, false)$
END DO

DO $ii_1 = 1, nrow, 1$
   $iw_3 = \mu(iw_1, iw_6)$
   $len_2 = \mu(len_1, len_3)$
   $k_1 = \mu(k_0, k_4)$
   $j_2 = \mu(j_1, j_3)$
   $c_1 = \mu(c_0, c_2)$
   $k1_1 = \mu(k1_0, k1_2)$
   $k2_1 = \mu(k2_0, k2_2)$
   $jc_1 = \mu(jc_0, jc_2)$
   $ic_1 = \mu(ic_0, ic_2)$
   DO $k_2 = imask(ii_1), imask(ii_1 + 1) - 1, 1$
      $iw_4 = \mu(iw_3, iw_5)$
      $iw_5 = \alpha(iw_4, jmask(k_2), true)$
   END DO
   $k1_2 = ia(ii_1)$
   $k2_2 = ia(ii_1 + 1) - 1$
   $ic_2 = \alpha(ic_1, ii_1, len_2 + 1)$
   DO $k_3 = k1_2, k2_2, 1$
      $len_3 = \mu(len_2, len_5)$
      $j_3 = \mu(j_2, j_4)$
      $c_2 = \mu(c_1, c_4)$
      $jc_2 = \mu(jc_1, jc_4)$
      $j_4 = ja(k_3)$
      IF $(iw_4(j_4))$ THEN
         $len_4 = len_3 + 1$
         $jc_3 = \alpha(jc_2, len_4, j_4)$
         $c_3 = \alpha(c_2, len_4, a(k_3))$
      END IF
      $len_5 = \gamma(iw_4(j_4), len_4, len_3)$
      $c_4 = \gamma(iw_4(j_4), c_3, c_2)$
      $jc_4 = \gamma(iw_4(j_4), jc_3, jc_2)$
   END DO
   DO $k_4 = imask(ii_1), imask(ii_1 + 1) - 1, 1$
      $iw_6 = \mu(iw_4, iw_7)$
      $iw_7 = \alpha(iw_6, jmask(k_4), false)$
   END DO
END DO
```

Figure 7: GSA form of the loop shown in Fig. 6.

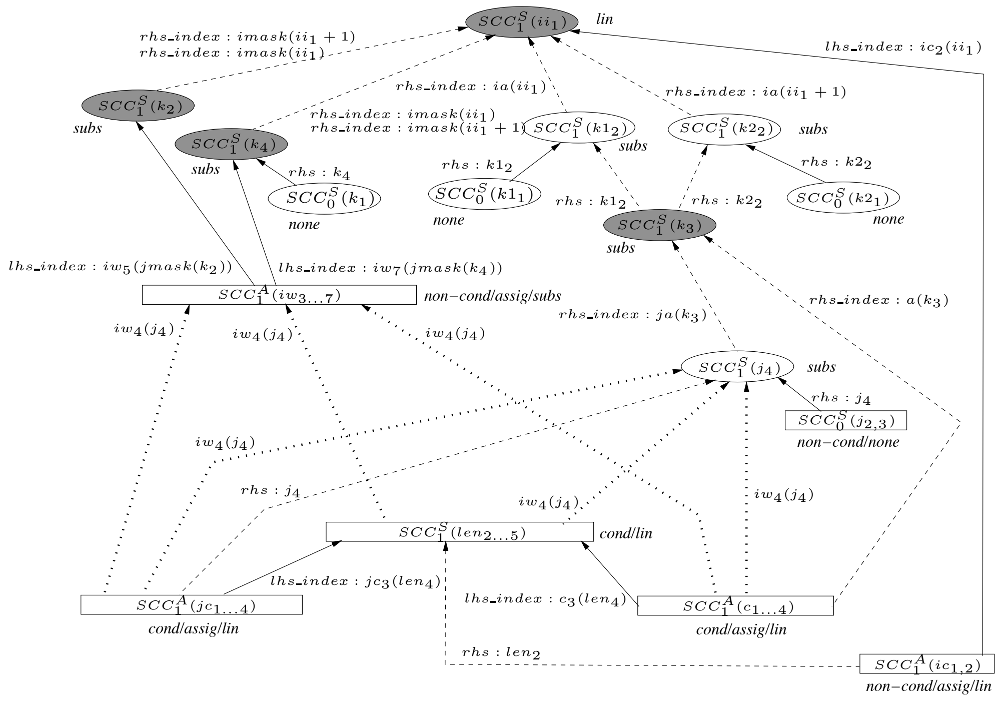


Figure 8: SCC graph of the code presented in Fig. 6.

4. The temporary array iw is initialized in do_{j_1} (value $false$ of iw_2).

5. The elements of array iw are set to a value different from $false$ in do_{k_2} ($true$ of iw_5), and those marks are deleted in do_{k_4} ($false$ of iw_7).

After the successful detection, the demand-driven algorithm addresses the control chain $SCC_1^S(len_{2...5}) \rightsquigarrow SCC_1^A(iw_{3...7})$. It does not enable neither kernel separation nor kernel classification. However, it is relevant because it enables the identification of those kernels whose execution is controlled by the contents of the temporary array iw (in the example, the *cond/lin* and *cond/assig/lin* basic kernels associated with len and jc, respectively). The compiler performs the following checks during the execution of the corresponding transfer function:

1. The conditional expression $iw_4(j_4)$ consists of an equality comparison between the value of the auxiliary array iw and the value of the marks ($true$ in iw_5).

2. The loop do_{k_2} precedes do_{k_3} in the CFG, and do_{k_3} precedes do_{k_4} in the CFG.

In the next step, the consecutively written array jc is detected as explained in Section 4.2 through the analysis of the structural chain $SCC_1^A(jc_{1...4}) \Rightarrow SCC_1^S(len_{2...5})$. As a result, the class *cond/cwa* is added to the NWSN subgraph class. Hereafter, no more loop-level kernels are recognized because the remaining nodes represent the computation of temporary scalar variables. Consequently, the class derived for the NWSN subgraph of $SCC_1^A(jc_{1...4})$ is *cond/cwa*.

Apart from the consecutively written arrays jc and c, the loop do_{h_1} also calculates a *non-cond/assig/lin* kernel using the array ic. This kernel is recognized through the classification of the NWSN subgraph associated with $SCC_1^A(ic_{1,2})$, which proceeds as follows. As the target SCC of the structural use-def chain $SCC_1^A(ic_{1,2}) \Rightarrow SCC_1^S(ii_1)$ is associated with the temporary loop index variable ii, no loop-level kernel is detected. The algorithm continues with the analysis of the non-structural chain $SCC_1^A(ic_{1,2}) \not\Rightarrow SCC_1^S(len_{2...5})$. It does not enable the detection of new kernels, but it provides useful information for the parallelization of the loop nest. In particular, it indicates that the values of the array entries $ic(ii)$, which are determined by the induction variable len, must be post-processed after the parallel execution of do_{ii} in order to preserve the sequential semantics. Finally, the classification of the NWSN subgraph finishes with the insertion of the class of the NWSN, *non-cond/assig/lin*, in the NWSN subgraph class. As a result, all the loop-level kernels that appear in the level-2 loop nest of Fig. 6 have been successfully recognized. Thus, the coarse-grain parallelism implicit in the loop has been uncovered.

6. EXPERIMENTAL RESULTS

6.1 Experimental Conditions

We have developed a prototype of approximately 30,000 lines of C++ code. The core of the prototype are the algorithms to recognize basic and loop-level computational kernels (see Fig. 1). We have used the support given by the internal representation of the Polaris compiler [4]. Polaris also provides the GSA form and the CFG of a Fortran77 source code.

Table 1: Number of complex loops detected by our approach and by Polaris.

	Detected by our approach	Also detected by Polaris
Level-1 parallel loops	33	15
non-cond/reduc	4	4
non-cond/assig/subs	12	0
cond/assig/subs	3	0
non-cond/reduc/subs	9	9
cond/reduc/subs	1	1
non-cond/cwa	1	1
cond/cwa	2	0
scalar-minimum-w/loc	1	0
Level-2 parallel loops	39	17
non-cond/assig/lin	6	1
cond/assig/lin	2	1
non-cond/assig/subs	6	1
cond/assig/subs	1	0
non-cond/reduc/lin	1	1
non-cond/reduc/subs	10	9
cond/reduc/subs	4	4
non-cond/cwa	1	0
cond/cwa	8	0
Level-4 parallel loops	1	0
non-cond/reduc/subs	1	0

Our benchmark suite is the *SparsKit-II* library [14], which consists of a set of costly routines to perform operations with sparse matrices. The routines are organized in four modules: *matvec*, that includes basic matrix-vector operations (matrix-vector products and triangular system solvers) with different types of sparse storage formats; *blassm*, which supplies basic linear algebra operations for sparse matrices (e.g. matrix-matrix products and sums); *unary*, that provides unary operations with sparse matrices (e.g. extract a submatrix from a sparse matrix and perform mask operations with matrices); and *formats*, which is devoted to format conversion routines for different types of sparse storages. This library is a representative benchmark suite because it contains most of the complex computational kernels enumerated in the introduction of the paper. Moreover, a subset of those kernels was considered relevant for the analysis of full-scale sparse applications in [10].

6.2 Recognition Results

The experiments presented in this section focus on complex loop nests because the analysis of regular computations is a well-established topic. Table 1 compares the effectiveness of our approach with the Polaris parallelizing compiler. The results corresponding to the Polaris column show the number of parallel loops detected by our approach that are also detected by Polaris. The table is organized in sets that present the results for the different nesting levels (level-1 refers to innermost loops). The first row summarizes the totals, and the subsequent rows detail the results for each loop-level kernel class. In some cases, Polaris only extracts parallelism from the inner loops of the level-2 loop nests handled by our approach.

The improvement in automatic parallelization effectiveness is approximately 56%. Most of the increase results from the recognition of (non-)conditional consecutively written arrays (denoted as *(non-)cond/cwa*), and (non-)conditional irregular assignments (class *(non-)cond/assig/subs*). Polaris cannot parallelize loops that contain *cond/cwa* kernels because it cannot compute a closed form expression for the corresponding conditional induction variable. An interesting example of potentially parallel *cond/cwa* is the level-2 loop do_{100} of the routine *rperm* of the module *unary*. It contains a conditional linear induction variable that is incremented during the execution of an inner loop, and it is reinitialized to a loop-variant expression at the beginning of each iteration in the outer loop do_{100}. This complex form of induction variable is not handled by Polaris. However, this level-2 loop can be executed in parallel with appropriate run-time support. Our infrastructure provides an adequate environment for the parallelization of this loop. The details, which are outside the scope of this paper, are shown in [1].

The recognition of irregular assignment computations enables the parallel execution of a wider set of loop nests, for instance, the level-2 loops do_1 of the routines *diamua* and *amudia* of the module *blassm*. However, the detection of this kernel is even more relevant because it provides support for the recognition of variations of other kernels. The detection of the *cond/cwa* kernel computed in the level-2 loop nest analyzed in Section 5 (loop do_{100} of the routine *amask* of *unary*) illustrates this issue.

The *SparsKit-II* library includes some complex loop nests that contain other interesting computational kernels. Table 1 shows the existence of a *scalar-minimum-w/loc* semantic kernel (level-1 loop do_1 of the routine *blkfnd* of module *unary*). The class *scalar-minimum-w/loc* represents the computation of both the minimum value of a set of values, and the position of that value within the set. Polaris also fails to extract coarse-grain parallelism from two loop nests that compute the well-known irregular reduction kernel (captured by the class *non-cond/reduc/subs*). The most interesting case is a level-4 loop that appears in the routine *vbrmv* of the module *matvec*, which includes a non-conditional linear induction variable whose closed form expression cannot be computed by Polaris.

7. RELATED WORK

Several approaches have been proposed for the automatic extraction of loop-level parallelism. Automatic program comprehension addresses the automatic parallelization of sequential codes by taking into account the semantics of the code. Keßler [8] proposes a speculative program comprehension method for a set of computational kernels with sparse vectors and matrices. The code is analyzed with the aim of recognizing syntactical variations that are frequently used in full-scale applications. Program comprehension enables aggressive code transformations such as local algorithm replacement using available parallel algorithms and machine-specific library routines. The information could also be used as a guide for applying optimizing transformations tuned for each sparse kernel specifically. However, unlike the approach presented in this paper, the scope of application is limited because the semantics of the code is considered.

Gerlek, Stoltz and Wolfe [6] describe a method that addresses the recognition of integer-valued scalar computational kernels even in the presence of complex control constructs. The technique basically consists of a classification scheme that recognizes the type of kernel calculated in the statements of the SCCs that appear in the use-def chain graph of the SSA form. The scheme can only handle expressions that involve scalar integer-valued variables. Thus, expressions with array references, which appear very often in real applications, are outside the scope of that technique. In contrast, our technique based on the GSA form supports not only integer-valued scalar expressions, but also floating-point-valued expressions and references to array variables. In particular, the array references with subscripted subscript expressions that characterize irregular codes can be analyzed. This extension could seem to be of little significance. However, the spectrum of computational kernels that can be detected by the compiler is extended considerably.

Previous works on detection of parallelism in irregular codes addressed the problem of recognizing specific and isolated kernels (usually using pattern-matching to analyze the source code), for instance, irregular reductions [12] or irregular assignments [9]. In this paper we presented a compiler infrastructure that enables the detection of a wide range of structural and semantic kernels in a unified manner. The recognition of semantic kernels also allows the parallelization of a wider set of complex loops. Furthermore, unlike classic source code pattern-matching, our approach can be easily extended to recognize new kernels by checking additional characteristics during the execution of the transfer functions.

Suganuma et al. [15] present a technique limited to the detection of complex scalar reduction constructs, including semantic reductions, which are also detected by our approach. As in [6], the approach is based on the analysis of the strongly connected components of a dependence graph. However, the graph is constructed directly from the source code of the program. Thus, it does not take advantage of the information about the flow of values provided by intermediate representations such as SSA or GSA.

Complex loops usually contain induction variables that introduce loop-carried dependences at run-time. In classical dependence analysis, the occurrences of non-conditional induction variables are substituted for closed form expressions in order to remove the loop-carried dependences introduced by these variables. Different techniques have been proposed for the computation of closed form expressions [6, 12, 22]. In general, the approach described above cannot be applied to loops with conditional induction variables. Wolfe [17] and Wu et al. [19] address the problem by analyzing how the induction variable changes during the execution of a loop. The study focuses on determining whether the induction variable is (strictly) increasing or decreasing. This information is later used in dependence tests for discarding the existence of loop-carried dependences in the references to array variables. Our compiler infrastructure is compatible with the techniques mentioned above. In fact, it can be used to provide information to decide which is the most appropriate technique in each case, and apply these techniques on demand.

8. CONCLUSIONS

This paper has addressed the automatic detection of coarse-grain parallelism in complex loop nests, as the compiler technology for the analysis of regular codes is well-established. We have proposed a GSA-based compiler infrastructure that enables the recognition of a wide variety of computational kernels independently of the semantics and the quality of the code. Unlike other detection methods that focus on partial aspects, our scheme is general, handles scalar/array and structural/semantic kernels in a unified manner, and is easily extensible. Furthermore, the case studies presented in this paper show the potential of our infrastructure for the recognition of complex variations of the kernels. The experiments have shown that our infrastructure enables the detection of coarse-grain parallelism where the Polaris parallelizing compiler fails.

As future work, we intend to measure the effectiveness of our kernel recognition scheme in the analysis of complex loop nests included in other representative benchmark suites. We believe that this study would lead to the identification of new computational kernels, and to the development of new parallelizing transformations.

9. REFERENCES

[1] M. Arenaz. *Compiler Framework for the Automatic Detection of Loop-Level Parallelism*. PhD thesis, Department of Electronics and Systems, University of A Coruña, Spain, Mar. 2003. Available at `www.des.udc.es/~juan/publicationsjuan.html`.

[2] M. Arenaz, J. Touriño, and R. Doallo. A compiler framework to detect parallelism in irregular codes. In *14th International Workshop on Languages and Compilers for Parallel Computing, LCPC 2001*, Cumberland Falls, KY, Aug. 2001.

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

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

[5] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[6] M. P. 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, Jan. 1995.

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

[8] C. W. Keßler. Applicability of program comprehension to sparse matrix computations. In *3rd International 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. A. Padua. On the automatic parallelization of sparse and irregular Fortran programs. In *4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers, LCR'98*, pages 41–56, Pittsburgh, PA, May 1998.

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

[12] W. M. Pottenger and R. Eigenmann. Idiom recognition in the Polaris parallelizing compiler. In *9th ACM International Conference on Supercomputing, ICS'95*, pages 444–448, Barcelona, Spain, July 1995.

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

[14] Y. Saad. *SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations*. Available at `www-users.cs.umn.edu/~saad/software/SPARS-KIT/sparskit.html`.

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

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

[17] M. Wolfe. Beyond induction variables. In *1992 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'92*, pages 162–174, San Francisco, CA, June 1992.

[18] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[19] P. Wu, A. Cohen, J. Hoeflinger, and D. A. Padua. Monotonic evolution: An alternative to induction variable substitution for dependence analysis. In *15th ACM International 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, May 2001.

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

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