DEPARTMENT OF ELECTRONICS AND SYSTEMS
UNIVERSITY OF A CORUÑA, SPAIN

PhD THESIS

Compiler Framework for the Automatic Detection of Loop-Level Parallelism

Author:
Manuel Arenaz Silva

PhD Advisors:
Dr. Juan Touriño Domínguez
Dr. Ramón Doallo Biempica

A Coruña, January 2003

Dr. Juan Touriño Domínguez
Profesor Titular de Universidad
Dpto. de Electrónica y Sistemas
Universidad de A Coruña

Dr. Ramón Doallo Biempica
Catedrático de Universidad
Dpto. de Electrónica y Sistemas
Universidad de A Coruña

CERTIFICAN

Que la memoria titulada "*Compiler Framework for the Automatic Detection of Loop-Level Parallelism*" ha sido realizada por D. Manuel Carlos Arenaz Silva bajo nuestra dirección en el Departamento de Electrónica y Sistemas de la Universidad de A Coruña y concluye la Tesis Doctoral que presenta para optar al grado de Doctor en Informática.

A Coruña, 3 de Enero de 2003

Fdo.: Juan Touriño Domínguez
Codirector de la Tesis Doctoral

Fdo.: Ramón Doallo Biempica
Codirector de la Tesis Doctoral

Fdo.: Ramón Doallo Biempica
Director del Dpto. de Electrónica y Sistemas

To my parents, for everything.

Acknowledgments

One does not realise the considerable effort required to write a PhD Thesis until one has actually written it. It is at that moment, on looking back, that the important and essential part played by the PhD advisors can be more clearly appreciated. It is for that reason that I want to thank Dr. Ramón Doallo and Dr. Juan Touriño for directing me throughout this work.

I also want to acknowledge *Rosa Vázquez* for her patience and comprehension during the period of time that I have been writing this thesis.

I gratefully thank to the following institutions for funding this work: *Department of Electrónica y Sistemas of A Coruña* for the human and material support, *University of A Coruña* for financing my attendance at some conferences, and *Xunta de Galicia* and *Spanish Goverment* for the projects FEDER 1FD97-0118, PGIDT01-PXI10501PR and MCYT-FEDER TIC2001--3694-C02-02.

Manuel Arenaz

Resumen de la Tesis

Introducción

El desarrollo de aplicaciones eficientes para las máquinas paralelas disponibles en la actualidad es una tarea complicada que exige un gran conocimiento de las características de la máquina por parte del programador. Una aproximación que permite simplificar esta tarea consiste en proporcionar al programador una tecnología de compilación capaz de generar automáticamente código paralelo a partir de un código fuente diseñado para ser ejecutado de manera secuencial en un único procesador.

El proceso de paralelización automática de un código secuencial consta de dos etapas bien diferenciadas: detección de las partes del código que se pueden ejecutar en paralelo, y generación de código paralelo que no altere la semántica del código secuencial. En esta tesis se aborda el problema de detección en el ámbito de los códigos irregulares.

Las técnicas actuales de detección de paralelismo están basadas en el análisis de las dependencias que existen entre las sentencias del programa. Estas técnicas son efectivas en el ámbito de los códigos regulares debido a que, en este tipo de códigos, las expresiones de los índices de las referencias a arrays a menudo se pueden expresar como funciones lineales o afines de los índices de los bucles en que aparecen. Sin embargo, los códigos irregulares son todavía un reto debido a que dichas expresiones contienen referencias a arrays cuyo valor, en general, no se puede determinar en tiempo de compilación. Esta característica hace que el compilador no pueda obtener información precisa acerca de las dependencias del programa.

Este trabajo de investigación ha dado lugar a varias publicaciones en el campo de la detección automática de paralelismo [4, 7]. Los conocimientos que han permitido llevarlo a cabo fueron adquiridos a través del estudio detallado del código fuente de un conjunto de aplicaciones irregulares. Como resultado de estos estudios han surgido una serie de publicaciones en el ámbito de la generación automática de código paralelo, siendo los más relevantes [5, 6, 8].

Metodología Utilizada

En esta tesis presentamos una técnica de detección de paralelismo a nivel de bucle orientada, principalmente, al análisis de códigos irregulares. La metodología utilizada consta de dos partes bien diferenciadas. La primera parte consiste en la construcción de un entorno de compilación capaz de extraer del código fuente la información relativa a los tipos de cálculos (en adelante llamados *kernels*) que se realizan durante la ejecución de un bucle. En la segunda parte se analiza la información recabada por el entorno de compilación con el fin de identificar aquellos bucles que es posible ejecutar en paralelo. Los detalles acerca de cada una de estas partes se describen a continuación.

La detección automática de paralelismo requiere tener un conocimiento preciso acerca del flujo de valores que se producirá entre las variables de un programa durante la ejecución del mismo. El entorno de compilación propuesto está basado en la representación *Gated Single Assignment (GSA)* de los programas, la cual permite analizar sintácticamente el flujo de valores. La construcción del entorno consta de los siguientes pasos:

1. *Traducción del código fuente del programa a la forma GSA.*

2. *Reconocimiento de los kernels básicos que se calculan durante la ejecución del bucle.*

   El objetivo que se persigue en esta fase es descomponer el cuerpo del bucle en un conjunto de kernels más sencillos e identificar las dependencias que existen entre ellos. El procedimiento empleado es el siguiente:

   (a) *Búsqueda de las componentes fuertemente conexas (SCCs) que contiene el grafo de dependencias de la representación GSA.*

   (b) *Reconocimiento del kernel básico asociado a cada SCC.* Esta tarea se lleva a cabo mediante un algoritmo de clasificación de SCCs que analiza de forma exhaustiva todas las operaciones que se realizan en las sentencias de la SCC. Este algoritmo permite reconocer una gran variedad de kernels básicos, como por ejemplo, formas complejas de variables de inducción, operaciones de reducción con variables escalares y array (incluidas operaciones de tipo mínimo/máximo), asignaciones irregulares o reducciones irregulares.

(c) *Clasificación de las dependencias entre SCCs.* La búsqueda de las dependencias se realiza durante la ejecución del algoritmo de clasificación de SCCs aprovechando el análisis exhaustivo de todas las expresiones que componen el cuerpo del bucle. En esta fase se distinguen varios tipos de dependencias con el fin de identificar las más relevantes desde el punto de vista del reconocimiento de kernels más complejos que los representados mediante SCCs.

Toda la información obtenida durante la ejecución de las fases (b) y (c) se representa en una estructura de datos que denominamos *grafo de SCCs*, la cual se utilizará como soporte para las etapas posteriores de construcción del entorno de compilación.

3. *Reconocimiento del conjunto de kernels que se calculan en el bucle.*

   En general, los cálculos que se realizan en un bucle se pueden representar bien como kernels básicos bien como kernels más complejos que son el resultado de combinar varios kernels básicos. Para la realización de esta tarea hemos diseñado un algoritmo que está basado en el análisis del grafo de SCCs. Algunos ejemplos de nuevos tipos de cálculos reconocidos mediante este algoritmo son mínimo/máximo con posición o *consecutively written array*.

Una vez finalizada la construcción del entorno de compilación, el compilador dispone de información acerca del conjunto de kernels que se calculan durante la ejecución de un bucle. En la literatura se han propuesto una gran variedad de métodos que permiten transformar el código secuencial de algunos tipos de kernels irregulares en un código paralelo equivalente. La idea básica de nuestra técnica de detección automática de paralelismo consiste en utilizar la información proporcionada por el entorno para determinar cuales de dichas transformaciones se pueden aplicar en cada caso. Nótese que con esta aproximación el compilador utiliza las transformaciones de código como soporte para asegurar que, en tiempo de ejecución, las dependencias propias de cada kernel se respetan y, de esta manera, se preserva la semántica del código secuencial.

Conclusiones

En esta tesis se han abordado, principalmente, dos temas relacionados con el análisis automático de los programas orientado a la extracción de paralelismo del código fuente. Por una parte, hemos propuesto una técnica de compilación que permite reconocer una gran variedad de kernels mediante el análisis de anidamientos de bucles que contienen computación de tipo regular e irregular. Por otra parte, hemos descrito la manera de utilizar nuestro esquema de reconocimiento como un entorno de compilación que hace posible la detección automática de paralelismo en códigos irregulares.

Las principales aportaciones de la tesis se pueden resumir en los siguientes puntos:

- Hemos propuesto un algoritmo de clasificación de SCCs basado en la forma GSA de los programas que permite reconocer una gran variedad de kernels básicos tanto de computación regular como irregular.
- Hemos propuesto un algoritmo de clasificación del grafo de SCCs de un bucle. Este algoritmo está orientado al reconocimiento del conjunto de kernels que se calculan durante la ejecución del bucle. Estos kernels pueden consistir tanto en kernels básicos como en kernels más complejos.
- Hemos demostrado que los algoritmos descritos en los apartados anteriores pueden ser utilizados como potentes herramientas de recopilación de información que proporcionan al compilador el soporte necesario para la implementación de otras técnicas de compilación. En particular, hemos descrito como aplicar este entorno de compilación en el ámbito de la detección automática de paralelismo en bucles irregulares. Las principales características de nuestra técnica de detección son:
  - Detección, en tiempo de compilación, de kernels de computación irregular que son paralelizables con la ayuda de soporte en tiempo de ejecución para preservar las dependencias del bucle. El soporte lo proporcionan las técnicas de transformación de código orientadas a la paralelización de bucles con computación irregular.
  - Detección de paralelismo en un amplio abanico de kernels de características muy dispares, incluso en bucles que contienen estructuras de control complejas.

- Hemos comparado la efectividad de nuestro método de detección con el compilador paralelizador Polaris. Hemos presentado resultados experimentales que demuestran que nuestro método es capaz de extraer paralelismo en bucles irregulares en los que Polaris no tiene éxito.

Contents

List of Tables

List of Figures

Frequently Used Terms

Term	*Meaning*
x_k	Identifier of a scalar or array variable
v	Identifier of a scalar variable.
a	Identifier of an array variable.
$a(s)$	Reference to an element of an array variable.
s	Subscript expression of an array reference.
tmp	Temporary scalar variable.
e	Expression.
K	Constant expression.
$e_1 \overset{GSA}{\equiv} e_2$	GSA equivalent expressions e_1 and e_2.
rhs	Right-hand side expression of a statement.
ϕ	Special operator of SSA-based program forms.
μ	ϕ associated with loop headers.
γ	ϕ associated with if-endif construct.
α	ϕ associated with array assignment statements.
stm	Statement of a program.
do_h	Loop whose index variable is the scalar h. Sometimes h will denote the label of a Fortran77 loop.
$< x_1, \ldots, x_n >$	Cycle composed of the statements that define the variables $x_1, \ldots, x_n$ in the GSA form of a loop.
$SCC(x_{1\ldots n})$	Strongly connected component composed of the statements that define the variables $x_1, \ldots, x_n$ in the GSA form of a loop.
$SCC_C^S(x_{1\ldots n})$	Scalar strongly connected component of cardinality C.

Term	*Meaning*
$SCC^A_C(x_{1...n})$	Array strongly connected component of cardinality C.
$\|SCC(x_{1...n})\|$	Cardinality of the component $SCC(x_{1...n})$
$SCC(x_{1...n}) \rightarrow SCC(y_{1...m})$	Direct use-def chain between two components $SCC(x_{1...n})$ and $SCC(y_{1...m})$.
$SCC(x_{1...n}) \xrightarrow{+} SCC(y_{1...m})$	Indirect use-def chain.
$SCC(x_{1...n}) \Rightarrow SCC(y_{1...m})$	Structural use-def chain.
$SCC(x_{1...n}) \nRightarrow SCC(y_{1...m})$	Non-structural use-def chain.
$SCC(x_{1...n}) \rightsquigarrow SCC(y_{1...m})$	Control use-def chain.
$[e]^{e_{ref}}_{p:d,E}$	Contextual class of an expression e. The abbreviation $[e]$ is also used.
$[stm]$	Class of a statement stm.
$[SCC(x_{1...n})]$	Class of a component $SCC(x_{1...n})$.
T_e	Transfer function of an expression e.
T_{stm}	Transfer function of a statement stm.
$SCC(x_{1...n}) \sqsubseteq do_h$	SCC contained in the do_h.
$SCC(x_{1...n}) \sqsubset do_h$	SCC exclusively contained in the do_h.

Chapter 1

Introduction

One of the major reasons for the widespread acceptance of computers, both in academia and industry, is the existence of powerful tools that contribute to improve the productivity of frequent tedious tasks. The design and implementation of computer applications is greatly influenced by the programming language used by the software developer. High-level languages provide better support for human understanding and, thus, ease the programming task and increase the productivity of software developers. However, the executable programs generated from these languages are not usually as efficient as expected. In an effort to improve the quality of executable code, programmers have learnt how to optimize code to get more performance out of individual processors and memory hierarchy. On the other hand, advances in computing power have come with architectural innovations that make the machine more difficult to program. As a result, programmers that develop applications for todays parallel systems have also learned to explicitly manage the increasing complexity of memory hierarchies and the mechanisms for interprocessor communication.

It is not a recent observation that compiler technology plays an important role in the generation of executable code that is efficiently targeted for present parallel machines. If compilers could provide efficient language implementations for architectural innovations, programmers would not have to worry about the characteristics of the underlying architecture. There are two major approaches for the development of parallel applications, both of them with strong arguments in their favor: using parallel programming languages, and transforming sequential programs into a parallel counterpart. This thesis focuses on the latter approach; it concentrates on the development of compiler support for automatically transforming sequential code to take advantage of parallelism. It should be noted that starting from a sequential language has very practical advantages. The most important one is that existing sequential software could be re-targeted for todays parallel computers with a minimum of programming effort. Moreover, many users in science and engineering who are not expert programmers could continue to use the conventional programming language they are used to working with. It would be the job of the compiler to adapt the code to perform well on the underlying parallel machine.

Research on the topic of this thesis has established the foundations for the implementation of several compilers with automatic parallelization capabilities. Well-known parallelizing compilers available today, both in industry (HP [23], SGI [43]) and academia (Polaris [11], SUIF [21]), rest on different compiler technologies that are based on a common concept: the concept of dependence analysis. The reason for this is that dependences capture a fundamental property of the algorithm coded in a program: the flow of values during the execution of the program. Parallelizing compilers uncover implicit parallelism in sequential programs by running dependence tests that determine whether the code can be executed in parallel preserving the sequential semantic [2, 9, 31, 51]. It should be noted that dependence analysis provides support for uncovering implicit parallelism, but also specifies how the execution of the program may be reordered.

Gathering information about the flow of values during the execution of a program is a critical task that the compiler has to accomplish prior to the execution of dependence tests. In the literature, methods for constructing data structures that represent this information precisely have been proposed [1, 51]. In the last decade, however, an intermediate program representation called *Static Single Assignment (SSA)* [14] that exhibits this information syntactically has become quite popular. The key idea is to rename the variables in a program according to a specific naming discipline which assures that left-hand sides of assignment statements are pairwise disjoint. As a consequence, each use of a variable is reached by one definition at most. This is a property that was shown to be useful for simplifying the analysis of the code of a program. Other intermediate representations that extend the capabilities of

SSA [12, 29, 46] have been successfully used in the scope of program analysis. SSA-based representations are used in modern parallelizing (e.g. Polaris [11]) and optimizing (e.g. GNU GCC [17]) compilers.

An important challenge for current parallelizing compilers is the analysis of programs that contain subscripted subscripts. The information provided by SSA is insufficient because the flow of values is captured for scalar variables only. Some extensions of SSA that handle array references have been proposed to address the analysis of codes of these characteristics. *Array Static Single Assignment (Array SSA)* [27] has been applied as a concrete form of parallel code generation that supports the automatic parallelization of programs [27], and to conditional constant propagation of scalar and array references [28]. On the other hand, *Gated Single Assignment (GSA)* [46] was applied to automatic privatization of variables, which was shown to be very important for improving the accuracy of dependence tests.

Current compiler technology has been successful in the parallelization of programs that contain regular computations. The reason for this is that, in regular codes, the subscript expressions of array references can often be rewritten as linear or affine functions of the index variables of the enclosing loops. However, irregular codes are still a challenge due to the presence of subscripted subscript expressions whose value cannot be determined at compile-time, and that make it impossible for the compiler to analyze dependences precisely. The automatic parallelization of irregular codes has been addressed by developing ad-hoc techniques that deal with specific computational kernels with well-defined characteristics [27, 35].

The main contributions of this thesis are:

1. A new compiler framework where loop nests are represented in an intermediate form that enables the recognition of the computational kernels calculated during the execution of the loop nest. The framework is constructed on top of the GSA form of the program.

2. The description of how the information provided by our framework enhances the efficacy of current compiler technology in the scope of automatic detection of parallelism in irregular applications.

The results of this research work in the field of automatic detection of parallelism have been published in [4, 7]. The background that allowed the development of this compiler framework was obtained by studying a set of sparse/irregular codes in detail. As a result of these studies, some research works in the scope of parallel code generation were also presented. The most relevant publications are [5, 6, 8].

Overview of the Contents

This thesis is organized into five chapters whose contents are summarized in the following paragraphs.

Chapter 2, *Automatic Detection of Parallelism*, is intended to give the reader a clear description of the goals and the basic ideas behind the material presented in this thesis. The chapter begins with a review of the different approaches that can be found in the literature in the scope of automatic detection of parallelism. The discussion emphasizes the limitations of current techniques for the analysis of irregular applications. In an early stage of the work, a suite of irregular codes was analyzed by hand. As a result, it was found that a set of basic computations appear in the majority of the loop nests. The differences between loop nests come from the different ways these basic computations are combined. In this chapter, the set of basic computations considered in the thesis is described. Finally, we present the motivations and the key ideas that led us to develop the compiler framework.

Chapters 3 and 4 cover two algorithms that are the core of our compiler framework. The explanation of the internals of the algorithms is illustrated with some example codes extracted from our irregular benchmark suite. Experimental results that show the effectiveness of the algorithms are presented at the end of each chapter.

Chapter 3, *Recognition of Basic Computational Kernels*, presents an algorithm that identifies the set of basic computations that appear in the source code of a loop nest, and determines how these basic computations are combined in the loop. As a result of the execution of this algorithm, the source code of the loop nest is represented as a graph that summarizes a wide set of information that will be useful for the subsequent stages of the compiler framework. The theoretical foundations of the algorithm are well-established. In particular, the basic computations are identified by searching the strongly connected components that compose the dependence graph of the GSA program form. The type of computations, and how they are combined, is determined through a classification scheme that is applied to the set of components.

Chapter 4, *Recognition of Loop-Level Computational Kernels*, presents an algorithm that recognizes the type of computations performed in a loop as a whole. This goal is accomplished through the analysis of the graph that represents how the basic computations are combined in the loop. A detailed description of the recognition of the computations performed in some loop nests extracted from our benchmark suite is presented. The codes are representative examples of loop patterns that appear very frequently in irregular applications. The chapter covers simple loops that are detected with current compiler technology, as well as coarser-grain loop nests that are not covered.

Chapter 5, *Generation of Parallel Code*, describes how our framework provides the compiler with efficient support for the transformation of a sequential code into a parallel counterpart. The approach is illustrated with the example codes studied in Chapter 4. The chapter finishes with a comparison of our approach against the technology implemented in the Polaris parallelizing compiler, which is capable of analyzing irregular applications.

Chapter 2

Automatic Detection of Parallelism

The parallelization of a sequential program basically consists of changing the execution order specified by the programmer so that the same results are computed using a set of processors. Parallelizing compilers are faced with two major challenges. The most important is finding the minimal set of constraints that the parallel program has to fulfill for the semantics of the sequential code to be preserved. The other is finding a correct reordering of the execution of the program so that good performance can be obtained from the target parallel machine.

Constraints are traditionally represented as dependences between statements of the program. Dependences capture an important strategy for preserving correctness in imperative languages: they preserve the relative order of load and store operations for each memory location in the program (they do not preserve the relative order of reads to the same location, but this cannot affect a program's meaning). Dependences are a passive guide for uncovering implicit parallelism, but also specify how the execution of a program may be reordered. In Section 2.1, a review of the state of the art on automatic detection of parallelism in sequential codes is presented. The discussion focuses on remarking the limitations of current compiler technology for the analysis of irregular applications.

The framework we propose addresses the recognition of the kernels that are computed during the execution of a loop. We apply the term *computational* *kernel* to a set of statements from the loop body whose execution provides useful values for other parts of the program. In an early stage of this work, we studied in detail a set of real codes that contain a great deal of irregular computations. We observed that, in general, loop nests perform complex computations that are a combination of simpler ones. In Section 2.2, we describe a collection of kernels that covers the majority of the cases found in our benchmark suite.

The chapter finishes with a general overview of the structure of our compiler framework. The key ideas that motivated its design are also presented.

2.1 State of the Art

Parallelizing compilers are based on data dependence analysis techniques that enable the detection of the parallelism that is implicit in sequential programs. Data dependence analysis [51] determines whether two statement instances must execute in the order specified in the sequential program to guarantee correct results. The problem for scalar variables has been studied extensively and can be solved with data flow analysis [1]. In real applications, the compiler is faced with the analysis of loop nests that touch different array entries in different loop iterations, very often with the added complexity introduced by if-endif constructs that define complex control flows. The data dependence analysis problem for arrays is equivalent to integer programming, and thus a general solution to the problem would have exponential time complexity [34]. In most cases, this problem is simplified by focusing only on the subscript expressions that can be described as linear or affine functions of the index variables of the enclosing loops. Several polynomial-time data dependence tests can be found in the literature: the Banerjee test [9], the I-Test [30, 36] or the VI-Test [37]. The experimental results in [37] show that the VI-Test improves the accuracy of the original I-Test and, in many cases, it performs as well as the Omega test at much less computational cost. The Omega Test [38] is a general purpose integer constraint satisfaction algorithm that can accurately handle complex loop regions and even provide exact data dependence information. It is more accurate than the other tests, but at a higher computational cost. In fact, it has a worst case exponential time complexity. The accuracy of these tests is limited due to non-linear constraints, which are frequently found in scientific benchmarks.

Programs are not typically written with dependence testing in mind. Programmers tend to write code that exploits specific characteristics that are only implemented in certain versions of languages and compilers. Furthermore, there are many programming practices that have been developed to overcome weaknesses in compiler optimization strategies, but whose result is often code that defeats the best dependence analyzer. To address problems of this sort, a number of code transformations can be applied prior to dependence testing with the goal of making testing more accurate. The accuracy is improved because, as a side effect of these transformations, subscript expressions are usually rewritten as linear or affine functions of the loop index variables, which can be handled by dependence tests. Transformations, such as loop normalization, induction variable substitution, constant propagation, dead code elimination and privatization are some examples of techniques that were found to be effective for this purpose. There is a voluminous literature on this issue [13, 25, 40, 47, 49, 50, 51].

Dependence testing in the presence of induction variables was found to be critical for the automatic parallelization of sequential codes. In classical dependence analysis, induction variable occurrences are substituted for closed form expressions in order to remove the loop-carried dependences introduced by these variables. Different approaches have been proposed for the computation of closed form expressions. Pottenger and Eigenmann [35] use idiom recognition to detect simple induction variable forms. Mathematical approaches based on solving systems of equations were also proposed [56]. In real applications, more complex forms are computed. Gerlek et al. [18] propose a sophisticated classification scheme that is based on the analysis of the dependence graph of the SSA form [14]. Complex induction variables are detected by analyzing the characteristics of the statements associated with the strongly connected components of that graph. In the literature, this problem is addressed from a different perspective by Wolfe [50] and by Wu et al. [52, 53]. The basic idea consists of analyzing how the induction variable changes during the execution of a loop. The study focusses on determining whether the induction variable is (strictly) increasing or (strictly) decreasing. This information is later used in dependence tests that discard the existence of loop-carried dependences in the references to array variables.

Suganuma et al. [44] present a technique that can detect reduction constructs in general complex loops. As in [18], the approach is based on the analysis of the strongly connected components of a dependence graph. However, the graph is constructed from the source code of the program. Thus, it does not take advantage of the information about the flow of values provided by SSA-based program representations.

Current parallelizing compilers, which are mainly based on data dependence analysis, cannot detect all the parallelism available in sequential programs. Sometimes this is because the necessary information for dependence tests to be successful cannot be gathered at compile-time. However, many times, the reason is that the data flow analysis techniques and the dependence tests used by the compiler are not sufficiently sophisticated. The limitations of classical dependence tests are specially relevant for irregular codes because of the presence of subscripted subscript expressions. In general, the value of subscripted subscripts cannot be determined at compile-time because of limitations of current technology, but, more frequently, because they depend on the input data of the program. Subscripted subscripts are handled as unknown symbolic quantities that introduce non-linearities in the equations that describe the subscript expressions. Pottenger and Eigenmann [35] address this problem using a strategy that combines idiom recognition and dependence analysis. The method is applied to the detection of irregular reduction computations, as well as to induction variables. First, potentially parallel loops are identified by searching for statements of the loop body that fulfill the characteristics of irregular reductions. Next, dependence analysis is used to discard the existence of dependences that preclude the parallelization of the loop. Finally, a parallelizing transformation specially designed for irregular reductions is applied [20, 22, 32] whenever possible. This approach was implemented in the Polaris parallelizing compiler. It should be noted that it is not necessary to know at compile-time the value of the subscripted expressions that characterize irregular reduction computations. The implicit parallelism is uncovered at compile-time, but dependences are handled at run-time by the parallel code that is generated. Furthermore, the efficiency of the parallel code depends on the run-time value of subscripted subscripts; in the worst case, no parallelism would be available.

The dependence tests described so far perform compile-time analysis. Thus, they are conservative when the necessary information cannot be extracted from the source code. At run-time, much more parallelism could be uncovered because all program information is available. Rauchwerger and Padua [39] presented a general purpose technique that detects loop-carried data dependences during the execution of a loop. The compiler generates code that speculatively executes a loop in a fully parallel manner (*DOALL* loop) and that determines at run-time whether the loop was, in fact, fully parallel. If the subsequent test finds that the loop was not fully parallel, then it will be re-executed sequentially.

Automatic program comprehension allows a compiler to address the automatic parallelization of sequential codes from a different point of view. Keßler [26] proposes a speculative program comprehension and parallelization method suitable for sparse matrix codes. The code is analyzed with the aim of recognizing syntactical variations of a set of sparse computational kernels that are frequently used in full-scale applications, for instance, operations with sparse vectors and matrices. The recognition is performed by taking into account the semantics of the program. For this reason, 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.

2.2 Collection of Computational Kernels

Research on parallelizing compilers usually addresses the recognition and the transformation of computational kernels with well-known characteristics [5, 6, 18, 33, 35, 44]. A set of kernels that are important for the analysis of full-scale sparse/irregular applications is presented in [32]. The focus of this section is the description of a collection of computational kernels that cover a wide range of the regular/irregular kernels found in our benchmark suite, the *SparsKit-II* library [41], which contains a collection of costly routines for the manipulation of sparse matrices. We do not intend to define a comprehensive taxonomy, but to familiarize the reader with concepts and terms that will be used in this thesis.

The first sections focus on kernels whose result is stored in a scalar variable of the program. Some well-known examples are induction variables and scalar reduction operations. It should be noted that the computations associated with these *scalar kernels* usually involve operations with array variables, which may even contain subscripted index expressions. On the other hand, a set of kernels whose result is stored in array variables are also introduced. Typical examples are reduction and recurrence operations with arrays. We will refer to these computations as *array kernels*.

2.2.1 Induction Variables

In the literature [18, 35, 56], the term *induction variable* is usually used for representing the type of scalar, integer-valued variables that are updated in all the iterations of a loop, and for whom a well-defined closed form expression can be calculated. We will use this term in a more general sense. If the variable is updated in every loop iteration, we will call it a *non-conditional induction variable*. In real programs, however, the variable is usually conditionally updated during the execution of the loop. In this case, we will call it a *conditional induction variable*.

This work focuses on conditional and non-conditional *basic linear induction variables*, where a scalar, integer-valued variable is defined in terms of itself and some combination of integer-valued loop-invariant expressions; occurrences of other induction variables are not allowed. Although compiler technology can handle more complex situations [18], basic linear induction variables represent about 85% of the cases found in our benchmark suite. From now on, the term induction variable will be used in this restricted sense. The loop shown in Fig. 2.1(a) is a fragment of a routine that computes a permutation of the rows of a sparse matrix stored in compressed row storage (CRS) format [10]. The inner loop do_k (in this notation the loop index variable is shown as a subscript) contains a non-conditional induction variable ko. A conditional induction variable, i, is shown in Fig. 2.5, which presents a synthetic code that will be used in Section 2.3 for providing a general overview of the compiler framework.

In loop nests, induction variables are often incremented in inner loops and set to an arbitrary value at the beginning of every iteration of an outer loop. We call these kernels *reinitialized (non-)conditional induction variable*. For the

```
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) Code fragment from a routine that performs a permutation of the rows of a sparse matrix in CRS format (*SparsKit-II*, module *unary*, routine *rperm*). The input matrix is a, ja, ia, and the output matrix is ao, jao, iao. The loop nest do_{ii} contains a reinitialized non-conditional induction variable ko. Furthermore, the inner loop do_k calculates a non-conditional consecutively written array, jao, and a conditional consecutively written array, ao.

```
DO h = 1, n
   t = 0
   DO k = ia(h), ia(h + 1) − 1
      t = t + a(k) ∗ x(ja(k))
   END DO
   y(h) = t
END DO
```

(b) Multiplication of a sparse matrix in CRS format by a vector (*SparsKit-II*, module *matvec*, routine *amux*). There is a reinitialized non-conditional scalar reduction t in do_h.

Figure 2.1: Example codes from the *SparsKit-II* library.

sake of brevity, we will use the notation *(non-)conditional* to refer to both the conditional and the non-conditional versions of a kernel. An example of this computational kernel is ko in the scope of the outer loop do_{ii} in Fig. 2.1(a). Note that, at run-time, ko is set to the value of the subscripted expression $iao(perm(ii))$, where iao and $perm$ are loop-invariant arrays.

2.2.2 Scalar Reduction Operations

A reduction is an operation that computes a result from a set of source arrays by reducing the number of dimensions on the basis of an associative operator. If the result is a scalar variable, we call it *scalar reduction operation.* If it is an array variable, we call it *array reduction operation.* Well-known examples are the sum (product) of the entries of an array, the dot product of two arrays and the minimum (maximum) value of an array. These computations are usually implemented as loops. Although the detection and parallelization of these code fragments is generally well-covered in the literature [2, 44], in this section we briefly define some types of reductions that will be studied later in this thesis.

A *scalar reduction* is a scalar variable that is defined in terms of itself and a combination of loop-invariant expressions and/or loop-variant subscripted expressions. The scalar variable may be either integer-valued or floating-point-valued. Like induction variables, *(non-)conditional scalar reductions* and *reinitialized (non-)conditional scalar reductions* can be found in real programs. The code presented in Fig. 2.1(b) multiplies a vector and a sparse matrix stored in CRS format. In the inner loop do_k, the variable t is a non-conditional scalar reduction that computes the inner product of the row number h and vector x. As the figure shows, the computations may involve subscripted subscripts of several indirection levels. In the context of the outer loop, do_h, t is a reinitialized non-conditional scalar reduction because it is set to zero at the beginning of each iteration.

Another interesting scalar reduction operation is the computation of the *minimum* (or the *maximum*) value of a set of values. It is usually implemented as a loop that, in each iteration, compares the value of the reduction variable with an element of the set. The code fragment shown in Fig. 2.2(a) calculates the length $minlen$ of the smallest row of a sparse matrix stored in CRS format, ia being the array of pointers to the beginning of the rows. The set of values is determined by the set of expressions $\{ia(h+1) - ia(h) \ : \ h = 2...nrow\}$, $nrow$ being the number of rows of the matrix. Programmers often need to determine the position of the minimum (or the maximum) value within the set. In Fig. 2.2(a), the variable $irow$ stores the position of the minimum, which, in this case, represents the number of the smallest row. We call this kernel *scalar minimum with location.* It should be noted that *array minimum (maximum) operations* whose result is an array variable, also appear in real codes. An example is the computation of the minimum with location of each row of a matrix, which will be described in Section 4.4.2.

2.2.3 Linked-List Traversal

A distinguishing characteristic of induction variables is that there is an expression that allows the computation of the next value of the variable starting from its current value. In real codes, arbitrary lists of values that do not have

```
minlen = ia(2) − ia(1)
irow = 1
DO h = 2, nrow
   len = ia(h + 1) − ia(h)
   IF (len < minlen) THEN
      minlen = len
      irow = h
   END IF
END DO
```

(a) Computation of the length of the smallest row and the row number of a sparse matrix in CRS format (*SparsKit-II*, module *unary*, routine *rperm*). The loop do_h contains a minimum with location kernel. The reduction variable consists of the pair *minlen* and *irow*.

```
i = 1
DO h = 1, n
   /* Other kernels */
   ...i...
   i = f(i)
END DO
```

(b) Synthetic code that computes a non-conditional linked-list traversal kernel, i.

```
flag = true
DO h = 1, n
   IF (flag.AND.(...)) THEN
      flag = false
      /* Other kernels */
      ...
   END IF
   /* Other kernels */
   ...
END DO
```

(c) Synthetic code that contains a scalar find-and-set kernel, *flag*.

```
DO h = 1, n
   IF (diag(h) ≠ 0) THEN
      diag(h) = 1/diag(h)
   ELSE
      diag(h) = 1
   END IF
END DO
```

(d) Code fragment from a routine that scales the rows of a sparse matrix (*SparsKit-II*, module *unary*, routine *dscaldg*). The array of scale factors *diag* is computed by executing an array find-and-set kernel.

Figure 2.2: Example codes from the *SparsKit-II* library (cont.).

such an expression are often used. We call this kind of computation *linked-list traversal*. A *linked-list traversal* is a variable that is defined in terms of itself, and the variable is a part of the subscript expression of an array reference. Other types of operators can only appear in the subscript expression of the first array reference. The synthetic loop of Fig. 2.2(b) shows a scalar variable i that is associated with a non-conditional linked-list traversal. It is supposed that i is used in other computations within the loop nest (represented as $\ldots i \ldots$ in the figure). *Conditional* and *reinitialized (non-)conditional linked-list traversals* can also be found in programs.

2.2.4 Masked Operations

Masked operations are computational kernels that modify the value of a variable if it fulfills some properties. A typical example is shown in Fig. 2.2(c), where the loop body contains a set of statements that are executed only in the first loop iteration. This mechanism is usually implemented by means of a scalar variable that can only take two different values. In the example, the scalar $flag$ is set to the logical value $true$ before the execution of the loop. In the first iteration, the value of $flag$ is changed so that the condition of the if-endif construct is evaluated to $false$ in subsequent iterations. Note that $flag$ is not used in any other calculations within the loop body. We call this kernel *scalar find-and-set*.

Another typical example consists of changing the value of a set of array entries that fulfill a certain condition. The loop do_h presented in Fig. 2.2(d) was extracted from a routine that multiplies each row of a sparse matrix by a scaling factor. The scaling factors are stored in the array variable $diag$. We call this kernel *array find-and-set*.

2.2.5 Array Operations

An *array operation* is a kernel whose result is stored in an array variable. In previous sections several types of array operations were introduced: *array minimum (maximum)*, *array minimum (maximum) with location* and *array find-and-set*. They were described in those sections because of their similarities with the corresponding scalar kernels. Next, other important array operations are described.

Array operations are usually implemented as loops that perform scalar operations on the array elements. Several types can be distinguished on the basis of the characteristics of the scalar operations. Let $a(s) = e$ be a statement that assigns the value of expression e to the s-th entry of the array variable a. We say that it is an *array assignment* if there is not any occurrence of variable a in e. If $a(s) = e$ is executed in every loop iteration, then it is a *non-conditional array assignment*. If it is conditionally computed during loop execution, the kernel is called *conditional array assignment*. Different types of array assignments will be distinguished by the properties of the index s. For instance, an *irregular assignment* [6] is a kernel that basically consists of array assignments whose index is a subscripted expression. The loop do_j shown in Fig. 2.3(a) was extracted from a routine that performs a permutation (defined by the vector $perm$) of a sparse matrix in CRS format (a, ia, ja). The array of pointers iao is computed by executing a *non-conditional irregular assignment*. Note that $i + 1$ is a subscripted expression because the scalar variable i takes the value of a different array entry, $i = perm(j)$, in each do_j iteration.

In a similar manner, we say that $a(s) = e$ is a *(non-)conditional array reduction* if there is only one occurrence $a(s)$ of variable a in e. Note that the left-hand side expression and the occurrence have the same subscript expression s. The loop nest do_i presented in Fig. 2.3(b) calculates a *conditional irregular reduction* [20], which is a kernel where the index expression, $j + 1$, is subscripted. Note that, in the example, j introduces two levels of indirection due to the fact that the index expression of ja is the index variable of the inner loop do_k. Furthermore, at run-time, several reduction assignment statements are executed in each do_i iteration.

The last case is a *(non-)conditional array recurrence*, where there are a set of occurrences $a(s_1), ..., a(s_m)$ of variable a in e. Note that $s, s_1, ..., s_m$ are different expressions, in general. The loop do_j shown in Fig. 2.3(c) computes a *(non-)conditional array recurrence*, iao .

An interesting kernel is the *consecutively written array* [32], which consists of writing consecutive array entries in consecutive locations. Unlike the kernels described so far, this one is implemented as a combination of an induction variable of step one that defines the array entries to be written during loop execution, and an array assignment whose left-hand side subscript expression is a linear function of the induction variable. The loop nest do_{ii} for the permu-

```
DO j = 1, nrow
   i = perm(j)
   iao(i + 1) = ia(j + 1) − ia(j)
END DO
```

(a) Loop extracted from *SparsKit-II*, module *unary*, routine *rperm*, which contains an array assignment operation (see Fig. 2.1(a) for a description of the routine). Loop do_j determines the lengths of the rows of the permuted matrix. The results of the non-conditional irregular assignment kernel are stored in the array *iao*.

```
DO i = 1, nrow
   DO k = ia(i), ia(i + 1) − 1
      j = ja(k)
      IF (j ≠ i) THEN
         iwk(j + 1) = iwk(j + 1) + 1
      END IF
   END DO
END DO
```

(b) Loop nest extracted from a routine that changes the storage format of a sparse matrix (*SparsKit-II*, module *formats*, routine *ssrcsr*) A conditional irregular reduction iwk is computed in do_i.

```
iao(1) = 1
DO j = 1, nrow
   iao(j + 1) = iao(j + 1) + iao(j)
END DO
```

(c) Code extracted from *SparsKit-II*, module *unary*, routine *rperm* (see Fig. 2.1(a) for a description of the routine). The array of pointers of the permuted matrix *iao* is computed by executing a non-conditional array recurrence kernel.

Figure 2.3: Example codes from the *SparsKit-II* library (cont.).

tation of the rows of a sparse matrix (see Fig. 2.1(a)) contains such a type of computations. The inner loop do_k copies one row of the input matrix (a, ja, ia) into the arrays ao and jao of the output sparse matrix. This task is carried out by using the non-conditional linear induction variable ko as the subscript expression for the definition of ao and jao. We call these kernels *conditional consecutively written array* and *non-conditional consecutively written array*, respectively. In the scope of the outer loop do_{ii}, a consecutive subarray of ao and jao is modified in each iteration.

2.3 Framework Overview

Data dependence analysis is the fundamental technology that is used by parallelizing compilers for uncovering implicit parallelism in sequential codes. Dependence tests are methods that determine whether two references to the same variable in a given set of loops might access the same memory location. These methods usually assume a number of properties for loops. Thus, the compiler needs to gather information regarding which loops meet these requirements. Furthermore, as discussed in Section 2.1, the accuracy of dependence tests depends on the ability of the compiler to rewrite the original code in order to make subscript expressions amenable to dependence testing. This approach was found to be effective for the analysis of regular codes. However, as will be shown next, it is not the case for irregular codes. Consider the irregular reduction of Fig. 2.3(b). Dependence tests should try to determine if there are two loop iterations where the subscript expression $j+1$ takes the same value. If a test is successful, the loop cannot be executed in parallel. In general, the value of j cannot be determined at compile-time because the value of arrays ja and ia is not known until run-time. As a result, the tests will fail. There are two main reasons for the unsuccessful tests. On the one hand, the limitations of the information-gathering techniques used in today's parallelizing compilers. On the other hand, the fact that the value of the subscript expression usually depends on program inputs that are known at run-time only.

The automatic parallelization of irregular computations can be addressed at compile-time if the compiler is provided with support for handling dependences at run-time. The key idea consists of generating parallel code that uses run-time information to assure that dependences are not broken during the execution of the loop. This run-time support rids the compiler of having to determine whether the subscript expression may or may not introduce loop-carried dependences. Thus, its job is limited to recognizing the irregular computational kernel and running dependence tests to discard the existence of other dependences that preclude the parallel execution. Finally, if testing is successful, an appropriate parallelizing transformation can be applied. The parallelization of loops that contain irregular reduction computations is addressed in [35] using this approach. It uses source code pattern-matching to detect the irregular reduction kernel. A similar approach could be used for parallelizing loops with irregular assignments [6] or with *DOACROSS* computations [33]. However, it would be necessary to develop a specific method for the detection of each kernel type.

Detection techniques that rest on the analysis of the source code have two major drawbacks: dependence on the source code quality and difficulty in analyzing complex control constructs. Gerlek, Stoltz and Wolfe [18] proposed a sophisticated classification scheme that recognizes complex induction variables even in loops that have a complicated control flow. It is based on the translation of the source code into *Static Single Assignment* form (commonly abbreviated SSA) [14], which is a program representation that provides the compiler with valuable information about the use of variables during the execution of a program. A major limitation of this method is that it can only handle expressions that contain references to integer-valued scalar variables. Our key idea is to generalize the classification scheme of [18] so that both integer-valued and floating-point-valued scalar and array variables can be handled. As will be explained later, the relevance of this generalized scheme comes from the fact that it is the basis for the construction and the design of the detection technique proposed in this thesis.

In the rest of the chapter, we will show how our scheme enables the recognition of the scalar and array kernels that are computed during the execution of a loop nest. Furthermore, the information gathered by this classification method will be used as a compiler framework that supports the automatic detection and parallelization of sequential codes. This organization is shown in the block diagram depicted in Fig. 2.4. The chain of solid boxes shows the intermediate program representations used by the compiler during the process of transformation of the source code of the loop body into parallel

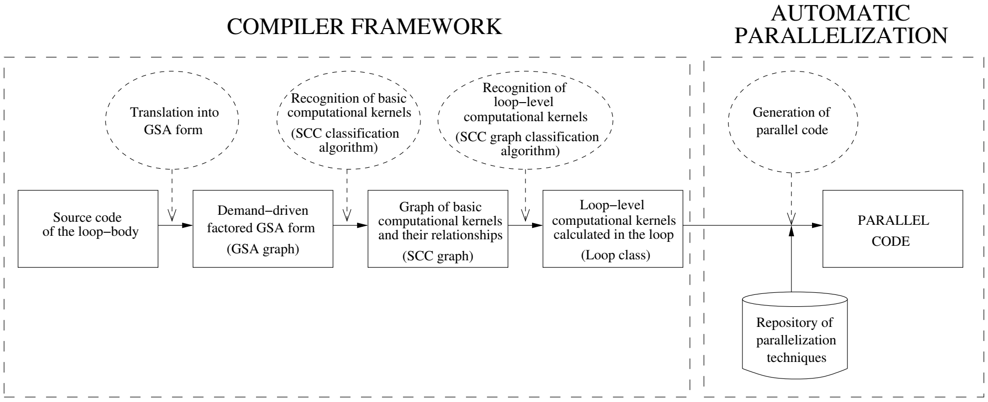


Figure 2.4: Block diagram of the automatic detection technique.

code. The compiler framework construction and the automatic parallelization process (dashed rectangular boxes) are decomposed into a set of stages that are represented as dashed ovals. In the following sections, the different stages are described using as a guide the consecutively written array kernel (see Section 2.2.5) presented in Fig. 2.5(a).

2.3.1 Translation into GSA form

Flow of values of program variables is a fundamental property of the algorithm coded in a program. In order to detect the computational kernels computed in a loop nest, the compiler needs to find precise information about the value and the use of variables at various points in the program. For instance, for the induction variable i of Fig. 2.5(a) to be recognized, the compiler must be able to determine that, in every loop iteration, i is assigned its value in the previous iteration plus a constant value 1. The first loop iteration is the only exception to this general rule. In that case, i is computed by adding 1 to the value of i before the loop, i.e. 1. Classical data flow analysis can be used to compute the reaching definitions at each use of the variable [2]. However, in [18] a different approach is used. SSA is an intermediate representation that has two key properties: (1) every use of any variable in the program has exactly one *reaching definition*; (2) at points where control flow joins, a special operation ϕ is inserted to merge the definitions of the variable that reach the point along different control flow paths. The SSA form of our example, the consecutively written array kernel, is shown in Fig. 2.5(b). Special operations

ϕ were inserted following the loop header and the if-endif construct. The first one, $i_2 = \phi(i_1, i_4)$, merges the value at the end of the previous iteration, i_4, and the value before the loop execution starts, i_1. The second one, $i_4 = \phi(i_3, i_2)$, merges the reaching definition from the beginning of the loop iteration, i_2, and the reaching definition corresponding to the conditionally executed scalar statement $i_3 = i_2 + 1$.

The information captured by SSA is not sufficient for the analysis of expressions that contain array references (see the statement $a(i_2) = tmp_2 + 2$ in Fig. 2.5(b)). Several extensions of SSA that provide reaching definition information for array variables have been proposed [46, 29]. As will be shown in Section 3.1, the *Gated Single Assignment* form (GSA) [46] is the most appropriate representation for being the basis of our compiler framework. It should be observed in Fig. 2.5(c) that ϕ operations in GSA are inserted at the loop header and after the if-endif construct, both for the scalar i and the array a. Unlike SSA, the array statement $a(i_2) = tmp_2 + 2$ is replaced with an additional ϕ operation $a_2 = \phi(a_1, i_2, tmp_2 + 2)$ that merges the value of the array before the statement, $a_1 = \phi(a_0, a_3)$, and the new value that results from modifying the entry i_2 of the array. In the next section, we explain how the compiler can take advantage of this information in order to recognize the kernels computed in the loop do_h.

2.3.2 Recognition of Basic Computational Kernels

Program representations like SSA or GSA provide the compiler with very useful information about the use of variables at run-time. Gerlek, Stoltz and Wolfe [18] do not use the reaching definition information contained in SSA directly. With this information, they build a graph of *factored use-def chains* [51] where each use of a variable is represented by a single use-def chain that allows the compiler to locate the unique reaching definition straightforwardly. This graph is usually called *SSA graph* in the literature. From now on, we will use that term. The key observation for the detection of induction variables is that the flow of values associated with this kernel during loop execution corresponds to a strongly connected component in the SSA graph. We will refer to these computations as *basic kernels*. The *strongly connected components (SCCs)* of a graph are the maximal cycles of the graph, that is, the cycles

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

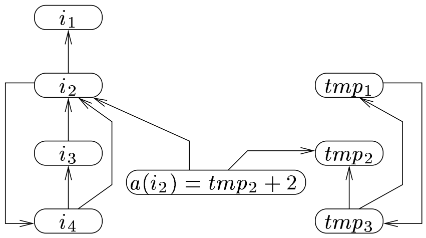
(a) Source code.

```
i_1 = 1
DO h_1 = 1,n,1
   i_2 = φ(i_1, i_4)
   tmp_1 = φ(tmp_0, tmp_3)
   IF (c(h_1)) THEN
      tmp_2 = f(h_1)
      a(i_2) = tmp_2 + 2
      i_3 = i_2 + 1
   END IF
   i_4 = φ(i_3, i_2)
   tmp_3 = φ(tmp_2, tmp_1)
END DO
```

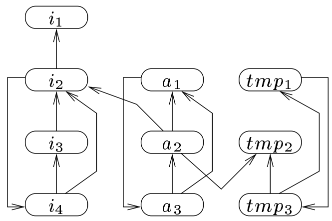
(b) SSA form.

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

(c) GSA form.



(d) SSA graph.



(e) GSA graph.

Figure 2.5: SSA and GSA representations of a generic consecutively written array computational kernel.

that are not proper subsets of any other cycle [51, Chapter 3]. Let us look at the SSA graph in Fig. 2.5(d). The scalar assignment statements, which define unique scalar variables, are represented by nodes whose label is the left-hand side of the statement. For array statements, the whole statement is used as a label. The edges denote the use-def chains between statements. The computations corresponding to the loop index have been omitted for the sake of clarity. The statements $i_2 = \phi(i_1, i_4)$, $i_3 = i_2 + 1$ and $i_4 = \phi(c(h_1), i_3, i_2)$ form a SCC in the SSA graph. This is due to the cyclic nature of loops, which makes loop-carried dependences associated with i appear during loop execution. Regarding GSA (see the GSA graph of Fig. 2.5(e)), it should be noted that there is a SCC related to i. But, and what is more interesting, the statements $a_1 = \phi(a_0, a_3)$, $a_2 = \phi(a_1, i_2, tmp_2 + 2)$ and $a_3 = \phi(c(h_1), a_2, a_1)$ form another SCC that captures the loop-carried dependences that arise from the computation of the array a.

An important contribution of this thesis is the proposal of a classification scheme that enables the recognition of the scalar and array computational kernels associated with SCCs in the GSA graph. As in [18], the method basically consists of looking for occurrences of a variable in the expressions that compose the statements of a SCC. The key difference is that the method must be able to identify occurrences of both scalar and array variables. During loop execution, i is incremented by a constant value 1 in those iterations where the condition $c(h)$ is fulfilled. In the SCC, there is one occurrence of i in the source code expression $i+1$. As the expression is a sum operator that adds the constant value 1 to i, the SCC is classified as a conditional induction variable of step 1. A similar analysis of the SCC related to the array a will conclude that there is no occurrence of the array reference $a(i)$ in the source code expression tmp. Thus, the compiler can classify the computations associated with array a as a conditional array assignment operation (see Section 2.2.5).

This stage of the compiler framework construction extracts valuable information from the source code of a loop body. This information will be represented in a graph whose nodes are the basic computational kernels and whose edges are the relationships among these kernels. This graph, which will be referred to as *SCC graph*, is the basis for the recognition of the kernels computed in a loop. It should be noted that the relationships, which are represented as use-def chains between SCCs, can be identified during the classification of the SCCs because it basically consists of finding occurrences of the variables associated with other SCCs. The details about the implementation of the SCC classification algorithm will be presented in Chapter 3.

2.3.3 Recognition of Loop-Level Computational Kernels

The goal of our compiler framework is to recognize the type of computations performed during the execution of a loop nest. Once the basic kernels have been identified, it is necessary to analyze their relationships in order to know how they are combined within the loop body. This analysis enables the recognition of more complex kernels. Consider the GSA graph of Fig. 2.5(e), which contains a use-def chain from $a_2 = \phi(a_1, i_2, tmp_2 + 2)$ to $i_2 = \phi(i_1, i_4)$ due to the use of i in the left-hand side subscript of the array assignment statement. In the SCC graph, this relationship is represented as a use-def chain from the SCC of a to the SCC of i. The use-def chain provides the compiler with an appropriate scenario for the recognition of the conditional consecutively written array kernel of the example. Note that, in order to be successful, some additional checks are needed. On the one hand, the subscript expression i of the array assignment operation, a, consists of an occurrence of the conditional induction variable of step 1, i.e. i. This information is available in the SCC graph. On the other hand, it is necessary to assure that every time an array entry is written, the induction variable is updated. This condition is fulfilled if both a and i are computed at the same point of the program. This task can be accomplish through the analysis of the *control flow graph (CFG)* of the program. A detailed explanation of the detection of complex computational kernels will be presented in Chapter 4.

2.3.4 Generation of Parallel Code

Finally, a parallelizing compiler generates parallel code for the computational kernels that have been recognized. In order to take advantage of available research results on parallelizing transformations for regular and irregular computations, the compiler can use a repository of the transformations that can be applied to each kernel type (see Fig. 2.4). In the example of Fig. 2.5, the computations of the loop do_h were recognized as a conditional consecutively written array kernel. Its characteristics assure that each loop iteration actually executed writes an array entry that is not touched in any other iteration. With this information the compiler can conclude that, by applying an appropriate code transformation, the loop can be executed in parallel safely. The generation of parallel code stage will be explained in Chapter 5.

The compiler framework proposed in this thesis is a powerful information-gathering tool whose application is not restricted to the scope of automatic parallelization of sequential codes. Other possible applications of the framework will be outlined at the end of this work.

Chapter 3

Recognition of Basic Computational Kernels

In general, the computations performed in the body of a loop can be decomposed into a set of basic computational kernels that are combined in a specific manner. In this chapter, we present a method for the recognition of both the basic kernels computed in a loop nest, and the relationships that exist between those kernels. The work by Gerlek, Stoltz and Wolfe [18] describes the theoretical foundations that are the basis of our technique. The method proposed in that work addresses the recognition of integer-valued scalar computational kernels even in the presence of complex control constructs. A well-known example of such kernels are non-conditional and conditional induction variables. The method consists of a classification scheme that proceeds as follows. First, the loop body is translated into SSA form. Next, a partitioning of the scalar statements of the loop body is computed. This task is accomplished by searching the strongly connected components (SCCs) that appear in the SSA graph, which is the graph of factored use-def chains corresponding to the SSA form. The core of the method consists of the classification of the statements of each SCC. The classification scheme provides the compiler with information about the type of scalar, integer-valued computational kernel that is calculated in each SCC. Furthermore, whenever possible, a closed form expression for the corresponding kernel is determined.

We present a new classification scheme that recognizes the kernels computed in the statements of the SCCs that arise in the GSA graph, i.e. the graph that represents the factored use-def chains of the GSA form. The scheme in [18] 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. The new algorithm is a generalization that 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 also 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, at least, to the collection of basic kernels described in Section 2.2, namely, induction variables (basic linear and more complex induction variables), scalar reduction operations (sum, product, minimum, maximum, etc.), linked-list traversals, masked operations with scalars and arrays, and both regular and irregular array operations. It should be noted that the recognition of the *reinitialized* scalar kernels (induction variables, scalar reductions and linked-lists) cannot be accomplished with the classification scheme presented in this chapter because the SCCs do not contain enough information. That problem will be addressed in the following chapter.

A detailed block diagram of this stage of the compiler framework construction, *Recognition of basic computational kernels*, is depicted in Fig. 3.1. Starting from the GSA form, the SCCs of the GSA graph are classified. As will be shown later, the classification process performs an exhaustive traversal of the expressions that compose the statements of the SCC. During this process, the use-def chains between SCCs will be identified. Finally, the use-def chains will be classified in a subsequent phase. The information gathered during the analysis of the code is represented in a structure called *SCC graph*, which is the basis of the last stage of the framework, *Recognition of Loop-Level Computational Kernels* (see Chapter 4).

The rest of the chapter is organized as follows. The gated single assignment program form is described in Section 3.1. Basic definitions and notations that are needed to explain the algorithms for the construction of the framework are introduced in Section 3.2. The key ideas behind the decomposition of the loop body into SCCs are described in Section 3.3. A taxonomy of SCC classes is formally defined in Section 3.4. The algorithms for the classification

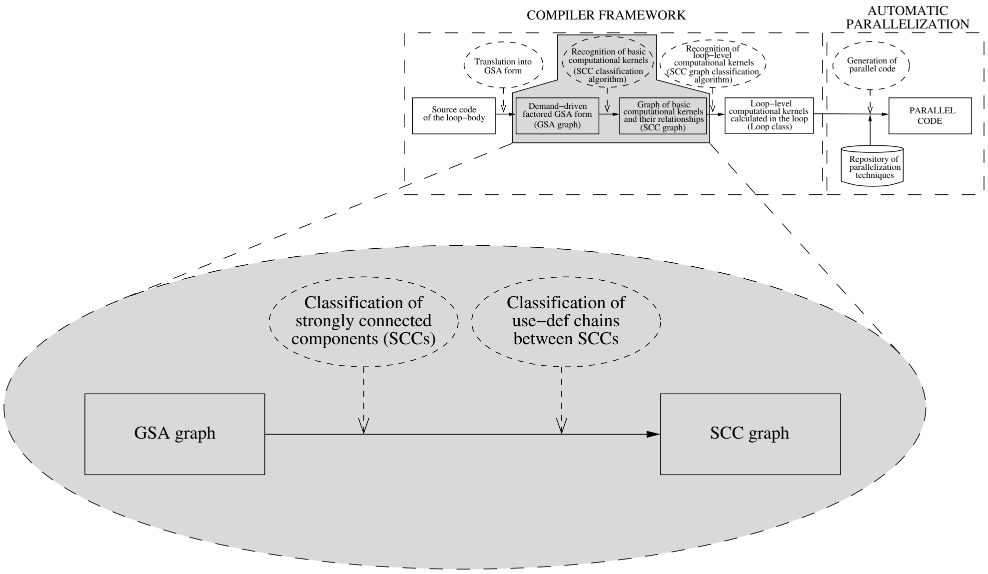


Figure 3.1: Recognition of basic computational kernels (detail of the compiler framework depicted in Fig. 2.4).

of the SCCs and the use-def chains between SCCs are described in detail in Section 3.5. In order to help the reader to understand the algorithms precisely, a unique example code will be used as a guide for the explanations. A detailed description of the SCC classification algorithm applied to this example is presented in Section 3.6. The chapter finishes with the presentation of experimental results in Section 3.7.

3.1 Gated Single Assignment Form

Flow of values of program variables is a fundamental property of the algorithm coded in a program. Flow of values was traditionally studied by means of methods that analyze the source code of a program in order to compute the set of variables that are used in each statement, as well as the points of the program where the values of those variables are defined (*Reaching Definition Analysis* [1]). In the last decade, a different approach has been successfully adopted for implementing code transformations in a more efficient manner. It basically consists of translating the source code into an intermediate program form where reaching definition information is represented syntactically. This task is accomplished by inserting a set of special operators (called ϕ generically) in 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. This approach was shown to be effective, as demonstrated by the large number of code transformation and optimization techniques that were implemented on top of these intermediate representations. Some examples are constant propagation [49], global value numbering [3, 40], partial redundancy elimination [13], strength reduction [25], register promotion [42], detection of scalar computational kernels [18] , automatic parallelization of programs [27, 50], conditional constant propagation of scalar and array references [28], and privatization of variables [45].

The most popular intermediate form is *Static Single Assignment (SSA)* [14], which captures reaching definition information at the statement-wise level. This means that the unique statement that defines the value of a variable at compile-time, may have several run-time instances that write successive values to the same variable (e.g. a statement included in a loop body). An extension of SSA that captures control flow information, *predicated static single assignment (PSSA)* [12], was also proposed. However, both SSA and PSSA have a major limitation for the analysis of real applications: they only represent reaching definition information for scalar variables.

In order to solve the problem described above, several extensions that handle array variables have been proposed. Knobe and Sarkar [27, 29] present two formulations of *Array SSA*: *Partial Array SSA (PA-SSA)*, used in static analysis of programs (i.e. at compile-time), and *Full Array SSA (FA-SSA)*, used in dynamic analysis of programs (i.e. at run-time). PA-SSA captures data-flow information at the statement-wise level. Thus, arrays are managed as a unique entity, like scalars in SSA. FA-SSA is a run-time implementation of PA-SSA that transforms the special operators inserted in the code into code that is executed at run-time. This mechanism enables the analysis of different instances of array assignment statements separately, and thus it enables to perform precise analysis of array subscript expressions. Another extension called *Gated Single Assignment (GSA)* was used by Tu and Padua [46] to address automatic privatization of array variables. Unlike FA-SSA, the special operators are not translated into run-time code in the GSA form. Actually, GSA and PA-SSA are very similar representations that support static program analysis. A key difference is that GSA captures the information associated with the conditional expressions of the conditional statements of a program.

The technique presented in this thesis addresses the recognition of loop-level computational kernels at compile-time. For this purpose, both GSA and PA-SSA can be used. However, we have chosen GSA because it presents several advantages. First, GSA represents control flow information. Thus, during the execution of the SCC classification algorithm, control information is immediately available in the statements of the SCCs. In PA-SSA, further analysis of the source code is needed. Consider the conditional consecutively written array computations shown in Fig. 3.2(a). Focus on the corresponding GSA and PA-SSA forms (Figs. 3.2(b) and 3.2(d), respectively), where the generic ϕ operators have been replaced with more specific operators in the GSA form (these special operators will be described in Section 3.1.1). In GSA form, the special operator in statement $a_3 = \gamma(c(h_1), a_2, a_1)$ captures the conditional expression $c(h_1)$ of the if-endif construct; the statement $a_4 = \phi(a_3, a_1)$ in PA-SSA does not. Second, array assignment statements are represented in GSA using only one statement ($a_2 = \alpha(a_1, i_2, tmp_2 + 2)$ in Fig. 3.2(b)), while two statements are used in PA-SSA ($a_2(i_2) = tmp_2 + 2$ and $a_3 = \phi(a_2, a_1)$ in Fig. 3.2(d)). This compact representation has important implications from the viewpoint of the construction of our framework. On the one hand, the statements associated with array definitions in GSA belong to a unique SCC in the GSA graph (see the set of nodes a_1, a_2, a_3 in Fig. 3.2(c)), which simplifies the SCC classification algorithm. Note that two SCCs appear in PA-SSA form: one composed of the ϕ-statements (a_1, a_3, a_4 in Fig. 3.2(e)), and the other one composed of the array statement (a_2 in Fig. 3.2(e)). On the other hand, the translation of real codes into PA-SSA form could lead to code explosion.

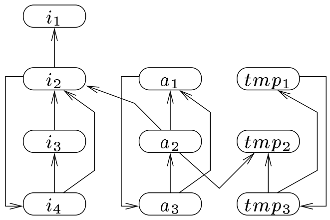
3.1.1 Special Operators in GSA Form

The translation of a source code into GSA form involves carrying out two main tasks: (1) placement of the ϕ special operators at the confluence nodes of the control flow graph, and (2) renaming of variables so that the left-hand sides of the assignment statements define distinct variables. An efficient construction method for GSA is proposed in [46]. From the viewpoint of program analysis, it is interesting to be able to distinguish the generic ϕ operators by the point

```
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.

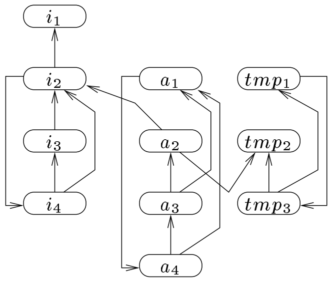
```
i_1 = 1
DO h_1 = 1, n, 1
   i_2 = μ(i_1, i_4)
   a_1 = μ(a_0, a_3)
   tmp_1 = μ(tmp_0, tmp_3)
   IF (c(h_1)) THEN
      tmp_2 = f(h_1)
      a_2 = α(a_1, i_2, tmp_2 + 2)
      i_3 = i_2 + 1
   END IF
   i_4 = γ(c(h_1), i_3, i_2)
   a_3 = γ(c(h_1), a_2, a_1)
   tmp_3 = γ(c(h_1), tmp_2, tmp_1)
END DO
```



(c) GSA graph.

(b) GSA form.

```
i_1 = 1
DO h_1 = 1, n, 1
   i_2 = φ(i_1, i_4)
   a_1 = φ(a_0, a_4)
   tmp_1 = φ(tmp_0, tmp_3)
   IF (c(h_1)) THEN
      tmp_2 = f(h_1)
      a_2(i_2) = tmp_2 + 2
      a_3 = φ(a_2, a_1)
      i_3 = i_2 + 1
   END IF
   i_4 = φ(i_3, i_2)
   a_4 = φ(a_3, a_1)
   tmp_3 = φ(tmp_2, tmp_1)
END DO
```



(e) PA-SSA graph.

(d) PA-SSA form.

Figure 3.2: GSA and PA-SSA representations of the generic consecutively written array kernel shown in Fig. 2.5.

of the program where the new statements are inserted. The following types of ϕ's are considered:

$\mu(x_{out}, x_{in})$, which appears at loop headers and selects the initial x_{out} and loop-carried x_{in} values of a variable.

$\gamma(c, x_{in}, x_{out})$, 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_{in} if c is true; x_{out}, if c is false.

$\alpha(a_{prev}, s, rhs)$, whose semantics is that the element s of an array variable a receives the value rhs while the other elements take the values of the previous definition of the array, denoted as a_{prev}.

Several syntaxes are possible. However, the really important thing is the semantics. Without loss of generality, the algorithms presented later in Section 3.5 are based on the syntax described above. From now on, we will refer to the statements of the source code of the loop body explicitly as *source code statements*. The term *statement* will be used for the assignment statements of the GSA form. The statements that contain μ, γ and α operators will be referred to as μ-statements, γ-statements and α-statements, respectively. Note that we do not need to talk about conditional statements in GSA form because control information is implicitly represented in the γ operator.

3.1.2 Useful Properties of GSA for the Analysis of Codes

The success of intermediate program representations such as SSA or GSA comes from the fact that they have some properties that are very useful for the analysis of codes. Our technique for automatic detection of parallelism takes advantage of a set of properties that are described in this section.

An immediate consequence of the placement of ϕ special operators and the renaming of variables is that anti-dependences and output-dependences on scalar variables are removed from the code. Regarding array variables, these dependences related to memory reuse are removed for the array as a whole, not at the array element level. Furthermore, reaching definition information is immediately available for scalar and for array variables (not for array entries).

An important characteristic of GSA form is that it supplies syntactical support for analyzing at compile-time whether or not two expressions take the same value during the execution of a program. The following definition introduces this concept.

Definition 3.1. *Let e_1 and e_2 be two expressions in the GSA form. We say that e_1 and e_2 are* ***GSA-equivalent expressions****, $e_1 \overset{GSA}{\equiv} e_2$, if e_1 and e_2 always take the same value during the execution of the program.*

In the source code, a unique variable name is used along all the execution paths. The name of the variable does not contain any information about the point of the program where the value of an occurrence is computed. Thus, a compiler has to use data flow analysis techniques for determining if two different occurrences of a variable will take the same value at run-time. In contrast, in GSA form different definitions of the same variable are assigned different names, so reaching definition information (at the statement-wise level) is represented syntactically. Next, we present a theorem that, using such information, provides a sufficient condition for two expressions to be GSA-equivalent.

Theorem 3.1. *Let e_1 and e_2 be two expressions in GSA form. If e_1 and e_2 are syntactically identical ($e_1 = e_2$), then both expressions take the same value during the execution of the program, i.e. they are GSA-equivalent ($e_1 \overset{GSA}{\equiv} e_2$).*

Proof. The key observation is that different occurrences of a scalar variable are assigned the same name in GSA form if they are located between two consecutive scalar assignment statements executed between consecutive confluence nodes of the control flow graph. As there is not any intermediate definition of confluence node, the scalar variable keeps its value unchanged during the execution of that fragment of the program. As a result, all the occurrences will take the same value at run-time. Regarding array variables, the occurrences of an array are assigned the same name under the same conditions stated above, the difference being that it is the array as a whole that keeps its value unchanged. Consequently, two array references will take the same value if their subscript expressions are syntactically identical.

For the proof to be completed, the general case of an arbitrary operator must be considered. In that case, if e_1 and e_2 are syntactically identical, then they compute the same operation with the same arguments evaluated in the same order. The arguments may be scalar variables, array references or other arbitrary operators. As the arguments are syntactically identical, the expressions e_1 and e_2 will take the same value at run-time. $\square$

Two observations should be noted regarding Theorem 3.1. On the one hand, the run-time value of the expressions is not known; we can only assure that they will take the same value during the execution of the program. On the other hand, two expressions that are syntactically different in GSA form can take the same value at run-time.

The following lemma, which is an immediate consequence of Theorem 3.1, establishes a sufficient condition for two DO loops of a program in GSA form to have the same iteration space.

Lemma 3.1. *Let do_h and $do_{h'}$ be two DO loops that appear in the GSA form of a program. Let $init_h$, $limit_h$ and $step_h$ be the init, limit and step expressions that define iteration space of do_h. Let $init_{h'}$, $limit_{h'}$ and $step_{h'}$ be the init, limit and step expressions that define iteration space of $do_{h'}$. If these expressions are pairwise syntactically identical (i.e. $init_h = init_{h'}$, $limit_h = limit_{h'}$ and $step_h = step_{h'}$), then the iteration spaces of do_h and $do_{h'}$ are equal.*

3.2 Basic Definitions and Notations

In this section, we introduce definitions and notations that will be used throughout this chapter for describing the construction of our compiler framework. Let $SCC(x_{1\ldots n})$ denote a strongly connected component composed of n nodes of the GSA graph of a program. The nodes are associated with the set of statements that define the variables x_k $(k = 1, \ldots, n)$.

The following five definitions are used as criteria for the taxonomy of SCCs presented later in Section 3.4.

Definition 3.2. *Let $SCC(x_{1\ldots n})$ be a strongly connected component. The component is **trivial** if it consists of exactly one node of the GSA use-def chain graph ($n = 1$). Otherwise, the component is **non-trivial** ($n > 1$).*

Definition 3.3. *Let $SCC(x_{1\ldots n})$ be a non-trivial strongly connected component. Let $c_1, \ldots, c_m$ be the set of conditional expressions associated with the*

γ-statements of the SCC. The component is ***structural*** *if there is not any occurrence of $x_1, \ldots, x_n$ in $c_1, \ldots, c_m$. Otherwise, the component is* ***semantic***.

Definition 3.4. *The* ***cardinality*** *of a component, $|SCC(x_{1\ldots n})|$, is defined as the number of different variables of the source code that are associated with $x_1, \ldots, x_n$.*

Definition 3.5. *Let $SCC(x_{1\ldots n})$ be a strongly connected component of cardinality zero or one. The component is* ***scalar*** *if $x_1, \ldots, x_n$ are different definitions of a unique scalar variable x of the source code. Otherwise, the component is* ***array***.

Definition 3.6. *Let $SCC(x_{1\ldots n})$ be a strongly connected component. The component is* ***conditional*** *if it contains at least one γ-statement, i.e., if at least one assignment statement is enclosed within an if-endif construct. Otherwise, it is* ***non-conditional***.

The notation $SCC(x_{1\ldots n})$ will be used for referring to components in a generic manner. However, giving a detailed description of the compiler framework will usually require the specification of some characteristics of the components. The notations $SCC_C^S(x_{1\ldots n})$ and $SCC_C^A(x_{1\ldots n})$ will denote a scalar and an array component of cardinality C, respectively.

This thesis focuses on the analysis of SCCs with cardinality zero or one because they represent approsimately 88% of the SCCs found in our benchmark suite (see Section 3.7 for more details). The classification of SCCs with higher cardinality, and the computational kernels that they represent, is a research topic left for future study. The following lemma, which is an immediate consequence of Def. 3.4, presents an important relationship between the variables of the SCCs in GSA form and the corresponding variables in the source code.

Lemma 3.2. *Let $SCC(x_{1\ldots n})$ be a component with cardinality zero or one. The variables x_k ($k = 1, \ldots, n$) are different definitions of a unique variable x of the source code.*

The consequences of this lemma will be analyzed in Section 3.5. It basically simplifies the design of the SCC classification algorithm.

The goal of the algorithm described in this chapter is the recognition of the basic computational kernels that are calculated in a loop nest. As will be shown in Section 3.5, we propose a classification scheme that is able to recognize the type of kernel computed in an expression e, a statement $x = rhs$ and a strongly connected component $SCC(x_{1...n})$. We use the concept of *class* for representing the different types of kernels. The notations $[e]$, $[x = rhs]$ and $[SCC(x_{1...n})]$ will be used for the class of an expression, a statement and a component, respectively.

The taxonomy of SCCs that will be presented in Section 3.4 shows the different classes of SCCs that may appear in a GSA graph. The class of a scalar component $[SCC_C^S(x_{1...n})]$ is represented as a pair, c_1/c_2, that indicates the conditionality, c_1, and the type of scalar kernel, c_2 (see, e.g., linear induction variable, linked-list traversal, etc. in Section 2.2). An example is *non-cond/lin*, which denotes a non-conditional linear induction variable. The class of an array component $[SCC_C^A(x_{1...n})]$ is represented by a triplet, $c_1/c_2/c_3$, composed of the conditionality c_1, the type of array operation c_2 (see, e.g., assignments, reductions and recurrences in Section 2.2.5), and the type of scalar kernel, c_3, computed by the index expression of the array reference that appears in the left-hand side of the statements of the component. An example is *cond/reduc/subs*, which denotes a conditional irregular reduction.

Our compiler framework is based on the representation of loop nests as SCC graphs that show the relationships (i.e. the dependences) that exist between the basic kernels computed in the SCCs of the GSA graph. In the following definitions we extend the concept of dependence between statements to the concept of dependence between SCCs, which will be used in Section 3.5.

Definition 3.7. *Let stm_1 and stm_2 be two statements included in the components $SCC(x_{1...n})$ and $SCC(y_{1...m})$, respectively. We say that there is a **direct use-def chain** between the components, $SCC(x_{1...n}) \rightarrow SCC(y_{1...m})$, if there is a direct use-def chain between stm_1 and stm_2, i.e., if the left-hand side or the right-hand side expressions of stm_1 contain at least one occurrence of the variable defined in stm_2.*

The concept defined above can be generalized to the concept of **indirect use-def chain** between two strongly connected components, which will be denoted as $SCC(x_{1...n}) \stackrel{+}{\rightarrow} SCC(y_{1...m})$.

Definition 3.8. *A component $SCC(x_{1\ldots n})$ is* ***independent*** *if it is not the source of any use-def chain.*

Definition 3.9. *Two components $SCC(x_{1\ldots n})$ and $SCC(y_{1\ldots m})$ are* ***mutually dependent*** *iff $SCC(x_{1\ldots n}) \xrightarrow{+} SCC(y_{1\ldots m})$ and $SCC(y_{1\ldots m}) \xrightarrow{+} SCC(x_{1\ldots n})$.*

At this point we can define the SCC graph, which is the output of the classification algorithm described in Section 3.5, as well as the input of the loop classification scheme presented in Chapter 4. First, we define several classes of use-def chains. The goal of these classes is to highlight relevant dependences from the viewpoint of the last stage of the compiler framework construction, *Recognition of Loop-Level Computational Kernels* in Fig. 3.1. The details can be consulted in Chapter 4.

Definition 3.10. *Let $SCC(x_{1\ldots n}) \rightarrow SCC(y_{1\ldots m})$ be a use-def chain between two SCCs of cardinality zero or one. Let e be an expression within a statement of $SCC(x_{1\ldots n})$ that contains an occurrence y_k of the variables $y_1, \ldots, y_m$ of the target component $SCC(y_{1\ldots m})$. We say that $SCC(x_{1\ldots n}) \rightarrow SCC(y_{1\ldots m})$ is a* ***control use-def chain*** *if e is the conditional expression of a γ-statement of $SCC(x_{1\ldots n})$. If e is the left-hand side or the right-hand side expression of a source code assignment statement associated with $SCC(x_{1\ldots n})$, then other two types of chains are distinguished. We say that $SCC(x_{1\ldots n}) \rightarrow SCC(y_{1\ldots m})$ is a* ***structural use-def chain*** *if one of the following properties is fulfilled:*

1. *$SCC(x_{1\ldots n})$ and $SCC(y_{1\ldots m})$ are scalar strongly connected components whose variables $x_1, \ldots, x_n, y_1, \ldots, y_m$ are different definitions of a unique scalar variable in the source code.*

2. *$SCC(x_{1\ldots n})$ is an array component of class $c_1/c_2/c_3$, $SCC(y_{1\ldots m})$ is a scalar component of class c_4/c_5, e consists of a reference to the array variable associated with $SCC(x_{1\ldots n})$, and the classes c_3 and c_5 are equal.*

Otherwise, $SCC(x_{1\ldots n}) \rightarrow SCC(y_{1\ldots m})$ is a ***non-structural use-def chain****.*

Definition 3.11. *Given a program in GSA form, we define the* ***SCC graph*** *as the directed graph whose nodes are the SCCs that appear in the GSA graph, and whose edges are the control, structural and non-structural use-def chains between pairs of SCCs.*

The GSA form captures the flow of values in a program by inserting special μ, γ and α operators in the code. Some special types of SCCs are defined on the basis of the special operators they contain.

Definition 3.12. *A component $SCC(x_{1...n})$ is **wrap-around** if it is only composed of μ-statements. Otherwise, it is **non-wrap-around**.*

Definition 3.13. *A strongly connected component $SCC(x_{1...n})$ is **virtual** if it contains a combination of μ and γ-statements only, with at least one γ-statement.*

3.3 Strongly Connected Components in GSA Graphs

The first stage of the construction of the SCC graph is the identification and classification of the SCCs that appear in GSA graph of the loop body. Several methods for searching SCCs in graphs have been proposed in the literature. The classical Tarjan algorithm [51] or more recent searching methods [16, 24] can be used for this purpose. Gerlek, Stoltz and Wolfe [18] proposed a classification scheme where any given SCC will first demand the classification of any SCC it requires for its own classification. The Tarjan algorithm supports this *demand-driven* classification scheme naturally because, in that algorithm, a SCC is not found until all the descendent SCCs in the graph have been found. This characteristic enables to perform the SCC search and the SCC classification in one step only.

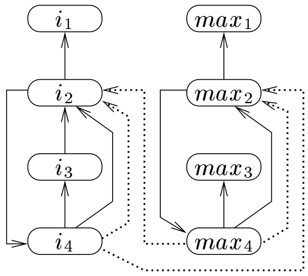
The SCCs are intended to represent the basic computational kernels that are calculated during the execution of the loop. In order to achieve this goal, we will build the SCCs by ignoring the control use-def chains of the GSA graph. Let us study the effects of this decision with the code fragment shown in Fig. 3.3. Control use-def chains are highlighted by depicting them as dotted edges in the GSA graph of Fig. 3.3. If the control use-def chains introduced by the conditional expression $max_2 < a(i_2)$ are ignored, the loop body contains two components $SCC(i_{2...4})$ and $SCC(max_{2,4})$ that represent a conditional induction variable and a maximum scalar reduction, respectively. However, if those dependences are considered, the two computational kernels are captured by a unique component $SCC(i_{2...4}, max_{2,4})$. Unlike the latter approach, the

```
max = a(1)
i = 1
DO h = 1,n
   IF (max < a(i)) THEN
      max = a(i)
      i = i + 1
   END IF
END DO
```

(a) Source code.

```
max_1 = a(1)
i_1 = 1
DO h_1 = 1,n,1
   i_2 = μ(i_1,i_4)
   max_2 = μ(max_1,max_4)
   IF (max_2 < a(i_2)) THEN
      max_3 = a(i_2)
      i_3 = i_2 + 1
   END IF
   i_4 = γ(max_2 < a(i_2),i_3,i_2)
   max_4 = γ(max_2 < a(i_2),max_3,max_2)
END DO
```

(b) GSA form.



(c) GSA graph.

Figure 3.3: Effect of searching SCCs in GSA graphs with information about control use-def chains.

first enables the separation of the basic kernels. The information about the flow of values that is provided by the control use-def chains is determinant for the recognition of the kernels computed in a loop. Thus, although it is not considered for the analysis of the SCCs, it will be taken into account during the execution of the SCC graph classification algorithm presented in Chapter 4.

As stated above, the *conditional consecutively written array* computations analyzed in Section 2.3 will be used to illustrate the construction of our compiler framework, as well as its application to the automatic parallelization of sequential codes. The source code and the corresponding GSA form and GSA graph were presented in Fig. 3.2. In particular, focus on the induction variable i, which is represented by the definitions i_1, i_2, i_3 and i_4 in the GSA form. Starting at the μ that defines i_2, the external definition i_1 determines the value at the beginning of the first loop iteration. On subsequent iterations, the value is that of the internal definition i_4 at the statement $i_4 = \gamma(c(h_1), i_3, i_2)$. The γ operator represents that, if the condition $c(h_1)$ is true in the loop iteration h_1, then i_4 takes the value i_3 defined in the body of the if-endif construct. Otherwise, it takes the value i_2 defined at the μ-statement at the beginning of the iteration. The right-hand side of the statement $i_3 = i_2 + 1$ also fetches the value i_2 assigned by the μ operator. The flow of values during the execution of the loop is represented in the GSA use-def chain graph of Fig. 3.2(c). A directed edge denotes a use-def chain between two statements of the GSA form. The labels of the nodes represent the left-hand side of the statements. The flow of values described above for the induction variable i defines two cycles that we denote as $< i_2, i_4 >$ and $< i_2, i_3, i_4 >$. The key observation is that there is a component $SCC_1^S(i_2, i_3, i_4)$ associated with the induction variable i, as i is defined in terms of itself in the loop body.

A similar scenario arises for the computation of array a. The key observation here is that there is not any occurrence of a in the right-hand side of the source code statement $a(i) = tmp + 2$. In GSA form, however, the corresponding statement $a_2 = \alpha(a_1, i_2, tmp_2 + 2)$ contains one occurrence of a. The reason for this is that the α operator represents the definition of the element $a_2(i_2)$ in terms of the previous value of the whole array. In this case, the value a_1 that was assigned by the μ operator inserted at the beginning of the loop body. Apart from the occurrence inserted by the α operator, array reductions and array recurrences contain at least one occurrence in the right-hand side expression. This discussion leads to the important conclusion that every array assignment statement is contained in one SCC in GSA form. In our example code, the flow of values for array a is represented by two cycles $< a_1, a_3 >$ and $< a_1, a_2, a_3 >$. The maximal cycle is a strongly connected component $SCC_1^A(a_1, a_2, a_3)$.

On the other hand, the loop body contains a scalar variable tmp that is not defined in terms of itself. In GSA form, there is a trivial component $SCC_1^S(tmp_2)$ (see Def. 3.2) associated with $tmp_2 = f(h_1)$. The important fact here is that tmp is assigned a value in the loop iterations where the condition $c(h_1)$ is true. This data flow is represented in GSA form by inserting two new statements $tmp_1 = \mu(tmp_0, tmp_3)$ and $tmp_3 = \gamma(c(h_1), tmp_2, tmp_1)$ that define a non-trivial virtual component $SCC_0^S(tmp_1, tmp_3)$ (see Defs. 3.2

and 3.13). Note that the cardinality (Def. 3.4) is zero because the two statements of the SCC were inserted in the code during the translation of the program into GSA form.

As shown above, induction variables are represented by non-trivial SCCs. The index of the DO-loop do_{h_1} computes a non-conditional induction variable. However, the special syntax of the DO-loop headers causes this induction variable to be represented by a trivial component $SCC_1^S(h_1)$. For the sake of clarity, this information was omitted from the GSA graph of Fig. 3.2(c).

3.4 Taxonomy of GSA Strongly Connected Components

The *class of a SCC* represents the type of basic computational kernel that is calculated in the statements of the component. The analysis of the characteristics of the statements allows several classes of SCCs to be distinguished. The *class of a SCC* will represent the type of kernel associated with the component (the set of kernels considered in this work was described in Section 2.2). In this section, a taxonomy of the classes of SCCs that may appear in the GSA graph is presented. Notations for the classes of scalar SCCs and array SCCs were defined in Section 3.2.

The taxonomy is defined using five properties of the strongly connected components: trivial/non-trivial (Def. 3.2), structural/semantic (Def. 3.3), cardinality (Def. 3.4), scalar/array (Def. 3.5) and conditionality (Def. 3.6). A graphical depiction is shown in Fig. 3.4. It should be noted that the classification of SCCs with cardinality greater than one is outside the scope of this thesis because they represent a small percentage of the classes of SCCs that appear in our benchmark suite. Furthermore, the expressions within statements are supposed to be simplified algebraically. If the left-hand side variable of the statement is a scalar, after simplification the right-hand side expression will contain at most one occurrence of the variable. Some exceptions to this rule are described in [18], but the codes included in our benchmark suite do not contain such exceptional cases. For array statements, the same will occur except if there is, at least, one subscript expression of a right-hand side array reference that does not match the subscript expression of the left-hand side.

- SCC classes
  - Trivial
    - none *{none}*
    - invariant *{inv}*
    - linear *{lin}*
    - reduction *{red}*
    - subscripted *{subs}*
    - unknown *{unk}*
  - Non−trivial
    - Structural
      - Cardinality 0
        - (non−)conditional/none *{(non−)cond/none}*
        - (non−)conditional/invariant *{(non−)cond/inv}*
        - (non−)conditional/linear *{(non−)cond/lin}*
        - (non−)conditional/reduction *{(non−)cond/red}*
        - (non−)conditional/subscripted *{(non−)cond/subs}*
        - (non−)conditional/unknown *{(non−)cond/unk}*
      - Cardinality 1
        - Scalar
          - (non−)conditional/linear *{(non−)cond/lin}*
          - (non−)conditional/reduction *{(non−)cond/red}*
          - (non−)conditional/list *{(non−)cond/list}*
          - (non−)conditional/unknown *{(non−)cond/unk}*
        - Array
          - (non−)conditional/assignment/invariant *{(non−)cond/assig/inv}*
          - (non−)conditional/assignment/linear *{(non−)cond/assig/lin}*
          - (non−)conditional/assignment/subscripted *{(non−)cond/assig/subs}*
          - (non−)conditional/assignment/unknown *{(non−)cond/assig/unk}*
          - (non−)conditional/reduction/invariant *{(non−)cond/reduc/inv}*
          - (non−)conditional/reduction/linear *{(non−)cond/reduc/lin}*
          - (non−)conditional/reduction/subscripted *{(non−)cond/reduc/subs}*
          - (non−)conditional/reduction/unknown *{(non−)cond/reduc/unk}*
          - (non−)conditional/recurrence/invariant *{(non−)cond/recur/inv}*
          - (non−)conditional/recurrence/linear *{(non−)cond/recur/lin}*
          - (non−)conditional/recurrence/subscripted *{(non−)cond/recur/subs}*
          - (non−)conditional/recurrence/unknown *{(non−)cond/recur/unk}*
    - Semantic
      - Cardinality 0
        - scalar−maximum
        - scalar−minimum
        - scalar−find−and−set
        - scalar−unknown
      - Cardinality 1
        - array−maximum
        - array−minimum
        - array−find−and−set
        - array−unknown

Figure 3.4: Taxonomy of strongly connected components in GSA graphs. Abbreviations of SCC classes are written in italic font within braces. Classes that contain the word *unknown* represent computations that do not fulfill the properties of the other SCCs.

3.4.1 Trivial SCCs

Trivial components, $SCC(x)$, consist of only one node that is associated with a scalar assignment statement $x = rhs$, where x is a scalar variable and rhs is an expression that does not contain any occurrence of x. Variable x is not defined in terms of itself, otherwise the component would be composed of at least two nodes. The class of trivial SCCs will not be decomposed into subclasses according to the criteria used in the taxonomy of Fig. 3.4. The following lemmas are immediate consequences of the definition of a trivial SCC (see Def. 3.2).

Lemma 3.3. *The cardinality of a trivial SCC is always one. The only exceptions are virtual and wrap-around SCCs, whose cardinality is zero.*

Lemma 3.4. *Trivial SCCs are structural and scalar.*

The first trivial class shown in the taxonomy is *none*, which does not represent any type of kernel. It will be used in the SCC classification algorithm (see Section 3.5) as the initial value of all the classes. The last class *unknown*, captures those trivial SCCs whose computational kernel cannot be successfully classified. Several trivial classes are distinguished by the characteristics of the right-hand side expression rhs. If rhs does not contain array references, two classes of trivial SCCs are distinguished:

- *invariant*: Let $\{x_1, \ldots, x_n\}$ be the set of scalar/array variables that are referenced in rhs. The component is invariant if $\forall x_k$ $(k = 1, \ldots, n)$, x_k is loop-invariant, i.e. it is not defined within the loop body.

- *linear*: Let $\{x_1, \ldots, x_n\}$ be the set of variables that are referenced in rhs. The SCC is linear if (1) $\exists x_k$ $(k \in \{1, \ldots, n\})$, where x_k is a linear induction variable (see *induction variables* in Section 2.2.1) defined in another SCC within the loop body, and (2) $\forall x_j$ $(j \in \{1, \ldots, n\},\ k \neq j)$, x_j is loop-invariant.

- *reduction*: The component is reduction if the set of variables $\{x_1, \ldots, x_n\}$ contains at least one occurrence of a scalar reduction variable (see scalar reduction operations in Section 2.2.2). The remaining occurrences may be either linear induction variables or loop-invariant variables.

The trivial classes defined above are associated with some sequences defined in [18]. However, expressions involving array references are not considered in that work. In order to cope with such expressions, we define a new class of trivial component:

- *subscripted*: The component is subscripted if the expression rhs fulfills the same properties as linear trivial SCCs, the difference being that at least one x_j is an occurrence of an array variable whose subscript expression is loop-variant, i.e., its value changes in each loop iteration.

The generic implementation of the consecutively written array kernel shown in Fig. 3.2(a) contains a scalar temporary variable tmp that is represented as a trivial component $SCC_1^S(tmp_2)$ of class *subscripted* in the GSA graph of Fig. 3.2(c), the subscript h_1 being the loop-variant expression. The source code of the gather operation presented in Fig. 3.5 contains two scalar variables l and m that were inserted for illustrative purposes only. They are represented as $SCC_1^S(l_2)$ and $SCC_1^S(m_2)$ of class *invariant* and *linear*, respectively. In the example, some trivial SCCs of cardinality zero also appear, namely, $SCC_0^S(l_2)$, $SCC_0^S(m_2)$ and $SCC_0^S(j_2)$.

3.4.2 Non-trivial SCCs

Non-trivial components consist of at least two nodes of the GSA graph. As a consequence, non-trivial SCCs may contain μ-statements, γ-statements, α-statements or scalar assignment statements, but they always have at least one μ-statement.

i. Structural SCCs

Two criteria will be used for further classification of structural SCCs (see the taxonomy of Fig. 3.4): cardinality (Def. 3.4) and scalar/array (Def. 3.5).

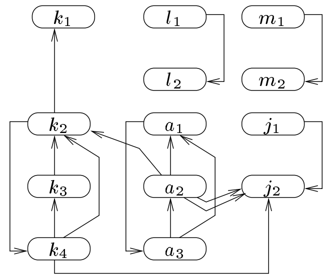
Cardinality zero. This class of structural SCC consists of a combination of μ and γ-statements only. It arises when a scalar variable is defined in a point of the loop body such that there is at least one confluence node between that point and the loop entry. Two typical scenarios are the definition of a scalar variable in the body of an if-endif construct, and in the body of an inner

$k = 1$
DO $h = 1, n$
  $l = b(2)$
  $m = h$
  $j = f(h)$
  IF $(j \neq 0)$ THEN
    $a(k) = b(j)$
    $k = k + 1$
  END IF
END DO

(a) Source code.

$k_1 = 1$
DO $h_1 = 1, n$
  $k_2 = \mu(k_1, k_4)$
  $j_1 = \mu(j_0, j_2)$
  $m_1 = \mu(m_0, m_2)$
  $l_1 = \mu(l_0, l_2)$
  $a_1 = \mu(a_0, a_3)$
  $l_2 = b(2)$
  $m_2 = h_1$
  $j_2 = f(h_1)$
  IF $(j_2 \neq 0)$ THEN
    $a_2(k_2) = \alpha(a_1, b(j_2))$
    $k_3 = k_2 + 1$
  END IF
  $k_4 = \gamma(j_2 \neq 0, k_3, k_2)$
  $a_3 = \gamma(j_2 \neq 0, a_2, a_1)$
END DO

(b) GSA form.



(c) GSA graph.

Figure 3.5: Gather operation of an array with irregular access pattern.

loop within a loop nest. The consecutively written array of Fig. 3.2(a) contains a scalar variable that corresponds to the first scenario described above. Consequently, a structural component $SCC_0^S(tmp_{1,3})$ arises in the GSA graph.

Cardinality one. This class represents the computations associated with one variable of the original source code that is defined in terms of itself. The SCC contains at least one μ-statement. Structural SCCs with cardinality one are divided into scalar SCCs and array SCCs.

Some scalar sequences associated with scalar structural SCCs are presented in [18]: *linear induction variables*, and a set of *monotonic sequences*. The example code shown in Fig. 3.2(a) contains a conditional linear induction variable (see Section 2.2.1) that is represented by a structural component $SCC_1^S(i_2, i_3, i_4)$ of class *conditional/linear*. In order to cope with array references, we define two additional classes:

- *(non-)conditional/reduction* (see Section 2.2.2): The operations within the component contain at least one loop-variant array reference. An example code is the computation of the sum/product of the elements of an array.
- *(non-)conditional/list* (see Section 2.2.3): No arithmetic operator is allowed within the component. Only array references whose subscript ex-

pressions consist of one occurrence of the variable defined in the SCC are allowed. An example code is the traversal of a linked list implemented by means of an array, i.e., $i = next(i)$ within the loop body.

Regarding array structural SCCs, several subclasses are also distinguished. This class of SCCs enables the recognition of a set of array operations described in Section 2.2.5. Let $a(s) = rhs$ be an array assignment statement where the $s - th$ element of the array a is assigned the value of expression rhs. Different classes are distinguished according to the number of occurrences of the left-hand side array reference, $a(s)$, that appear in rhs:

- *(non-)conditional/assignment/*$[s]$, if there are zero occurrences.
- *(non-)conditional/reduction/*$[s]$, if there is one occurrence matching $a(s)$.
- *(non-)conditional/recurrence/*$[s]$, if the expression rhs contains a set of occurrences $\{a(s_1), \ldots, a(s_r)\}$ such that $\exists s_k \; k \in \{1, \ldots, r\} \; / \; s_k \neq s$.

The occurrences cannot appear in the subscript expression of any array reference. The class [s] denotes the class of the left-hand side subscript expression s, which is determined by checking the same properties as the right-hand side expressions associated with trivial components (see Section 3.4.1). Note that in the taxonomy of Fig. 3.4 the trivial classes *none* and *reduction* are not shown as possible values for [s]. In the first case, this is because *none* will be used in the SCC classification algorithm (see Section 3.5) just as an initial value before running the algorithm. The class *reduction* was excluded because there is not any case in our benchmark suite. In fact, it is not a common programming practice to use reduction variables to reference the entries of an array. The computations of array a in Fig. 3.2(a) are captured as an array structural component $SCC_1^A(a_1, a_2, a_3)$ of class *cond/assig/lin.*

ii. Semantic SCCs

Semantic SCCs are associated with kernels where the current value of a variable is tested in an if-endif conditional statement before computing the next value of the variable. Semantic SCCs are conditional because they always contain at least one γ-statement. Like structural SCCs, semantic SCCs are classified according to their cardinality (Def. 3.4). The detection of some kernels associated with semantic SCCs (e.g. minimum (maximum) reduction operations) was also addressed in [44].

Cardinality zero. Like cardinality-0 structural SCCs, these components appear in loop bodies where a scalar variable v is defined inside an if-endif construct. Let c represent the condition defined by the γ-statements of the SCC. Several subclasses are distinguished according to the characteristics of c:

- *Scalar minimum* and *scalar maximum* (see Section 2.2.2): The condition c matches $v < e$, $v \leq e$, $e > v$ or $e \geq v$ for minimum. It matches $e < v$, $e \leq v$, $v > e$ or $v \geq e$ for maximum. A representative source code pattern is the computation of the minimum value of an n-dimensional array (see $SCC(m_3, m_5)$ in the innermost loop do_{h_2} of Fig. 3.6).
- *Scalar find-and-set* (see Section 2.2.4): The conditional expression c matches $(v == e).AND.e'$ where e and e' are expressions that do not contain occurrences of v. Furthermore, variable v must be set to a value different to e before starting the execution of the loop.

As shown in Section 2.2, the recognition of the kernels maximum, minimum and find-and-set requires checking of some properties, both in a conditional expression and in a scalar assignment statement. Within the scope of these components, only the information about the condition is available. Thus, the SCC classification scheme can only detect components that are candidates to be classified as *scalar-minimum (maximum)* and *scalar-find-and-set*. As will be shown in Chapter 4, all the information will be available during the execution of the SCC graph classification scheme.

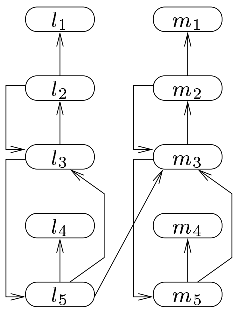
Cardinality one. These semantic components appear in loop bodies where an array variable is defined inside an if-endif construct. Consequently, they contain at least one α-statement. Three classes are distinguished: *array-maximum*, *array-minimum* and *array-find-and-set*. These classes verify the same conditions as *scalar-maximum*, *scalar-minimum* and *scalar-find-and-set*, respectively, the difference being that an array reference is involved, not a scalar variable. In this case, the assignment statement belongs to the SCC.

```
m = a(f(1))
l = f(1)
DO row = n1, n2
   DO h = begin(row), end(row)
      IF (a(f(h)) < m) THEN
         m = a(f(h))
         l = f(h)
      END IF
   END DO
END DO
```

(a) Source code.

```
m1 = a(f(1))
l1 = f(1)
DO row1 = n1, n2
   l2 = μ(l1, l3)
   m2 = μ(m1, m3)
   h1 = μ(h0, h2)
   DO h2 = begin(row1), end(row1)
      l3 = μ(l2, l5)
      m3 = μ(m2, m5)
      IF (a(f(h2)) < m3) THEN
         m4 = a(f(h2))
         l4 = f(h2)
      END IF
      l5 = γ(a(f(h2)) < m3, l4, l3)
      m5 = γ(a(f(h2)) < m3, m4, m3)
   END DO
END DO
```

(b) GSA form.



(c) GSA graph.

Figure 3.6: Computation of the minimum value of the rows of a sparse matrix.

Thus, the necessary information is available and the computational kernel is detected in the SCC classification algorithm. A simple example of *array-find-and-set* is $SCC_1^A(a_1, a_2, a_3)$ in the masked operation shown in Fig. 3.7.

3.5 Strongly Connected Component Classification Algorithm

The basis of our compiler framework is the representation of the source code of the loop nests as SCC use-def chain graphs. The information needed for the construction of the SCC graph (see Def. 3.11) is retrieved from the source code during the execution of the SCC classification algorithm. The explanation of this algorithm is the main topic of our work [7].

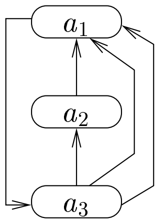
The main steps of the algorithm, which is the first stage of the construction of our framework (see Fig. 3.1), are described in Section 3.5.1. It basically consists of performing an exhaustive analysis of the statements and the expressions that compose the loop body. The algorithms for the classification of the statements and the expressions of the GSA form are presented in Sections 3.5.2 and 3.5.3, respectively.

```
DO h = 1,n
   IF (a(f(h)) ≠ 0) THEN
      a(f(h)) = 0
   END IF
END DO
```

(a) Source code.

```
DO h1 = 1,n
   a1 = μ(a0,a3)
   IF (a1(f(h1)) ≠ 0) THEN
      a2(f(h1)) = α(a1,0)
   END IF
   a3 = γ(a1(f(h1)) ≠ 0,a2,a1)
END DO
```

(b) GSA form.



(c) GSA graph.

Figure 3.7: Source code for computing a masked operation.

3.5.1 Algorithm Overview

The set of strongly connected components that represent a loop nest are classified using a non-deadlocking demand-driven algorithm that proceeds as follows. For each non-classified component, $SCC(x_{1...n})$, the component is pushed onto a stack of SCCs and its classification procedure is started:

1. If $SCC(x_{1...n})$ is independent (see Def. 3.8), then it will be successfully classified by the algorithms that will be presented in Sections 3.5.2 and 3.5.3. Once $[SCC(x_{1...n})]$ has been determined, $SCC(x_{1...n})$ is popped from the stack, and the classification process of the SCC located on top of the stack continues at the same point where it had been deferred.

2. If $SCC(x_{1...n})$ is not independent, then it is the source of a set of use-def chains. During the analysis of the statements of $SCC(x_{1...n})$, every occurrence of a variable defined in another component $SCC(y_{1...m})$ is found. For each occurrence, a use-def chain $SCC(x_{1...n}) \rightarrow SCC(y_{1...m})$ is set, the classification process of $SCC(x_{1...n})$ is deferred, and the classification of $SCC(y_{1...m})$ is started. Once all the occurrences have been processed, the classification of $SCC(x_{1...n})$ will be successfully completed. The final stage of the algorithm is the classification of the use-def chains whose source is $SCC(x_{1...n})$. According to Def. 3.10, the class of both the source and sink components is needed. Thus, use-def chains cannot be classified until $[SCC(x_{1...n})]$ has been determined. It should be noted that GSA enables finding control use-def chains while

processing the statements of a component because the γ operator captures the conditional expressions of if-endif constructs. As explained in Section 3.1, this is an advantage of the GSA form with respect to the PA-SSA representation, which does not supply such information.

The algorithm described above reaches a deadlock state when mutually dependent SCCs (see Def. 3.9) exist in the loop body, i.e., when the SCC graph contains cycles (an example loop was presented in Section 3.3). Such situations are detected by using the stack of SCCs as follows. If a use-def chain $SCC(x_{1...n}) \rightarrow SCC(y_{1...m})$ is found, the contents of the stack are checked before starting the classification of $SCC(y_{1...m})$. If $SCC(y_{1...m})$ is already in the stack, it means that an indirect use-def chain $SCC(y_{1...m}) \xrightarrow{+} SCC(x_{1...n})$ exists. Consequently, $SCC(x_{1...n})$ and $SCC(y_{1...m})$ are mutually dependent.

The taxonomy of SCCs presented in Section 3.4 defines the properties of each SCC class. The class of both structural and semantic components, $[SCC(x_{1...n})]$, is computed as follows:

1. If the component is trivial, $SCC(x_1)$, then $[SCC(x_1)] = [x_1 = e_1]$.
2. If it is non-trivial, $[SCC(x_{1...n})]$=$[x_1 = \mu(x_0, x_n)]$, where $x_1 = \mu(x_0, x_n)$ is the statement corresponding to the outermost loop of the loop nest.

Finally, if the SCC is semantic, the algorithm determines the appropriate subclass by checking the properties of the corresponding conditional expressions. It should be noted that, as described above, the problem of calculating $[SCC(x_{1...n})]$ is reduced to determine the class of a statement of the GSA representation. During the classification process, the statements of a SCC are classified on demand. In order to assure that each statement is processed only once, a stack of statements is used.

The SCC classification algorithm is *demand-driven* because if references to variables defined in statements of other SCCs exist, then such SCCs are classified first. As will be shown later in this section, this demand-driven mechanism is supported by a *contextual classification scheme* that may compute different classes for a given expression. Consider the loops presented in Fig. 3.8. Let us focus on the expression $f(i)$ to illustrate why contextual classification is needed. In Fig. 3.8(a), a linked-list traversal computational kernel is computed using the scalar variable i. The right-hand side of statement $i = f(i)$ is the expression $f(i)$, which just consists of a reference to an array variable f whose subscript expression i is the variable used to compute the linked-list traversal. According to the definition of a linked-list traversal, $[f(i)]$ and $[i = f(i)]$ should be assigned the class *non-conditional/list* (see the taxonomy of Fig. 3.4). The loop of Fig. 3.8(b) calculates a scalar reduction using the variable r, $f(i)$ being an operand of the sum operator that appears in $r = r + f(i)$. In this case, $[f(i)]$ should be assigned the class *subscripted*, which represents loop-variant expressions. On the other hand, $[r = r + f(i)]$ should be set to *non-conditional/reduction* because a loop-variant value is summed in each loop iteration. Note that if f and i were loop-invariant variables, the scalar r would be a linear induction variable. As a result, $[f(i)]$ would be *invariant*, and $[r = r + f(i)]$ would be *non-conditional/linear*.

In the rest of this chapter, the notation $[e]_{p:l,E}^{e_{ref}}$ will be used to represent the contextual class of an expression e. The parameters that define the context are: e_{ref}, the reference expression; l, the level of e within E, e being a subexpression of another expression E; and p, which indicates if E is a subexpression of the left-hand side of a statement ($\blacktriangleleft$), the right-hand side of a statement ($\blacktriangleright$), or the conditional expression of a γ-statement (denoted as ?). The concept *level of an expression* is related to the representation of expressions as trees (see [51, Chapter 3]). In this thesis, we will assume that the level of the root node is 0. We will also consider that the level of the subexpression e within E is the level of the root node of e in the tree that represents E.

We are describing a demand-driven classification scheme, whose main characteristic is that the classification of an expression requires its operands to be classified first. Thus, the class of an expression is calculated by means of a *transfer function* (denoted as T) that combines the classes of the operands. The concept of transfer function is also applied to statements, where the operands are the left-hand side and the right-hand side expressions. The expression e or the statement *stm* to which the transfer function is applied will be denoted as T_e or T_{stm}, respectively. The main complexity of the algorithms presented in the following sections is the definition of appropriate transfer functions that enable the recognition of the collection of computational kernels described in Section 2.2.

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

(a) Source code of a loop that contains a linked-list computational kernel.

```
DO h = 1, f_size
   ...
   i = h + 1
   r = r + f(i)
END DO
```

(b) Loops where a scalar reduction operation is computed.

Figure 3.8: Example codes to illustrate why expressions have to be analyzed with a contextual classification scheme.

3.5.2 Classification of GSA Statements

Let $SCC(x_{1...n})$ and $x_j = rhs_j$ be, respectively, a SCC of cardinality zero or one and a GSA statement of the component. The class of a statement is determined by applying a transfer function that takes into account the classes of the left-hand side and right-hand side expressions, x_j and rhs_j. The transfer functions for the different types of GSA statements are defined as follows. The first function copes with the special syntax of the statements associated with loop headers:

$$T_{\text{DO-stm}} : [do\, v = e_{init}, e_{limit}, e_{step}] = \begin{cases} unk & \text{if } [e_{init}]^{v}_{\blacktriangleright:1,e_{init}} {=} unk \text{ or} \\ & [e_{limit}]^{v}_{\blacktriangleright:1,e_{limit}} {=} unk \text{ or} \\ & [e_{step}]^{v}_{\blacktriangleright:1,e_{step}} {=} unk \\ subs & \text{if } [e_{init}]^{v}_{\blacktriangleright:1,e_{init}} {=} subs \text{ or} \\ & [e_{limit}]^{v}_{\blacktriangleright:1,e_{limit}} {=} subs \text{ or} \\ & [e_{step}]^{v}_{\blacktriangleright:1,e_{step}} {=} subs \\ lin & \text{otherwise} \end{cases} \quad (3.1)$$

The index variable computes a linear induction variable except if at least one of the *init*, *limit* and *step* expressions contains array references (class *subs*), or if it cannot be successfully classified (class *unk*).

The transfer functions for the virtual μ and γ-statements are rather simple. The statement inherits the class of the right-hand side expression, i.e. that of the GSA operator that captures the flow of values in loop headers and if-endif constructs.

$$
\begin{aligned}
T_{\mu\text{-stm}} &: \quad [x = \mu(x_{out}, x_{in})] = [\mu(x_{out}, x_{in})]^{x}_{\blacktriangleright:0,\mu(x_{out},x_{in})} && (3.2)\\
T_{\gamma\text{-stm}} &: \quad [x = \gamma(c, x_{in}, x_{out})] = [\gamma(c, x_{in}, x_{out})]^{x}_{\blacktriangleright:0,\gamma(c,x_{in},x_{out})} && (3.3)
\end{aligned}
$$

The key idea behind the classification of scalar/array assignment statements is to scan the right-hand side expression looking for subexpressions that match a scalar/array reference. This reference expression is the left-hand side of the statement in the source code. This task is accomplished in the contextual classification algorithm that is applied to expressions. For this reason, the transfer functions of the scalar and array assignment statements are defined in a simple manner:

$$
\begin{aligned}
T_{\text{scalar-stm}} &: \quad [v = rhs] = [rhs]^{v}_{\blacktriangleright:0,rhs} && (3.4)\\
T_{\alpha\text{-stm}} &: \quad [a = \alpha(a_{prev}, s, rhs)] = [\alpha(a_{prev}, s, rhs)]^{a(s)}_{\blacktriangleright:0,\alpha(a_{prev},s,rhs)} && (3.5)
\end{aligned}
$$

The transfer functions presented above (Eqs. (3.1) to (3.5)) reduce the classification of a statement to determining the contextual class of the right-hand side expression. In the following section the contextual classification of expressions is described in detail.

3.5.3 Contextual Classification of Expressions

Let $SCC(x_{1\ldots n})$ be a component of cardinality zero or one. Let $x_j = rhs_j$ be a statement of $SCC(x_{1\ldots n})$. Let e be a generic expression, represented as $\oplus(e_1, \ldots, e_r)$, which is a subexpression at level l of another expression E, which may be either x_j, rhs_j or the arguments of a GSA operator. The contextual class of the expression, $[e]^{e_{ref}}_{p:l,E}$, is determined by performing a post-order traversal of e, which is naturally supported by the demand-driven GSA form. First, the classes of the operands $[e_i]^{e_{ref}}_{p:(l+1),E}$ $(i = 1, \ldots, r)$ are computed. Finally, a transfer function $T_{\oplus}$ combines $[e_i]^{e_{ref}}_{p:(l+1),E}$ $(\forall i)$ to derive $[e]^{e_{ref}}_{p:l,E}$. Note that the parameters e_{ref}, p and E are preserved across the post-order traversal, while the level l increases as the traversal advances.

In the rest of this section, we propose a set of transfer functions that enable the classification of expressions. Different types of expressions are considered: loop-invariant expressions, identifiers of scalar/array variables, array references, arithmetical and logical expressions, and special GSA expressions. The transfer functions presented in this work have been adjusted to classify the expressions that appear in our benchmark suite. However, although some changes may be needed, these functions are expected to enable the recognition of the set of kernels considered in this work in other real codes.

Loop-invariant Expressions

A loop-invariant expression, K, includes constants and references to variables that are not modified in the body of a loop. The transfer function T_K for the classification of K is:

$$T_K : [K]^{e_{ref}}_{p:l,E} = \begin{cases} inv & \text{if } e_{ref} \equiv v \text{ is scalar} \\ assig/[s_a]^{a(s_a)}_{\blacktriangleleft:1,a(s_a)} & \text{if } e_{ref} \equiv a(s) \text{ is array, } l = 0 \\ inv & \text{if } e_{ref} \equiv a(s) \text{ is array, } l > 0 \end{cases} \tag{3.6}$$

where the symbol "$\equiv$" means that the left-hand side and the right-hand side expressions match.

In Section 3.2, we introduced notations for the classes of scalar and array SCCs. Scalar and array classes were denoted by tuples whose first component represents the conditionality of the SCC. For the sake of clarity, this information was removed from the classes shown in T_K. However, no information was lost because T_K does not modify the first component of the tuple. In fact, the first component of the tuple is only modified in the transfer function of the γ special operator, T_γ, which will be presented later in this section. This notation will also be used in the description of the remaining functions.

Identifiers of Scalar/Array Variables

The transfer function, T_y, of the identifier of a scalar/array variable, y, proceeds as follows. First, the SCC where the value of y is defined is searched in the set of SCCs of the loop. Next, several cases are distinguished:

1. *There is not any SCC where y is defined.* Thus, y is a loop-invariant variable, and is classified by applying the transfer function of Eq. (3.6).

2. *The definition statement of y and the statement that contains the occurrence of y are included in the same component $SCC(x_{1...n})$.* According

to Lemma 3.2, the identifier y matches x_k ($k \in \{1,\ldots,n\}$) because $|SCC(x_{1\ldots n})|$ is zero or one. Let $x_k = rhs_k$ and $x_j = rhs_j$ ($j \neq k$) be the statements where the variable is defined and referenced, respectively. Let $stack_{stm}$ be the stack of statements whose classification process has been deferred. The class is determined as follows:

$$T_y : [x_k]_{p:l,E}^{e_{ref}} = \begin{cases} [x_k = rhs_k] & \text{if } x_k \notin stack_{stm} \\ lin & \text{if } x_k \in stack_{stm}, \\ & \text{and } x_k = rhs_k \text{ is a } \mu\text{-statement}, \\ & \text{and } x_k \text{ is a scalar variable}, \\ & \text{and } x_j = rhs_j \text{ is a scalar or } \alpha\text{-statement} \\ none & \text{if } x_k \in stack_{stm}, \\ & \text{and } x_k = rhs_k \text{ is a } \mu\text{-statement}, \\ & \text{and } x_k \text{ is a scalar variable}, \\ & \text{and } x_j = rhs_j \text{ is a } \mu \text{ or } \gamma\text{-statement} \\ none & \text{if } x_k \in stack_{stm}, \\ & \text{and } x_k = rhs_k \text{ is a } \mu\text{-statement}, \\ & \text{and } x_k \text{ is an array variable} \\ unk & \text{otherwise} \end{cases} \tag{3.7}$$

In the first entry of the transfer function, the classification process of the definition statement $x_k = rhs_k$ is launched because $x_k = rhs_k$ is a statement of $SCC(x_{1\ldots n})$ that has not been visited yet. The second entry also deserves to be commented because it copes with the detection of occurrences of e_{ref} in the statement whose classification process is in progress, i.e., $x_j = rhs_j$. When the occurrence x_k corresponds to a scalar variable defined in a μ-statement of $SCC(x_{1\ldots n})$, it means that there will be a loop-carried dependence during the execution of the loop. This information enables the recognition of computational kernels such as induction variables or scalar reduction operations. In fact, the occurrence x_k is assigned a candidate class *lin*, which will be adjusted later by the transfer functions of the other types of expressions, for example, arithmetical expressions.

3. *The variable is defined in a different component* $SCC(y_{1\ldots m})$. Let $y_j = rhs_j$ ($j \in \{1,\ldots,m\}$) be the definition statement of the variable in $SCC(y_{1\ldots m})$.

Let $x_k = rhs_k$ ($k \in \{1, \ldots, n\}$) be the statement with the occurrence of the variable in $SCC(x_{1\ldots n})$. Let $stack_{scc}$ be the stack of SCCs whose classification process has been deferred. The transfer function proceeds as described next. First, the class $[SCC(y_{1\ldots m})]$ is computed by running the SCC classification algorithm. When this algorithm finishes, $[y_j]_{p:l,E}^{e_{ref}}$ is calculated as follows. First, we show the cases where the occurrence cannot be successfully classified:

$$T_y : [y_j]_{p:l,E}^{e_{ref}} = \begin{cases} unk & \text{if } SCC(y_{1\ldots m}) \in stack_{scc} \\ unk & \text{if } SCC(y_{1\ldots m}) \notin stack_{scc}, \\ & \text{and } x_k = rhs_k \text{ is a scalar or } \alpha \text{ statement,} \\ & \text{and } [SCC(y_{1\ldots m})] = unk \end{cases} \tag{3.8}$$

The classification is successful under the following conditions:

$$T_y : [y_j]_{p:l,E}^{e_{ref}} = \begin{cases} none & \text{if } x_k = rhs_k \text{ is a } \mu \text{ or } \gamma \text{ statement} \\ [SCC(y_{1\ldots m})] & \text{if } x_k = rhs_k \text{ is a scalar or } \alpha \text{ statement,} \\ & \text{and } [SCC(y_{1\ldots m})] \neq unk, \\ & \text{and } SCC(y_{1\ldots m}) \text{ is scalar, } l = 0, \\ & \text{and } SCC(x_{1\ldots n}) \text{ is scalar} \\ assig/none & \text{if } x_k = rhs_k \text{ is a scalar or } \alpha \text{ statement,} \\ & \text{and } [SCC(y_{1\ldots m})] \neq unk, \\ & \text{and } SCC(y_{1\ldots m}) \text{ is scalar, } l = 0, \\ & \text{and } SCC(x_{1\ldots n}) \text{ is array} \\ [SCC(y_{1\ldots m})] & \text{if } x_k = rhs_k \text{ is a scalar or } \alpha \text{ statement,} \\ & \text{and } [SCC(y_{1\ldots m})] \neq unk, \\ & \text{and } SCC(y_{1\ldots m}) \text{ is scalar, } l > 0 \\ subs & \text{if } x_k = rhs_k \text{ is a scalar or } \alpha \text{ statement,} \\ & \text{and } [SCC(y_{1\ldots m})] \neq unk, \\ & \text{and } SCC(y_{1\ldots m}) \text{ is array,} \\ & \text{and } SCC(x_{1\ldots n}) \text{ is scalar} \\ [SCC(y_{1\ldots m})] & \text{if } x_k = rhs_k \text{ is a scalar or } \alpha \text{ statement,} \\ & \text{and } [SCC(y_{1\ldots m})] \neq unk, \\ & \text{and } SCC(y_{1\ldots m}) \text{ is array,} \\ & \text{and } SCC(x_{1\ldots n}) \text{ is array} \end{cases} \tag{3.9}$$

It should be noted that the condition $SCC(y_{1...m}) \notin stack_{scc}$ is fulfilled in all cases. Finally, a use-def chain $SCC(x_{1...n}) \rightarrow SCC(y_{1...m})$ is defined in the SCC graph. A label that gathers useful information is attached next to the use-def chains in the graph. The label is calculated in the following manner:

$$label = \begin{cases} E' & \text{if } p = ? \\ lhs\_index : E' & \text{if } p = \blacktriangleleft,\ \textit{subscript-level} > 0 \\ rhs\_index : E' & \text{if } p = \blacktriangleright,\ \textit{subscript-level} > 0 \\ rhs : E' & \text{if } p = \blacktriangleright,\ \textit{subscript-level} = 0 \end{cases} \tag{3.10}$$

where E' is the scalar/array fetch of minimum level within E that contains the occurrence y_j. The term *subscript-level* represents the indirection level of y_j within E, i.e., the number of array references that there are in the path from the root node of E until the node of y_j.

Finally, we want to remark the fact that, as shown above, the demand-driven mechanisms used to classify SCCs and statements are properly launched in this transfer function as the occurrences of the different variables are found during the analysis.

Array References

The contextual classification of expressions, $[e]^{e_{ref}}_{p:l,E}$, basically searches for occurrences of e_{ref} in e. Occurrences of scalar variables are recognized by the transfer function of the identifier of a variable. In contrast, occurrences of array references are analyzed in the transfer function, $T_{a(s_a)}$, presented next. Let $x(s_x)$ and $a(s_a)$ represent the reference and the target expressions e_{ref} and e, respectively. Several cases are distinguished. First, assume that a and x represent different occurrences of the same array variable of the source code:

$$T_{a(s_a)} : [x(s_a)]^{x(s_x)}_{p:l,E} = \begin{cases} \textit{reduc}/[s_a]^{x(s_x)}_{\blacktriangleleft:1,E} & \text{if } s_a \overset{GSA}{\equiv} s_x \\ \textit{recur}/[s_a]^{x(s_x)}_{\blacktriangleleft:1,E} & \text{if } s_a \overset{GSA}{\not\equiv} s_x,\ [s_a]^{x(s_x)}_{p:(l+1),E} = [s_x]^{x(s_x)}_{\blacktriangleleft:1,E} \\ \textit{recur/unk} & \text{if } s_a \overset{GSA}{\not\equiv} s_x,\ [s_a]^{x(s_x)}_{p:(l+1),E} \neq [s_x]^{x(s_x)}_{\blacktriangleleft:1,E} \end{cases} \tag{3.11}$$

If the index expressions s_a and s_x are GSA-equivalent, i.e. their value coincides during the execution of the loop (see Def. 3.1), then we can conclude that the statement under classification computes an array reduction operation (first entry of Eq. (3.11)). Otherwise, the statement computes an array recurrence operation (second and third entries). In general, GSA equivalence will be proved by checking if s_a and s_x are syntactically identical (see Theroem 3.1).

Second, assume that the identifier of the array reference, a, and that of e_{ref}, x, do not represent the same array variable of the source code:

$$T_{a(s_a)} : [a(s_a)]^{x(s_x)}_{p:l,E} = \begin{cases} unk & \text{if } [s_a]^{x(s_x)}_{p:(l+1),E} = unk \\ assig/[s_x]^{x(s_x)}_{\blacktriangleleft:1,x(s_x)} & \text{if } [s_a]^{x(s_x)}_{p:(l+1),E} = none/\text{-},\ l = 0 \\ inv & \text{if } [s_a]^{x(s_x)}_{p:(l+1),E} = none/\text{-},\ l > 0, \\ & \text{and } [a]^{x(s_x)}_{p:l,E} = [s_a]^{x(s_x)}_{p:(l+1),E} = inv \\ subs & \text{if } [s_a]^{x(s_x)}_{p:(l+1),E} = none/\text{-},\ l > 0, \\ & \text{and otherwise} \\ unk & \text{if } [s_a]^{x(s_x)}_{p:(l+1),E} = reduc/\text{-} \\ unk & \text{if } [s_a]^{x(s_x)}_{p:(l+1),E} = recur/\text{-} \end{cases} \tag{3.12}$$

The symbol "−" represents any trivial class. The last two entries of Eq. (3.12) handle those array references that appear in subscript expressions. As mentioned in Section 3.4.2, these cases are not allowed in the statements that compose the non-trivial structural array SCCs. Thus, the array reference is classified as *unk*.

Finally, the transfer function is rather different if the reference expression, e_{ref}, is a scalar variable y:

$$T_{a(s_a)} : [a(s_a)]^{y}_{p:l,E} = \begin{cases} unk & \text{if } [s_a]^{y}_{p:(l+1),E} = unk \\ inv & \text{if } [a]^{y}_{p:l,E} = [s_a]^{y}_{p:(l+1),E} = inv \\ list & \text{if } [a]^{y}_{p:l,E} = inv \text{ and } [s_a]^{y}_{p:(l+1),E} = list \\ subs & \text{otherwise} \end{cases} \tag{3.13}$$

Note that the second entry of the transfer function enables the recognition of loop-invariant array references, and that the third entry copes with the detection of the linked-list traversal computational kernel.

Arithmetical and Logical Expressions

The class of an expression depends not only on the number of occurrences of e_{ref}, but also on the type of operations that are performed on those occurrences. The analysis of associative operators is specially important, as they enable the recognition of reduction and recurrence computations, both for scalar and array variables. The sum operator, $+$, is the associative operator most commonly used in this kind of computation. In general, the sum is a r-ary operator $+(e_1, \ldots, e_r)$. However, for the sake of simplicity, we define this transfer function, T_+, on the basis of the binary $+$ operator as follows. First, the class of the sum expression, $[+(e_1, \ldots, e_r)]^{x(s_x)}_{p:l,+(e_1,\ldots,e_r)}$, is set to an initial value. Next, for each operand e_i ($i \in \{1, \ldots, r\}$), the class of the operand is computed, $[e_i]^{x(s_x)}_{p:(l+1),+(e_1,\ldots,e_r)}$, and then combined with $[+(e_1, \ldots, e_r)]^{x(s_x)}_{p:l,+(e_1,\ldots,e_r)}$ by means of the transfer function of the binary $+$ operator. The partial and final results of T_+ are stored in the class of the sum expression $[+(e_1, \ldots, e_r)]^{x(s_x)}_{p:l,+(e_1,\ldots,e_r)}$.

The transfer function for the binary sum distinguishes two main situations: e_{ref} is a scalar variable and e_{ref} is an array reference. Table 3.1 represents the behavior in the first case. The first column shows the value of $[+(e_1, \ldots, e_r)]^{x(s_x)}_{p:l,+(e_1,\ldots,e_r)}$. The other columns show the class of the operand $[e_i]^{x(s_x)}_{p:(l+1),+(e_1,\ldots,e_r)}$ (head of the table) and the result of the transfer function (entries one to six). Possible values are some trivial classes of the taxonomy presented in Section 3.4: none (*none*), invariant (*inv*), linear (*lin*), reduction (*red*), subscripted (*subs*) and unknown (*unk*). The value *none* is used to initialize both $[+(e_1, \ldots, e_r)]^{x(s_x)}_{p:l,+(e_1,\ldots,e_r)}$ and $[e_i]^{x(s_x)}_{p:(l+1),+(e_1,\ldots,e_r)}$. The entries of the table that contain two classes show, in parenthese, additional conditions that are checked in the transfer function (l is the level parameter of the context of an expression).

When e_{ref} is an array fetch $x(s_x)$, the transfer function is defined in a quite different manner. For the sake of clarity, let c_1^+ and c_2^+ denote the elements of the tuple that represents the class of the sum operator $[T_+]$. Similar notations are used for the class of the i-th operand $[e_i]^{x(s_x)}_{p:(l+1),+(e_1,\ldots,e_r)}$ ($c_1^{e_i}$ and $c_2^{e_i}$), and for the class, $[s_x]^{x(s_x)}_{\blacktriangleleft:1,x(s_x)}$, of the left-hand index expression, s_x, of the array statement that is being classified ($c_1^{s_x}$ and $c_2^{s_x}$). Note that the latter expression is the index of the reference expression e_{ref}. The most relevant cases of the

T_+	*none*	*inv*	*lin*	*red*	*subs*	*unk*
none	*none*	*inv*	*lin*	*red*	*subs*	*unk*
inv	*inv*	*inv*	*lin*	*red*	*subs*	*unk*
lin	*lin*	*lin*	*lin*	$subs(l=1)$ $red(l \neq 1)$	$red(l=0)$ $subs(l \neq 0)$	*unk*
red	*red*	*red*	$subs(l=1)$ $red(l \neq 1)$	*red*	*red*	*unk*
subs	*subs*	*subs*	$red(l=0)$ $subs(l \neq 0)$	*red*	*subs*	*unk*
unk	*unk*	*unk*	*unk*	*unk*	*unk*	*unk*

Table 3.1: Transfer function T_+ for binary sum arithmetical expressions when the reference expression e_{ref} is a scalar variable.

transfer function are shown in the equation below:

$$T_+ : [+(e_1,\ldots,e_r)]^{x(s_x)}_{\blacktriangleright:l,+(e_1,\ldots,e_r)} = \begin{cases} assig/c_2^{s_x} & \text{if } c_1^+ = none,\ c_1^{e_i} = none,\ l=0 \\ c_1^{e_i}/c_2^{e_i} & \text{if } c_1^+ = none,\ c_1^{e_i} \neq none,\ l=0 \\ assig/c_2^{s_x} & \text{if } c_1^+ = assig,\ c_1^{e_i} = none,\ l=0 \\ c_1^{e_i}/c_2^{e_i} & \text{if } c_1^+ = assig,\ c_1^{e_i} \neq none,\ l=0 \\ recur/c_2^{e_i} & \text{if } c_1^+ = reduc,\ c_1^{e_i} = recur,\ l=0 \\ reduc/c_2^+ & \text{if } c_1^+ = reduc,\ c_1^{e_i} \neq recur,\ l=0 \\ recur/c_2^+ & \text{if } c_1^+ = recur,\ l=0 \end{cases} \tag{3.14}$$

The condition $l = 0$ captures the fact that the right-hand side of the statement consists of a sum expression. The cases presented above enable the recognition of the different types of array operations (assignments, reductions and recurrences). Note that whether an operand consists of an array reference that defines a reduction/recurrence computation is not checked in this transfer function, but in $T_{a(s)}$.

Another important associative operator is the product operator, $*$. The transfer function T_* is very similar to T_+. In fact, the relevant differences arise when e_{ref} is a scalar variable. Its behavior is represented in Table 3.2. Regarding other arithmetical operators, we have defined transfer functions for division, unary minus and unary sum as they were needed to complete the analysis of our benchmark suite. In the scope of this work, we assume that the transfer functions for other operators (e.g. modulo, square root, etc.) classify the target expression as *unknown*.

T_*	*none*	*inv*	*lin*	*red*	*subs*	*unk*
none	*none*	*inv*	*lin*	*unk*	*subs*	*unk*
inv	*inv*	*inv*	*unk*	*unk*	*subs*	*unk*
lin	*lin*	*unk*	*unk*	*unk*	*subs*	*unk*
red	*unk*	*unk*	*unk*	*unk*	*unk*	*unk*
subs	*subs*	*subs*	*subs*	*unk*	*subs*	*unk*
unk	*unk*	*unk*	*unk*	*unk*	*unk*	*unk*

Table 3.2: Transfer function T_* for binary product arithmetical expressions when the reference expression e_{ref} is a scalar variable.

$T_{logical}$	*none*	*inv*	*lin*	*red*	*subs*	*unk*
none	*none*	*inv*	*subs*	*subs*	*subs*	*unk*
inv	*inv*	*inv*	*subs*	*subs*	*subs*	*unk*
lin	*subs*	*subs*	*subs*	*subs*	*subs*	*unk*
red	*subs*	*subs*	*subs*	*subs*	*subs*	*unk*
subs	*subs*	*subs*	*subs*	*subs*	*subs*	*unk*
unk	*unk*	*unk*	*unk*	*unk*	*unk*	*unk*

Table 3.3: Transfer functions $T_=$, $T_{\neq}$, $T_<$, $T_>$, $T_{\leq}$, $T_{\geq}$, T_{NOT}, T_{AND}, T_{OR} for the binary logical expressions.

Finally, we describe the transfer function for the classification of logical expressions briefly. Logical expression may be found in conditional expressions associated with if-endif constructs and in assignment statements that set the value of logical variables. For the analysis of our benchmark suite, we have defined a common transfer function, $T_{logical}$, for the relational operators ($T_=$, $T_{\neq}$, $T_<$, $T_>$, $T_{\leq}$, $T_{\geq}$), and for the logical operators (T_{NOT}, T_{AND}, T_{OR}). A tabular representation is shown in Table 3.3. If both the left-hand and the right-hand subexpressions are *invariant* (i.e. *true* or *false*), the global expression is also set to *invariant*. Otherwise, the logical expression is classified as *subscripted* in order to represent that it can take arbitrary logical values during the execution of the loop.

Special GSA Expressions

The transfer functions for the special GSA expressions mainly check that all the statements of a component belong to the same class. The transfer function T_μ checks that the classes of the variables defined outside (e_{out}) and inside (e_{in})

the loop body are the same; otherwise, the μ expression cannot be successfully classified:

$$T_\mu : [\mu(e_{out}, e_{in})]^x_{\blacktriangleright:0,\mu(e_{out},e_{in})} = \begin{cases} [e_{in}]^x_{\blacktriangleright:1,e_{in}} & \text{if } [e_{out}]^x_{\blacktriangleright:1,e_{out}} = \mathit{none/none} \\ & \text{or } [e_{out}]^x_{\blacktriangleright:1,e_{out}} = \mathit{inv} \\ [e_{in}]^x_{\blacktriangleright:1,e_{in}} & \text{if } [e_{out}]^x_{\blacktriangleright:1,e_{out}} = [e_{in}]^x_{\blacktriangleright:1,e_{in}} \\ \mathit{unk} & \text{otherwise} \end{cases} \tag{3.15}$$

The first entry of Eq. (3.15) defines two exceptions to the behavior described above. The first condition copes with μ-statements associated with inner loops, which are usually part of a wrap-around SCC with several μ-statements. The second condition is associated with the header of the outermost loop, whose e_{out} is loop-invariant. In both cases, the class of e_{in} is inherited by the μ expression.

The transfer function T_γ has a distinguishing characteristic with respect to the other functions: it modifies the first element of the tuple that denotes the class of the expression. The first element, which captures the conditionality of the SCC, is set to *conditional* in any case. The behavior of T_γ is described by the equation presented below.

$$\begin{aligned} T_\gamma \quad : \quad & [\gamma(c, e_{in}, e_{out})]^x_{\blacktriangleright:l,\gamma(c,e_{in},e_{out})} = \\ & = \begin{cases} \mathit{unk} & \text{if } [c]^x_{\blacktriangleright:(l+1),c} = \mathit{unk} \\ [e_{in}]^x_{\blacktriangleright:(l+1),e_{in}} & \text{if } [e_{out}]^x_{\blacktriangleright:(l+1),e_{out}} = \mathit{none/none} \\ [e_{out}]^x_{\blacktriangleright:(l+1),e_{out}} & \text{if } [e_{in}]^x_{\blacktriangleright:(l+1),e_{in}} = \mathit{none/none} \\ [e_{in}]^x_{\blacktriangleright:(l+1),e_{in}} & \text{if } [e_{out}]^x_{\blacktriangleright:(l+1),e_{out}} = [e_{in}]^x_{\blacktriangleright:(l+1),e_{in}} \\ \mathit{unk} & \text{otherwise} \end{cases} \end{aligned} \tag{3.16}$$

The general rule, represented by the fourth entry, is applied to if-else-endif constructs. Entries two and three classify γ-statements associated with if-endif constructs. Due to the demand-driven nature of the algorithm, the class of e_{out} is set to the value *none* because e_{out} is defined in a μ-statement whose classification process is still in progress (see Eq. (3.7) of T_y).

The semantics of T_α is similar to the previous transfer functions. Let $\alpha(x_k, s_x, rhs_x)$ represent the α expression. For the sake of clarity, let c_1^x and c_2^x denote the elements of the tuple that represents $[x_k]^{x(s_x)}_{\blacktriangleright:0,x_k}$. Similar notations are used for $[s_x]^{x(s_x)}_{\blacktriangleleft:1,x(s_x)}$ (c_1^s and c_2^s) and $[rhs_x]^{x(s_x)}_{\blacktriangleright:0,rhs_x}$ (c_1^{rhs} and c_2^{rhs}).

$$T_\alpha : \; [\alpha(x_k, s_x, rhs_x)]^{x(s_x)}_{\blacktriangleright:0,\alpha(x_k,s_x,rhs_x)} = \begin{cases} unk & \text{if } c_1^x = none,\ c_2^x = none \\ & \text{and } c_1^{rhs} = none, \\ & \text{and } c_2^{rhs} = unk \\ c_1^{rhs}/c_2^s & \text{if } c_1^x = none,\ c_2^x = none \\ & \text{and } c_2^{rhs} = none \\ c_1^{rhs}/c_2^{rhs} & \text{if } c_1^x = none,\ c_2^x = none \\ & \text{and } c_2^{rhs} = c_2^s \\ c_1^{rhs}/unk & \text{if } c_1^x = none,\ c_2^x = none \\ & \text{and otherwise} \\ c_1^{rhs}/c_2^{rhs} & \text{if } c_1^x = c_1^{rhs},\ c_2^x = c_2^{rhs} \\ unk & \text{otherwise} \end{cases} \tag{3.17}$$

In this section, we have presented transfer functions that enable the contextual classification of the expressions that appear in the loop bodies of the routines included in our benchmark suite. Furthermore, we have explained how this contextual classification algorithm provides the support that is necessary for the implementation of the demand-driven SCC classification scheme. In order to clarify the structure of our scheme, pseudocodes of the most relevant function of the algorithms for the classification of SCCs, statements and expressions are shown in Figs. 3.9, 3.10 and 3.11, respectively. The implementation details about, for example, the stacks and the context of an expression, have been omitted for the sake of clarity. The procedures $Classify\_use\_def\_chain()$ and $Classify\_Semantic\_SCC()$ is not presented either. In the following section, we will describe the way these algorithms all work together.

3.6 Case Studies

In the previous section, we presented a set of algorithms that enable the representation of the body of a loop nest as a graph of SCCs and use-def chains between SCCs. In this section, we will describe how these algorithms all work together. Consider the consecutively written array kernel depicted in Fig. 3.2(a). The GSA graph consists of five SCCs (see Fig. 3.2(c)). It contains

```
ALGORITHM: Construct_SCC_graph()
INPUT:      Set of SCCs
OUTPUT:     SCC graph

PROCEDURE
   FOREACH SCC(x_{1...n}) DO
      Classify_SCC( SCC(x_{1...n}) )
   END FOREACH
   FOREACH SCC(x_{1...n}) → SCC(y_{1...m}) DO
      Classify_use_def_chain( SCC(x_{1...n}) → SCC(y_{1...m}) )
   END FOREACH
ENDPROCEDURE



ALGORITHM: Classify_SCC()
INPUT:      SCC(x_{1...n}): Strongly connected component
OUTPUT:     [SCC(x_{1...n})]: Class of the SCC

PROCEDURE
   IF SCC(x_{1...n}) has already been visited THEN
      RETURN
   END IF

   IF SCC(x_{1...n}) is trivial THEN
      [SCC(x_1)] = Classify_GSA_Statement( x_1 = rhs )
   ELSE
      [SCC(x_{1...n})] = Classify_GSA_Statement( x_1 = μ(x_0, x_n) )
      IF SCC(x_{1...n}) is semantic THEN
         [SCC(x_{1...n})] = Classify_Semantic_SCC( SCC(x_{1...n}) )
      END IF
   END IF
ENDPROCEDURE
```

Figure 3.9: Pseudocode of the SCC classification algorithm.

```
ALGORITHM: Classify_GSA_Statement()
INPUT:        stm: GSA statement
OUTPUT:     [stm]: Class of the GSA statement

PROCEDURE
   IF stm has already been visited THEN
      return
   END IF

   SWITCH Type of GSA statement stm
   CASE do v = e_init, e_limit, e_step:
      [e_init] = Classify_Expression(e_init)
      [e_limit] = Classify_Expression(e_limit)
      [e_step] = Classify_Expression(e_step)
      [stm] = T_DO-stm([e_init], [e_limit], [e_step])
      BREAK
   CASE x = μ(x_out, x_in):
      [μ] = Classify_Expression(μ(x_out, x_in))
      [stm] = T_μ-stm([μ])
      BREAK
   CASE x = γ(c, x_in, x_out):
      [γ] = Classify_Expression(γ(c, x_in, x_out))
      [stm] = T_γ-stm([γ])
      BREAK
   CASE a = α(a_prev, s, rhs):
      [α] = Classify_Expression(α(a_prev, s, rhs))
      [stm] = T_α-stm([α])
      BREAK
   CASE v = rhs:
      [rhs] = Classify_Expression(rhs)
      [stm] = T_scalar-stm([rhs])
      BREAK
   DEFAULT :
      [stm] = unknown
      BREAK
   ENDSWITCH
ENDPROCEDURE
```

Figure 3.10: Pseudocode of the algorithm for the classification of the statements of GSA form.

```
ALGORITHM: Classify_Expression()
INPUT:      e: GSA expression
OUTPUT:    [e]: Class of the GSA expression

PROCEDURE
   SWITCH Type of GSA expressions e
   CASE x:
      [e] = T_y(x)
      BREAK
   CASE a(s):
      [a] = Classify_Expression(a)
      [s] = Classify_Expression(s)
      [e] = T_a(s)([a], [s])
      BREAK
   CASE μ(x_out, x_in):
      [x_out] = Classify_Expression(x_out)
      [x_in] = Classify_Expression(x_in)
      [e] = T_μ([x_out], [x_in])
      BREAK
   CASE α(a, s, rhs):
      [a] = Classify_Expression(a)
      [s] = Classify_Expression(s)
      [rhs] = Classify_Expression(rhs)
      [e] = T_α([a], [s], [rhs])
      BREAK
   CASE γ(c, x_in, x_out):
      [x_in] = Classify_Expression(x_in)
      [x_out] = Classify_Expression(x_out)
      [c] = Classify_Expression(c)
      [e] = T_γ([x_in], [x_out], [c])
      BREAK
   CASE +, *, /, =, ≠, <, ≤, >, ≥, not, and, or:
      [e] = none
      FOREACH operand e_i
         [e_i] = Classify_Expression(e_i)
         [e] = T_operator([e], [e_i])
      END FOREACH
      BREAK
   CASE unary+, unary−:
      [e_1] = Classify_Expression(e_1)
      [e] = T_operator([e_1])
      BREAK
   CASE K:
      [K] = T_K(K)
      BREAK
   DEFAULT : /* Includes procedure/function calls */
      [e] = unknown
      BREAK
   ENDSWITCH
ENDPROCEDURE
```

Figure 3.11: Pseudocode of the contextual algorithm for the classification of expressions.

two non-trivial SCCs, $SCC_1^S(i_2, i_3, i_4)$ and $SCC_1^A(a_1, a_2, a_3)$, that capture the conditional induction variable i and the conditional array assignment operation a, respectively. There are two trivial SCCs, $SCC_1^S(h_1)$ and $SCC_1^S(tmp_2)$, which represent the loop index variable h_1 (it is not depicted in the figure), and the temporary variable tmp_2. As tmp_2 is computed inside the if-endif construct, an additional virtual component, $SCC_0^S(tmp_1, tmp_3)$, appears in the GSA graph. The SCC classification algorithm is demand-driven. Thus, the set of SCCs can be classified in any order because the algorithm will analyze the components as needed. Without loss of generality, let us assume that the SCCs are processed in the following order: $SCC_1^S(i_2, i_3, i_4)$, $SCC_0^S(tmp_1, tmp_3)$, $SCC_1^A(a_1, a_2, a_3)$, $SCC_1^S(h_1)$ and $SCC_1^S(tmp_2)$.

The class $[SCC_1^S(i_2, i_3, i_4)]$ is determined first. A tree representation of the classification process is depicted in Fig. 3.12. The picture consists of two trees. The tree on the left illustrates the decomposition of $[SCC_1^S(i_2, i_3, i_4)]$ into the classification of the statements and the expressions that form 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 are a reminder of this correspondence. The class of each expression, statement or SCC is the result of applying the appropriate transfer function to the classes associated with the child nodes. The dotted edges remark this bottom-up process.

The execution of the SCC classification algorithm corresponds to the depth-first traversal of the tree on the left-hand side. As stated in Section 3.5.1, $[SCC_1^S(i_2, i_3, i_4)]$ is reduced to determining $[i_2 = \mu(i_1, i_4)]$, where $i_2 = \mu(i_1, i_4)$ is associated with the outermost loop of the loop nest, i.e. do_{h_1}. The class of a statement inherits the contextual class of its right-hand side expression, $[\mu(i_1, i_4)]^i_{\blacktriangleright:0,\mu(i_1,i_4)}$. The class of an expression is computed by classifying the arguments first. The first argument of the μ expression is the occurrence i_1 corresponding to the initialization of the induction variable before the execution of the loop do_h. According to Eq. (3.6), $[i_1]^i_{\blacktriangleright:0,i_1}$ is *inv*. The second argument is an occurrence i_4 of the value of i at the end of the previous loop iteration. The definition statement belongs to $SCC_1^S(i_2, i_3, i_4)$, so the demand-driven classification algorithm launches the computation of $[i_4 = \gamma(c(h_1), i_3, i_2)]$ (see

Eq. (3.7)). The classification process continues in a similar manner. When the childs of a node have been classified, the transfer function of the corresponding operator is applied. Thus, $[c(h_1)]^{i}_{?:0,c(h_1)}$ is set to *subs* after the application of $T_{a(s)}$ with the parameters *inv* and *lin* (see Eq. (3.13)). At the end of the depth-first traversal $[SCC_1^S(i_2, i_3, i_4)]$ is set to *cond/lin*.

The computation of $[h_1]^{i}_{?:0,c(h_1)}$ deserves special mention, because it launches the classification of a different component, $SCC_1^S(h_1)$ (see Eq. (3.9)). As stated in the demand-driven algorithm described in Section 3.5.1, the classification process of $SCC_1^S(i_2, i_3, i_4)$ is deferred at this moment. As a consequence, $SCC_1^S(i_2, i_3, i_4)$ is pushed onto the stack of SCCs and the classification of $SCC_1^S(h_1)$ is started. The component $SCC_1^S(h_1)$ is independent because the statement DO $h_1 = 1, n, 1$ does not contain occurrences of the variables associated with other components. Thus, the demand-driven classification process is not launched again. The details of the computation of $[SCC_1^S(h_1)]$ are depicted inside boxes in the trees of Fig. 3.12. Once $[SCC_1^S(h_1)]$ has been determined to be *lin*, $SCC_1^S(i_2, i_3, i_4)$ is popped from the stack and its classification process continues. At the end of the execution of the SCC classification algorithm, the class *cond/lin* that represents the *conditional linear induction variable* kernel has been derived.

When the classification of $SCC_1^S(i_2, i_3, i_4)$ finishes, two components of the loop have been successfully derived: $SCC_1^S(i_2, i_3, i_4)$ and $SCC_1^S(h_1)$. The SCC classification algorithm continues with the computation of $[SCC_0^S(tmp_1, tmp_3)]$. The trees of Fig. 3.13 illustrate this process. Note that the demand-driven mechanism launches $[SCC_1^S(h_1)]$ and $[SCC_1^S(tmp_2)]$. In the first case, the class $[SCC_1^S(h_1)]$ is not computed because it has already been determined. In the second case, $[SCC_1^S(tmp_2)]$ is actually computed. Finally, only the class $[SCC_1^A(a_1, a_2, a_3)]$ has to be calculated for the set of components of do_h to be processed. The corresponding trees are depicted in Fig. 3.14.

The information gathered from the source code during the execution of the SCC classification algorithm is represented in the SCC graph of the loop nest. The SCC graph of our case study, the consecutively written array kernel of Fig. 3.2(a), is depicted in Fig. 3.15. The different types of SCCs are represented by nodes with different shapes: rectangle (non-trivial structural SCC), oval (trivial SCC), and shaded oval (trivial SCC that represent a loop index variable). The label of the node is the SCC, which provides informa-

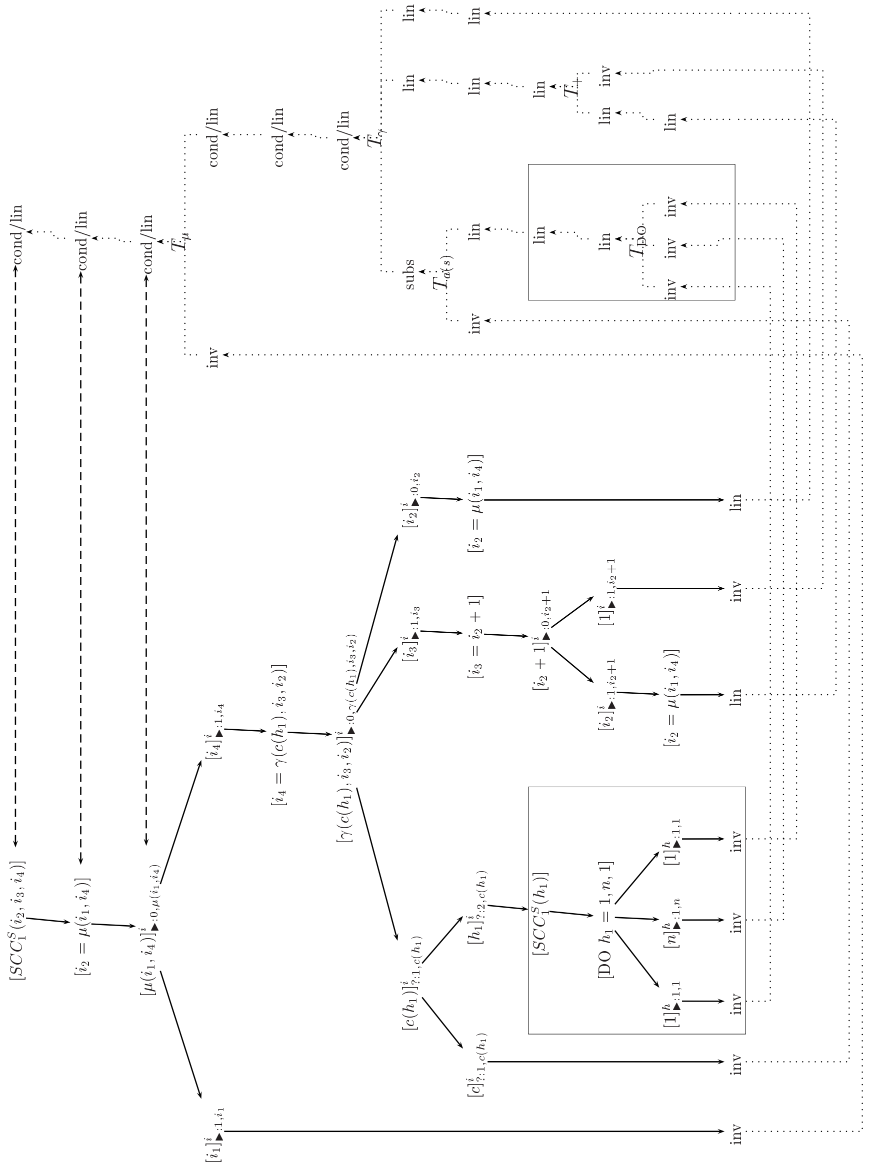


Figure 3.12: Classification of the components $SCC_1^S(i_2,i_3,i_4)$ and $SCC_1^S(h_1)$ of the SCC graph shown in Fig. 3.15.

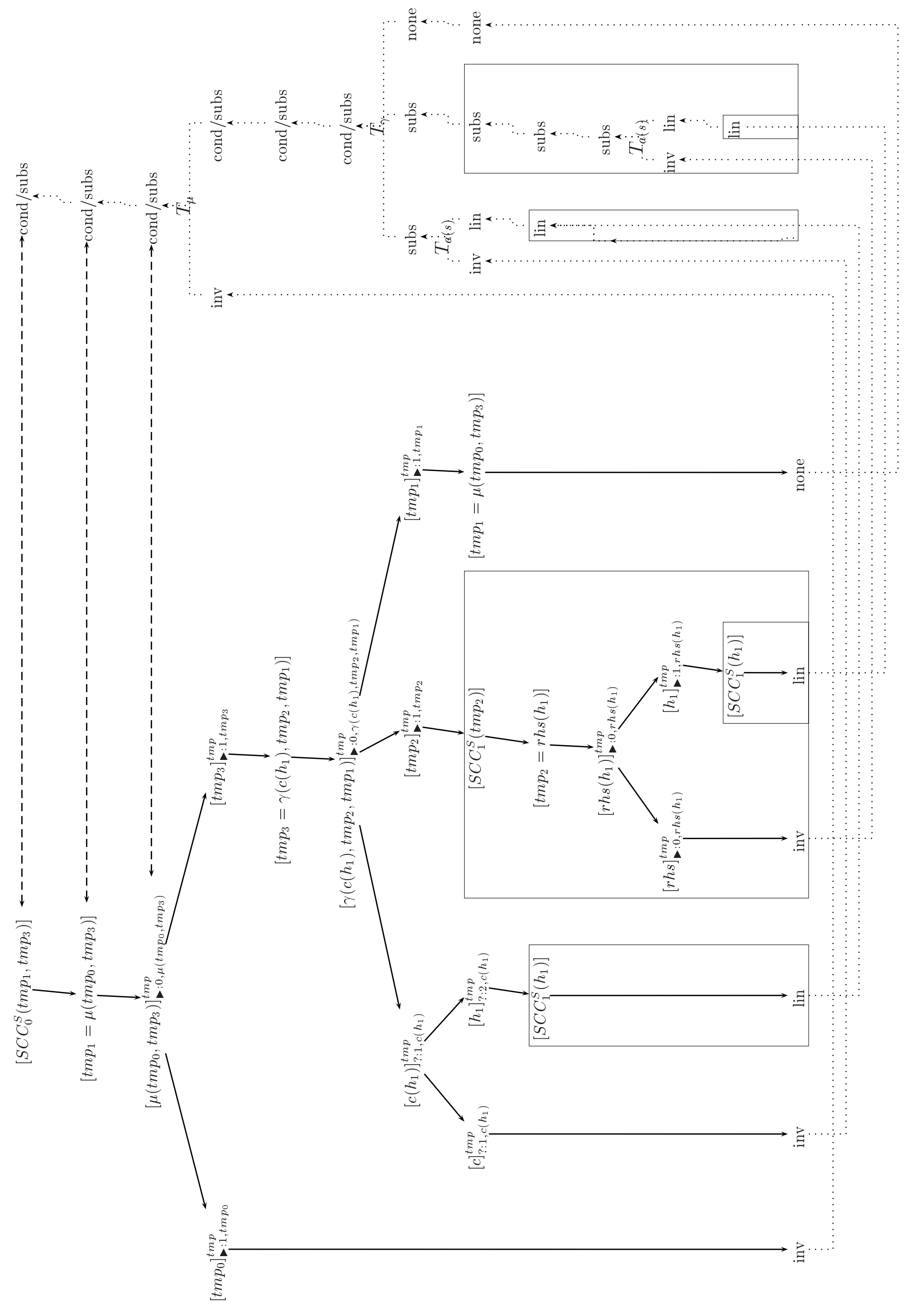


Figure 3.13: Classification of the components $SCC_0^S(tmp_1, tmp_3)$ and $SCC_1^S(tmp_2)$ of the SCC graph shown in Fig. 3.15.

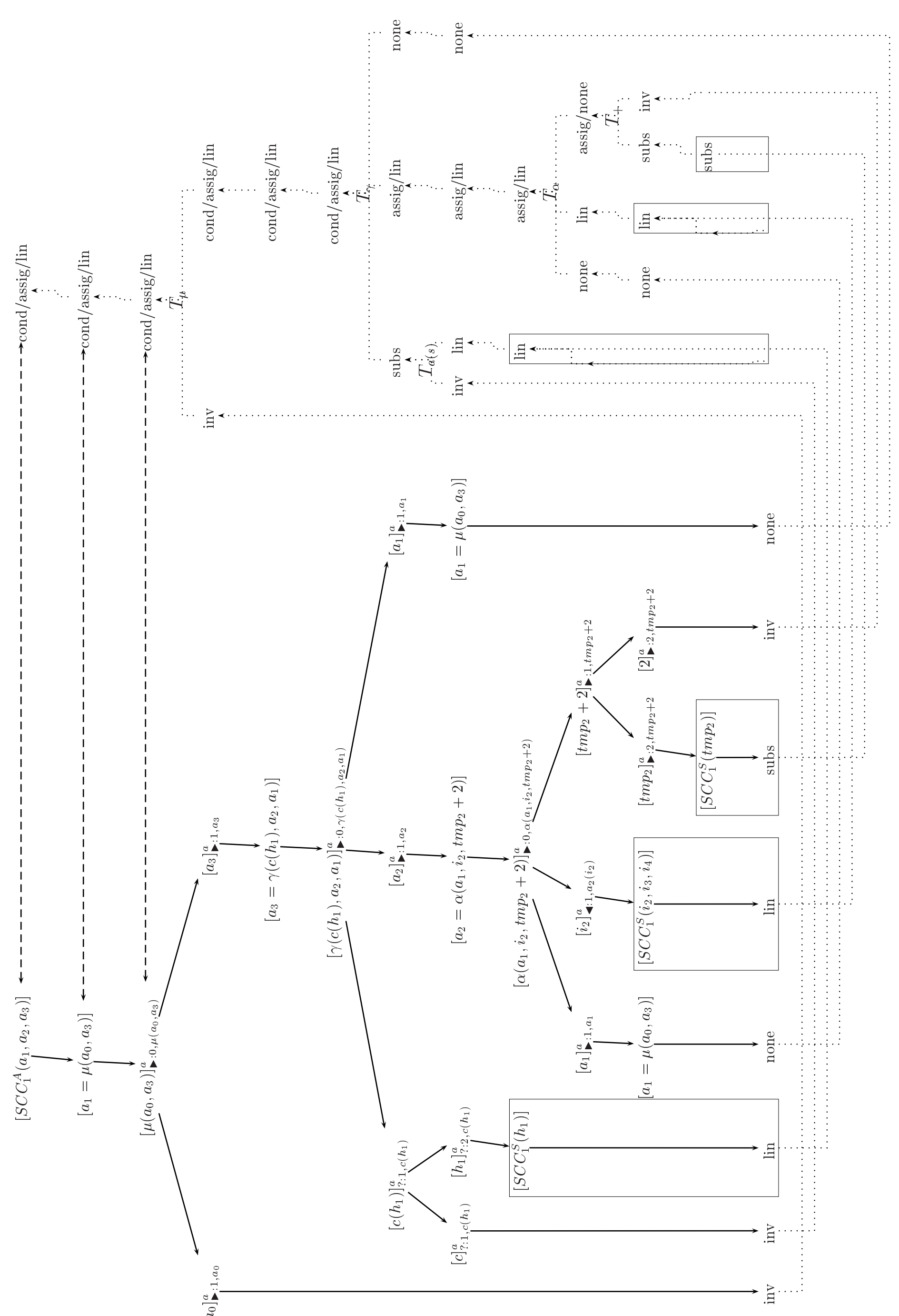


Figure 3.14: Classification of the component $SCC_1^A(a_1, a_2, a_3)$ of the SCC graph shown in Fig. 3.15.

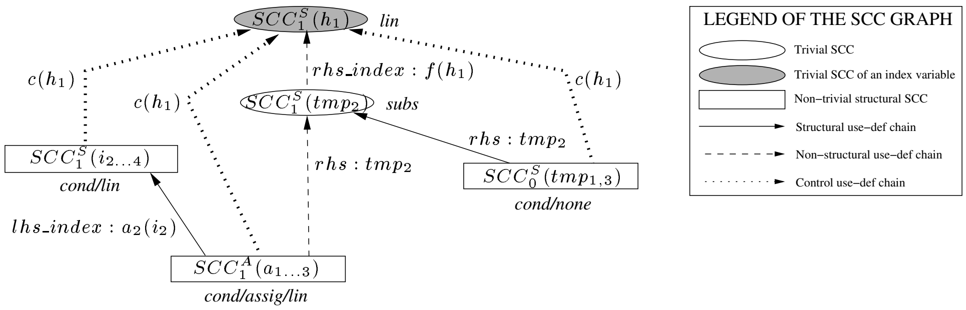


Figure 3.15: SCC graph of the consecutively written array kernel shown in Fig. 3.2(a).

tion about several properties of the SCCs: scalar/array, cardinality and set of statements in the GSA form. Furthermore, the class of each SCC is printed next to the corresponding node. The edges of the graph represent the use-def chains between the SCCs. Solid, dashed and dotted lines are used for structural, non-structural and control use-def chains, respectively. The labels of the edges are determined during the execution of the SCC classification algorithm according to Eq. (3.10).

Let us analyze the construction of the SCC graph of our case study. As shown in the block diagram of Fig. 3.1, the first step of the construction is the execution of the SCC classification algorithm. After this stage, most of the SCC graph has been constructed because the information about the classes of the SCCs and the use-def chains between pairs of SCCs is available. For the graph to be completed, the class of the use-def chains must be determined. As explained below, this task is a straightforward application of Def. 3.10. The loop contains three SCCs that include a γ-statement: $SCC_1^S(i_2, i_3, i_4)$, $SCC_0^S(tmp_1, tmp_3)$ and $SCC_1^A(a_1, a_2, a_3)$. In all the cases, the conditional expression is $c(h_1)$. Thus, the SCC graph contains three control use-def chains whose target is the strongly connected component $SCC_1^S(h_1)$: $SCC_1^S(i_2, i_3, i_4) \rightsquigarrow SCC_1^S(h_1)$, $SCC_0^S(tmp_1, tmp_3) \rightsquigarrow SCC_1^S(h_1)$ and the chain $SCC_1^A(a_1, a_2, a_3) \rightsquigarrow SCC_1^S(h_1)$. On the other hand, there are two structural use-def chains. The chain $SCC_0^S(tmp_1, tmp_3) \Rightarrow SCC_1^S(tmp_2)$ is structural because it captures a dependence between two scalar SCCs that are associated with the same variable in the source code, namely, *tmp*. Furthermore, the array and the scalar components $SCC_1^A(a_1, a_2, a_3)$ and $SCC_1^S(i_2, i_3, i_4)$ 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 $SCC_1^A(a_1, a_2, a_3) \Rightarrow SCC_1^S(i_2, i_3, i_4)$ is classified as structural. Finally, the SCC graph contains two non-structural chains $SCC_1^S(tmp_2) \not\Rightarrow SCC_1^S(h_1)$ and $SCC_1^A(a_1, a_2, a_3) \not\Rightarrow SCC_1^S(tmp_2)$. In Chapter 4, we will describe how the information included in the SCC graph can be used for the recognition of the consecutively written array kernel computed in the loop of our case study.

3.7 Experimental Results

In this section we will analyze the efficacy of the SCC classification algorithm presented in this chapter. We have implemented a prototype of the algorithm using the infrastructure provided by the Polaris parallelizing compiler [11]. The characteristics of the prototype and the benchmark suite are described in Section 3.7.1. Experimental results in terms of number of occurrences of each SCC class are presented in Section 3.7.2. The relevance of each SCC class for the analysis of *SparsKit-II* is also discussed. Finally, the cases where the SCC classification algorithm fails are analyzed in Section 3.7.3.

3.7.1 Experimental Conditions

We have developed a C++ prototype that constructs our compiler framework and takes a Fortran77 source code as input. For the first stage of the framework (Fig. 2.4), a translator of Fortran77 code into GSA form provided by Polaris was used. The subsequent stages are devoted to the recognition of computational kernels. The implementation of these phases requires the analysis of the code from the viewpoint of, for example, the dependences between statements and the control flow of the program. For this purpose, we have used the support provided by the internal representation of Polaris [15]. The prototype consists of approximately $30,000$ lines of C++ code (the routines of Polaris were not accounted).

During the design and implementation of the prototype we have found several bugs in the translator of source code into GSA form provided by Polaris. These bugs result in incorrect GSA representations, for example, in some loop bodies that contain GOTO statements. As our framework is constructed on top of GSA, the prototype preprocesses the GSA form of the routines to detect those *loops that are not amenable for classification*. We also use this term to refer to loops that contain jump statements (e.g. goto, break) and procedure/function calls.

Our benchmark suite is the *SparsKit-II* library [41], 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 with different types of sparse storage formats (matrix-vector products and triangular system solvers).
- *blassm*, which supplies a basic linear algebra for sparse matrices. It contains routines that compute different types of matrix-matrix products and sums.
- *unary*, that provides unary operations with sparse matrices (e.g. extracting a submatrix from a sparse matrix, filter out elements of a matrix according to their magnitude, or performing a mask operation with matrices).
- *formats*, which is devoted to format conversion routines for different types of sparse storages.

We have chosen this library because it includes parallelizable computational kernels that are important for full-scale sparse/irregular applications. The importance of a subset of these kernels was already pointed-out in [32]. A description of the kernels considered in this thesis was presented in Section 2.2. The programs in *SparsKit-II* are small, which enabled the completion of a previous hand analysis in a reasonable time.

The main characteristics of the modules in terms of number of routines, number of loop nests and number of loops actually analyzed are presented in Table 3.4. In the latter case, we distinguish between loops that are amenable for classification and those that are not. The numbers are presented for the modules *matvec*, *blassm*, *unary* and *formats*, as well as for *SparsKit-II* as a whole. The percentage of loops not amenable for classification in *unary* (33%) and *formats* (16%) is significant. However, the number of loop nests actually affected is low. The reason for this is that, in most cases, the innermost loops contain GOTO statements that also make surrounding loops not amenable for classification. It should be noted that the number of loop nests is less than the number of loops actually analyzed. This is a consequence of the strategy that we use for the analysis of the loop nests. In particular, the outermost loop is analyzed first. Then, if the computational kernels could not be recognized, the analysis of the inner loops is accomplished. We found this strategy effective for the analysis of *SparsKit-II*. However, it is quite aggressive when full-scale applications are considered because outer loops are usually devoted to control tasks and, hence, contain calls to procedures.

3.7.2 SCC Recognition Results

Detailed statistics about the number of occurrences of each SCC class in the modules of *SparsKit-II* are presented in Table 3.5. In the first column, the SCC taxonomy of Fig. 3.4 is depicted as a tree that reveals the subclasses of each SCC class. The next block of four columns show the numbers for the modules *matvec*, *blassm*, *unary* and *formats*. Blank entries represent zero occurrences of the corresponding class. The last two columns summarize the total number ($\#SCCs$) and the percentage ($\%SCCs$) for each SCC class in *SparsKit-II* as a whole. The percentages are calculated with respect to the total number of SCCs, i.e., the sum of trivial, non-trivial and unknown SCCs. The rows are organized in four sets. The first three sets contain the statistics corresponding to trivial SCCs, non-trivial structural SCCs and non-trivial semantic SCCs, respectively. The first row of each set presents the total numbers for the set. The last set summarizes the total number of trivial, non-trivial and *unknown* SCCs in the four modules and in *SparsKit-II*.

The class of trivial SCCs is the most frequent class of component in *SparsKit-II*. However, it has little relevance for kernel recognition because it mainly corresponds to computations associated with temporary scalar variables. In fact, 56%, 42%, 50% and 52% represent the computation of the loop index variables of *matvec*, *blassm*, *unary* and *formats*, respectively. Furthermore, the cardinality-zero trivial SCCs (31%, 28%, 28% and 32%, respectively) correspond to virtual components (see Def. 3.13) that capture the flow of values at run-time, but do not provide useful information from the recognition viewpoint. The details about this will be presented in Chapter 4.

Characteristics of *SparsKit-II*	*matvec*	*blassm*	*unary*	*formats*	*SparsKit-II*
Routines	17	11	42	35	105
Loop nests	28	27	88	113	256
Individual loops actually analyzed	40	41	126	172	379
Amenable for classification	39	38	85	144	306
Not amenable for classification	1	3	41	28	73

Table 3.4: Summary of characteristics of the source code of the *SparsKit-II* library.

The trivial class *lin* is associated with loop index variables whose iteration space does not change during the execution of the loop nest. Typical examples are the outermost loop and any loop whose *init*, *limit* and *step* expressions are loop-invariant. For illustrative purposes, consider the source code of the lower triangular system solver shown in Fig. 3.16(a). The index variable of do_k is represented by a *lin* trivial SCC class. On the other hand, the class *subs* represents loop indices whose iteration space is defined on the basis of loop-variant expressions. A well-known example is the inner loop of a loop nest that traverses a sparse matrix stored in CRS format. The index variable of do_j in Fig. 3.16(a) fits into this category.

Regarding non-trivial SCC classes, the cardinality-one structural SCCs are the most numerous classes in *SparsKit-II*. In particular, *non-cond/assig/lin* (8%) and *cond/assig/lin* (3%) stand out from the other SCC classes. Most of the non-conditional assignments correspond to simple loops devoted to the initialization of an array variable where the left-hand side subscript expression consists of an occurrence of the index variable of the surrounding loop. The class of the index expression, *lin*, indicates that the array entries are written according to a linear access pattern. On the contrary, the conditional assignments are mainly related to the computation of conditional consecutively written array kernels. This observation is reflected in the statistics of Table 3.5, where the number of *cond/lin* occurrences (i.e. conditional linear induction variables) is almost as high as the number of *cond/assig/lin* occurrences. Finally, note that the prototype has been able to recognize

	matvec	*blassm*	*unary*	*formats*	#SCCs	%SCCs
Trivial SCCs	105	131	232	444	912	57
Cardinality=0	32	37	64	131	264	17
none	32	37	64	131	264	17
Cardinality=1	73	94	168	313	648	40
inv	1	4	4	11	20	1
lin	40	38	95	176	349	22
red				1	1	0
subs	32	52	69	125	278	17
Non-trivial Structural SCCs	34	89	136	232	491	30
Cardinality=0	6	16	32	64	118	7
cond/none			16	21	37	2
non-cond/none	6	14	16	43	79	5
Cardinality=1	28	73	104	168	373	23
cond/lin		15	12	13	40	3
cond/red				2	2	0
non-cond/lin	1	1	7	27	36	2
non-cond/red	6		5	4	15	1
cond/assig/inv		2		2	4	0
cond/assig/lin		18	14	19	51	3
cond/assig/subs		6	3	2	11	1
cond/reduc/inv				1	1	0
cond/reduc/subs		1		3	4	0
cond/recur/lin			2	1	3	0
non-cond/assig/lin	12	18	39	59	128	8
non-cond/assig/red				1	1	0
non-cond/assig/subs		11	8	5	24	2
non-cond/reduc/lin	1	1	1	1	4	0
non-cond/reduc/subs	8		6	9	23	1
non-cond/recur/lin			7	19	26	2
Non-trivial Semantic SCCs	0	4	7	3	14	1
Cardinality=0	0	4	1	2	7	0
Scalar-minimum			1		1	0
Scalar-maximum				2	2	0
Scalar-find-and-set		4			4	0
Cardinality=1	0	0	6	1	7	0
Array-find-and-set			6	1	7	0
Trivial SCCs	105	131	232	444	912	57
Non-trivial SCCs	34	93	143	235	505	31
Unknown SCCs	13	16	26	140	195	12

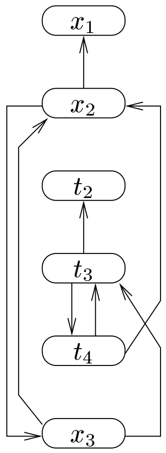
Table 3.5: Basic kernels recognized by the SCC classification algorithm in the *SparsKit-II* library.

```
x(1) = y(1)
DO k = 2,n
   t = y(k)
   DO j = ial(k),ial(k+1) - 1
      t = t - al(j) * x(jal(j))
   END DO
   x(k) = t
END DO
```

(a) Source code.

```
x1(1) = y(1)
DO k1 = 2,n,1
   x2 = μ(x1,x3)
   t1 = μ(t0,t3)
   j1 = μ(j0,j2)
   t2 = y(k1)
   DO j2 = ial(k1),ial(k1+1) - 1,1
      t3 = μ(t2,t4)
      t4 = t3 - al(j2) * x2(jal(j2))
   END DO
   x3 = α(x2,k1,t3)
END DO
```

(b) GSA form.



(c) GSA graph.

Figure 3.16: Unit lower triangular system solver for a CRS matrix using standard forward elimination (extracted from *SparsKit-II*, module *matvec*, routine *lsol*). The code contains a SCC of cardinality two.

SCCs that capture the properties of a set of relevant computational kernels such as *(non-)cond/red* for scalar reductions, *(non-)cond/assig/subs* for irregular assignments, *(non-)cond/reduc/subs* for irregular reductions, and *(non-)cond/recur/lin* for array recurrence operations.

Semantic SCCs represent a low percentage of the total number of components. Nevertheless, we consider that the detection of this SCC class is a significant advance for two reasons. On the one hand, the corresponding source code kernels appear quite often in scientific applications that work with very large matrices. The computational cost associated with these kernels usually depends on matrix size. On the other hand, in general, the computations performed in loop bodies can be represented as a combination of semantic and structural SCCs. Thus, the detection of semantic SCCs can enable the parallelization of a wider class of loops. The 14 semantic components that appear in *SparsKit-II* are divided as follows: 7 cardinality-zero SCCs (1, 2 and 4 of classes *scalar-minimum*, *scalar-maximum* and *scalar-find-and-set*, respectively) and 7 cardinality-one SCCs (in particular, *array-find-and-set*).

3.7.3 Failures in SCC Recognition

The efficacy of our prototype for the analysis of the *SparsKit-II* library is high. In fact, the results presented in Table 3.5 show that 88% of the SCCs that appear in the loop that are amenable for classification were successfully recognized. However, it is interesting to study the impact of the current limitations of our prototype for the SCC classification algorithm. Statistics about this issue are presented in Table 3.6. The first column shows the different reasons for a SCC to be unsuccessfully classified. The number of *unknown* SCCs in *SparsKit-II* is 201, which represents 13% of the total number of SCCs of the library (the percentage was calculated with respect to the sum of the total number of trivial, non-trivial and *unknown* SCCs presented in Table 3.5). It should be noted that up to 147 of the SCCs of *formats* (18%) were not successfully classified. This is mainly due to the fact that some routines of the module contain deep loop nests whose innermost loops include *unknown* SCCs, which also leads to the unsuccessful classification of the enclosing loops.

The main limitation of the prototype is related to the classification of SCCs with cardinality greater than one (4% of the SCCs of *SparsKit-II*). These components appear in loops that contain mutually dependent variables. For example, 10 SCCs of cardinality 2 arise in the routines of *matvec* that contain equation system solvers. The code shown in Fig. 3.16(a) solves a lower triangular system. The system matrix is stored in CRS format. Note that in the GSA graph (Fig. 3.16(c)), there is a strongly connected component $SCC(x_2, x_3, t_3, t_4)$ that represents the flow of values for the mutually dependent variables t and x during the execution of do_k.

Another typical computation that is represented by a SCC of cardinality greater than one is the swap operation of two variables. Consider the code of Fig. 3.17, which is a fragment of a bubble-sort algorithm that orders the entries of a sparse matrix according to column indices. In each do_j iteration, two pairs of array entries are swapped if they fulfill a certain condition. In our compiler framework, these computations are represented by two SCCs of cardinality 3. The first one is composed of the source code statements $ko = jao(k)$, $jao(k) = jao(j)$ and $jao(j) = ko$. It is a semantic SCC because the conditional expression $(jao(k) > jao(j))$ contains occurrences of array jao. The second one is a structural SCC that consists of $tmp = ao(k)$, $ao(k) = ao(j)$

Limitation of the prototype	Number of SCCs				
	matvec	*blassm*	*unary*	*formats*	*SparsKit-II*
Cardinality> 1	10	4	16	32	62
Multidimensional array references	3	0	0	51	54
Statements of different classes	0	11	2	16	29
Other limitations	0	1	7	48	56
Total	13	16	25	147	201

Table 3.6: Relevance of current limitations of our prototype of the SCC classification algorithm for the analysis of *SparsKit-II*.

and $ao(j) = tmp$.

The presence of references to multidimensional arrays in the loop body is the next limitation that causes the prototype to fail (3%). However, the components of *SparsKit-II* that present these characteristics do not capture any computational kernel that deserves special mention, but the common kernels studied in this thesis.

The least relevant limitation is the classification of SCCs with statements of different classes (2%). However, these SCCs represent interesting computations. Consider the loop of Fig. 2.2(d), which computes an *array-find-and-set* kernel using the array variable *diag* (see Section 2.2.4). The computational kernel is represented by a semantic SCC that contains two array statements $diag(h) = 1/diag(h)$ and $diag(h) = 1$. The prototype classifies the statements as *cond/reduc/linear* and *cond/assig/lin*, respectively. As a result, the transfer function for the γ-statement (see Eq. (3.16)) makes the component being classified as *unknown*.

The product and sum operations with sparse matrices included in *blassm* also contain some loops whose computations are represented by SCCs with statements of different classes (do_{100} in routine *amub*; do_{300} in *aplb*, *apmbt* and *aplsbt*; do_4 in *aplsca*). Consider the loop do_{kb} of Fig. 3.18. There is a non-trivial structural SCC composed of two statements $c(len) = scal * b(kb)$ and $c(jpos) = c(jpos) + scal * b(kb)$ that are located in different branches of an if-endif construct. The first statement, which belongs to the class *cond/assig/lin*, sets the initial value of a new entry in the array c of the product matrix. In subsequent iterations, the second statement, which belongs to class *cond/reduc/subs*, accumulates new values on the new entry indicated by an element of the temporary array iw.

We finish this section with a brief description of some of the other limitations of the SCC classification algorithm:

- *SCCs of cardinality zero or one that is not defined in our taxonomy.* The taxonomy of SCC classes presented in Section 3.4 can be extended to capture new computational kernels. In Fig. 3.19 the innermost loop do_{jj} of a level-4 loop nest is shown. It contains a linear induction variable $a0$ that is conditionally incremented during the execution of do_{jj}. As the conditions $(a0 \geq ia(kvstr(i)+ii+1))$ and $(jj = ja(a0)))$ depend on the variable itself, the computations are represented by a scalar semantic SCC that has not been defined in our taxonomy. In *SparsKit-II* there are other semantic components that are not recognized by the current version of the prototype (e.g. $do_{i(5)}$ in routine *csrvbr* of the module *formats*).

- *Presence of occurrences of variables that are associated with unknown SCCs.* Consider the loop do_{jj} in Fig. 3.19. There is a SCC that represents the computation of array b. One of the reasons for this component to be unsuccessfully classified is the presence of the occurrence $a0$ in the statement $b(b0) = a(a0)$. As explained above, the component for a0 is an *unknown* semantic SCC of cardinality one.

- *Arithmetical operators not considered in the transfer functions.* In Section 3.5.3 we defined transfer functions for a set operators. Expressions that contain other operators (e.g. module, power) will be classified as *unknown* and will lead to the unsuccessfull classification of the SCC they are included in.

```
DO j = indu(i) - 1, ipos + 1, -1
   k = j - 1
   IF (jao(k) > jao(j)) THEN
      ko = jao(k)
      jao(k) = jao(j)
      jao(j) = ko
      IF (value2 ≠ 0) THEN
         tmp = ao(k)
         ao(k) = ao(j)
         ao(j) = tmp
      END IF
   END IF
END DO
```

Figure 3.17: Loop that contains structural and semantic SCCs with cardinality two. It was extracted from a routine that converts the storage format of sparse matrix from symmetric sparse row into CRS (*SparsKit-II*, module *formats*, routine *ssrcsr*).

```
DO kb = ib(jj), ib(jj + 1) - 1
   jcol = jb(kb)
   jpos = iw(jcol)
   IF (jpos = 0) THEN
      len = len + 1
      jc(len) = jcol
      iw(jcol) = len
      IF (values) THEN
         c(len) = scal * b(kb)
      END IF
   ELSE
      IF (values) THEN
         c(jpos) = c(jpos) + scal * b(kb)
      END IF
   END IF
END DO
```

Figure 3.18: Loop that contains SCCs with statements of different classes. It was extracted from a routine that performs the sparse matrix by sparse matrix product (*SparsKit-II*, module *blassm*, routine *amub*).

```
DO jj = kvstc(jb(j)), kvstc(jb(j) + 1) - 1
   IF (a0 ≥ ia(kvstr(i) + ii + 1)) THEN
      b(b0) = 0.d0
   ELSE
      IF (jj = ja(a0)) THEN
         b(b0) = a(a0)
         a0 = a0 + 1
      ELSE
         b(b0) = 0.d0
      END IF
   END IF
   b0 = b0 + neqr
END DO
```

Figure 3.19: Code that contains non-recognized semantic SCCs. Extracted from a routine that converts the CRS storage format into variable block row format (*SparsKit-II*, module *formats*, routine *csrvbr*).

Chapter 4

Recognition of Loop-Level Computational Kernels

The goal of the compiler framework presented in this thesis is to recognize the types of computational kernels that are calculated during the execution of a loop. In the previous chapter, we presented a classification scheme that is able to recognize the basic kernels computed by the statements of a SCC. Consider a set of statements $x_1, \ldots, x_n$ in GSA form that represent the operations corresponding to the computation of a kernel. The kernel will be detected by the SCC classification algorithm if $x_1, \ldots, x_n$ belong to a unique SCC. However, in general, $x_1, \ldots, x_n$ may be associated with a set of SCCs and a set of use-def chains among these SCCs. In order to actually recognize the kernel, it is necessary for the compiler to analyze the SCC graph constructed in the first stage of our compiler framework. This stage, *Recognition of Basic Computational Kernels* in Fig. 2.4, was described in Chapter 3.

The recognition of loop-level kernels involves two main tasks. On the one hand, the analysis of the SCC graph in order to identify the set of kernels computed in a loop. And, on the other hand, the actual recognition of the different kernels. From now on, these tasks will be referred to as *kernel separation* and *kernel classification*, respectively. This chapter is organized as follows. In Section 4.1, we introduce definitions and notations. In Section 4.2, we explain several issues of kernel separation from the point of view of the analysis of the SCC graph. In Section 4.3, we present a SCC graph classification algorithm that accomplishes the recognition of loop-level kernels. This algorithm addresses both kernel separation and kernel classification. In Section 4.4, a set of example codes extracted from our benchmark suite is analyzed in detail. The explanations will show that, in order to classify loop-level kernels, the compiler needs to analyze other data structures apart from the SCC graph, for example, the *control flow graph (CFG)* of the program. The chapter finishes with the presentation of experimental results in Section 4.5.

4.1 Basic Definitions and Notations

The SCC graph classification algorithm addresses the recognition of the set of kernels computed in a loop. We will use the term **loop class** to refer to this set of kernels. The individual kernels of the loop class, which were described in Section 2.2, will be denoted as follows. In Chapter 3, we presented a SCC classification algorithm that is able to recognize a subset of kernels: linear induction variable, scalar reduction, linked-list traversal, array assignments, array reductions, array recurrences, scalar minimum (maximum), array minimum (maximum), scalar find-and-set and array find-and-set. In these cases, we will use the abbreviations depicted in the SCC taxonomy of Fig. 3.4. There is a subset of more complex kernels that cannot be detected with the SCC classification algorithm. These kernels will be denoted as follows:

- Induction variable whose value is reinitialized to a loop-invariant value as *(non-)cond/lin-r/inv*. Similar notations will be used for scalar reduction (*(non-)cond/red-r/inv*) and linked-list traversal (*(non-)cond/list-r/inv*).
- Induction variable reinitialized to a loop-variant value in an outer loop as *(non-)cond/lin-r/subs*. Scalar reductions and linked-list traversals will be referred to as *(non-)cond/red-r/subs* and *(non-)cond/list-r/subs*, respectively.
- Scalar minimum (maximum) with location as *scalar-minimum-w/loc* (the corresponding scalar kernel as *scalar-maximum-w/loc*). For operations with arrays we will use *array-minimum-w/loc* and *array-maximum-w/loc*.
- Consecutively written array as *(non-)cond/cwa*.

We define two additional classes that do not correspond to any kernel described in Section 2.2: *scalar-location* and *array-location*. These classes will be used in

Section 4.4.2 for the detection of scalar and array minimum (maximum) with location kernels.

The following definition introduces a concept that will be used for accomplishing the kernel separation task during the execution of the SCC graph classification algorithm.

Definition 4.1. *Let $SCC(x_{1...n})$ be a non-wrap-around component of a SCC use-def chain graph. The* ***non-wrap-around source node (NWSN) subgraph*** *of $SCC(x_{1...n})$ is the subgraph composed of the set of nodes and edges that are reachable from $SCC(x_{1...n})$.*

The SCC graph captures information regarding the classes and the dependences between the basic kernels computed in a loop. However, that information is not enough, for example, for the recognition of reinitialized scalar reductions or consecutively written arrays. As will be shown in Section 4.4, the analysis of data structures such as the *control flow graph (CFG)* is also needed. Next, some definitions that will be helpful for describing some case studies are introduced.

Definition 4.2. *A $SCC(x_{1...n})$ is* ***multi-loop*** *if it contains more than one scalar statement or α-statement that belong to the body of several loops located at the same nesting level.*

Definition 4.3. *A $SCC(x_{1...n})$* ***is contained*** *in loop do_h, $SCC(x_{1...n}) \sqsubseteq do_h$, if all the statements $x_1, \ldots, x_n$ belong to the body of do_h.*

Definition 4.4. *We say that a $SCC(x_{1...n})$* ***is exclusively contained*** *in loop do_h, $SCC(x_{1...n}) \sqsubset do_h$, if all the statements $x_1, \ldots, x_n$ belong to the body of do_h, and do not belong to the body of any inner loop.*

In this chapter, the classes of use-def chains that were defined in Def. 3.10 will be referred to within the text. The notations $SCC(x_{1...n}) \Rightarrow SCC(y_{1...m})$, $SCC(x_{1...n}) \nRightarrow SCC(y_{1...m})$ and $SCC(x_{1...n}) \rightsquigarrow SCC(y_{1...m})$ will be used for representing structural, non-structural and control use-def chains, respectively.

4.2 Kernel Separation Issues of the SCC Graph Classification Algorithm

Programmers have learnt to optimize sequential code for obtaining the highest performance from uniprocessor architectures. These optimizations were shown to be effective for improving the efficiency of the sequential execution of the program. However, as stated in Section 2.1, the use of such programming practices results in source codes that require sophisticated techniques for the automatic analysis of programs (e.g. for automatic detection of parallelism).

A common optimization consists of computing a set of independent computational kernels in a unique loop nest. This approach enables, for example, the reuse of temporary results that are needed for various kernels. The synthetic code shown in Fig. 4.1 will be used to illustrate the key ideas behind kernel separation. The legend of the SCC graph (Fig. 4.1(c)) distinguishes several types of nodes: rectangle (non-trivial structural SCC), hexagon (non-trivial semantic SCC), oval (trivial SCC), and shaded oval (trivial SCC that represent a loop index variable). Although there is no hexagonal node in this SCC graph, this type of node will appear in other graphs presented in subsequent sections. The edges depicted as solid, dashed and dotted lines represent structural, non-structural and control use-def chains, respectively. The labels of the edges are determined by means of Eq. (3.10) during the execution of the SCC classification algorithm.

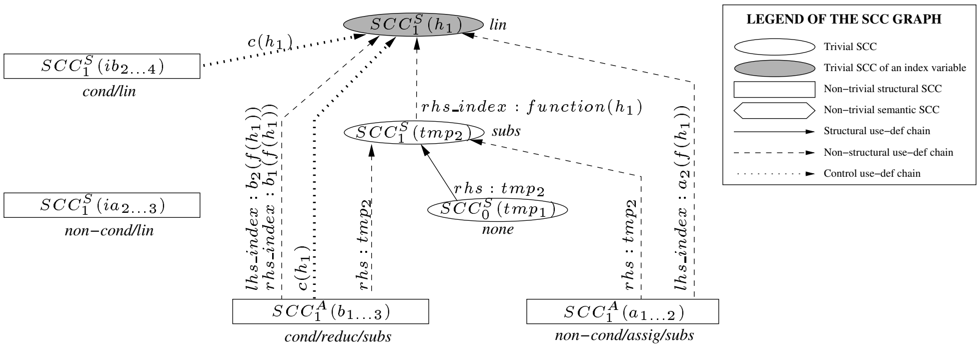
The loop do_h of Fig. 4.1(a) contains four computational kernels: a non-conditional irregular assignment (variable a), a conditional irregular reduction (variable b) and two linear induction variables that are used as counters (one non-conditional, ia, and one conditional, ib). The SCC graph that represents do_h consists of two connected subgraphs (see Fig. 4.1(c)). The first one contains a unique component $SCC_1^S(ia_{2...3})$ of class *non-cond/lin*. Thus, the kernel associated with ia is identified through the analysis of this simple subgraph. The second subgraph needs further analysis for the separation of the remaining kernels. The reason that the three computational kernels belong to a unique connected subgraph is that all of them use a common set of temporary variables, namely, the loop index of the outermost loop do_h and the temporary scalar tmp. The key observation that supports the separation process is that the SCC graph contains a source node for each of the non-wrap-around

```
ia = 0
ib = 0
DO h = 1, n
    tmp = function(h)
    a(f(h)) = tmp
    ia = ia + 1
    IF (c(h)) THEN
        b(f(h)) = b(f(h)) + 2 * tmp
        ib = ib + 1
    END IF
END DO
```

(a) Source code.

```
ia_1 = 0
ib_1 = 0
DO h_1 = 1, n, 1
    b_1 = μ(b_0, b_3)
    a_1 = μ(a_0, a_2)
    ia_2 = μ(ia_1, ia_3)
    ib_2 = μ(ib_1, ib_4)
    tmp_1 = μ(tmp_0, tmp_2)
    tmp_2 = function(h_1)
    a_2 = α(a_1, f(h_1), tmp_2)
    ia_3 = ia_2 + 1
    IF (c(h_1)) THEN
        b_2 = α(b_1, f(h_1), b_1(f(h_1)) + 2 * tmp_2)
        ib_3 = ib_2 + 1
    END IF
    b_3 = γ(c(h_1), b_2, b_1)
    ib_4 = γ(c(h_1), ib_3, ib_2)
END DO
```

(b) GSA form.



(c) SCC graph.

Figure 4.1: Synthetic code to illustrate kernel separation.

components $SCC_1^S(ib_{2...4})$, $SCC_1^A(b_{1...3})$ and $SCC_1^A(a_{1...2})$ that represent the remaining three computational kernels. Note that the SCC graph includes a fourth source node that consists of the wrap-around component $SCC_0^S(tmp_1)$. The statements of the wrap-around SCCs are inserted in the code during the construction of the GSA form to represent the flow of values. However, from the detection of parallelism viewpoint, they do not provide the compiler with any useful information. As a result, wrap-around SCCs can be ignored safely during the execution of the SCC graph classification algorithm.

There are some exceptions to the general separation scheme described above. After the decomposition of the SCC graph, some special connected subgraphs that do not provide useful information for kernel separation may appear. Consider the synthetic loop shown in Fig. 4.2. The index variable of loop do_h is not referenced within the loop body, i.e. it is just used as a counter. As a result, the loop header is represented by a connected subgraph that is composed of a unique component $SCC_1^S(h_1)$ (see Fig. 4.2(c)). A similar situation arises for the index variables of the inner loops of a loop nest. The corresponding loop headers are represented by a subgraph of two components: one associated with the loop header, and another composed of the μ-statements inserted in GSA form at the beginning of the loop body of the surrounding loops. The two connected subgraphs described above share a common characteristic: they consist of SCCs associated with source code temporary variables that are not referenced in any statement within the loop body. These exceptions, as well as the general separation procedure described above, have been considered for the design of the SCC graph classification algorithm presented in the following section.

4.3 SCC Graph Classification Algorithm

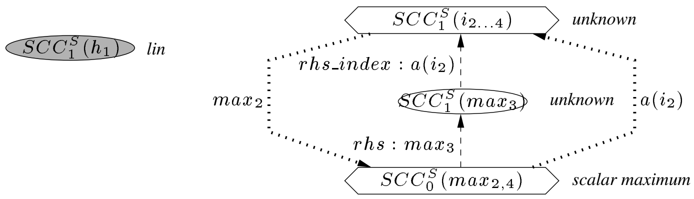
The SCC graph classification algorithm, which is the main topic of our work [7], accomplishes loop-level kernel recognition by performing kernel separation and kernel classification in a unique traversal of the SCC graph. The algorithm proceeds as follows. First, the SCC graph is divided into a set of connected subgraphs. For each connected subgraph, a demand-driven classification algorithm starts from each non-wrap-around source node. This procedure separates and classifies the set of kernels that are computed in the statements

```
max = a(1)
i = 2
DO h = 1,n
   IF (max < a(i)) THEN
      max = a(i)
      i = i + 1
   END IF
END DO
```

(a) Source code.

```
max_1 = a(1)
i_1 = 1
DO h_1 = 1,n,1
   i_2 = μ(i_1, i_4)
   max_2 = μ(max_1, max_4)
   IF (max_2 < a(i_2)) THEN
      max_3 = a(i_2)
      i_3 = i_2 + 1
   END IF
   i_4 = γ(max_2 < a(i_2), i_3, i_2)
   max_4 = γ(max_2 < a(i_2), max_3, max_2)
END DO
```

(b) GSA form.



(c) SCC graph.

Figure 4.2: Example code whose SCC graph contains cycles (loop already presented in Fig. 3.3)

of the SCCs included in the NWSN subgraph (see Def. 4.1). We call this set of kernels *NWSN subgraph class*. Finally, the *loop class* is determined by applying a *transfer function* to all the NWSN subgraph classes. This transfer function basically consists of the union of the sets of kernels recognized during the analysis of all the NWSN subgraphs. However, as will be shown in Section 4.4.1, there are some exceptions where some kernels of the *NWSN subgraph class* are not transfered into the *loop class*.

The core of the SCC graph classification algorithm is the procedure for the classification of NWSN subgraphs. It basically consists of a post-order traversal of the NWSN subgraph. When a node $SCC(x_{1...n})$ 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. For each use-def chain, the successor is classified and, after that, a transfer function that determines the type of kernel represented by the target and source SCCs of the use-def chain is applied. We call this kernel *use-def chain class*. The classification of the NWSN subgraph fails if either the class of the target SCC or the class of the source SCC have not been successfully computed by the SCC classification scheme presented in Chapter 3. In that case, the NWSN subgraph class and the loop class are 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, then the transfer function is applied directly. At the end of the NWSN classification algorithm, the class of the SCC associated with the NWSN is added to the NWSN subgraph class.

As shown in Section 3.3, the SCC graph may contain cycles due to the fact that control use-def chains are not taken into account for the construction of the graph. The example code of Fig. 4.2 contains two mutually dependent components (see Def. 3.9). Thus there is a cycle composed of $SCC(i_{2...4}) \rightsquigarrow SCC(max_{2,4})$ and $SCC(max_{2,4}) \rightsquigarrow SCC(i_{2...4})$. These situations can be detected by using a stack of SCCs as follows. When a node is visited, the corresponding SCC is inserted into the stack. Next, the contents of the stack are checked before analyzing the target SCC of a use-def chain. If the target SCC is in the stack, then there is a cycle in the SCC graph. This detection mechanism was also used in the SCC classification algorithm described in the previous chapter.

A pseudocode of the SCC graph classification algorithm is depicted in Fig. 4.3. For the sake of clarity, we have omitted some implementation details, for instance, the management of the stack and the checking of whether a SCC has been successfully classified. We have presented two procedures: *Classify_SCC_Graph()* and *Classify_Node()*. The latter carries out the post-order traversal of the SCC graph by analyzing the target nodes of the use-def chains before the application of the corresponding transfer function. The procedures whose names are *TF_NWSN_Subgraphs()*, *TF_Structural_UseDef_Chains()*, *TF_Control_UseDef_Chains()* and *TF_Non_Structural_UseDef_Chains()* represent the transfer functions of the NWSN subgraphs and structural, control and non-structural use-def chains. Note that due to the demand-driven nature of *Classify_Node()*, the NWSN subgraph classification algorithm is implicitly implemented by running *Classify_Node()* with the non-wrap around source node as argument. The details about the division of the SCC graph into connected subgraphs are not shown in the pseudocode.

```
ALGORITHM: Classify_SCC_Graph()
INPUT:     scc_graph: SCC use-def chain graph
OUTPUT:    [loop]: Loop class

PROCEDURE
   FOREACH NWSN_subgraph  IN scc_graph
      [NWSN_subgraph] = Classify_Node(source_node of NWSN_subgraph)
      [loop] = TF_NWSN_Subgraphs([loop],[NWSN_subgraph])
   END FOREACH
   IF [loop] is unknown THEN
      FOREACH inner_loop
         Classify_SCC_Graph(inner_loop)
      END FOREACH
   END IF
ENDPROCEDURE
```

(a) Procedure *Classify_SCC_Graph()*.

```
ALGORITHM: Classify_Node()
INPUT:     node: Node of the SCC graph
OUTPUT:    [NWSN_subgraph]: Class of the NWSN subgraph

PROCEDURE
   FOREACH chain  IN set of structural use-def chains
      [chain] = Classify_Node(target_node of chain)
      [NWSN_subgraph] = TF_Structural_UseDef_Chains([NWSN_subgraph],[chain])
   END FOREACH
   FOREACH chain  IN set of control use-def chains
      [chain] = Classify_Node(target_node of chain)
      [NWSN_subgraph] = TF_Control_UseDef_Chains([NWSN_subgraph],[chain])
   END FOREACH
   FOREACH chain  IN set of non-structural use-def chains
      [chain] = Classify_Node(target_node of chain)
      [NWSN_subgraph] = TF_Non_Structural_UseDef_Chains([NWSN_subgraph],[chain])
   END FOREACH
ENDPROCEDURE
```

(b) Procedure *Classify_Node()*.

Figure 4.3: Pseudocode of the SCC graph classification algorithm.

The transfer functions of the use-def chains provide the compiler with a mechanism to detect the typical scenarios where the different kernels considered in this work are computed. Furthermore, during their execution, the compiler may carry out any test that enables the recognition of a particular kernel. In the rest of this section, we just outline the goals of the transfer functions for the structural, control and non-structural use-def chains because, in general, each kernel requires the checking of different conditions. In Section 4.4, we will describe the tests that are needed for several case studies extracted from our benchmark suite.

Transfer Function of Structural Use-Def Chains

The transfer functions of structural use-def chains is presented in Table 4.1. The first two columns define the characteristics that must fulfill the target SCC and the source SCC. The last column shows the use-def chain class for each possible value of the label associated with it: *lhs_index*, *rhs* and *rhs_index* (see Eq. (3.10)). The rows are organized in sets. The first row of each set shows the notations of the target SCC and the source SCC, which supplies the following information about components: trivial/non-trivial (Def. 3.2), cardinality (Def. 3.4) and scalar/array (Def. 3.5). The remaining rows show the classes of the target SCC, the source SCC and the use-def chain. The notation of the SCC class provides information about structural/semantic (Def. 3.3) and conditionality (Def. 3.6). The symbol "=" represents that the use-def chain class and the NWSN subgraph class are not changed during the execution of the transfer function. Finally, blank entries represent use-def chains that correspond to non-recognized computational kernels (the use-def chain class is set to *unknown*). This transfer function has been defined to enable the recognition of the kernels computed in our benchmark suite. However, it is expected to work well with other real codes.

Structural use-def chains (see Def. 3.10) were defined so that they show which dependences are essential for the recognition of loop-level kernels. As a result, this transfer function enable the detection of a wide range of kernels. Next, we will enumerate some relevant kernels. For example, consider the consecutively written array shown in Fig. 3.2(a), which was used as a guide for the description of the SCC classification algorithm in Chapter 3. The SCC graph

Target SCC	Source SCC	Structural use-def chain class		
		lhs_index	rhs	rhs_index
$[SCC_1^S(y_1)]$	$[\ SCC_0^S(x_{1\ldots n})\ ]$			
lin	*cond/none*		*scalar-location*	
subs	*cond/none*		*scalar-location*	
subs	*scalar-minimum*		*scalar-minimum*	
subs	*scalar-maximum*		*scalar-maximum*	
$[SCC_1^S(y_1)]$	$[\ SCC_1^S(x_{1\ldots n})\ ]$			
inv	*(non-)cond/lin*		*(non-)cond/lin-r/inv*	
inv	*(non-)cond/red*		*(non-)cond/red-r/inv*	
subs	*(non-)cond/lin*		*(non-)cond/lin-r/subs*	
subs	*(non-)cond/red*		*(non-)cond/red-r/subs*	
$[SCC_1^S(y_1)]$	$[\ SCC_1^A(x_{1\ldots n})\ ]$			
lin	*(non-)cond/assig/lin*	*(non-)cond/assig/lin*		
lin	*(non-)cond/reduc/lin*	*(non-)cond/reduc/lin*		=
lin	*(non-)cond/recur/lin*	*(non-)cond/recur/lin*		=
lin	*array-minimum*		*array-minimum*	
lin	*array-maximum*		*array-maximum*	
subs	*(non-)cond/assig/subs*	*(non-)cond/assig/subs*		
subs	*(non-)cond/reduc/subs*	*(non-)cond/reduc/subs*		=
$[SCC_1^S(y_{1\ldots m})]$	$[\ SCC_1^A(x_{1\ldots n})\ ]$			
cond/lin	*cond/assig/lin*	*cond/cwa*		
non-cond/lin	*non-cond/assig/lin*	*non-cond/cwa*		
cond/lin-r/subs	*cond/assig/lin*	*cond/cwa*		
non-cond/lin-r/subs	*non-cond/assig/lin*	*non-cond/cwa*		

Table 4.1: Transfer function for the analysis of structural use-def chains.

of the loop contains a structural use-def chain $SCC_1^A(a_{1...3}) \Rightarrow SCC_1^S(i_{2...4})$ (see Fig. 3.15). According to the entry 16 of the transfer function of Table 4.1, the *cond/cwa* kernel can be recognized. The details regarding additional tests will be explained in Section 4.4.3.

Structural use-def chains also enable the detection of array operations that involve either regular (e.g. *(non-)cond/assig/lin*, *(non-)cond/reduc/lin*) or irregular (e.g. *(non-)cond/assig/subs*, *(non-)cond/reduc/subs*) subscript expressions. A *(non-)cond/assig/subs* kernel will be studied in Section 4.4.1.

As stated in Section 3.4.2, semantic reduction operations with scalars cannot be detected in the SCC classification algorithm because of lack of information. The candidate kernels pointed out in that stage (e.g. *scalar-minimum*, *scalar-maximum*, *scalar-find-and-set*) are confirmed or discarded in this transfer function. A fragment of code from our benchmark suite that computes a scalar minimum with location will be analyzed in Section 4.4.2.

Finally, note that induction variables reinitialized to an invariant or subscripted value (e.g. *(non-)cond/lin-r/inv*, *(non-)cond/lin-r/subs*) can also be recognized. The same holds for scalar reductions (e.g. *(non-)cond/red-r/inv*, *(non-)cond/red-r/subs*).

Transfer Function of Control Use-Def Chains

Control use-def chains are usually associated with conditional expressions that involve temporary variables computed in the loop nest. The *cond/cwa* kernel shown in Figs. 3.2(a) and 3.15 contains an if-endif construct whose condition, $c(h_1)$, consists of a reference to a loop-invariant array c. The SCC graph shows a control use-def chain from each component $SCC_1^S(i_{2...4})$, $SCC_1^A(a_{1...3})$ and $SCC_0^S(tmp_{1,3})$. As the target component $SCC_1^S(h_1)$ represents a temporary variable (in particular, a loop index), the use-def chains are not relevant from the kernel recognition viewpoint. The transfer function of control use-def chains is presented in Table 4.2. Note that the situations described above correspond to "=" entries in the table (entries 3, 5, and 1, respectively).

This transfer function will also be used for other purposes. First, it enables the recognition of complex kernels that consist of one basic semantic kernel and one basic structural kernel whose execution depends on the same condition

Target SCC	Source SCC	Control use-def chain class
$[SCC_1^S(y_1)]$	$[\ SCC_0^S(x_{1 \ldots n})\ ]$	
lin	*cond/none*	=
subs	*cond/none*	=
subs	*scalar-minimum*	=
$[SCC_1^S(y_1)]$	$[\ SCC_1^S(x_{1 \ldots n})\ ]$	
lin	*(non-)cond/lin*	=
subs	*(non-)cond/subs*	=
$[SCC_1^S(y_1)]$	$[\ SCC_1^A(x_{1 \ldots n})\ ]$	
lin	*(non-)cond/assig/lin*	=
lin	*(non-)cond/assig/subs*	=
$[SCC_0^S(y_{1 \ldots m})]$	$[\ SCC_0^S(x_{1 \ldots n})\ ]$	
scalar-minimum	*scalar-location*	*scalar-minimum-w/loc*
scalar-maximum	*scalar-location*	*scalar-maximum-w/loc*
$[SCC_1^A(y_{1 \ldots m})]$	$[\ SCC_1^A(x_{1 \ldots n})\ ]$	
array-minimum	*cond/assig/lin*	*array-minimum-w/loc*
array-maximum	*cond/assig/lin*	*array-maximum-w/loc*
array-find-and-set	*cond/lin*	*array-find-and-set*
non-cond/assig/lin	*cond/lin*	=
non-cond/assig/lin	*cond/assig/lin*	=
non-cond/assig/lin	*(non-)cond/cwa*	=

Table 4.2: Transfer function for the analysis of control use-def chains.

within the loop body. A typical example is the computation of the kernel scalar (array) minimum with location, which is studied in detail in Section 4.4.2. On the other hand, it can be used to detect temporary scalar and array variables that control the execution of other computational kernels in the loop nest. The recognition of these kernels is addressed in the analysis of the consecutively written array kernel presented in Section 4.4.4.

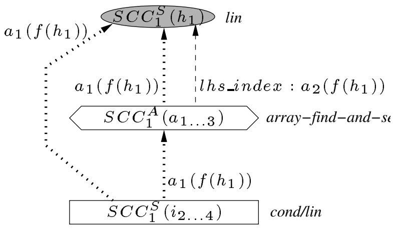
Finally, control use-def chains also provide a mechanism for kernel separation. Consider the code shown in Fig. 4.4 that sets to zero the non-zero elements of an array (variable a), and that counts the number of updated elements (variable i). The transfer function of Table 4.2 (entry number 12) adds the class *array-find-and-set* of the semantic $SCC_1^A(a_{1 \ldots 3})$ to the class of the NWSN subgraph. At the end of the demand-driven process, the class *cond/lin* of $SCC_1^S(i_{2 \ldots 4})$ is also added to the NWSN subgraph class. This way both kernels are separated and recognized.

```
i = 0
DO h = 1,n
   IF (a(f(h)) ≠ 0) THEN
      a(f(h)) = 0
      i = i + 1
   END IF
END DO
```

(a) Source code.

```
i1 = 0
DO h1 = 1,n,1
   i2 = μ(i1,i4)
   a1 = μ(a0,a3)
   IF (a1(f(h1)) ≠ 0) THEN
      a2 = α(a1,f(h1),0)
      i3 = i2 + 1
   END IF
   i4 = γ(a1(f(h1)) ≠ 0,i3,i2)
   a3 = γ(a1(f(h1)) ≠ 0,a2,a1)
END DO
```

(b) GSA form.



(c) SCC graph.

Figure 4.4: Kernel separation through control use-def chains.

Transfer Function of Non-Structural Use-Def Chains

Unlike structural and control chains, non-structural use-def chains do not enable the recognition of loop-level computational kernels. In general, they will be used by the compiler as a mechanism to gather information that is relevant for subsequent compiler optimizing and parallelizing transformations. It should be noted that, however, non-structural use-def chains may lead the classification scheme to be unsuccessful in some situations. In Section 4.5.1 some examples of codes with these characteristics will be presented.

An example application may be the recognition of the access pattern of a set of arrays during the execution of a loop. Consider the non-structural use-def chain $SCC_1^S(tmp_2) \nRightarrow SCC_1^S(h_1)$ in the SCC graph of Fig. 3.15. The label of the chain, $rhs\_index : f(h_1)$, indicates that the index variable of the loop, do_{h_1}, is referenced in the subscript expression of an array, f, in the right-hand side expression of the statement $tmp_2 = f(h_1)$. As the subscript expression, h_1, consists of an occurrence of index variable, the compiler knows that the entries of f are accessed as dictated by the loop index. This information may be useful, for example, in the scope of automatic parallelization of sequential codes. If do_{h_1} is parallelized, then the entries of array f may be scattered among the processors in the same manner as the iterations of do_{h_1} in order to minimize the overhead of the parallel code. In Chapter 5, we will describe the parallelization of a code where non-structural chains provide essential information for the generation of parallel code.

4.4 Case Studies

In this section we present a detailed analysis of a set of loop nests extracted from our benchmark suite. These case studies are intended to provide the reader with a general overview of the loop classification algorithm. In Section 4.4.1, a simple loop pattern called *irregular assignment* [5, 6] is studied. In Section 4.4.2, several codes for the computation of minimum reduction operations, both with scalar and array variables, are analyzed. Finally, Sections 4.4.3 and 4.4.4 are devoted to the recognition of several forms of *consecutively written arrays* [32].

4.4.1 Irregular Assignment

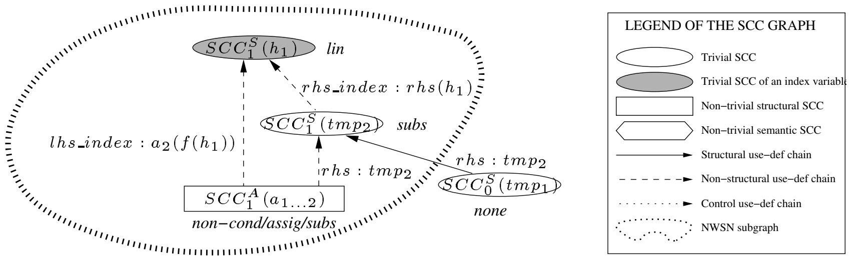
An *irregular assignment* consists of a loop where, at each iteration h, a write operation of the array element $a(f(h))$ is performed, f being the subscript array. The expression whose value is assigned to $a(f(h))$ does not contain occurrences of a, thus the code is free of loop-carried true data dependences. 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). An example abstraction of this computational kernel, as well as the corresponding GSA form and SCC graph, is shown in Fig. 4.5. In Fig. 4.5(a), array entries $a(f(h))$ are assigned the value of the expression $tmp + K$, where tmp is a temporary variable and K is a constant expression. The value of tmp in each loop iteration is represented as $b(h)$ in the figure. Irregular assignment computations are a type of array operation (see Section 2.2.5) that can be found in codes from different application fields such as computer graphics [19], finite elements [48], or operations for the manipulation of sparse matrices [41].

```
a(...) = ...
DO h = 1,n
   tmp = b(h)
   a(f(h)) = tmp + K
END DO
... = ...a(...)...
```

(a) Source code.

```
DO h1 = 1,n,1
   a1 = μ(a0,a2)
   tmp1 = μ(tmp0,tmp2)
   tmp2 = b(h1)
   a2(f(h1)) = α(a1,tmp2 + K)
END DO
```

(b) GSA form.



(c) SCC graph.

Figure 4.5: Irregular assignment computations.

Several code transformations that enable the parallel execution of irregular assignments can be found in the literature [6, 27]. The detection could be carried out through a specific technique that combines source code pattern-matching and data dependence analysis. This approach was successfully applied to irregular reductions in the Polaris parallelizing compiler [35]. In this section, however, we address the detection problem using the classification schemes presented in the thesis. These schemes provide support for the efficient recognition of a wide range of kernels in a unified manner.

The irregular assignment kernel shown in the loop do_h of Fig. 4.5(a) is recognized as follows. First, the SCC classification algorithm performs an exhaustive analysis of the loop in GSA form, do_{h_1}, which is presented in Fig. 4.5(b). As a result, do_{h_1} is represented as the SCC graph shown in Fig. 4.5(c). Next, the loop classification algorithm splits this graph into a set of NWSN subgraphs. One NWSN subgraph that starts from $SCC_1^A(a_{1...2})$ is found in do_{h_1} (see the dotted curve of Fig. 4.5(c)). It consists of three nodes, $SCC_1^S(h_1)$, $SCC_1^S(tmp_2)$ and $SCC_1^A(a_{1...2})$, which are related through non-structural use-def chains only. The loop classification scheme performs a post-order traversal of the NWSN subgraph. The NWSN $SCC_1^A(a_{1...2})$ is visited first. As it is the source of two non-structural use-def chains, $SCC_1^A(a_{1...2}) \not\Rightarrow SCC_1^S(h_1)$ and $SCC_1^A(a_{1...2}) \not\Rightarrow SCC_1^S(tmp_2)$, the target components are visited before applying the corresponding transfer function. When $SCC_1^S(h_1)$ is visited, the demand-driven process finds a sink node that was successfully classified. As explained in Section 4.3, the transfer function of non-structural use-def chains does not enable the recognition of kernel classes. Thus, the analysis of the behavior of that transfer function will not be addressed in this section. As a consequence, the demand-driven process can continue with the analysis of $SCC_1^A(a_{1...2}) \not\Rightarrow SCC_1^S(tmp_2)$. The process continues after classifying the chains $SCC_1^S(tmp_2) \not\Rightarrow SCC_1^S(h_1)$ and $SCC_1^A(a_{1...2}) \not\Rightarrow SCC_1^S(tmp_2)$ in that order. Finally, the class of the NWSN $SCC_1^A(a_{1...2})$ is added to the NWSN subgraph class. Thus, the subgraph is classified as *non-cond/assig/subs*.

It should be noted that the loop classification algorithm is not concerned with the analysis of the output dependences that arise on array a during the execution of the loop, as these dependences are represented by the class of the $SCC_1^A(a_{1...2})$. The remaining true dependence is captured as a non-structural use-def chain $SCC_1^A(a_{1...2}) \not\Rightarrow SCC_1^S(tmp_2)$. This non-structural chain, which is associated with the temporary variable tmp, does not provide the compiler with relevant information for the recognition of the irregular assignment computed in do_{h_1}.

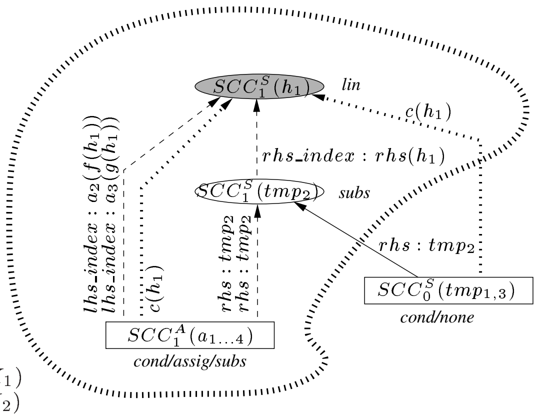
In real codes, more complex irregular assignment computations can be found. The example loop do_h shown in Fig. 4.6(a) contains multiple assignment statements that may potentially write on different entries of array a during the execution of a loop iteration. The statements are executed if a condition $c(h)$ is fulfilled. The SCC graph that represents do_h is depicted in Fig. 4.6(c). It should be noted that the SCC graph is similar to that of the simpler case shown in Fig. 4.5(c). In fact, the two types of irregular assignments are represented by SCCs whose class is *(non-)cond/assig/subs*, the only difference being the conditionality of the SCC (see $SCC_1^A(a_{1...2})$ and $SCC_1^A(a_{1...4})$ in the SCC graphs of Figs. 4.5(c) and 4.6(c), respectively). The characteristics that distinguish these irregular assignments are represented in the SCC graphs as follows. First, the NWSN subgraph of $SCC_1^A(a_{1...4})$ contains a control chain $SCC_1^A(a_{1...4}) \rightsquigarrow SCC_1^S(h_1)$. According to the entry 7 of Table 4.2, the classification process can continue with the analysis of non-structural use-def chains.

```
a(...) = ...
DO h = 1,n
   IF (c(h)) THEN
      tmp = b(h)
      a(f(h)) = tmp + K1
      a(g(h)) = tmp + K2
   END IF
END DO
... = ...a(...)...
```

(a) Source code.

```
DO h1 = 1,n,1
   a1 = μ(a0,a4)
   tmp1 = μ(tmp0,tmp2)
   IF (c(h1)) THEN
      tmp2 = b(h1)
      a2(f(h1)) = α(a1,tmp2 + K1)
      a3(g(h1)) = α(a2,tmp2 + K2)
   END IF
   tmp3 = γ(c(h1),tmp2,tmp1)
   a4 = γ(c(h1),a3,a1)
END DO
```

(b) GSA form.



(c) SCC graph.

Figure 4.6: Irregular assignment computations with multiple assignment statements.

The subgraph also contains two non-structural chains from $SCC_1^A(a_{1...4})$ to each trivial component $SCC_1^S(h_1)$ and $SCC_1^S(tmp_2)$. There is a pair of use-def chains $SCC_1^A(a_{1...4}) \nRightarrow SCC_1^S(h_1)$ and $SCC_1^A(a_{1...4}) \nRightarrow SCC_1^S(tmp_2)$ for each assignment statement of array a in the source code. For the sake of clarity, the two chains are depicted in Fig. 4.6(c) as a unique dashed edge with two labels. The behavior of the loop classification algorithm does not change significantly. In fact, the NWSN subgraph inherits $[SCC_1^A(a_{1...4})]$, i.e. the class *cond/assig/subs* of the NWSN.

The SCC graph contains another NWSN subgraph associated with the node of $SCC_0^S(tmp_{1,3})$. According to Tables 4.1 and 4.2, the NWSN subgraph should inherit $[SCC_0^S(tmp_{1,3})]$, i.e. *cond/none*. However, this class will not be considered because it is associated with a virtual component that arises from the definition of the scalar temporary variable *tmp* within an if-endif construct. This case is an exception to the general behavior of the transfer function that merges the classes of the NWSN subgraphs (see Section 4.3).

The final step of the loop classification algorithm is the merging of the classes of both NWSN subgraphs. The resulting loop class is *cond/assig/subs*.

4.4.2 Minimum with Location

An important characteristic of our compiler framework is that it enables the recognition of structural and semantic kernels in a unified manner. The minimum with location kernel (see Section 2.2.2) is a representative example of the combination of structural and semantic basic kernels (see Def. 3.3). A specific technique for the detection of this type of kernel was presented in [44].

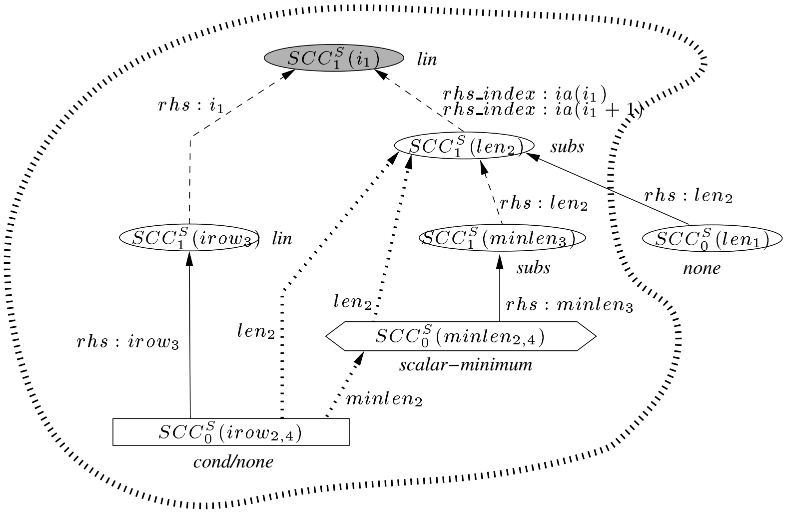
Consider the loop do_i shown in Fig. 4.7 which calculates the length $minlen$ of the smallest row of a sparse matrix stored in CRS format, as well as the corresponding row number $irow$. In our framework, the SCC graph of do_i contains one NWSN subgraph. The loop classification scheme proceeds as follows. Let us focus on structural and control use-def chains. The control chains $SCC_0^S(irow_{2,4}) \rightsquigarrow SCC_1^S(len_2)$, and $SCC_0^S(minlen_{2,4}) \rightsquigarrow SCC_1^S(len_2)$ will not be considered because, as len_2 is detected as a temporary scalar variable, they do not provide useful information. First, the compiler addresses the analysis of the structural chain $SCC_0^S(irow_{2,4}) \Rightarrow SCC_1^S(irow_3)$. As the right-hand side expression of the statement $irow_3 = i_1$ consists of an occurrence of the index variable i_1 of the outermost loop, $[SCC_0^S(irow_{2,4}) \Rightarrow SCC_1^S(irow_3)]$ is set to the class *scalar-location* (entry 1 of Table 4.1). Second, the structural chain $[SCC_0^S(minlen_{2,4}) \Rightarrow SCC_1^S(minlen_3)]$ is set to *scalar-minimum* (entry 3 of Table 4.1). Note that at this point of the analysis, the necessary information for detecting a scalar minimum kernel is available as the condition $len_2 < minlen_2$ fulfills the properties defined in Section 3.4.2 for the SCC class *scalar-minimum*. The check consists of the condition $len_2 < minlen_2$ matching $e < v$, where v is an occurrence of the reduction variable $minlen$ defined in the scalar statement $minlen_3 = len_2$ of the target component $SCC_1^S(minlen_3)$, and e consists of the right-hand side of that statement. Finally, according to the entry 8 of Table 4.2, the control chain $SCC_0^S(irow_{2,4}) \rightsquigarrow SCC_0^S(minlen_{2,4})$ enables the recognition of a *scalar-minimum-w/loc* kernel. For this kernel to be successfully detected, the compiler must check that the scalar statements $minlen_3 = len_2$ and $irow_3 = i_1$ belong to the same basic block within the loop body. The demand-driven clas-

```
minlen = ia(2) - ia(1)
irow = 1
DO i = 2, nrow
   len = ia(i + 1) - ia(i)
   IF (len < minlen) THEN
      minlen = len
      irow = i
   END IF
END DO
```

(a) Source code.

```
minlen_1 = ia(2) - ia(1)
irow_1 = 1
DO i_1 = 2, nrow, 1
   len_1 = μ(len_0, len_2)
   irow_2 = μ(irow_1, irow_4)
   minlen_2 = μ(minlen_1, minlen_4)
   len_2 = ia(i_1 + 1) - ia(i_1)
   IF (len_2 < minlen_2) THEN
      minlen_3 = len_2
      irow_3 = i_1
   END IF
   irow_4 = γ(len_2 < minlen_2, irow_3, irow_2)
   minlen_4 = γ(len_2 < minlen_2, minlen_3, minlen_2)
END DO
```

(b) GSA form.



(c) SCC graph.

Figure 4.7: Computation of minimum and its location (extracted from *SparsKit-II*, module *unary*, subroutine *blkfnd*).

sification algorithm finishes by setting the NWSN subgraph class and the loop class to *scalar-minimum-w/loc*.

The example loop described above is simple. However, it is a representative case of more complex kernels that involve subscripted subscripts. The code shown in Fig. 4.8 calculates the minimum value m of the rows $n1 \ldots n2$ of a sparse matrix stored in CRS format (a, ia and ja). The location l within array a is also computed. The SCC graph of Fig. 4.8(c) is very similar to that of Fig. 4.7(c). In fact, the loop classification scheme is faced with similar structural and control use-def chains, the differences being the classes of the source and the target SCCs.

A more complex kernel is presented in Fig. 4.9. It consists of calculating the minimum of each row of a sparse matrix in CRS format and their locations. The main difference with respect to the previous cases is that minima and locations are stored in two array variables m and l, respectively. Thus, the array definition statements belong to the same SCC as the corresponding μ and γ-statements (two SCCs arise when scalars are involved as shown in Figs. 4.7 and 4.8). In the examples, the locations are represented by $SCC_1^A(l_{1\ldots5})$, whose class is *cond/assig/lin*. The minima are captured as a $SCC_1^A(m_{1\ldots5})$, whose class is *array-minimum*.

4.4.3 Consecutively Written Array

The *consecutively written array* computational kernel (see Section 2.2.5) is usually implemented by using an induction variable of step one to define which array entries are written during loop execution. The loop nest do_{ii} presented in Fig. 4.10 computes a permutation of the rows of a sparse matrix stored in CRS format (a, ia and ja). The result is also stored in a sparse matrix in CRS format (ao, iao and jao). The inner loop do_k initializes the arrays ao and jao with the values corresponding to one row of the matrix. This task is carried out by using the linear induction variable of step one, ko, as the subscript expression for the definition of ao and jao. The condition $values$, which is used to determine at run-time if the entries ao of the sparse matrix are computed, is loop-invariant. Thus, if $values$ is true, the assignment statement $ao(ko) = a(k)$ is executed in every loop iteration.

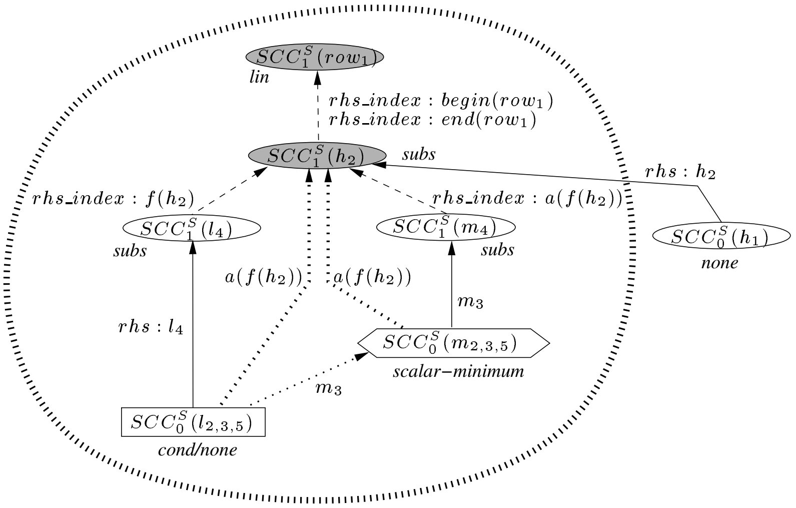
The parallel execution of the consecutively written array kernel of the in-

```
m = a(f(1))
l = f(1)
DO row = n1, n2
  DO h = begin(row), end(row)
    IF (a(f(h)) < m) THEN
      m = a(f(h))
      l = f(h)
    END IF
  END DO
END DO
```

(a) Source code.

```
m1 = a(f(1))
l1 = f(1)
DO row1 = n1, n2
  l2 = μ(l1, l3)
  m2 = μ(m1, m3)
  h1 = μ(h0, h2)
  DO h2 = begin(row1), end(row1)
    l3 = μ(l2, l5)
    m3 = μ(m2, m5)
    IF (a(f(h2)) < m3) THEN
      m4 = a(f(h2))
      l4 = f(h2)
    END IF
    l5 = γ(a(f(h2)) < m3, l4, l3)
    m5 = γ(a(f(h2)) < m3, m4, m3)
  END DO
END DO
```

(b) GSA form.



(c) SCC graph.

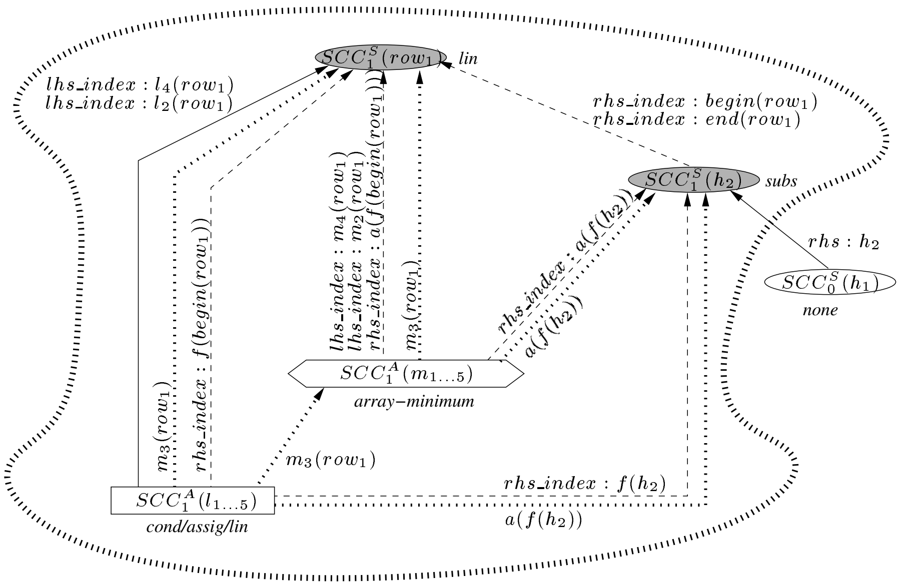
Figure 4.8: Computation of the minimum value of a set of rows of a sparse matrix, and its location.

```
DO row = 1,n
   m(row) = a(f(begin(row)))
   l(row) = f(begin(row))
   DO h = begin(row) + 1, end(row)
      IF (a(f(h)) > m(row)) THEN
         m(row) = a(f(h))
         l(row) = f(h)
      END IF
   END DO
END DO
```

(a) Source code.

```
DO row1 = 1,n
   m1 = μ(m0, m3)
   l1 = μ(l0, l3)
   h1 = μ(h0, h2)
   m2(row1) = α(m1, a(f(begin(row1))))
   l2(row1) = α(l1, f(begin(row1)))
   DO h2 = begin(row1) + 1, end(row1)
      m3 = μ(m2, m5)
      l3 = μ(l2, l5)
      IF (a(f(h2)) > m3(row1)) THEN
         m4(row1) = α(m3, a(f(h2)))
         l4(row1) = α(l3, f(h2))
      END IF
      m5 = γ(a(f(h2)) > m3(row1), m4, m3)
      l5 = γ(a(f(h2)) > m3(row1), l4, l3)
   END DO
END DO
```

(b) GSA form.



(c) SCC graph.

Figure 4.9: Computation of minimum of each row of a matrix and their locations.

nermost loop do_k can be accomplished using, for instance, induction variable substitution techniques or the array splitting and merging parallelizing transformation. However, it is possible for the compiler to extract coarser-grain parallelism from the outer loop do_{ii} by checking that the set of consecutive entries of arrays jao and ao that are written in the iterations of do_{ii} do not overlap at run-time. The strategies stated above will be described in more detail in Chapter 5. The rest of this section will focus on the detection problem.

Consider the loop do_{ii} of Fig. 4.10. As shown in Fig. 4.10(c), the corresponding SCC graph consists of one connected subgraph that contains two NWSN subgraphs associated with $SCC_1^A(jao_{1...3})$ and $SCC_1^A(ao_{1...4})$. Let us focus on the NWSN subgraph of $SCC_1^A(jao_{1...3})$. During the post-order traversal of this subgraph, structural use-def chains are processed in the following order. First, $SCC_1^S(ko_{3,4}) \Rightarrow SCC_1^S(ko_2)$ points out the possible existence of a *non-cond/lin-r/subs* (see entry 7 of Table 4.1). For the reinitialized induction variable to be detected, the compiler must check the position of the statements of the SCC within the body of do_{ii}:

1. The increments of ko are performed in do_k only, i.e. $SCC_1^S(ko_{3,4}) \sqsubseteq do_k$ (see Def. 4.3).
2. The reinitialization of ko is carried out in do_{ii} but not in do_k, i.e. $SCC_1^S(ko_2) \sqsubset do_{ii}$ (see Def. 4.4)
3. The statement ko_2 precedes the inner loop do_k in the control flow graph of the loop body.

All these conditions are fulfilled in the outermost loop do_{ii}, so ko is recognized as a *non-cond/lin-r/subs*.

The following step of the NWSN subgraph classification algorithm is the analysis of the structural use-def chain $SCC_1^A(jao_{1...3}) \Rightarrow SCC_1^S(ko_{3,4})$. In this case, the compiler must confirm the existence of the consecutively written array jao. For this purpose, we have used the approach of [32]:

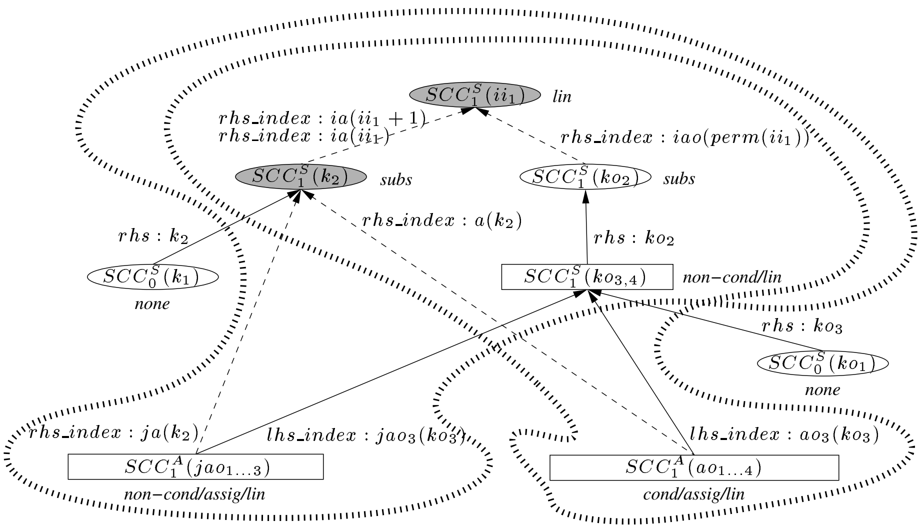
1. All the operations on ko are increments (or decrements) of the value 1.
2. Every time an array entry $jao(ko)$ is written, the corresponding induction variable ko is updated. This task is accomplished throughout the analysis of the CFG of do_{ii}.

```
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.

```
DO ii_1 = 1, nrow, 1
   jao_1 = μ(jao_0, jao_2)
   k_1 = μ(k_0, k_2)
   ko_1 = μ(ko_0, ko_3)
   ao_1 = μ(ao_0, ao_2)
   ko_2 = iao(perm(ii_1))
   DO k_2 = ia(ii_1), ia(ii_1 + 1) - 1, 1
      jao_2 = μ(jao_1, jao_3)
      ko_3 = μ(ko_2, ko_4)
      ao_2 = μ(ao_1, ao_4)
      jao_3(ko_3) = α(jao_2, ja(k_2))
      IF (values_1) THEN
         ao_3(ko_3) = α(ao_2, a(k_2))
      END IF
      ao_4 = γ(values_1, ao_3, ao_2)
      ko_4 = ko_3 + 1
   END DO
END DO
```

(b) GSA form.



(c) SCC graph.

Figure 4.10: Permutation of the rows of a sparse matrix (extracted from *SparsKit-II*, module *unary*, subroutine *rperm*).

Note the advantage of using our compiler framework for the recognition of consecutively written arrays. The framework provides precise information about what loops can potentially compute this kernel at run-time. Thus, specific detection techniques such as [32] can be applied on demand to confirm that the kernel is computed in the loop.

Regarding the NWSN subgraph of $SCC_1^A(ao_{1...4})$, the classification process is similar to that of the subgraph described above. A remarkable difference is that the component represents a *cond/cwa* that actually computes a *non-cond/cwa* kernel. This is because the condition $values$ is loop-invariant, so the array either is updated in every iteration or it is not modified during the execution of the loop.

4.4.4 Consecutively Written Array with Sparse Temporary Array

More complex consecutively written arrays can be found in sparse/irregular programs. Consider the example code shown in Fig. 4.11 and its SCC graph, which is depicted in Fig. 4.12. This subroutine builds a sparse matrix in CRS 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 stored in a sparse storage format ($imask$, $jmask$).

The code consists of two loop nests. In the first loop, do_{j_1}, the initialization of the temporary array variable 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, which are recognized as explained in Section 4.4.3. This example is specially interesting because of the use of the variable iw to determine the iterations where c and jc are computed. 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. In the second loop do_{k_4}, these marks are unset. There is a loop do_{k_3} where the value of iw is tested, and that is located in the middle of the loops do_{k_2} and do_{k_4}. If the mark is set, the corresponding elements $c(len)$ and $jc(len)$ are calculated, len being a *cond/lin* induction variable. Otherwise, no processing is performed.

```
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
```

(a) Source code.

```
DO j₁ = 1, ncol, 1
   iw₁ = μ(iw₀, iw₂)
   iw₂(j₁) = α(iw₁, false)
END DO

DO ii₁ = 1, nrow, 1
   iw₃ = μ(iw₁, iw₆)
   len₂ = μ(len₁, len₃)
   k₁ = μ(k₀, k₄)
   j₂ = μ(j₁, j₃)
   c₁ = μ(c₀, c₂)
   k1₁ = μ(k1₀, k1₂)
   k2₁ = μ(k2₀, k2₂)
   jc₁ = μ(jc₀, jc₂)
   ic₁ = μ(ic₀, ic₂)
   DO k₂ = imask(ii₁), imask(ii₁ + 1) - 1, 1
      iw₄ = μ(iw₃, iw₅)
      iw₅(jmask(k₂)) = α(iw₄, true)
   END DO
   k1₂ = ia(ii₁)
   k2₂ = ia(ii₁ + 1) - 1
   ic₂(ii₁) = α(ic₁, len₂ + 1)
   DO k₃ = k1₂, k2₂, 1
      len₃ = μ(len₂, len₅)
      j₃ = μ(j₂, j₄)
      c₂ = μ(c₁, c₄)
      jc₂ = μ(jc₁, jc₄)
      j₄ = ja(k₃)
      IF (iw₄(j₄)) THEN
         len₄ = len₃ + 1
         jc₃(len₄) = α(jc₂, j₄)
         c₃(len₄) = α(c₂, a(k₃))
      END IF
      len₅ = γ(iw₄(j₄), len₄, len₃)
      c₄ = γ(iw₄(j₄), c₃, c₂)
      jc₄ = γ(iw₄(j₄), jc₃, jc₂)
   END DO
   DO k₄ = imask(ii₁), imask(ii₁ + 1) - 1, 1
      iw₆ = μ(iw₄, iw₇)
      iw₇(jmask(k₄)) = α(iw₆, false)
   END DO
END DO
```

(b) GSA form.

Figure 4.11: Filter the contents of a sparse matrix using a mask vector (*SparsKit-II*, module *unary*, subroutine *amask*).

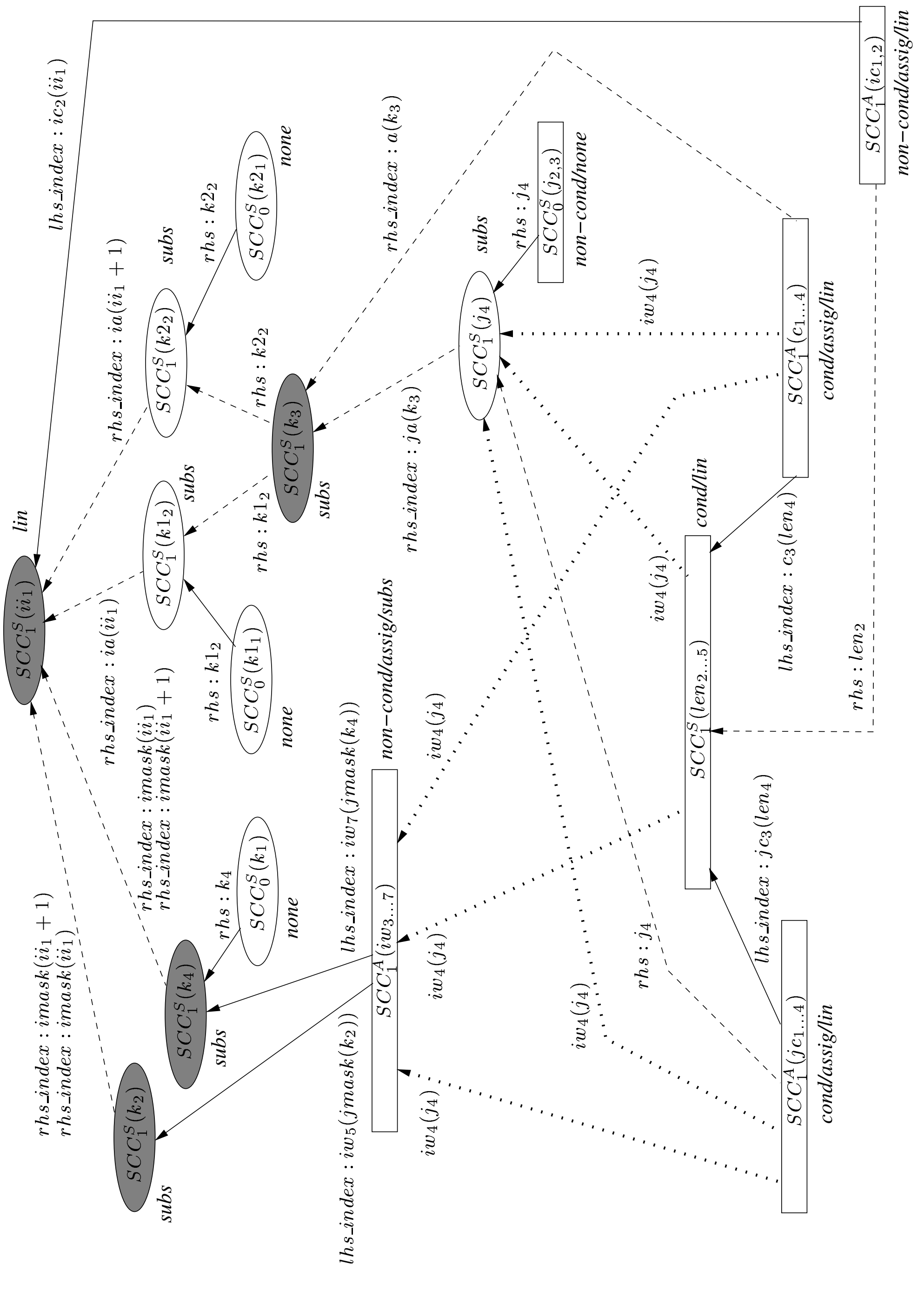


Figure 4.12: SCC graph of the code presented in Fig. 4.11.

In our compiler framework, do_{ii_1} is represented by a SCC use-def chain graph that contains three NWSNs (see Fig. 4.12): $SCC_1^A(jc_{1...4})$, $SCC_1^A(c_{1...4})$ and $SCC_1^A(ic_{1...2})$. During the analysis of the SCC graph, the demand-driven algorithm is faced with the classification of $SCC_1^A(iw_{3...7})$. For the masking mechanism to be recognized by the compiler, $SCC_1^A(iw_{3...7})$ must fulfill the following properties:

1. The $[SCC_1^A(iw_{3...7})]$ is *non-cond/assig/subs*, with two structural use-def 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. $SCC_1^A(iw_{3...7})$ is a multi-loop component (see Def. 4.2) that contains two α-statements that belong to the body of two different loops: do_{k_2} and do_{k_4}.

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

   (a) Check that the iteration spaces are equal. Applying Lemma 3.1, 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, and thus they are GSA-equivalent (see Def. 3.1). In the GSA form of Fig. 4.11(b), these expressions are $imask(ii_1)$, $imask(ii_1 + 1) - 1$ and 1 both in do_{k_2} and do_{k_4}.

   (b) Check that the left-hand side indices of the α-statements are GSA--equivalent expressions. This condition is fulfilled because both expressions are syntactically identical (see Theorem 3.1) except for the occurrences of the index variables k_2 and k_4. However, as shown above, k_2 and k_4 are GSA-equivalent (i.e. $k_2 \overset{GSA}{\equiv} k_4$). Thus, we can conclude that $jmask(k_2) \overset{GSA}{\equiv} jmask(k_4)$.

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 than $false$ in do_{k_2} ($true$ of iw_5), and those marks are deleted in do_{k_2} ($false$ of iw_7).

The analysis of the control use-def chains $SCC_1^A(jc_{1...4}) \rightsquigarrow SCC_1^A(iw_{3...7})$, $SCC_1^A(c_{1...4}) \rightsquigarrow SCC_1^A(iw_{3...7})$, and $SCC_1^S(len_{2...5}) \rightsquigarrow SCC_1^A(iw_{3...7})$ depicted in Fig. 4.12 is addressed next. The target component of these chains is

$SCC_1^A(iw_{3...7})$. In order to identify those computational kernels (in the example, the *(non-)cond/cwa* kernels associated with jc and c) whose execution is driven by the contents of the sparse and temporary array iw, the compiler performs the following checks during the execution of the transfer function:

1. The conditional expression $iw_4(j_4)$ that guards the *(non-)cond/cwa* kernels 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.

Note that no kernel class is added to the NWSN class (see the symbol "$-$" in the last entry of Table 4.2). At the end of the NWSN subgraph classification process, the classes of the NWSNs, $SCC_1^A(jc_{1...4})$ and $SCC_1^A(c_{1...4})$, will be added, i.e. *cond/cwa*.

Finally, the third NSWN subgraph associated with $SCC_1^A(ic_{1...2})$ is classified. According to the entry 9 of Table 4.1, the structural use-def chain $SCC_1^A(ic_{1...2}) \Rightarrow SCC_1^S(ii_1)$ is classified as *non-cond/assig/lin*. This class is inherited by the NWSN subgraph and, at the end, it is added to the loop class, which results to be $\{$*cond/cwa*,*non-cond/assig/lin*$\}$.

4.5 Experimental Results

The goal of the SCC graph classification algorithm is the recognition of the set of computational kernels calculated in a loop. In this section we present statistics about the efficacy of this algorithm for the analysis of the *SparsKit-II* library.

Table 4.3 presents, for each nesting level and for each module of *SparsKit-II*, the number of loops that are amenable for classification and those that are not (see Section 3.7.1). Further statistics are shown for amenable loops. In particular, the number of loops whose computational kernels were successfully recognized. Next four columns show the numbers for the modules *matvec*, *blassm*, *unary* and *formats*, respectively. The last column summarizes, for each loop class, the total number of occurrences in *SparsKit-II*. Percentages with respect to the total number of loops in *SparsKit-II* are also presented. The rows of the table are organized in sets. Each set corresponds to a nesting level, level-1 being the innermost level. The last set summarizes the information for all the nesting levels.

Experimental results show that 19% of the loops of *SparsKit-II* are not amenable for classification due to the presence of jump statements, the existence of procedure calls or an incorrect translation into GSA form. From the remaining loops 81%, 47% contain computational kernels that can be recognized by our prototype. The 96% of the amenable loops have nesting levels 1 and 2. In fact, the *SparsKit-II* library contains only 16 loops with nesting levels 3 and 4. The effectiveness decreases as the nesting level rises because outer loops usually compute more complex kernels. In fact, all the recognized loops concentrate in levels 1 and 2 (38% and 9%, respectively). The only exception is a level-4 loop nest in the module *matvec*.

Table 4.4 shows, for each nesting level and for each module, the wide variety of kernels classes that were successfully recognized by our prototype. The first column presents the kernel classes in the form of a tree. The last column summarizes the total number of occurrences for each kernel class. Blank entries mean zero occurrences of the class. In our compiler framework, the class of a loop represents the set of kernels computed in the loop. For this reason, the total number of kernels (207) is greater than the total number of loops recognized by the prototype, 176 (see totals in Table 4.3).

The most numerous classes are *non-cond/assig/lin* and *non-cond/recur/lin*. These classes mainly correspond to level-1 simple loops that initialize the entries of an array variable that will be referenced in subsequent loop nests within a routine (see the discussion about the *non-cond/assig/lin* SCC class in Section 3.7.2). Our approach has detected well-known kernels such as irregular reductions (21 *non-cond/reduc/subs*, and 4 *cond/reduc/subs*), even in level-4 loops. But also it has detected multiple irregular assignment kernels (18 *non-cond/assig/subs*, and 4 *cond/assig/subs*), consecutively written arrays (15 *non-cond/cwa*, and 15 *cond/cwa*), and semantic kernels such as *scalar-minimum-w/loc* and *array-find-and-set* (1 and 3 level-1 loops, respectively).

An interesting characteristic of our classification scheme is that it enables the recognition of loops that include a combination of structural and semantic

	matvec	*blassm*	*unary*	*formats*	*SparsKit-II*
Level-1 loops	22	32	86	104	244 (65%)
Not amenable for classification	0	3	22	16	41 (11%)
Amenable for classification	22	29	64	88	203 (54%)
Recognized	20	22	51	49	142 (38%)
Level-2 loops	16	8	35	60	119 (31%)
Not amenable for classification	1	0	16	11	28 (7%)
Amenable for classification	15	8	19	49	91 (24%)
Recognized	5	2	12	14	33 (9%)
Level-3 loops	1	1	4	6	12 (3%)
Not amenable for classification	0	0	2	1	3 (1%)
Amenable for classification	1	1	2	5	9 (2%)
Recognized	0	0	0	0	0 (0%)
Level-4 loops	1	0	1	2	4 (1%)
Not amenable for classification	0	0	1	0	1 (0%)
Amenable for classification	1	0	0	2	3 (1%)
Recognized	1	0	0	0	1 (0%)
Total number of loops	40	41	126	172	379 (100%)
Not amenable for classification	1	3	41	28	73 (19%)
Amenable for classification	39	38	85	144	306 (81%)
Recognized	26	24	63	63	176 (47%)

Table 4.3: Effectiveness of the SCC graph classification algorithm for the recognition of the loop-level kernels computed in *SparsKit-II*.

basic computational kernels. The code of Fig. 4.13(a) presents an interesting loop that calculates two computational kernels: an *array-find-and-set* semantic kernel, *iwk*, and a *cond/cwa* structural kernel, *kvstc*. The loop class is successfully recognized by the prototype.

As shown in Table 4.3, 19% of the loops of *SparsKit-II* are not amenable for classification. However, we have checked a great deal of these loops contain computational kernels that could be recognized with our classification scheme. In many cases, applying restructuring techniques to the source code before running the classification algorithms could lead the prototype to the recognition of the kernels. Some examples including structural and seman-

	matvec	*blassm*	*unary*	*formats*	*SparsKit-II*
Level-1 loops	20	26	56	57	159
Structural kernels	20	26	52	57	155
non-cond/lin			1		1
non-cond/red	5		5		10
non-cond/assig/lin	11	12	25	24	72
non-cond/assig/subs		7	4	1	12
non-cond/reduc/lin		1	1	1	3
non-cond/reduc/subs	4		2	3	9
non-cond/recur/lin			7	16	23
non-cond/cwa		1	2	4	7
cond/lin		2	1	1	4
cond/assig/inv				1	1
cond/assig/lin				1	1
cond/assig/subs		1	2		3
cond/reduc/inv				1	1
cond/reduc/subs		1			1
cond/cwa		1	2	4	7
Semantic kernels	0	0	4	0	4
scalar-minimum-w/loc			1		1
array-find-and-set			3		3
Level-2 loops	5	2	23	17	47
Structural kernels					
non-cond/assig/lin	1		5	1	7
non-cond/assig/subs		2	2	2	6
non-cond/reduc/lin	1				1
non-cond/reduc/subs	3		3	5	11
non-cond/cwa			6	2	8
cond/assig/lin			1	1	2
cond/assig/subs				1	1
cond/reduc/subs				3	3
cond/cwa			6	2	8
Level-4 loops	1	0	0	0	1
Structural kernels	0	0	0	0	0
non-cond/reduc/subs	1				1

Table 4.4: Loop-level kernels recognized by the SCC graph classification algorithm in the *SparsKit-II* library.

```
nc = 1
kvstc(1) = 1
DO i = 2, ncol + 1
   IF (iwk(i) ≠ 0) THEN
      nc = nc + 1
      kvstc(nc) = i
      iwk(i) = 0
   END IF
END DO
nc = nc − 1
```

(a) Level-1 loop extracted from *SparsKit-II*, module *unary*, routine *csrkvstc*. The routine finds a block column partitioning of sparse matrix in CRS.

```
DO k = k2, k1, −1
   j = jb(k)
   IF (test.and.(j < ii)) THEN
      test = false
      ko = ko − 1
      b(ko) = diag(ii)
      jb(ko) = ii
      iw(ii) = ko
   END IF
   ko = ko − 1
   b(ko) = a(k)
   jb(ko) = j
END DO
```

(b) Innermost loop extracted from *SparsKit-II*, module *blassm*, routine *apldia*. The routine adds a diagonal matrix to a general sparse matrix.

Figure 4.13: Interesting codes where the SCC graph classification algorithm fails although the SCC graph contains no unknown SCC.

tic kernels are: 1 *non-cond/reduc/subs* in *matvec*; 3 *cond/cwa* in *blassm*; 2 *non-cond/reduc/subs*, 6 *scalar-maximum*, and 9 *(non-)cond/cwa* in *unary*; and 5 *scalar-maximum (minimum)*, 1 *scalar-maximum-w/loc*, and 13 *cond/cwa* in *formats*.

4.5.1 Failures in SCC Graph Recognition

The main reasons that make the SCC graph classification algorithm fail are the existence of *unknown* SCCs, the existence of use-def chains that have not been considered in the transfer functions of the algorithm, and the limitations of the methods that are used for the detection of specific computational kernels. According to Table 4.3, 34% of the total number of loops in *SparsKit-II* are not successfully recognized. The main reason for failure is the presence of *unknown* SCCs in the SCC graph, which represents up to 30% of loops. Because of its relevance, this issue will be addressed first. Next, we will focus on the remaining 4% of the loops, which are represented by SCC graphs free of *unknown* SCCs.

As discussed in Section 3.7.3, the SCC classification algorithm may fail for several reasons, the most relevant being the existence of SCCs with cardinality greater than one, multidimensional array references, and SCCs with statements of several classes. In Fig. 4.13(b) an interesting loop that computes a semantic *scalar-find-and-set* kernel, *test*, jointly with other structural kernels is presented. The scalar variable *test* is used to distinguish the first loop iteration so that special values can be stored in array entries $b(ko)$, $jb(ko)$ and $iw(ii)$, where ko is a linear induction variable and ii is a loop-invariant value. The classification of do_k fails because there is a structural SCC of cardinality two that represents the computations of the variables j and jb. This SCC arises because the statements $j = jb(k)$ and $jb(ko) = j$ may introduce loop-carried dependences during the execution of the loop. A similar loop can also be found in the routine *aplsca* of *blassm*. Other example loops with SCCs of cardinality greater than one were shown in Section 3.7.

In the following, let us focus on the 4% of non-recognized loops whose SCC graphs are free of *unknown* SCCs. The transfer functions presented in Section 4.3 for the classification of structural, control and non-structural use-def chains cover a wide range of the kernels that can be found in *SparsKit-II*. However, there are kernels that cannot be successfully recognized. Consider the loop do_j of Fig. 4.14(a), which contains two structural basic kernels: *non-cond/assig/subs* (variable *index*) and a *non-cond/reduc/subs* (variable *icnt*). The distinguishing characteristic of this loop is that *icnt* is referenced in the left-hand side index expression of the array assignment statement $index(icnt(ivalj)) = j$. As a result, the SCC graph of do_j contains a non-structural use-def chain between both components. This is an example of a use-def chain between two structural SCCs that leads the transfer function of non-structural chains to classify the chain as *unknown*. Thus, the use-def chain and, hence, the SCC graph are classified as *unknown*. This example illustrates the fact that the transfer functions proposed in this thesis could be adapted to cover a wider range of computational kernels. A similar situation is found in other loops of *SparsKit-II* (e.g. do_{30} in the routine *levels* of the module *unary*).

On the other hand, our framework provides information that the compiler can use as a guide for the detection of specific computational kernels. Consider the loop do_{jj} of Fig. 4.14(b), which contains a non-conditional consecutively

```
DO j = n, 1, -1
   ivalj = ival(j)
   index(icnt(ivalj)) = j
   icnt(ivalj) = icnt(ivalj) - 1
END DO
```

(a) Loop nest extracted from *SparsKit-II*, module *formats*, routine *dcsort*. The routine is devoted to computation related to sorting algorithms.

```
DO jj = 1, numc
   b(b1) = a(a0)
   b1 = b1 + neqr
   a0 = a0 + 1
END DO
```

(b) Innermost loop extracted from *SparsKit-II*, module *formats*, routine *csrvbr*. The routine converts the CRS storage format of a sparse matrix into variable block row format.

Figure 4.14: Interesting codes where the SCC graph classification algorithm fails although the SCC graph contains no unknown SCC (cont.).

written array b. In the SCC graph, these computations would be represented by means of a use-def chain between two SCCs: a structural scalar SCC for the induction variable $b1$, and a structural array SCC for the array assignment operation of array b. As the use-def chain is structural and the left-hand side index expression of the statement $b(b1) = a(a0)$ is an occurrence of $b1$ (i.e. the label of the use-def chain is lhs_index), the compiler can use the algorithm described in Section 4.4.3 for the detection of consecutively written arrays (see the blank entry number 17 of Table 4.1). Note that the detection technique fails if the compiler cannot determine that $neqr$ takes the value 1 during the execution of do_{jj}. As a consequence, it would be necessary to extend the capabilities of the detection technique in order to recognize the kernel of the figure. The prototype has detected 5 loops in *SparsKit-II* in which the detection technique described in Section 4.4.3 has failed.

The classification of the SCC graph of the loop do_{jj} of Fig. 4.14(b) will also fail for another reason. The loop contains a non-conditional linear induction variable $a0$ that is used to access a read-only array a located in the right-hand side of the assignment statement $b(b1) = a(a0)$. Thus, there is a non-structural use-def chain between the SCC of array b and the SCC of the linear induction variable $a0$. The application of the corresponding transfer function leads the compiler to classify the SCC graph as *unknown*. A similar situation is found in the loop do_{100} of the routine *lnkcsr* of *formats*. Instead of a non-conditional induction variable, a linked-list kernel is used to access a read-only array.

Chapter 5

Generation of Parallel Code

The kernel recognition technique presented in the previous chapters rests on two classification algorithms that perform an exhaustive analysis of the code of a loop. These algorithms could be used as an information-gathering framework that provides support for other optimizing and parallelizing techniques. In this chapter, we focus on the application of this compiler framework to the scope of automatic parallelization of sparse/irregular codes.

The rest of the chapter is organized as follows. In Section 5.1, an automatic parallelization method based on our compiler framework is presented. The remaining sections of the chapter are devoted to the analysis of the case studies presented in Section 4.4, that is, the automatic parallelization of irregular assignments (Section 5.2), minimum reduction operations (Section 5.3) and consecutively written arrays (Section 5.4). Finally, experimental results that compare the effectiveness of our method with the Polaris parallelizing compiler are presented in Section 5.5.

5.1 Automatic Parallelization Method

The key idea of our method consists of using a repository of the parallelizing transformations that were proposed in the literature for each computational kernel (see Fig. 2.4). In the scope of irregular codes, well-known examples are *irregular reductions* [20, 22, 32, 55], *irregular assignments* [6, 27] or *DOACROSS loops* [33, 54]. Once the kernels computed in a loop have been recognized, the compiler can select the most efficient transformation from the repository. This selection phase is critical for maximizing the performance of the parallel code. The most appropriate transformation should be determined not only by considering the characteristics of the target parallel architecture (shared memory, distributed memory, distributed/shared memory, etc...), but also the parameters of the application itself. For instance, in [55] a method to select the most appropriate technique for the parallel execution of irregular reductions is presented. Similar approaches could be used for other computational kernels.

The last stage of this algorithm is the generation of parallel code according to the selected technique. If there is no technique for a given kernel in the repository, then a generic approach could be applied; for instance, one based on speculative parallel execution of irregular loops [39]. General methods can be applied to any loop with irregular computations. However, efficiency usually falls with respect to code transformations that are tuned for the efficient execution of a specific computational kernel on a specific target architecture. In order to generate parallel code according to a given parallelizing transformation, it is necessary for the compiler to extract information from the loop body, for instance, what variables store the results of the computations, what array variables determine the access patterns of other variables in irregular computations, or what is the most appropriate point of the program for the insertion of the code of a run-time test. It should be noted that, in general, each parallelizing transformation requires that the compiler retrieves specific information. In the following, a brief description of the information-gathering process for the case studies of Chapter 4 is presented.

5.2 Irregular Assignment

The detection of several syntactical variations of irregular assignments was described in Section 4.4.1. In this section, we will focus on explaining how our compiler framework can provide compiler support for the generation of parallel code. Two parallelizing transformations were proposed in the literature. The first approach, which is based on the concept of array expansion, is presented in Section 5.2.1. The second one, which uses the inspector-executor model, is described in Section 5.2.2. The discussion will be oriented to reveal the relevant information for the implementation of each parallelizing technique, as well as to show how such information can be gathered during the construction of our framework.

5.2.1 Array Expansion Approach

The use of the Array SSA program representation as support for the automatic parallelization of irregular assignments is described in [27]. Parallel code for P processors is generated as follows. Each array definition a in the sequential code, is replaced by two new array variables, $a_1(1 : a_{size})$ and $@a_1(1 : a_{size})$. The value of the array element defined in the sentence is stored in a_1. The @-array stores the last loop iteration at which the elements of array a_1 were modified. Each processor p is assigned a set of iterations of the sequential loop, and the arrays are expanded as $a_1(1 : a_{size}, 1 : P)$ and $@a_1(1 : a_{size}, 1 : P)$. As a result, distinct processors, p_1 and p_2, can write into different memory locations concurrently, $a_1(f(h_1), p_1)$ and $a_1(f(h_2), p_1)$, while computing different iterations h_1 and h_2, respectively. The entries of array a are computed by a reduction operation. Consider the irregular assignment of Fig. 5.1(a). This method replaces the sequential loop do_h with the three-phase parallel code shown in Fig. 5.1(b). In the first stage, each processor p initializes its region of the expanded arrays, $a_1(1 : a_{size}, p)$ and $@a_1(1 : a_{size}, p)$. Next, processor p executes the set of loop iterations that it was assigned, $iterations(p)$, preserving the order of the sequential execution. At the end of the second stage, partial results computed by different processors are stored in separate memory locations. In the last stage, the value of each array element $a(j)$ $(j = 1, ..., a_{size})$ is determined by means of a reduction operation that obtains the value of the element of $a_1(j, 1 : P)$ with the highest iteration number in $@a_1(j, 1 : P)$. Each processor computes the final value of a subset of array elements $a(j)$, which is denoted as $array\_elements(p)$. Note that processors must be synchronized at the end of the execution stage to ensure that the computation of arrays a_1 and $@a_1$ has finished before performing the reduction at the finalization stage.

For the array expansion approach to be implemented, the compiler needs to extract the following information from the source code:

1. The array variable that stores the result. Consider the example of Fig. 5.1(a). The array a is identified as the source code variable whose

computations are represented by the component $SCC_1^A(a_{1...2})$ of class *non-cond/assig/subs*.

2. The source code statements that perform write operations on the array, as their location within the CFG is necessary for the insertion of the @-arrays. Within our framework, those statements are the source code statements associated with the α-statements of the SCC that represents the irregular assignment computations. In the example of Fig. 5.1(a), the statement of $SCC_1^A(a_{1...2})$ is $a_2(f(h_1)) = \alpha(a_1, tmp_2 + K)$.

In order to apply that parallelizing transformation, the compiler needs further information that is not gathered during the analysis of the loop body: the range of the array (a_{size}), the number of processors (P) and the mapping of computations to processors ($iterations(p)$ in the execution stage and $array\_elements(p)$ in the finalization stage). The compiler may obtain this information in several ways: user supplied parameters, lexical and syntactical analysis of the source code before translation into the internal representation, or default values determined by the compiler.

An optimization to perform element-level dead code elimination at run-time is presented in [27]. In irregular assignments, the same array element may be computed several times, although only the last value is used after the loop ends. Consequently, intermediate values need not be computed. Classical dead code elimination typically removes assignment statements from the source code. This technique eliminates unnecessary array element definitions at run-time. In order to apply this optimization, no further information is needed.

Table 5.1 summarizes the relevant information from the point of view of the generation of parallel code. The first column shows the computational kernel. There is an entry for each kernel described in this chapter, namely, irregular assignment, minimum with location and consecutively written array. The second column indicates the parallelizing transformation. The last column presents both the information extracted from the source code by the compiler, as well as other information that is necessary for the implementation of each parallelizing technique. The minimum with location kernel and the consecutively written array kernel will be addressed in Sections 5.3 and 5.4, respectively.

```
a(...) = ...
DO h = 1, f_size
   tmp = b(h)
   a(f(h)) = tmp + K
END DO
... = ...a(...)...
```

(a) Source code (also presented in Fig. 4.5).

```
DOALL p = 1, P
   ! ---- Initialization stage ----
   a_1(1 : a_size, p) = 0
   @a_1(1 : a_size, p) = 0

   ! ----- Execution stage -----
   DO h ∈ iterations(p)
      tmp = b(h)
      a_1(f(h), p) = tmp + K
      @a_1(f(h), p) = h
   END DO
END DOALL

! ---- Finalization stage ----
DOALL p = 1, P
   DO j ∈ array_elements(p)
      p_max = MAXLOC(@a_1(j, 1 : P))
      IF (@a_1(j, p_max) > 0) THEN
         a(j) = a_1(j, p_max)
      END IF
   END DO
END DOALL
```

(b) Parallel code.

Figure 5.1: Parallelization of irregular assignment computations using an approach based on array expansion. Element-level dead code elimination is not considered.

5.2.2 Inspector-Executor Approach

We proposed in [6] a technique targeted for distributed-shared memory architectures that is based on the inspector-executor model is presented. The key idea consists of reordering loop iterations so that data write locality is exploited on each processor. The sequential loop is replaced with a two-phase parallel code (see Fig. 5.2). In the *inspector* stage (see Figure 5.2(a)), array a is split into subarrays of consecutive locations, a_p $(p = 1, ..., P)$, and the computations associated with each block are assigned to different processors. Load-balancing is preserved by building subarrays a_p of different sizes. As a result, the loop iteration space $(1, ..., f_{size})$ is partitioned into sets f_p that perform write operations on different blocks a_p. In the *executor* stage (see Figure 5.2(b)), each processor p executes the conflict-free computations associated with the loop iterations contained in a set f_p.

<table>
<tr><th>Computational Kernel</th><th>Parallelizing transformation</th><th>Relevant information for the generation of parallel code</th></tr>
<tr><td rowspan="2">Irregular Assignment</td><td>Array Expansion (with dead code elimination)</td><td>a, P, a_{size}
α-statements in CFG
Mapping of sequential iterations
Mapping of array entries</td></tr>
<tr><td>Inspector-Executor (with dead code elimination)</td><td>a, P, a_{size}, f_{size}
lhs of α-statements of the SCC
Location of the inspector code</td></tr>
<tr><td>Minimum with Location</td><td>Parallel reduction</td><td>P
Mapping of sequential iterations
Reduction variable</td></tr>
<tr><td rowspan="2">Consecutively written array</td><td>Splitting and merging</td><td>a, P</td></tr>
<tr><td>$DOALL$ loop with run-time test</td><td>a, P
I, L</td></tr>
</table>

Table 5.1: Relevant information of several loop-level kernels for the generation of parallel code.

The implementation of the inspector-executor approach in a parallelizing compiler requires the following information to be extracted from the loop body:

1. The array that stores the results (variable a).

2. The subscript expressions that define the access pattern for the array during the execution of the loop. This expression is $f(h)$ in the irregular assignment of Fig. 5.1(a). During the construction of the framework, the SCC classification algorithm analyzes the statement $a_2(f(h_1)) = \alpha(a_1, tmp_2 + K)$ of the component that represents the irregular assignment, $SCC_1^A(a_{1\ldots2})$. During the analysis, the contextual classification of the left-hand side subscript expression $f(h)$ is performed. When, at the end, the class of $f(h)$ is determined to be *subs*, the compiler can gather the expression $f(h)$. When the classification of $SCC_1^A(a_{1\ldots2})$ finishes and the class of $SCC_1^A(a_{1\ldots2})$ is determined to be *non-cond/assig/subs*, the compiler knows that the left-hand side subscript expression of the irregular assignment is $f(h)$.

3. The location within the program that enables the reuse of the inspector code. Next, we briefly outline an algorithm that can be used in the

```
! Frequency distribution
his(1 : a_size) = 0
DO h = 1, f_size
   his(f(h)) = his(f(h)) + 1
END DO

! Accumulative frequency distribution
DO h = 2, a_size
   his(h) = his(h) + his(h − 1)
END DO

! Computation of the linked lists
Refs = (his(a_size)/P) + 1
count(1 : P) = 0
DO h = 1, f_size
   thread = (his(f(h))/Refs) + 1
   IF (count(thread) = 0) THEN
      next(f_size + thread) = h
   ELSE
      next(prev(thread)) = h
   END IF
   prev(thread) = h
   count(thread) = count(thread) + 1
END DO
```

(a) Inspector code.

```
a(...) = ...
DOALL p = 1, P
   h = next(f_size + p)
   DO k = 1, count(p)
      tmp = b(h)
      a(f(h)) = tmp + K
      h = next(h)
   END DO
END DOALL
... = ...a(...)...
```

(b) Executor code.

Figure 5.2: Parallelization of the irregular assignment of Fig. 5.1(a) using an approach based on the inspector-executor model. Element-level dead code elimination is not considered.

scope of our framework as it takes advantage of the demand-driven implementation of the GSA form. Let $\{v_1, ..., v_n\}$ be the set of variables that are referenced in the subscript expressions that define the access pattern for array a. Let $\{s_1, ..., s_n\}$ represent the nodes of the CFG that are associated with the definition statements of variables $\{v_1, ..., v_n\}$. The inspector can be inserted just after the first node s of the CFG that is reachable from $s_1, ..., s_n$. The demand-driven implementation of the GSA form provides an efficient solution to the statement-level reaching definition problem. Thus, the identification of $\{s_1, ..., s_n\}$ from $\{v_1, ..., v_n\}$ is straightforward. Next, the statement s can be determined through the analysis of the CFG. Several considerations should be taken into account. For example, it is not desirable that s belongs to a loop body, in order to prevent the inspector from being computed several times unnecessarily.

Further information that must be available at compile-time is the range of the arrays a and f (a_{size} and f_{size}, respectively) and the number of processors (P). The information is summarized in Table 5.1.

This approach also supports element-level dead code elimination at run-time. As in the array expansion technique, no further information is needed.

5.3 Minimum with Location

In Section 4.4.2, the detection of an example loop nest that computes a scalar minimum with location kernel was described in detail. The parallelization strategy for this kernel is similar to the parallelization of sum scalar reduction operations, for instance. The method consists of two stages. In the first stage, each processor calculates the local results corresponding to a set of loop iterations (the sets are constructed so that they define a partition of the loop iteration space). Next, the local results are combined to compute the result of the reduction operation. In the example code shown in Fig. 4.7, the local result is a pair $< minlen, irow >$. The pairs are combined so that the result consists of the pair with the lowest $minlen$ value. Hereafter, we will refer to the pair $< minlen, irow >$ as the reduction variable.

In order to implement the parallelizing transformation described above, the compiler has to extract the reduction variable from the loop body. In the scope of our compiler framework, the reduction variable $< minlen, irow >$ can be determined during the execution of the SCC graph classification algorithm, more specifically, during the analysis of the control use-def chain $SCC_0^S(irow_{2,4}) \rightsquigarrow SCC_0^S(minlen_{2,4})$ (see the SCC graph of Fig. 4.7(c)) whose target and source SCCs represent the computation of the minimum and the computations of the location, respectively. As shown in Table 5.1, further information must be available at compile-time: the number of processors P, and the mapping of loop iterations to processors.

5.4 Consecutively Written Array

In Sections 4.4.3 and 4.4.4, we described the detection of several syntactical variations of the consecutively written array (CWA) kernel. Hereafter, we will explain the parallelization strategy briefly, and we will describe the support that can be provided by our compiler framework. The section finishes with the description of a run-time test that is needed for the parallelization of one of the consecutively written arrays presented as an example (see Section 5.4.1).

Consecutively written array computations [32] consist of writing consecutive array locations in consecutive loop iterations. Non-conditional CWAs contain non-conditional induction variables. Current optimizing/parallelizing compilers parallelize this kernel by calculating a closed form expression for the induction variable and replacing the references to the variable with such an expression. In our framework, the information about the conditionality of the induction variable is intrinsically represented in the SCC class *non-cond/lin*. Furthermore, as our SCC classification algorithm is a generalization of the method proposed in [18], the closed form expression could be determined during the classification process as proposed in that work.

In general, the approach described above cannot be applied to conditional CWAs (except if the compiler can determine that the conditions are loop-invariant). In these cases, the *array splitting and merging* transformation described in [32] can be used. The technique consists of replacing the original loop with a three-phase code. In Fig. 5.3(b), a parallel implementation for shared-memory architectures of the CWA of Fig. 5.3(a) is shown. In an *initialization stage*, a private copy of array a is allocated to each processor p. In the *execution stage*, the processors work on private copies $a(1 : a_{size}, p)$ from position 1 in parallel. The iterations of the sequential loop are distributed among processors according to a block scheme (denoted as $consecutive\_iterations(p)$). After computation, each processor knows the number of array elements $l(p)$ modified in $a(1 : a_{size}, p)$. In the *finalization stage*, the starting position of the original array for each processor is calculated as $i(p) = (\sum_{h=1}^{p-1} l(h)) + 1$. Finally, the private copies are copied back (merged) to the original array a. This reconstruction process is depicted in Fig. 5.4. The dashed edges show the data movements during the execution of the copy back process.

As shown in Table 5.1, the implementation of the array splitting and merging method requires the array variable that stores the result to be identified by the compiler. Furthermore, the number of processors P must be available at compile-time. Our compiler framework provides support for recognizing the

```
i = 1
DO h = 1, n
   IF (...) THEN
      a(i) = Exp
      i = i + 1
   END IF
END DO
```

(a) Source code of a conditional consecutively written array kernel.

```
DOALL p = 1, P
   ! —— Initialization stage ——–
   a_aux(1 : a_size, p) = 0

   ! ——— Execution stage ———-
   l(p) = 1
   DO h ∈ consecutive_iterations(p)
      IF (...) THEN
         a_aux(i(p), p) = Exp
         l(p) = l(p) + 1
      END IF
   END DO
   l(p) = l(p) − 1
END DOALL

! ——– Finalization stage ——
DOALL p = 1, P
   ! —- Determine start position ——
   I(p) = 1
   DO h = 1, p − 1
      I(p) = I(p) + l(h)
   END DO
   ! —- Copy back array elements ——
   DO h = 1, l(p)
      a(I(p) + h) = a_aux(h, p)
   END DO
END DOALL
```

(b) Parallel code.

Figure 5.3: Array splitting and merging parallelizing transformation.

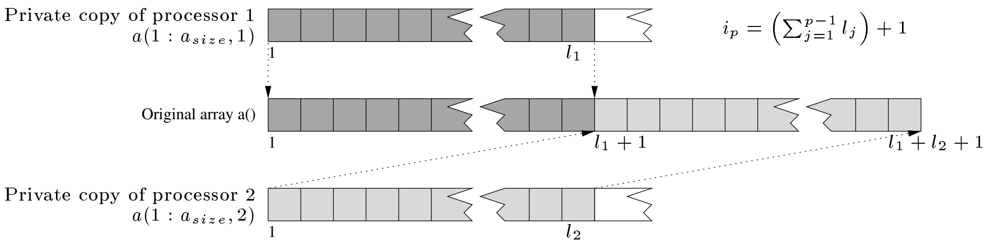


Figure 5.4: Illustration of the array splitting and merging technique for two processors.

array of results in the loop body. Consider the example shown in Fig. 4.10. The task is accomplished during the analysis of the structural use-def chains $SCC_1^A(jao_{1...3}) \Rightarrow SCC_1^S(ko_{3,4})$ and $SCC_1^A(ao_{1...4}) \Rightarrow SCC_1^S(ko_{3,4})$. The arrays of result, jao and ao, correspond to the array variables of the source components $SCC_1^A(jao_{1...3})$ and $SCC_1^A(ao_{1...4})$, respectively.

In Section 4.4.4, we described the recognition of the kernels computed in the level-2 loop nest do_{ii} of Fig. 4.11. We showed how the SCC classification algorithm determines that the loop class of do_{ii} consists of two kernels: *cond/cwa* (variables jc and c) and *non-cond/assig/lin* (variable ic). From the point of view of automatic parallelization, do_{ii} could be parallelized with the array splitting and merging transformation described above. However, there is an important fact related to ic that has to be considered in order to generate correct parallel code. In each do_{ii} iteration, $ic(ii)$ is set to $len + 1$, where len is the *cond/lin* induction variable of step 1 used to compute the consecutively written arrays jc and c. In the parallel code, each processor p calculates a set of consecutive elements of ic. As len is a local variable in each processor, the values stored in ic have to be corrected at the end of the loop execution in order to preserve the semantic of the sequetial code. In particular, each processor p has to add to its elements of ic the sum of the total number of elements computed by processors $1, \ldots, (p-1)$. Within our framework, the non-structural use-def chain $SCC_1^A(ic_{1...2}) \not\Rightarrow SCC_1^S(len_{2...5})$ of Fig. 4.12 provides the compiler with the necessary information for the generation of correct parallel code.

5.4.1 Run-Time Test

In Section 4.4.3, we described the recognition of a *(non-)cond/cwa* kernel in a level-2 loop nest that was extracted from the routine *rperm* of the module *unary* of the *SparsKit-II* library. This consecutively written array has a distinguishing characteristic that consists of using a reinitialized induction variable to access the elements of the array. In the code of Fig. 4.10, the linear induction variable of step one, ko, is reinitialized to a different loop-variant value, $iao(perm(ii))$, at the beginning of each iteration of the outermost loop do_{ii}. Thus, each do_{ii} iteration performs write operations on a subarray of arrays ao and jao. So, if the subarrays do not overlap, do_{ii} can be executed in a fully parallel manner ($DOALL$ loop). The pattern of write operations is depicted in Fig. 5.5 for two processors. The subarray written by each processor $p \in \{1, 2\}$ is defined by the starting position i_p and the length l_p.

In the computational kernel described above, the compiler cannot determine the entries of each subarray because, in general, the value of $iao(perm(ii))$ is known at run-time only. As a result, the array splitting and merging technique cannot be applied because the compiler cannot assure that consecutive array entries are modified in consecutive loop iterations. A parallelization strategy for this kernel could consist of executing the loop do_{ii} in a fully parallel manner ($DOALL$ loop), provided that the subarrays corresponding to the loop iterations do not overlap. This condition can be checked by inserting in the parallel code the following run-time test that we propose. Let $I = \{i_1, i_2, ..., i_n\}$ and $L = \{l_1, l_2, ..., l_n\}$ be, respectively, the sets of starting positions and lengths of the subarrays , $a_1, a_2, ..., a_n$, defined in each iteration of a loop. The subarrays $a_1, a_2, ..., a_n$ do not overlap if there is no subarray whose starting position corresponds to an entry of another subarray:

$$\nexists a_j (j \in \{1, ..., n\}) \ / \ i_k \leq i_j \leq i_k + l_k + 1 \ , \ \forall k \in \{1, \ldots, n\} \ , \ k \neq j \tag{5.1}$$

For this strategy to be implemented, the compiler needs to determine sets I and L. In the example code shown in Fig. 4.10(a), the starting positions are given by

$$I = \{iao(perm(ii)) \ \ ; \ ii = 1 \ldots nrow\}$$

Within our framework, this expression is identified when the SCC graph classification algorithm processes the structural chain $SCC_1^S(ko_{3,4}) \Rightarrow SCC_1^S(ko_2)$ (see the SCC graph of Fig. 4.10(c)). Note that $iao(perm(ii))$ is the right-hand side of the statement $ko_2 = iao(perm(ii_1))$ associated with the target component $SCC_1^S(ko_2)$. On the other hand, the lengths are

$$L = \{(ia(ii_1 + 1) - 1) - (ia(ii_1)) + 1 \ \ ; \ ii = 1 \ldots nrow\}$$

i.e. the number of iterations of the inner loop do_k for each do_{ii} loop iteration. This expression can be computed at the end of the classification of the component $SCC_1^S(k_2)$ associated with the innermost loop do_{k_2} (see Fig. 4.10). The compiler must be provided with support for symbolic computations.

Finally, the compiler must determine an appropriate point of the program for the run-time test to be inserted. A correct location is just before do_{ii} in the

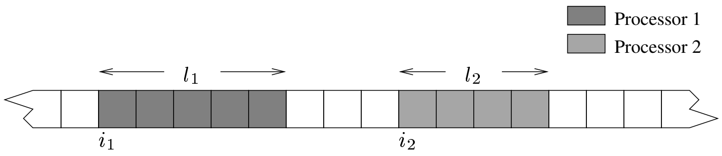


Figure 5.5: Write operations performed by two processors when a CWA is defined using a reinitialized induction variable of step one.

control flow graph of the program. However, as arrays *iao*, *perm* and *ia* are invariant with respect to do_{ii}, a more efficient location would be the point of the program where the results of the test can be reused for several execution of do_{ii}. Within our framework, this information can be extracted using the method proposed in Section 5.2 for inserting the inspector code.

5.5 Experimental Results

In this chapter, we have described a technique for the automatic parallelization of codes at the loop-level based on the information provided by our compiler framework. In order to evaluate the effectiveness of our approach, we have analyzed the routines of the *SparsKit-II* library with the Polaris parallelizing compiler. Polaris [11] is a source-to-source restructuring compiler that is organized in passes. The first passes perform source code transformations that are intended to make the code more amenable to dependence analysis. Next, dependence tests are run to discard the existence of loop-carried dependences that preclude the parallelization of a loop. Note that the reason the failure of Polaris is the failure of the dependence tests. In contrast, our approach fails when the loop class cannot be determined, i.e. when the SCC graph fails to recognize the computational kernels computed in the loop.

Table 5.2 summarizes the results of our experiments in terms of the number of loops that can be parallelized with our approach and with Polaris. For each nesting level, the number of loops with *regular*, *regular-irregular* and *irregular* computations is presented. The distinguishing characteristic is the presence of subscripted subscripts in the source code. In regular computations, the index expression of array references can usually be expressed as a linear or affine function of the loop index variables. In general, this property does not hold for irregular computations. A mixed situation is also possible: there is no subscripted subscript in the code, but there are if-endif constructs whose conditional expressions contain array references. Depending on the characteristics of the index expressions of the array references included in the body of the if-endif statement, the access pattern for those array references may be either regular or irregular. We will use the term *regular-irregular* computations to refer to these cases. Level-3 has been omitted because the numbers are zero for the two approaches. The last set of rows shows the total number of loops for all nesting levels. Comprehensive experimental results for each loop of *SparsKit-II* are presented in Appendix A.

According to the table, Polaris detects more parallelism than our approach (189 vs 152, respectively). However, the main difference comes from the analysis of regular loops (126 vs 84), which is the strength of the compiler technology included in Polaris. It should be noted that the detection of parallelism in regular codes was not our main objective bacause it is well covered in the literature (see Section 2.1). The rest of this section will focus on the comparison of both approaches with respect to the loops that contain irregular and regular-irregular computations. In this case, Polaris is less effective than our approach (63 vs 68). However, these total numbers are of little significance because they do not show the real differences between both compiler technologies. The explanation of such differences is the main topic of the discussion below.

The first consideration is related to the strategies that Polaris and our approach use for the analysis of the loop nests of *SparsKit-II*. In our prototype, inner loops are analyzed only if the computational kernels of the outer loops were not successfully recognized. In contrast, Polaris analyzes all the loops included in a loop nest. As a consequence, we have checked that from the 63 parallel loops of Polaris, there are 18 loops that were not analyzed by our prototype although their computations would have been recognized, and even the loop would have been detected as parallel.

Let us focus on parallel irregular loops. Polaris had success with 7 level-2 and 2 level-1 irregular loops where our prototype failed (these cases are marked with the symbol "<" in the tables of Appendix A). Table 5.3 lists these loops for each nesting level. The first column shows the loop within *SparsKit-II*. The notation consists of a triplet composed by the name of the module, the name of the routine and the name of the loop. The second column presents

	Our approach	Polaris
Level-1 parallel loops	118	156
Irregular loops	31	35
Regular-irregular loops	4	5
Regular loops	83	116
Level-2 parallel loops	33	33
Irregular loops	32	23
Regular-irregular loops	0	0
Regular loops	1	10
Level-4 parallel loops	1	0
Irregular loops	1	0
Regular-irregular loops	0	0
Regular loops	0	0
Total number of parallel loops	152	189
Irregular loops	64	58
Regular-irregular loops	4	5
Regular loops	84	126

Table 5.2: Effectiveness of our approach for the automatic detection of loop-level parallelism in the *SparsKit-II* library. Comparison with the Polaris parallelizing compiler.

the loop class that the prototype should determine. The last column shows the reason why the prototype failed to recognize the loop class: references to multidimensional arrays (M), statements of different classes in the SCCs (S), and existence of goto statements (G) or Fortran intrinsic procedure calls (I) in the loop body. A complete list of failure reasons is described in Appendix A. It should be noted that the irregularity of the computations is not necessarily represented in kernel classes (e.g. *non-cond/reduc/subs*). The irregular computations may arise, for instance, as subscripted subscripts in the right-hand side of the statements (a particular case are the *init*, *limit* and *step* expressions of the loop headers). Except for the loop *formats*:*csrmsr*:do_1, the loop classes that appear in the table can be recognized by the SCC and SCC graph classification schemes described in the previous chapters. However, the kernels were not recognized because of the limitations of the current version of our prototype. The primary failure reason was the presence of multidimensional array references in the loop body (M). Regarding *formats*:*csrmsr*:do_1, the detection failed because the SCC classification algorithm cannot classify

	Loop class	Failure reason
Level-1 parallel loops		
matvec:*amuxe*:do_{25}	*non-cond/reduc/lin*	M
formats:*csrell*:do_5	*non-cond/assig/lin*	M
Level-2 parallel loops		
matvec:*amuxe*:do_{10}	*non-cond/reduc/lin*	M
matvec:*amuxd*:do_{10}	*non-cond/reduc/subs*	MI
unary:*dscaldg*:do_{110}	*non-cond/assig/lin*	GI
formats:*csrdns*:do_4	*non-cond/assig/lin*	M
formats:*csrell*:do_6	*non-cond/assig/lin*	M
formats:*csrmsr*:do_1	*unknown*	S
formats:*cooell*:$do_{i(3)}$	*non-cond/assig/lin*	M

Table 5.3: Loops in *SparsKit-II* where our approach failed and Polaris had success.

SCCs that contain statements of different classes (S).

From the remaining cases, our approach beats Polaris in 1 level-4, 14 level-2 and 15 level-1 loops with irregular computations (symbol ">" in the tables of Appendix A). Table 5.4 shows these loops as well as the source code variables that may introduce loop-carried dependences according to the dependence tests of Polaris (*Loop-carried dependences*). The last column (*Kernel class*) presents the class of computational kernel that captures such dependences in our approach. Polaris fails in loops that contain irregular assignment computations, either conditional or non-conditional (*(non-)cond/assig/subs*). The loop *unary*:*amask*:do_{100} deserves special mention, because the irregular assignment is devoted to the computation of temporary values that are not useful when the execution of the loop finishes. In fact, they are part of a more complex masking mechanism that controls the loop iterations that are actually executed at run-time. The automatic parallelization of this loop in the scope of our compiler framework was studied in Section 5.4

On the other hand, Polaris cannot parallelize loops that contain conditional consecutively written array kernels (*cond/cwa*) because it cannot compute a closed form expression for the corresponding conditional induction variable (*cond/lin*). The level-2 loop *unary*:*rperm*:do_{100} contains an interesting case of *cond/cwa* kernel where the induction variable is reinitialized to a loop-

	Loop-carried dependences	Kernel class
Level-1 parallel loops		
blassm:*aplb*:do_{200}, do_{301}	iw	*non-cond/assig/subs*
blassm:*apmbt*:do_{200}, do_{301}	iw	*non-cond/assig/subs*
blassm:*aplsbt*:do_{200}, do_{301}	iw	*non-cond/assig/subs*
blassm:*apldia*:do_{1}	b	*cond/assig/subs*
unary:*clncsr*:do_{110}	indu	*non-cond/assig/subs*
unary:*rperm*:do_{50}	iao	*non-cond/assig/subs*
unary:*dmperm*:do_{101}	ao	*non-cond/assig/subs*
unary:*extbdg*:do_{13}	ao,jao,ko	*cond/cwa*
	bdiag	*cond/assig/subs*
unary:*blkfnd*:do_{1}	minlen	*scalar-minimum-w/loc*
unary:*csrkvstc*:do_{k}	iwk	*cond/assig/subs*
formats:*ssrcsr*:do_{110}	indu	*non-cond/assig/subs*
formats:*csrbsr*:do_{j}	iw	*non-cond/assig/subs*
Level-2 parallel loops		
blassm:*diamua*:do_{1}	b	*non-cond/assig/subs*
blassm:*amudia*:do_{1}	b	*non-cond/assig/subs*
unary:*submat*:do_{100}	ao,jao,klen	*cond/cwa*
unary:*csort*:do_{6}	iwork	*non-cond/assig/subs*
unary:*transp*:do_{3}	iwk	*non-cond/assig/subs*
unary:*amask*:do_{100}	c,jc,len	*cond/cwa*
	iw	*non-cond/assig/subs*
unary:*rperm*:do_{100}	ao,jao	*(non-)cond/cwa*
unary:*dperm1*:do_{900}	b,jb,ko	*(non-)cond/cwa*
unary:*dperm2*:do_{900}	b,jb,ko	*(non-)cond/cwa*
unary:*dscaldg*:do_{2}	a	*non-cond/reduc/subs*
unary:*xtrows*:do_{100}	ao,jao,ko	*(non-)cond/cwa*
formats:*csrmsr*:do_{500}	ao,jao,iptr	*cond/cwa*
formats:*csrssk*:do_{4}	asky	*cond/assig/subs*
formats:*csrsss*:do_{7}	al,jal,kl	*cond/cwa*
formats:*csrvbr*:$do_{i(4)}$	iwk	*non-cond/assig/subs*
formats:*vbrcsr*:do_{j}	ao,ja	*non-cond/cwa*
Level-4 parallel loops		
matvec:*vbrmv*:$do_{i(2)}$	k	*non-cond/lin*

Table 5.4: Loops in *SparsKit-II* where our approach had success and Polaris failed.

variant expression at the beginning of each iteration in the outer loop. These complex forms of induction variables cannot be handled by Polaris either. The parallelization of this loop was analyzed in detail in Section 5.4.1.

Finally, Polaris cannot parallelize the irregular reduction computational kernel (*non-cond/reduc/subs*) that appears in *unary:dscaldg:do*$_2$ because the reduction operator is not the sum but the product.

Regarding loops that contain regular-irregular computations, there is 1 level-1 loop in the routine *dscaldg* of module *unary* where Polaris beats our prototype. As shown in Section 3.7.3, this kernel is represented by a semantic SCC that cannot be successfully classified because it contains statements of different classes. As the kernel is not recognized, the automatic parallelization process fails. On the other hand, there is also 1 level-1 loop in the routine *csrkvstc* of *unary* where it is our prototype that beats Polaris. Polaris fails because the loop contains a *cond/cwa* kernel.

Conclusions

This thesis has covered two main subjects in the field of compiler technology for the analysis of source codes. On the one hand, we have proposed a scheme for the recognition of computational kernels at loop-level. This scheme is able to handle loop nests that contain both regular and irregular computations. Recognition is accomplished without considering the semantics of the code. On the other hand, we have described the way our recognition scheme can be used as a compiler framework that enables the automatic detection of parallelism in irregular codes.

The main contributions of the thesis can be summarized in the following items:

- A method that enables the recognition of a wide variety of basic computational kernels that contain regular and irregular computations. The basic kernels of a loop are recognized through the analysis of the strongly connected components that appear in the GSA graph of the program. This SCC classification algorithm recognizes kernels whose result is stored in scalar variables (e.g. induction variables, scalar reductions) as well as in array variables (e.g. irregular assignments, array recurrences). Furthermore, the technique handles kernels that are defined using if-endif constructs, for instance, minimum (maximum) and find-and-set operations whose result may be stored either in a scalar or an array variable.

- A technique for the recognition of the set of computational kernels that are calculated during the execution of a loop. This algorithm is based on the analysis of the SCC graph, which is an intermediate representation of the loop body that captures relevant information about the dependences between the basic kernels. No constraint is imposed on the regularity or the irregularity of the computations. We have shown that this SCC classification algorithm enables the recognition of kernels whose complexity cannot be handled by the method indicated in the previous item (e.g. scalar minimum (maximum) with location, array minimum (maximum) with location, consecutively written array).

- The application of our kernel recognition scheme to the automatic detection of coarse-grain loop-level parallelism in irregular codes. We have shown that the SCC and SCC graph classification algorithms can be used as a powerful information-gathering framework that provides the compiler with the information that the implementation of parallelizing transformations require. The main characteristics of our detection technique are:

  - Compile-time detection of regular and irregular kernels whose computations can be executed in parallel. Detection is accomplished even in loops that contain complex control constructs. In the scope of irregular codes, the information needed for the generation of parallel code may not be available at compile-time (e.g. irregular reductions, irregular assignments). In these situations, the compiler takes advantage of run-time support provided by available code transformations that enable the parallel execution preserving the sequential semantic of the loop (e.g. techniques based on the inspector-executor model).
  - Detection of a wide range of structural and semantic kernels in a unified manner. 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). The recognition of semantic kernels could enable the parallelization of a wider set of loops that would not be parallelized otherwise.

- A comparison of the effectiveness of our detection method against the Polaris parallelizing compiler. Encouraging experimental results have been obtained by detecting parallelism in irregular loops where the pattern-matching techniques available in Polaris fail. Some examples are irregular assignments and consecutively written arrays. We have shown that the recognition of these kernels allows the detection of parallelism in loops with complex irregular computations. For instance, we presented a code where the detection of an irregular assignment allows the compiler to parallelize a complex loop that contains a consecutively written array kernel.

We conclude with the presentation of some tasks that could be accomplished in order to complement the work developed in this dissertation, and that may be considered for future research work:

- Measure the effectiveness of our kernel recognition scheme for the analysis of other representative benchmark suites, mainly for irregular computations. We expect that this study will lead us to:
  - Measure the impact of the current limitations of the SCC classification algorithm for the analysis of full-scale applications. We intend to use this information as a guide for the inclusion of new features.
  - Identify new frequently used computational kernels.
  - Study the possibilities of parallelizing these new kernels, and propose new efficient parallelizing transformations.
- Extend the compiler framework in order to perform interprocedural and alias analysis.
- Address the analysis of source codes written in other programming languages, for instance, C, C++ or Java. We intend to integrate our prototype in the GNU GCC compiler as the first step to accomplish this goal.
- Study the advantages and drawbacks of combining our compiler framework (and the subsequent automatic parallelization stage) with the existing techniques that address the detection of parallelism in regular codes efficiently. During the construction of the framework, the SCC graph classification algorithm analyzes all the dependences that will appear during the execution of a loop. The dependences associated with irregular computations could be analyzed as described in this thesis. Regarding the dependences related to regular access patterns, the compiler could

be used for the construction of equation systems that would later be analyzed with classical dependence tests. Furthermore, the efficiency of the tests could be improved by eliminating, for instance, loop-invariant array references that would cause the test to fail.

- Study the applicability of the framework in the scope of optimizing techniques for intensive I/O applications. From the point of view of the compiler, it would be necessary to determine the kind of data flow analysis needed by these techniques and address their integration within the classification schemes that construct our compiler framework. For instance, this extension would require the analysis of loops that contain I/O statements and the characterization of the access patterns of the I/O arrays.

- Provide support for automatic program comprehension techniques in the scope of irregular codes. In this thesis, we addressed the recognition of computational kernels that do not supply information about the semantics of the program. However, we could adapt the transfer functions of our classification schemes in order to check additional constraints that cope with semantics. This approach would enable more aggressive code transformations.

- Measure the effectiveness of induction variable substitution techniques for the computation of closed form expressions in loops with irregular computations. We will try to improve current technology by including support for symbolic computations within our framework.

Appendix A

Loop-Level Detection Results vs Polaris for *SparsKit-II*

In Chapters 3 and 4 we presented experimental results that show the effectiveness of the SCC classification algorithm and the SCC graph classification algorithm, respectively. As a benchmark suite we used the *SparsKit-II* library because it consists of a set of simple routines that contain a wide range of loops with irregular computations. In this appendix, we include detailed information about the effectiveness of Polaris and the automatic parallelization approach we outlined in Chapter 5.

Tables A.1, A.2, A.3 and A.4 compare both approaches for each loop of the modules *matvec*, *blassm*, *unary* and *formats* of *SparsKit-II*. The first column, R/I, shows whether the loop contains regular (R), irregular (I) or regular-irregular (R/I) computations. The criterium to distinguish these types of computations was explained in Section 5.5. The second column shows the identifiers of the loops within the corresponding module. They are indented according to their nesting level. The identifier consists of the routine name and the loop name. The loop name is denoted as do_x, x being a symbol that identifies the loop within the routine (e.g. the loop index variable possibly followed by a sequence number within brackets, the number associated with Fortran77 loop labels). The third and the sixth columns, P/S indicate whether the loop computations are detected as parallelizable (P) or serial (S) by Polaris and by our approach, respectively. A symbol "$-$" means that the loop was not analyzed by the corresponding compiler. The fifth column compares the effectiveness of Polaris and our detection technique: same effectiveness (=), Polaris is effective but our approach is not (<), and our approach is effective but Polaris not (>). Finally, the remaining columns present information about the failure reasons of both approaches. The fourth column, *Dependences*, presents the source code variables for which the dependence tests implemented in Polaris failed. The message printed by Polaris says `''Variable (may) have loop-carried dependences''`. Regarding our approach, this information is shown in the seventh and eighth columns of the table. The seventh, *unk*, shows the existence of *unknown* SCCs in the loop body. In this case, the reasons for the unsuccessful classification are indicated as follows: cardinality greater than one (C), references to multidimensional arrays (M), statements of different classes (S), and operators whose transfer function has not been defined (O). The eighth column shows whether the loop body is not amenable for classification due to the existence of goto statements (G), procedure call statements (C) or Fortran intrinsic procedure calls (I). The routines that do not contain DO loops have been omitted from the tables.

In some rows of the tables there is a symbol "$-$" in the columns P/S of both the Polaris compiler and our prototype. In most cases, it is due to the fact that the loop is a while loop that is implemented with GOTOs. Polaris recognizes while loops, but it does not analyze the control flow graph for finding loops implemented with GOTOs. The current version of our prototype does not handle any of these loops.

The tables contain blank entries in the seventh and eighth columns. This means that, although the loop is amenable for classification and all the SCCs were successfully classified, the SCC graph classification algorithm cannot derive a known class for the loop.

Table A.1: Loop-level detection results vs Polaris for the module *matvec* of *SparsKit-II*.

Loop		*Polaris compiler*			*Our approach*		
R/I	*Identifier*	*P/S*	*Dependences*		*P/S*	*unk*	*Jump*
I	amux:do_{100}	P		=	P		
I	amux:do_{99}	P			$-$		
R	amuxms:do_{10}	P		=	P		
I	amuxms:do_{100}	P		=	P		

continued on next page

continued from previous page

Loop		*Polaris compiler*			*Our approach*		
R/I	*Identifier*	*P/S*	*Dependences*		*P/S*	*unk*	*Jump*
I	amuxms:do_{99}	P			–		
R	atmux:do_1	P		$=$	P		
I	atmux:do_{100}	P		$=$	P		
I	atmux:do_{99}	P			–		
R	atmuxr:do_1	P		$=$	P		
I	atmuxr:do_{100}	P		$=$	P		
I	atmuxr:do_{99}	P			–		
R	amuxe:do_1	P		$=$	P		
I	amuxe:do_{10}	P		$<$	S	M	
I	amuxe:do_{25}	P		$<$	S	M	
R	amuxd:do_1	P		$=$	P		
I	amuxd:do_{10}	P		$<$	S	M	I
R	amuxd:do_9	P		$<$	S	M	
R	amuxj:do_1	P		$=$	P		
I	amuxj:do_{70}	P		$=$	P		
I	amuxj:do_{60}	P			–		
R	vbrmv:$do_{i(1)}$	P		$=$	P		
I	vbrmv:$do_{i(2)}$	S	k	$>$	P		
I	vbrmv:do_j	S	k		–		
R	vbrmv:do_{jj}	S	k		–		
R	vbrmv:do_{ii}	P			–		
I	lsol:do_{150}	S	x	$=$	S	C	
I	lsol:do_{100}	P		$=$	P		
I	ldsol:do_{150}	S	x	$=$	S	C	
I	lsol:do_{100}	P		$=$	P		
R	lsolc:do_{140}	P		$=$	P		
I	lsolc:do_{150}	S	x	$=$	S	C	
I	lsolc:do_{100}	P		$=$	P		
R	ldsolc:do_{140}	P		$=$	P		
I	ldsolc:do_{150}	S	x	$=$	S	C	
I	ldsolc:do_{100}	P		$=$	P		
I	ldsoll:do_{150}	S	x	$=$	S	C	
I	ldsoll:do_{100}	S	x	$=$	S	C	
I	ldsoll:do_{130}	P		$=$	P		
I	usol:do_{150}	S	x	$=$	S	C	
I	usol:do_{100}	P		$=$	P		
I	udsol:do_{150}	S	x	$=$	S	C	

continued on next page

continued from previous page

Loop		*Polaris compiler*			*Our approach*		
R/I	*Identifier*	*P/S*	*Dependences*		*P/S*	*unk*	*Jump*
I	udsol:do_{100}	*P*		=	*P*		
R	usolc:do_{140}	*P*		=	*P*		
I	usolc:do_{150}	*S*	x	=	*S*	*C*	
I	usolc:do_{100}	*P*		=	*P*		
R	udsolc:do_{140}	*P*		=	*P*		
I	udsolc:do_{150}	*S*	x	=	*S*	*C*	
I	udsolc:do_{100}	*P*		=	*P*		

Table A.2: Loop-level detection results vs Polaris for the module *blassm* of *SparsKit-II*.

Loop		*Polaris compiler*			*Our approach*		
R/I	*Identifier*	*P/S*	*Dependences*		*P/S*	*unk*	*Jump*
R	amub:do_{1}	*P*		=	*P*		
I	amub:do_{500}	*S*	c,ic,iw,jc,len,scal	=	*S*		
I	amub:do_{200}	*S*	c,iw,jc,len,scal	=	*S*		
I	amub:do_{100}	*S*	c,iw,jc,len	=	*S*	*S*	
I	amub:do_{201}	*S*	iw	>	*P*		
R	aplb:do_{1}	*P*		=	*P*		
I	aplb:do_{500}	*S*	c,ic,iw,jc,len	=	*S*		
I	aplb:do_{200}	*S*	iw	>	*P*		
I	aplb:do_{300}	*S*	c,iw,jc,len	=	*S*	*S*	
I	aplb:do_{301}	*S*	iw	>	*P*		
R	aplb1:do_{6}	*S*	c,jc	=	*S*		*G*
R/I	aplb1:do_{5}	–			–		
R	aplsb:do_{6}	*S*	c,j1,j2,jc,ka,kb,kc	=	*S*		*G*
R/I	aplsb:do_{5}	–			–		
R	aplsb1:do_{6}	*S*	c,j1,j2,jc,ka,kb,kc	=	*S*		*G*
R/I	aplsb1:do_{5}	–			–		
R	apmbt:do_{1}	*P*		=	*P*		
R	apmbt:do_{2}	*P*		<	*S*	*O*	
I	apmbt:do_{500}	*S*	c,ic,iw,jc,len	=	*S*		
I	apmbt:do_{200}	*S*	iw	>	*P*		
I	apmbt:do_{300}	*S*	c,ic,jc,len	=	*S*	*S*	
I	apmbt:do_{301}	*S*	iw	>	*P*		
R	apmbt:do_{501}	*P*		=	*P*		

continued on next page

continued from previous page

Loop		*Polaris compiler*			*Our approach*		
R/I	*Identifier*	*P/S*	*Dependences*		*P/S*	*unk*	*Jump*
R	aplsbt:do_1	P		=	P		
R	aplsbt:do_2	P		=	P		
I	aplsbt:do_{500}	S	c,ic,iw,jc,len	=	S		
I	aplsbt:do_{200}	S	iw	>	P		
I	aplsbt:do_{300}	S	c,ic,jc,len	=	S	S	
I	aplsbt:do_{301}	S	iw	>	P		
R	aplsbt:do_{501}	P		=	P		
I	diamua:do_1	S	b	>	P		
R	diamua:do_2	P			–		
R	diamua:do_3	P		=	P		
R	diamua:do_{31}	P		=	P		
I	amudia:do_1	S	b	>	P		
I	amudia:do_2	P			–		
R	amudia:do_3	P		=	P		
R	amudia:do_{31}	P		=	P		
I	aplsca:do_1	P		=	P		
I	aplsca:do_5	S	a,ia,ja,ko	=	S		
R/I	aplsca:do_4	S	a,ja,ko,test	=	S		
R	apldia:do_2	P		=	P		
R	apldia:do_3	P		=	P		
I	apldia:do_1	S	b	>	P		
I	apldia:do_5	S	b,ib,jb,ko	=	S		
R/I	apldia:do_4	S	b,jb,ko,test	=	S		

Table A.3: Loop-level detection results vs Polaris for the module *unary* of *SparsKit-II*.

Loop		*Polaris compiler*			*Our approach*		
R/I	*Identifier*	*P/S*	*Dependences*		*P/S*	*unk*	*Jump*
I	submat:do_{100}	S	ao,jao,klen	>	P		
R	submat:do_{60}	S	ao,jao,klen		–		
I	filter:do_{10}	S	b,index,jb	=	S		G
R	filter:do_{22}	P		=	P		
R	filter:do_{23}	S	norm	=	S		I
R	filter:do_{30}	S	b,index,jb	=	S		I
I	filterm:do_{10}	S	b,index,jb	=	S		G
R	filterm:do_{22}	P		=	P		

continued on next page

continued from previous page

Loop		Polaris compiler			Our approach		
R/I	*Identifier*	*P/S*	*Dependences*		*P/S*	*unk*	*Jump*
R	filterm:do_{23}	S	norm	=	S		I
R	filterm:do_{30}	S	b,index,jb	=	S		I
R	csort:do_1	P		=	P		
I	csort:do_3	P		=	P		
I	csort:do_2	P			–		
R	csort:do_4	S	iwork	=	S		
I	csort:do_5	S	iwork	=	S	C	
I	csort:do_{51}	S	iwork	=	S	C	
I	csort:do_6	S	iwork	>	P		
R	csort:do_{61}	P			–		
I	csort:do_7	S	ia,iwork	=	S	C	
R	csort:do_8	S	ia	=	S		
R	clncsr:do_{90}	P		=	P		
R	clncsr:do_{120}	S	a,indu,ja,k	=	S		G
I	clncsr:do_{100}	–		=	–		
I	clncsr:do_{110}	S	indu	>	P		
I	clncsr:do_{140}	S	a,j,ja,kfirst, klast,tmp	=	S		G
I	clncsr:do_{130}	–		=	–		
I	clncsr:do_{190}	S	a,ja	=	S	C	
R/I	clncsr:do_{160}	S	a,ja	=	S	C	
R/I	clncsr:do_{150}	S	a,ja	=	S	C	
R/I	clncsr:do_{180}	S	a,ja	=	S	C	
R/I	clncsr:do_{170}	S	a,ja	=	S	C	
R	copmat:do_{100}	P		=	P		
R	copmat:do_{200}	P		=	P		
R	copmat:do_{201}	P		=	P		
R	msrcop:do_{100}	P		=	P		
R	msrcop:do_{200}	P		=	P		
R	msrcop:do_{201}	P		=	P		
R	msrcop:do_{202}	P		=	P		
R	getelm:do_5	S	iadd	=	S		G
R	getelm:do_{10}	–			–		
R	getdia:do_1	P		=	P		
I	getdia:do_6	S	ko	=	S		G
I	getdia:do_{51}	S	diag,idiag,k	=	S		G
I	getdia:do_7	S	a,ia,ja,ko	=	S		

continued on next page

continued from previous page

Loop		Polaris compiler			Our approach		
R/I	*Identifier*	*P/S*	*Dependences*		*P/S*	*unk*	*Jump*
R	getdia:do_{71}	S	a,ja,ko	=	S		
R	transp:do_{1}	S	jcol	=	S		I
I	transp:do_{3}	S	iwk	>	P		
R	trasnp:do_{2}	P			–		
R	transp:do_{35}	P		=	P		
I	transp:do_{4}	P		=	P		
R	transp:do_{44}	S	ia	=	S		
R	transp:do_{80}	S	ia	=	S		
I	getl:do_{7}	S	ao,jao,ko	=	S		G
R/I	getl:do_{71}	S	ao,jao,ko	=	S		G
I	getu:do_{7}	S	ao,jao,ko	=	S		G
R/I	getu:do_{71}	S	ao,jao,ko	=	S		G
R	levels:do_{10}	P		=	P		
R	levels:do_{20}	S	levnum,nlev	=	S		I
R	levels:do_{15}	S	levi	=	S		I
R	levels:do_{21}	P		=	P		
I	levels:do_{22}	P		=	P		
R	levels:do_{23}	S	ilev	=	S		
I	levels:do_{30}	S	ilev,lev	=	S		
R	levels:do_{35}	S	ilev	=	S		
R	amask:do_{1}	P		=	P		
I	amask:do_{100}	S	c,iw,jc,len	>	P		
I	amask:do_{2}	S	iw		–		
R/I	amask:do_{200}	S	c,jc,len		–		
I	amask:do_{3}	S	iw		–		
I	rperm:do_{50}	S	iao	>	P		
R	rperm:do_{51}	S	iao	=	S		
I	rperm:do_{100}	S	ao,jao	>	P		
R	rperm:do_{60}	P			–		
I	cperm:do_{100}	P		=	P		
R	cperm:do_{1}	P		=	P		
R	cperm:do_{2}	P		=	P		
I	dperm1:do_{900}	S	b,jb,ko	>	P		
R	dperm1:do_{800}	P			–		
I	dperm2:do_{900}	S	b,jb,ko	>	P		
I	dperm2:do_{800}	P			–		
I	dmperm:do_{101}	S	ao	>	P		

continued on next page

continued from previous page

Loop		Polaris compiler			Our approach		
R/I	*Identifier*	*P/S*	*Dependences*		*P/S*	*unk*	*Jump*
R/I	dvperm:do_6	–			S		
R	dvperm:do_{200}	P		<	S	O	
R/I	ivperm:do_6	–			S		
R	ivperm:do_{200}	P		<	S	O	
I	retmx:do_{11}	S	k2,t2	=	S		I
R	retmex:do_{101}	S	t	=	S		I
R	diapos:do_1	P		=	P		
R	diapos:do_6	P		=	P		
R/I	diapos:do_{51}	P			–		
I	dscaldg:do_{110}	P		<	S		GI
R	dscaldg:do_{111}	P		=	P		
R/I	dscaldg:do_1	P		<	S	S	
I	dscaldg:do_2	S	a	>	P		
R	dscaldg:do_{21}	P			–		
R	extbdg:do_1	P		=	P		
I	extbdg:do_{11}	S	ao,bdiag,iao,jao, kb,ko	=	S		I
I	extbdg:do_{12}	S	ao,bdiag,jao,ko	=	S		
I	extbdg:do_{13}	S	ao,bdiag,jao,ko	>	P		
I	getbwd:do_3	S	ml,mu	=	S		I
R	getbwd:do_{31}	S	ml,mu	=	S		I
R	blkfnd:do_1	S	minlen	>	P		
R	blkfnd:do_{99}	S	ia,imsg,ja,nrow	=	S		G
R	blkfnd:do_{10}	P		<	S		G
R/I	blkchk:do_{20}	S	irow	=	S		G
R/I	blkchk:do_6	S	irow,j2	=	S		G
R/I	blkchk:do_7	S	j2	=	S		G
R	blkchk:do_5	S	j2	=	S		G
R	infdia:do_1	P		=	P		
I	infdia:do_3	P		=	P		
I	infdia:do_2	P			–		
R	infdia:do_{41}	P		=	P		
R	amubdg:do_1	P		=	P		
R	amubdg:do_2	P		=	P		
I	amubdg:do_7	S	iw	=	S	C	
I	amubdg:do_6	S	iw,last	=	S		
I	amubdg:do_5	S	iw,last	=	S		

continued on next page

continued from previous page

Loop		*Polaris compiler*			*Our approach*		
R/I	*Identifier*	*P/S*	*Dependences*		*P/S*	*unk*	*Jump*
R	amubdg:do_{61}	S	iw,last	$=$	S		
R	amubdg:do_{8}	P		$=$	P		
R	aplbdg:do_{1}	P		$=$	P		
R	aplbdg:do_{2}	P		$=$	P		
I	aplbdg:do_{7}	S	iw	$=$	S	C	
I	aplbdg:do_{5}	S	iw,last	$=$	S		
I	aplbdg:do_{6}	S	iw,last	$=$	S		
R	aplbdg:do_{61}	S	iw,last	$=$	S		
R	aplbdg:do_{8}	P		$=$	P		
R	rnrms:do_{1}	P		$<$	S		I
R	rnrms:do_{2}	S	scal	$=$	S		I
R	rnrms:do_{3}	P		$<$	S		I
R	rnrms:do_{4}	P		$<$	S	O	
R	cnrms:do_{10}	P		$=$	P		
I	cnrms:do_{1}	S	diag	$=$	S		I
I	cnrms:do_{2}	S	diag	$=$	S		I
R	cnrms:do_{3}	P		$<$	S		I
R/I	roscal:do_{1}	P		$=$	P		
R/I	coscal:do_{1}	P		$=$	P		
I	addblk:do_{10}	S	c,jc,ka,kamax, kb,kbmax,kc	$=$	S		G
I	addblk:do_{20}	$-$			S		G
R/I	get1up:do_{5}	S	k	$=$	S		G
I	xtrows:do_{100}	S	ao,jao,ko	$>$	P		
R	xtrows:do_{60}	P			$-$		
I	csrkvstr:do_{i}	S	jo,kvstr,nr	$=$	S		G
R/I	csrkvstr:do_{j}	S	j,kvstr,nr	$=$	S		G
I	csrkvstc:$do_{i(1)}$	S	iwk,ncol	$=$	S		I
I	csrkvstc:do_{k}	S	iwk	$>$	P		
R/I	csrkvstc:$do_{i(2)}$	S	kvstc,nc	$>$	P		
R/I	kvstmerge:do_{200}	$-$			S		G

Table A.4: Loop-level detection results vs Polaris for the module *formats* of *SparsKit-II*.

Loop		*Polaris compiler*			*Our approach*		
R/I	*Identifier*	*P/S*	*Dependences*		*P/S*	*unk*	*Jump*
R	csrdns:do_1	P		$<$	S	M	
R	csrdns:do_2	P		$<$	S	M	
I	csrdns:do_4	P		$<$	S	M	
I	csrdns:do_3	S	dns	$=$	S	M	
R	dnscsr:do_4	S	a,ja,next	$=$	S	M	
R	dnscsr:do_3	S	a,ja,next	$=$	S	M	
R	coocsr:do_1	P		$=$	P		
I	coocsr:do_2	P		$=$	P		
R	coocsr:do_3	S	k	$=$	S	C	
I	coocsr:do_4	S	ao,iao,jao	$=$	S		
R	coocsr:do_5	S	iao	$=$	S		
R	coicsr:do_{35}	P		$=$	P		
I	coicsr:do_4	P		$=$	P		
R	coicsr:do_{44}	S	iwk	$=$	S		
I	coicsr:do_5	$-$			$-$		
I	coicsr:do_6	$-$			$-$		
R	coicsr:do_{65}	$-$			$-$		
R	coicsr:do_{80}	P		$=$	P		
R	csrcoo:do_{10}	P		$=$	P		
R	csrcoo:do_{11}	P		$=$	P		
I	csrcoo:do_{13}	P		$=$	P		
R	csrcoo:do_{12}	P			$-$		
I	csrssr:do_7	S	ao,jao,ko	$=$	S		G
R	csrssr:do_{71}	S	ao,jao,ko	$=$	S		G
R	ssrcsr:do_{10}	P		$=$	P		
I	ssrcsr:do_{30}	P		$=$	P		
I	ssrcsr:do_{20}	P			$-$		
R	ssrcsr:do_{40}	S	iwk	$=$	S	C	
I	ssrcsr:do_{60}	S	kosav	$=$	S	C	
R	ssrcsr:do_{50}	P		$=$	P		
I	ssrcsr:do_{80}	S	ao,iwk,jao	$=$	S	C	
I	ssrcsr:do_{70}	S	ao,iwk,jao	$=$	S	C	
R	ssrcsr:do_{90}	P		$=$	P		
I	ssrcsr:do_{120}	S	ao,indu,jao,k	$=$	S		G
I	ssrcsr:do_{100}	$-$			$-$		

continued on next page

continued from previous page

Loop		Polaris compiler			Our approach		
R/I	*Identifier*	*P/S*	*Dependences*		*P/S*	*unk*	*Jump*
I	ssrcsr:do_{110}	S	indu	$>$	P		
I	ssrcsr:do_{140}	S	ao,j,jao,kfirst, klast,tmp	$=$	S		G
I	ssrcsr:do_{130}	–			–		
I	ssrcsr:do_{190}	S	ao,jao	$=$	S	C	
R/I	ssrcsr:do_{160}	S	ao,jao	$=$	S	C	
R/I	ssrcsr:do_{150}	S	ao,jao	$=$	S	C	
R/I	ssrcsr:do_{180}	S	ao,jao	$=$	S	C	
R/I	ssrcsr:do_{170}	S	ao,jao	$=$	S	C	
R	xssrcsr:do_{1}	P		$=$	P		
I	xssrcsr:do_{3}	P		$=$	P		
I	xssrcsr:do_{2}	P			–		
R	xssrcsr:do_{4}	S	indu	$=$	S		
I	xssrcsr:do_{6}	S	ao,jao,kosav	$=$	S		
R	xssrcsr:do_{5}	P		$=$	P		
I	xssrcsr:do_{8}	S	ao,indu,jao,ko	$=$	S		
I	xssrcsr:do_{9}	S	ao,indu,jao,k	$=$	S		G
R	csrell:do_{3}	S	ndiag	$=$	S		I
R	csrell:do_{4}	P		$<$	S	M	
R	csrell:do_{41}	P		$<$	S	M	
I	csrell:do_{6}	P		$<$	S	M	
I	csrell:do_{5}	P		$<$	S	M	
R/I	ellcsr:do_{6}	S	a,ja,kpos	$=$	S	M	
R/I	ellcsr:do_{5}	S	a,ja,kpos	$=$	S	M	
I	csrmsr:do_{1}	P		$<$	S	S	
R/I	csrmsr:do_{2}	P		$=$	P		
I	csrmsr:do_{500}	S	ao,iptr,jao	$>$	P		
I	csrmsr:do_{100}	S	ao,iptr,jao		–		
R	csrmsr:do_{600}	S	jao	$=$	S		
R	msrcsr:do_{1}	P		$=$	P		
I	msrcsr:do_{500}	S	ao,iptr,jao	$=$	S	S	
R/I	msrcsr:do_{100}	S	added,ao,idiag, iptr,jao	$=$	S		
R	csrcsc2:do_{1}	P		$=$	P		
I	csrcsc2:do_{3}	P		$=$	P		
I	csrcsc2:do_{2}	P			–		
R	csrcsc2:do_{4}	S	iao	$=$	S		

continued on next page

continued from previous page

Loop		*Polaris compiler*			*Our approach*		
R/I	*Identifier*	*P/S*	*Dependences*		*P/S*	*unk*	*Jump*
I	csrcsc2:do_6	S	ao,iao,jao	=	S		
I	csrcsc2:do_{62}	S	ao,iao,jao	=	S		
R	csrcsc2:do_7	S	iao	=	S		
I	csrlnk:do_{100}	S	ia,link	=	S		
R	csrlnk:do_{99}	P		=	P		
R	lnkcsr:do_{100}	S	ao,ipos,jao, next	=	S		G
R	lnkcsr:do_{99}	–			–		
R	csrdia:do_{41}	S	kmask	=	S		G
R	csrdia:do_{55}	P		<	S	M	
R	csrdia:do_{54}	P		<	S	M	
I	csrdia:do_6	S	ao,jao,ko	=	S	M	G
R	csrdia:do_{51}	S	ao,diag,jao,ko	=	S	M	G
R	csrdia:do_{52}	S		=	S	M	G
R	csrdia:do_7	P		=	P		
R	diacsr:do_{80}	S	a,ja,ko	=	S	M	G
R	diacsr:do_{70}	S	a,ja,ko	=	S	M	G
I	bsrcsr:do_2	S	ao,jao,krow	=	S	M	
R	bsrcsr:do_{23}	S	ao,jao,krow	=	S	M	
R	bsrcsr:do_{21}	P		<	S	M	
R	bsrcsr:do_{22}	P		<	S	M	
R	csrbsr:do_j	P		=	P		
I	csrbsr:do_{ii}	–			–		
I	csrbsr:do_{while}	–			–		
I	csrbsr:do_k	S	ao,iw,jao,ko	=	S	M	
R	csrbsr:do_i	P		<	S	M	
I	csrbsr:do_j	S	iw	>	P		
R	csrbnd:do_{15}	P		<	S	M	
R	csrbnd:do_{10}	P		<	S	M	
I	csrbnd:do_{30}	S	abd	=	S	M	
I	csrbnd:do_{20}	S	abd	=	S	M	
R	bndcsr:do_{30}	S	a,jo,ja,ko	=	S	M	G
R	bndcsr:do_{20}	S	a,i,j,ja,ko	=	S	M	G
I	csrssk:do_3	S	isky	=	S		I
R	csrssk:do_{31}	S	ml	=	S		I
R	csrssk:do_1	P		=	P		
I	csrssk:do_4	S	asky	>	P		

continued on next page

continued from previous page

Loop		*Polaris compiler*			*Our approach*		
R/I	*Identifier*	*P/S*	*Dependences*		*P/S*	*unk*	*Jump*
I	csrssk:do_{41}	S	asky		–		
R	csrssk:do_{50}	P		=	P		
R	csrssk:do_{60}	S	isky	=	S		
R	sskssr:do_{50}	S	ao,jao,kend, next	=	S		G
R	sskssr:do_{31}	S	ao,jao,next	=	S		G
R	csrjad:do_{10}	S	idiag,ilo	=	S		I
R	csrjad:do_{20}	P		=	P		
I	csrjad:do_{40}	P		=	P		
R	csrjad:do_{30}	P			–		
I	csrjad:do_{60}	S	ao,jao,ko,k1	=	S		
I	csrjad:do_{50}	P		=	P		
R	jadcsr:do_{137}	P		=	P		
I	jadcsr:do_{140}	P		=	P		
I	jadcsr:do_{138}	P			–		
R	jadcsr:do_{141}	S	kpos	=	S		
I	jadcsr:do_{200}	S	ao,iao,jao	=	S		
I	jadcsr:do_{160}	S	ao,iao,jao	=	S		
R	jadcsr:do_5	S	iao	=	S		
R	dcsort:do_{10}	P		=	P		
I	dcsort:do_{20}	P		=	P		
R	dcsort:do_{30}	S	icnt	=	S		
I	dcsort:do_{40}	S	icnt,index	=	S		
R	cooell:$do_{i(1)}$	P		<	S	M	
R	cooell:$do_{k(1)}$	P		<	S	M	
R	cooell:$do_{i(2)}$	S	nc	=	S	M	
I	cooell:$do_{k(2)}$	S	ao,jao,nc	=	S	CM	
I	cooell:$do_{i(3)}$	P		<	S	M	
R	cooell:do_j	P		<	S	M	
R	xcooell:do_4	P		<	S	M	
R	xcooell:$do_{4,1}$	P		<	S	M	
R/I	xcooell:do_{10}	S	ncmax	=	S	M	
R/I	xcooell:do_{30}	S	ac,jac,k	=	S	M	
R	xcooell:do_{45}	P		<	S	M	
R	xcooell:do_{44}	P		<	S	M	
R	xcooell:do_{55}	P		<	S	M	
R	xcooell:do_{54}	P		<	S	M	

continued on next page

continued from previous page

Loop		*Polaris compiler*			*Our approach*		
R/I	*Identifier*	*P/S*	*Dependences*		*P/S*	*unk*	*Jump*
R	csruss:do_1	*P*		=	*P*		
I	csruss:do_3	*P*		=	*P*		
I	csruss:do_2	*P*			–		
R	csruss:do_4	*S*	ial,iau	=	*S*		
I	csruss:do_7	*S*	al,au,iau,jal,	=	*S*	*C*	
			jau,kl				
I	csruss:do_{71}	*S*	al,au,iau,jal,	=	*S*	*C*	
			jau,kl				
R	csruss:do_8	*S*	iau	=	*S*		
R	usscsr:do_1	*P*		=	*P*		
I	usscsr:do_3	*P*		=	*P*		
I	usscsr:do_2	*P*			–		
R	usscsr:do_4	*S*	ia	=	*S*		
I	usscsr:do_6	*S*	a,ja	=	*S*		
R	usscsr:do_5	*P*		=	*P*		
I	usscsr:do_8	*S*	a,ia,ja	=	*S*		
I	usscsr:do_7	*S*	a,ia,ja	=	*S*		
R	usscsr:do_9	*S*	ia	=	*S*		
I	csrsss:do_7	*S*	al,jal,kl	>	*P*		
I	csrsss:do_{71}	*S*	al,jal,kl		–		
I	csrsss:do_8	*S*	au,ial	=	*S*	*C*	
I	csrsss:do_{81}	*S*	au,ial	=	*S*	*C*	
R	csrsss:do_9	*S*	ial	=	*S*		
R	ssscsr:do_1	*P*		=	*P*		
I	ssscsr:do_3	*P*		=	*P*		
I	ssscsr:do_2	*P*			–		
R	ssscsr:do_4	*S*	ia	=	*S*		
I	ssscsr:do_6	*S*	a,ja	=	*S*		
R	ssscsr:do_5	*P*		=	*P*		
I	ssscsr:do_8	*S*	a,ia,ja	=	*S*		
I	ssscsr:do_7	S	a,ia,ja	=	*S*		
R	ssscsr:do_9	*S*	ia	=	*S*		
I	csrvbr:$do_{i(1)}$	*S*	ncol	=	*S*		*I*
R	csrvbr:$do_{i(2)}$	*P*		=	*P*		
R	csrvbr:$do_{i(3)}$	*P*		=	*P*		
I	csrvbr:$do_{i(4)}$	*S*	iwk	>	*P*		
R	csrvbr:do_j	*P*			–		

continued on next page

continued from previous page

Loop		*Polaris compiler*			*Our approach*		
R/I	*Identifier*	*P/S*	*Dependences*		*P/S*	*unk*	*Jump*
I	csrvbr:$do_{i(5)}$	S	a,b,bo,jb,ko,kb	=	S		
I	csrvbr:do_{jj}	S	bo,j,jb,jo,kb	=	S		
I	csrvbr:do_{ii}	S	ao,b	=	S		
R	csrvbr:do_{jj}	S	b	=	S		
R	csrvbr:$do_{i(6)}$	P		=	P		
I	csrvbr:$do_{i(7)}$	S	iwk,jb,ko,kb	=	S		
I	csrvbr:do_{jj}	S	iwk	=	S		
R/I	csrvbr:do_{j}	S	jb,ko,kb	=	S		
I	csrvbr:$do_{i(8)}$	S	ao,b	=	S		
I	csrvbr:do_{ii}	S	ao,b	=	S		
I	csrvbr:do_{j}	S	ao,b,bo	=	S		
R/I	csrvbr:do_{jj}	S	ao,b	=	S		
I	vbrcsr:do_{i}	S	a,ao,ia,ja	=	S		
I	vbrcsr:do_{j}	S	ao,ja	>	P		
R	vbrcsr:do_{jj}	P			–		
R	vbrcsr:do_{ii}	S	ao,ja	=	S		
R	vbrcsr:do_{j}	P		<	S		
I	vbrcsr:do_{ii}	S	a,ao	=	S		
R	vbrcsr:do_{jj}	P		<	S		
I	csorted:do_{i}	S		=	S		G
R	csorted:do_{j}	S		=	S		G

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures.* Academic Press, 2002.

[3] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *15th Annual ACM Symposium on Principles of Programming Languages, POPL 1988*, pages 1–11, San Diego, CA, January 1988.

[4] Manuel Arenaz, Juan Tourino, and Ramón 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, August 2001. To be published in Lecture Notes in Computer Science.

[5] Manuel Arenaz, Juan Tourino, and Ramón Doallo. Irregular assignment computations on cc-NUMA multiprocessors. In *4th International Symposium on High Performance Computing, ISHPC-IV*, pages 361–369, Kansai Science City, Japan, May 2002.

[6] Manuel Arenaz, Juan Tourino, and Ramón 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, March 2002.

[7] Manuel Arenaz, Juan Tourino, and Ramón Doallo. Towards detection of coarse-grain loop-level parallelism in irregular computations. In *8th In-*

ternational European Conference on Parallel Processing, Euro-Par 2002, pages 289–298, Paderborn, Germany, August 2002.

[8] Manuel Arenaz, Juan Tourino, Ramón Doallo, and Carlos Vázquez. Efficient parallel numerical solver for the elastohydrodynamic Reynolds-Hertz problem. *Parallel Computing*, 27(13):1743–1765, December 2001.

[9] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[10] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* SIAM, 1994.

[11] William Blume, Ramón Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David A. Padua, Yunheung Paek, William M. Pottenger, Lawrence Rauchwerger, and Peng Tu. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[12] Lori Carter, Beth Simon, Brad Calder, Larry Carter, and Jeanne Ferrante. Predicated static single assignment. In *1999 International Conference on Parallel Architectures and Compilation Techniques, PACT 1999*, pages 245–255, Newport Beach, CA, October 1999.

[13] Fred C. Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. A new algorithm for partial redundancy elimination based on SSA form. In *1997 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 1997*, pages 273–286, Las Vegas, Nevada, June 1997.

[14] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems, TOPLAS 1991*, 13(4):451–490, October 1991.

[15] Keith Faigin, Stephen Weatherford, Jay Hoeflinger, David A. Padua, and Paul Petersen. The Polaris internal representation. *International Journal of Parallel Programming*, 22(5):553–586, October 1994.

[16] Lisa Fleischer, Bruce Hendrickson, and Ali Pinar. On identifying strongly connected components in parallel. In *14th International Parallel and Distributed Processing Symposium, IPDPS 2000*, pages 505–511, Cancun, Mexico, 2000.

[17] GNU GCC. Gnu compiler collection. Available at `http://gcc.gnu.org`.

[18] Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA. *ACM Transactions on Programming Languages and Systems, TOPLAS 1991*, 17(1):85–122, 1995.

[19] Andrew S. Glassner. *Graphic Gems*. Academic Press, 1993.

[20] Eladio Gutiérrez, Oscar G. Plata, and Emilio L. Zapata. Balanced, locality-based parallel irregular reductions. In *14th International Workshop on Languages and Compilers for Parallel Computing, LCPC 2001*, Cumberland Falls, KY, August 2001. To be published in Lecture Notes in Computer Science.

[21] Mary W. Hall, Jennifer-Ann M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, 1996.

[22] Hwansoo Han and Chau-Wen Tseng. Efficient compiler and run-time support for parallel irregular reductions. *Parallel Computing*, 26(13-14):1861–1887, 2000.

[23] Hewlett-Packard Company. Compaq KAP™ optimizers for Fortran90, Fortran77 and C languages. Available at `http://www.hp.com/techservers/software/kap.html`.

[24] Mehmet H. Karaata and Fawaz S. Al-Anzi. A dynamic self-stabilizing algorithm for finding strongly connected components. In *18th Annual ACM Symposium on Principles of Distributed Computing, PODC 1999*, page 276, Atlanta, GA, May 1999.

[25] Robert Kennedy, Fred C. Chow, Peter Dahl, Shin-Ming Liu, Raymond Lo, and Mark Streich. Strength reduction via SSAPRE. In *7th International*

Conference on Compiler Construction, CC 1998, pages 144–158, Lisbon, Portugal, 1998.

[26] Christoph W. Keßler. Applicability of program comprehension to sparse matrix computations. In *3rd International European Conference on Parallel Processing, Euro-Par 1997*, pages 347–351, Passau, Germany, August 1997.

[27] Kathleen Knobe and Vivek 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, January 1998.

[28] Kathleen Knobe and Vivek Sarkar. Enabling sparse constant propagation of array elements via Array SSA form. In *5th International Static Analysis Symposium, SAS 1998*, Pisa, Italy, September 1998.

[29] Kathleen Knobe and Vivek Sarkar. Enhanced parallelization via analyses and transformations on Array SSA form. In *Compilers for Parallel Computers, CPC 2000*, Aussois, France, January 2000.

[30] Xiangyun Kong, David Klappholz, and Kleanthis Psarris. The I-Test: An improved dependence test for automatic parallelization and vectorization. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):342–349, 1991.

[31] Konstantinos Kyriakopoulos and Kleanthis Psarris. Data dependence analysis for complex loop regions. In *30th International Conference on Parallel Processing, ICPP 2001*, pages 195–204, Valencia, Spain, September 2001.

[32] Yuan Lin and David 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.

[33] María J. Martín, David E. Singh, Juan Tourino, and Francisco 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, August 2002.

[34] Dror E. Maydan, John L. Hennessy, and Monica S. Lam. Efficient and exact data dependence analysis. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI)*, pages 1–14, Toronto, Ontario, Canada, June 1991.

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

[36] Kleanthis Psarris, Xiangyun Kong, and David Klappholz. The direction vector I-Test. *IEEE Transactions on Parallel and Distributed Systems*, 4(11):1280–1290, November 1993.

[37] Kleanthis Psarris and Konstantinos Kyriakopoulos. Data dependence testing in practice. In *1999 International Conference on Parallel Architectures and Compilation Techniques, PACT 1999*, pages 264–273, Newport Beach, CA, October 1999.

[38] William Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM (CACM)*, 35(8):102–114, August 1992.

[39] Lawrence Rauchwerger and David A. 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, February 1999.

[40] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *15th Annual ACM Symposium on Principles of Programming Languages, POPL 1988*, pages 12–27, San Diego, CA, January 1988.

[41] Y. Saad. *SPARSKIT: A basic tool kit for sparse matrix computations.* Available at `http://www.cs.umn.edu/Research/darpa/-/SPARSKIT/sparskit.html`.

[42] A.V.S. Sastry and Roy Dz-Ching Ju. A new algorithm for scalar register promotion based on SSA form. In *1998 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 1998*, pages 15–25, Montreal, Canada, June 1998.

[43] Inc. Silicon Graphics. Automatic parallelization option (APO) of the MIPSpro$^{\mathrm{TM}}$ compilers for C, C++, Fortran77 and Fortran90 languages. Available at `http://www.sgi.com/developers/-/devtools/languages/apo.html`.

[44] Toshio Suganuma, Hideaki Komatsu, and Toshio Nakatani. Detection and global optimization of reduction operations for distributed parallel machines. In *Proceedings of the 1996 International Conference on Supercomputing, ICS'96*, pages 18–25, Philadelphia, PA, May 1996.

[45] Peng Tu and David A. Padua. Automatic array privatization. In *6th International Workshop on Languages and Compilers for Parallel Computing, LCPC 1993*, pages 12–14, Portland, OR, August 1994.

[46] Peng Tu and David A. Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of the 9th ACM International Conference on Supercomputing, ICS'95*, pages 414–423, July 1995.

[47] Peng Tu and David A. Padua. Automatic array privatization. In *Compiler Optimizations for Scalable Parallel Systems Languages, Compilation Techniques, and Run Time Systems*, pages 247–284, 2001.

[48] Stefan Turek. *Featflow: Finite Element Software for the Incompressible Navier-Stokes Equations.* Available at `http://www.featflow.de`.

[49] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. In *12th Annual ACM Symposium on Principles of Programming Languages, POPL 1985*, pages 291–299, New Orleans, LA, January 1985.

[50] Michael Wolfe. Beyond induction variables. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI)*, pages 162–174, San Francisco, CA, June 1992.

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

[52] Peng Wu, Albert Cohen, Jay Hoeflinger, and David A. Padua. Induction variable analysis without idiom recognition: Beyond monotonicity. In

14th International Workshop on Languages and Compilers for Parallel Computing, LCPC 2001, Cumberland Falls, KY, August 2001. To be published in Lecture Notes in Computer Science.

[53] Peng Wu, Albert Cohen, Jay Hoeflinger, and David A. Padua. Monotonic evolution: An alternative to induction variable substitution for dependence analysis. In *2001 International Conference on Supercomputing, ICS 2001*, pages 78–91, Sorrento, Italy, June 2001.

[54] Cheng-Zhong Xu and Vipin 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.

[55] Hao Yu and Lawrence Rauchwerger. Adaptive reduction parallelization techniques. In *2000 International Conference on Supercomputing, ICS 2000*, pages 66–77, Santa Fe, NM, 2000.

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