XARK: An E**X**tensible Framework for **A**utomatic **R**ecognition of Computational **K**ernels

MANUEL ARENAZ, JUAN TOURIÑO, and RAMON DOALLO
University of A Coruña

The recognition of program constructs that are frequently used by software developers is a powerful mechanism for optimizing and parallelizing compilers to improve the performance of the object code. The development of techniques for automatic recognition of computational kernels such as inductions, reductions and array recurrences has been an intensive research area in the scope of compiler technology during the 90's. This article presents a new compiler framework that, unlike previous techniques that focus on specific and isolated kernels, recognizes a comprehensive collection of computational kernels that appear frequently in full-scale real applications. The XARK compiler operates on top of the Gated Single Assignment (GSA) form of a high-level intermediate representation (IR) of the source code. Recognition is carried out through a demand-driven analysis of this high-level IR at two different levels. First, the dependences between the statements that compose the strongly connected components (SCCs) of the data-dependence graph of the GSA form are analyzed. As a result of this intra-SCC analysis, the computational kernels corresponding to the execution of the statements of the SCCs are recognized. Second, the dependences between statements of different SCCs are examined in order to recognize more complex kernels that result from combining simpler kernels in the same code. Overall, the XARK compiler builds a hierarchical representation of the source code as kernels and dependence relationships between those kernels. This article describes in detail the collection of computational kernels recognized by the XARK compiler. Besides, the internals of the recognition algorithms are presented. The design of the algorithms enables to extend the recognition capabilities of XARK to cope with new kernels, and provides an advanced symbolic analysis framework to run other compiler techniques on demand. Finally, extensive experiments showing the effectiveness of XARK for a collection of benchmarks from different application domains are presented. In particular, the SparsKit-II library for the manipulation of sparse matrices, the Perfect benchmarks, the SPEC CPU2000 collection and the PLTMG package for solving elliptic partial differential equations are analyzed in detail.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processor—*Compilers, optimization*

This research was supported by the Ministry of Education and Science of Spain and FEDER funds of the European Union (Projects TIN2004-07797-C02 and TIN2007-67537-C03), and by the Galician Government (Projects PGIDIT05PXIC10504PN and PGIDIT06PXIB105228PR).
Authors' address: Department of Electronics and Systems, Faculty of Computer Science, Campus de Elviña s/n, 15071 A Coruña, Spain; email: {arenaz; juan; doallo}@udc.es.

General Terms: Algorithms, Languages, Experimentation

Additional Key Words and Phrases: Automatic kernel recognition, demand-driven algorithms, use-def chains, symbolic analysis, gated single assignment, strongly connected component

ACM Reference Format:

Arenaz, M., Touriño, J., and Doallo, R. 2008. **XARK**: An e**X**tensible framework for **A**utomatic **R**ecognition of computational **K**ernels. ACM Trans. Prog. Lang. Syst. 30, 6, Article 32 (October 2008), 56 pages. DOI = 10.1145/1391956.1391959 http://doi.acm.org/10.1145/1391956.1391959

1. INTRODUCTION

The development and maintenance of applications that make an efficient use of the computer architecture is a complex time-consuming task even for experienced programmers. The reasons for this are very varied. For instance, the programming style usually needs to be adapted to the characteristics of the compiler available in the target computer. This is a common practice in the embedded arena where codes are often written with plenty of scalar temporary variables in order to provide the compiler with an increased number of opportunities for optimization. Furthermore, in application domains where performance requirements are paramount, the developer must use either nonstandard extensions of programming languages or platform-optimized libraries written in assembly language. These distinctive features often affect code portability making it necessary to write different versions targeted for different computer architectures. The development of parallel applications adds a higher level of complexity as, in addition, the programmer must cope with the peculiarities of parallel computer architectures, namely, interconnection network, shared or distributed memory, parallel programming paradigm, etc.

The cost of software development and software maintenance is highly influenced by the uniprocessor and multiprocessor issues discussed above, specially in application domains where hardware technology changes very fast (e.g., computer graphics) because the code needs to be retargeted each time a different architecture is launched. It is not a recent observation that compiler technology plays an important role in reducing that cost. During the compilation process, optimizing compilers apply automatic program analysis techniques that gather information about the code. In the literature [Aho et al. 2006; Muchnick 1997; Wolfe 1996; Allen and Kennedy 2002], automatic program analysis is addressed from different perspectives, for instance, reaching definition analysis, dependence analysis, live analysis, kernel recognition and inter-procedural analysis. Although current optimizing compilers combine different types of techniques, more sophisticated approaches are still needed in order to handle the complexity of real applications.

Automatic kernel recognition is the process of discovering program constructs in the source code. In general, it can be sketched as matching a given source code against a set of program constructs. This problem has been studied for a wide variety of application areas that range from string matching and replacement in text edition, through detection of induction and reduction variables in parallelizing compilers, up to program synthesis and modification in

software engineering. Thus, kernel recognition techniques can be classified in four levels according to the information required in order to match the set of program constructs [Kozaczynski et al. 1992]: the text level, the syntactic level, the semantic level and the concept level. At the *text level*, programs are represented as ASCII files directly. The application of recognition techniques is limited to string matching and replacement. At the *syntactic level*, the source code is parsed in order to build an abstract syntax tree (AST) that preserves the logical information only. Thus, information to increase the readability of the code or assist parsing, such as indentation, keywords, comments, etc., is not captured. Examples of applications are variable renaming and one-to-one translation between language constructs. At the *semantic level*, the semantic specification of the programming language in which the program is written is captured by annotating the AST with data and control flow information. Constant propagation and common subexpression elimination are standard compiler techniques that fit in this category. In many situations, it is usually insufficient to understand only the syntax and the semantics of a program. For instance, in software maintenance, the programmers must have gained an adequate understanding of the functionality of the code, that is, what the code is supposed to do, before they can modify it. This information is captured at the concept level, by annotating program ASTs with both semantic information and abstract concepts. The concept level can be further divided. On the one hand, the *domain-independent concept level* describes the functionality of the code from the point of view of the programmer. Thus, the program is represented in terms of programming concepts such as inductions, scalar reductions, irregular reductions or array recurrences. Examples of kernel recognition techniques that work at this level are Arenaz et al. [2003], Gerlek et al. [1995], Pottenger and Eigenmann [1995], and Suganuma et al. [1996]. On the other hand, the *domain-specific concept level* takes into account the knowledge about problem solving that is handled by the experts in a given application domain. For instance, imposing constraints on the access patterns of the read-only arrays of a scalar reduction, enables the recognition of either the inner dot product of two vectors in the linear algebra domain or the convolution of two signals represented as vectors in the signal processing domain. Some approaches proposed in the literature are di Martino and Iannello [1996], Keßler and Smith [1999], and Paul and Prakash [1994]. The resulting five levels of program information are shown in Figure 1. Note that the levels are not mutually exclusive but inclusive. Therefore, recognizing at a higher level requires recognizing at lower levels.

The focus of this article is the presentation of an extensible compiler framework for automatic recognition of domain-independent concepts (from now on, *kernels*). The main contribution is three-fold. First, the explanation of a comprehensive collection of kernels that appear in regular and irregular codes organized into families of kernels that share common properties. Some examples are inductions, reductions, masked operations, irregular assignments, irregular reductions, array recurrences and consecutively written arrays. Note that current optimizing compilers recognize these kernels in isolated stages that are usually run in a strict, predefined order in different moments of the compilation process. While such a strict separation into stages may simplify the design

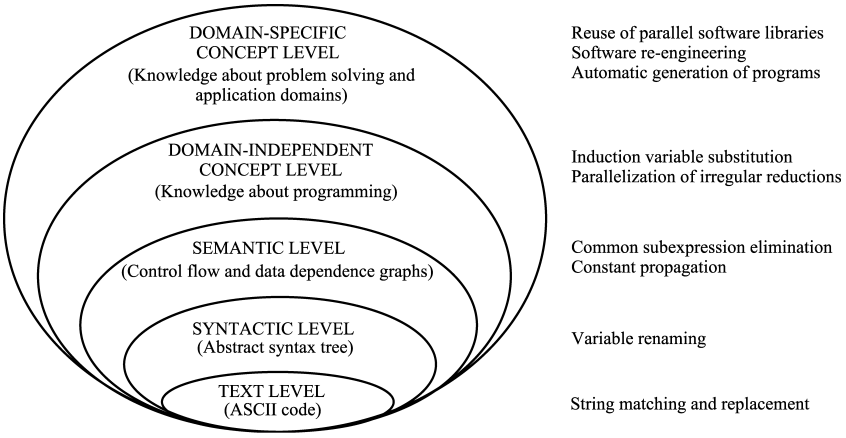


Fig. 1. Five-level classification of kernel recognition techniques according to their requirements about program information. For each level, typical examples of applications are presented.

and implementation, it generally leads to higher compilation-times than an all-in-one recognition stage. Second, the description of the internals of the XARK framework. XARK is based on a demand-driven classification method that analyzes the Gated Single Assignment (GSA) program representation. This article presents an algorithm for the recognition of the kernels represented by the strongly connected components of the data-dependence graph of the GSA form, and an algorithm for the recognition of more complex kernels that arise when the simpler kernels are combined in the same code. And third, extensive experimental results that show the efficacy of the recognition scheme for a set of well-known benchmark suites are included. The experiments demonstrate that XARK provides a more general solution than previous techniques for automatic kernel recognition at the domain-independent concept level. In addition, they show practical evidence that a relatively small set of kernels enables automatic recognition across different application domains.

The rest of the article is organized as follows: Section 2 gives a general overview of the XARK compiler. Basic ideas and terms that will be used in the explanations are also introduced. Section 3 describes in detail the collection of kernels considered in this work. Sections 4 and 5 present detailed descriptions of the internals of the recognition algorithms. Section 6 shows the experimental results. Section 7 discusses the robustness, the time complexity and the extensibility of the XARK compiler. Finally, Section 8 compares the approach with related work, and Section 9 concludes the paper.

2. OVERVIEW OF THE XARK COMPILER

Current optimizing and parallelizing compilers use several intermediate representations (IRs) during the translation of the source code into object code targeted for the underlying computer architecture. The level of abstraction of these IRs vary greatly from low-level IRs that represent the code in a manner that is very close to assembly language (e.g., RTL of GCC [GCC Internals]), to high-level IRs that resemble the original source code (e.g., GIMPLE

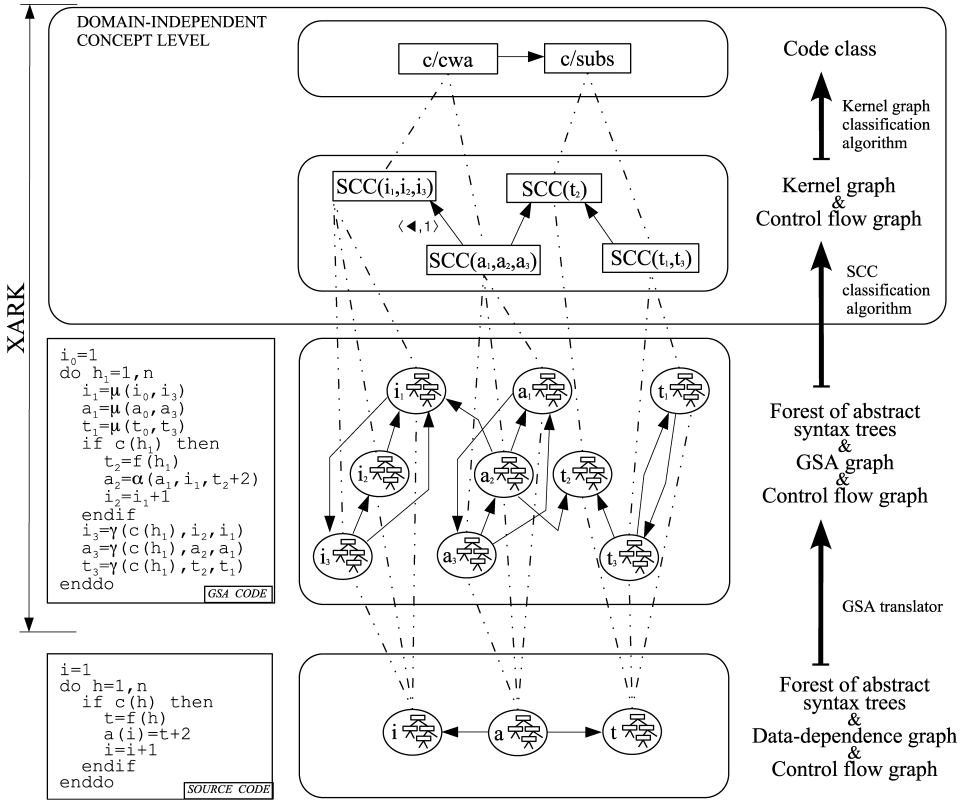


Fig. 2. Overview of the XARK compiler framework.

[Merrill 2003], Polaris IR [Faigin et al. 1994]). Code optimizations such as register allocation and instruction scheduling are carried out on top of low-level IRs because they are very dependent on the features of the underlying hardware (e.g., set of instructions, special registers). In contrast, program transformations for parallel code generation or locality improvement are usually implemented on top of high-level IRs because the source code is represented in a clearer manner.

Figure 2 presents an overview of the XARK compiler framework using as a guide the example code at the lower left corner, which will also be used to describe the XARK internals in detail throughout the paper. The code consists of a loop $do_{\mathtt{h}}$ that contains two kernels, namely, the computation of a *consecutively written array* `a` (denoted by *c/cwa*), and the computation of the loop-variant temporary variable `t` set to the value of the subscripted expression `f(h)` in each loop iteration (denoted by *c/subs*). At run-time, consecutive entries of the array `a` are written in consecutive memory locations determined by the value of the linear induction variable `i`. The complexity of this loop comes from the fact that `i` is incremented in one unit in those iterations where the condition `c(h)` is fulfilled (denoted in *c/subs* by the prefix character *c*). In general, the condition is not loop-invariant, so the value of `i` in each iteration cannot be calculated

as a function of the loop index variable h. The statement t=f(h) represents temporary computations that do not introduce loop-carried dependences during the execution of $do_{\mathtt{h}}$.

The framework is built on top of a high-level IR that captures the semantic specification of the programming language. It consists of a forest of abstract syntax trees (ASTs) that represent the statements of the source code, a data-dependence graph that represents data dependences between the ASTs where a variable is defined and used, and a control flow graph (CFG) where the ASTs are organized in basic blocks according to the control flow of the program. In Figure 2, the example loop is represented as a directed graph whose nodes and edges denote ASTs and data dependences between ASTs, respectively. For the sake of clarity, the details about the loop index variable (h) and about the CFG have been omitted. Note that the data dependences are depicted as use-def chains in order to emphasize the demand-driven nature of the algorithms used in XARK. The recognition process is divided in three stages: (1) *GSA translator* for building the Gated Single Assignment (GSA) form of the code; (2) *SCC classification algorithm* for the recognition of the simple kernels represented by the SCCs of the GSA graph, that is, the data-dependence graph of the GSA form; and (3) *kernel graph classification algorithm* for recognition of the compound kernels associated with sets of mutually dependent simple kernels. The rest of this section presents the key ideas behind each stage, and introduces the IRs built by XARK in order to summarize the relevant information extracted at each stage.

2.1 Translation into GSA Form

The construction of the XARK compiler begins with the translation of the source code into the Gated Single Assignment (GSA) form [Ballance et al. 1990]. GSA is an extension of the well-known Static Single Assignment (SSA) form [Cytron et al. 1991] where reaching definition information of scalar and array variables is represented syntactically. The construction of the GSA form involves two main tasks: first, placement of special operators (called ϕ generically) at the points of the program with multiple predecessors in the control flow graph; and second, renaming of program variables so that the left-hand sides of the assignment statements define distinct unique variables. Different kinds of ϕ operators are distinguished according to the point of the program where they are inserted:

—$\mu(x_{out}, x_{in})$, which appears at loop headers and selects the initial x_{out} and loop-carried x_{in} values of a variable.
—$\gamma(c, x_{true}, x_{false})$, which is located at the confluence node associated with a branch (e.g., if-endif construct), and captures the condition c for each definition to reach the confluence node: x_{true} if c is fulfilled; x_{false}, if c is not satisfied.
—$\alpha(a_{prev}, s, e)$, whose meaning is that the element s of an array variable a is set to the value e and the other elements take the values of the previous definition of the array, denoted as a_{prev}.

A detailed description of an efficient translation algorithm is beyond the scope of this paper and can be consulted in Tu and Padua [1995]. A GSA-like representation called *Partial Array SSA* was proposed in Knobe and Sarkar [1998, 2000]. However, the ϕ operators defined in GSA present several advantages from the point of view of XARK. First, the γ operator contains a predicate that captures the condition of an if-endif construct, which enables the analysis of conditions during the execution of the SCC classification algorithm. And second, the α operator provides a compact representation where array assignment statements are associated with a unique SCC of the GSA graph. This property prevents code explosion and widens the collection of kernels that can be recognized by the SCC classification algorithm.

In Figure 2, the GSA form is shown just above the source code of the consecutively written array. Note that new statements with special operators μ, γ and α have been inserted in the code, and that all the statements define different variables. Thus, each definition of a source code variable is represented by a different variable in GSA form (as usual, GSA variables are built by subscripting the source code variable). The construction of the GSA form results in the following changes in the compiler IR. First, each AST of a source code statement is replaced with the ASTs of the corresponding statements in the GSA code. And second, new data dependences that capture reaching definition information between the GSA statements are introduced. Figure 2 shows the forest of ASTs that represent the GSA code (the nodes are labeled with the unique GSA variables of the statements). The dashed dotted lines remark the relationship between the source code variables `a`, `i` and `t`, and the corresponding GSA variables. Note that, in the source code, the use-def chains are sets of edges that link each use to the reaching definitions. However, in GSA form, all use-def chains are singletons. Thus, the GSA graph captures the information about reaching definitions and preserves the data dependences between `a`, `i` and `t`.

2.2 Recognition of Simple Kernels

The second stage of XARK decomposes the GSA code into a set of mutually dependent kernels called *simple kernels*. The key observation is that simple kernels correspond to the statements of the SCCs of the GSA graph, which capture the flow of values of the source code variables at run-time. Focus on the statements of the GSA code of Figure 2 that represent the induction variable `i`, namely, the statements where the GSA variables $\texttt{i}_0$, $\texttt{i}_1$, $\texttt{i}_2$ and $\texttt{i}_3$ are modified. Starting at the μ operator that defines $\texttt{i}_1$, the external definition $\texttt{i}_0$ determines the value at the beginning of the first loop iteration. On subsequent iterations, $\texttt{i}_1$ takes the value calculated within the loop body during the execution of the statement $\texttt{i}_3=\gamma(\texttt{c}(\texttt{h}_1),\texttt{i}_2,\texttt{i}_1)$. The γ operator represents that, if the condition $\texttt{c}(\texttt{h}_1)$ is true in the loop iteration $\texttt{h}_1$, then $\texttt{i}_3$ takes the value $\texttt{i}_2$ defined in the body of the if-endif construct. Otherwise, it takes the value $\texttt{i}_1$ defined by the μ operator at the beginning of the iteration. The right-hand side of the statement $\texttt{i}_2=\texttt{i}_1+1$ also fetches the value $\texttt{i}_1$ assigned by the μ operator. Thus, the flow of values corresponding to the induction variable `i` during the execution

of the loop is represented by an SCC in the GSA graph. Regarding the computation of array a, the key observation is that, unlike the source code statement a(i)=t+2, the GSA statement $a_2=\alpha(a_1,i_1,t_2+2)$ defines the new value of array a in terms of its previous value explicitly. This is achieved by means of the α operator, which represents the definition of the element $a_2(i_1)$ in terms of the previous value of the whole array, in this case, the value a_1 determined by the μ operator inserted at the beginning of the loop body. The forest of ASTs and the GSA graph depicted in Figure 2 show the SCCs associated with i and a. The notation $SCC(x_1, \ldots, x_n)$ refers to the SCC composed of the ASTs related to the GSA variables $x_1, \ldots, x_n$ corresponding to different definitions of the source code variable x.

The discussion above leads to an important conclusion: the computations associated with every scalar variable defined in terms of itself and with every array variable are represented by an SCC in the GSA graph. Thus, during the second stage of the XARK compiler, the SCC classification algorithm analyzes the forest of ASTs combined with the GSA graph and the control flow graph, and determines the simple kernel associated with every source code variable. In the example, three kernels are recognized: a conditional linear induction (see Section 3.2) represented by the strongly connected component $SCC(i_1, i_2, i_3)$ associated with the GSA statements $i_1=\mu(i_0,i_3)$, $i_2=i_1+1$ and $i_3=\gamma(c(h_1),i_2,i_1)$; a conditional array assignment with a linear access pattern (see Section 3.1) captured by the statements $a_1=\mu(a_0,a_3)$, $a_2=\alpha(a_1,i_1,t_2+2)$ and $a_3=\gamma(c(h_1),a_2,a_1)$ of $SCC(a_1, a_2, a_3)$; and a conditional scalar assignment (see Section 3.1) that represents a loop-variant sequence of values that are not known at compile-time, which is captured by $t_2=f(h_1)$ of $SCC(t_2)$, as well as by $t_1=\mu(t_0,t_3)$ and $t_3=\gamma(c(h_1),t_2,t_1)$ of $SCC(t_1, t_3)$.

The information extracted from the source code during the execution of the SCC classification algorithm is summarized in an IR called *kernel graph*. Figure 2 presents a simplified version of the kernel graph that only contains the information that is relevant for this introductory section. The nodes and the edges of the kernel graph correspond to SCCs and to use-def chains between statements of different SCCs, respectively. A detailed description of the SCC classification algorithm and of the kernel graph will be presented in Section 4.

2.3 Recognition of Compound Kernels

The third stage of the XARK compiler uses the kernel graph classification algorithm in order to recognize *compound kernels* that consist of a set of mutually dependent simple kernels. In the example of Figure 2, the isolated detection of the simple kernels conditional linear induction (i) and conditional array assignment (a) does not provide enough information to recognize the consecutively written array kernel. Thus, for the compiler to have success, the dependences between both simple kernels have to be analyzed. In general, staged kernel analysis approaches build kernels up from simple kernels (e.g., inductions) to more complex kernels (e.g., reductions). However, staged approaches cannot recognize mutually dependent kernels, which require a comprehensive analysis of

cyclic kernel dependence directed graphs (e.g., to recognize consecutively written arrays).

The kernel graph exhibits the minimal set of properties that characterize a compound kernel. The so-called *scenario* consists of a set of properties of the use-def chains between SCCs as well as a set of properties of the corresponding SCCs. The key idea behind the kernel graph classification algorithm is the identification of scenarios and, in the following, the execution of appropriate compile-time tests to confirm or discard the existence of the corresponding compound kernel. In Figure 2, the kernel graph contains a use-def chain $SCC(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3) \rightarrow SCC(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)$ annotated with the label $< \blacktriangleleft, 1 >$. This information is used in the SCC classification algorithm to distinguish different situations in which a variable is used and thus to classify the use of the variable appropriately. The annotation of the use-def chain is a mechanism to carry out this information to the kernel graph classification algorithm so that the scenarios that characterize compound kernels can be detected. In this example, the label indicates that the left-hand side (denoted as $\blacktriangleleft$) of the source code assignment statement `a(i)=t+2` consists of one array reference whose subscript expression is an occurrence of the induction variable `i` (denoted as 1 in the label). Once this scenario has been detected, the compiler analyzes the control flow graph to prove that every time `a(i)=t+2` is executed, `i=i+1` is also executed. Furthermore, the compiler checks that `i` is incremented in one unit in every loop iteration where `c(h)` is fulfilled. In codes such as that of the example where this compile-time test is successful, XARK recognizes a consecutively written array kernel. A detailed description of the kernel graph classification algorithm will be presented in Section 5. Note that in order to capture the information about the initialization and updating of induction variables and about the access patterns of array variables, XARK uses the *chains of recurrences* formalism first introduced in Zima [1986] and later improved in Zima [1995], and van Engelen [2001].

The results of this stage are summarized in the code class $[\![c/cwa \rightarrow c/subs]\!]$, shown above the kernel graph in Figure 2. The code class exhibits the two kernels computed in the loop: the computation of the consecutively written array `a` (c/cwa), and the computation of the temporary variable `t` ($c/subs$). In addition, the code class captures the dependence relationship between the kernels, which is represented by a use-def chain $\rightarrow$ indicating that the values stored in `t` are used in the computation of `a`.

Overall, the code class provides the compiler with a hierarchical description of the program in terms of the kernels computed in the source code and the dependence relationships between such kernels. In addition, all the information extracted from the source code during the execution of the SCC and kernel graph classification algorithms is annotated in the intermediate representations built by the compiler. Thus, the code class abstracts the implementation details and, at the same time, meets the information requirements of other passes of parallelizing and optimizing compilers. Examples of successful application of XARK in the scopes of parallel code generation and compile-time prediction of memory hierarchy behavior have been presented in Arenaz et al. [2004] and Andrade et al. [2007], respectively.

3. COLLECTION OF KERNELS

In the early stages of this work the SparsKit-II library [Saad 1994] and several finite element numerical codes were analyzed manually in order to identify a set of kernels that are frequently used by software developers.

A program consists of a set of variables that represent memory locations where data are stored, and a set of statements that specify the way data are manipulated. In a similar manner, a kernel is related to a subset of variables and statements of a program. It should be noted that a kernel can be coded in many different ways. For instance, the two codes shown below carry out the same computations:

```
do h=1,n                    a(1)=...
   a(h)=...                 ...
enddo                       a(n)=...
```

The loop-based version is used extensively in scientific applications that work with large arrays. However, in embedded applications these loops are often unrolled in order to provide the compiler with more opportunities for optimization. It is the job of the compiler to recognize the variations of a given kernel, even in programs where the statements of the kernel are spread over the source code.

In order to describe the collection of kernels, some properties are introduced.

Definition 3.1. Let $\{v_1, \ldots, v_n\}$ be the variables of a kernel. It is a *scalar kernel* if $\{v_1, \ldots, v_n\}$ are scalar variables. It is an *array kernel* if $\{v_1, \ldots, v_n\}$ are array variables.

Definition 3.2. Let $\{v_1, \ldots, v_n\}$ be the variables of a kernel. It is a *gated kernel* if there is at least one occurrence of v_k ($k \in \{1 \ldots n\}$) in the condition of an if-endif construct that contains statements of the kernel. Thus, v_k influences the control flow of the statements of the kernel. Otherwise, it is a *non-gated kernel*.

Definition 3.3. Let $\{S_1, \ldots, S_n\}$ be the set of statements of a kernel. It is a *conditional kernel* if $\exists S_k$ ($k \in \{1 \ldots n\}$) such that S_k belongs to the body of an if-endif construct. Otherwise, it is a *non-conditional kernel*.

The rest of this section describes the collection of kernels organized into the eight families presented in Table I, detailing whether the kernels included in each family are scalar/array, gated/non-gated and conditional/nonconditional. The description is illustrated with examples of kernels that appear in source codes extracted from real applications.

3.1 Assignments

The simplest kernels considered in this work consist of storing a value in a given memory location. Within a program, this memory location may be accessed using either a scalar or an array variable. Thus, *scalar assignments* and *array assignments* are distinguished, respectively.

Let the statement $v = e$ represent a scalar assignment that stores the value of expression e in the memory location given by the scalar variable v, where e does not contain any occurrence of v. Both *non-conditional scalar assignment* and

Table I. Collection of Kernel Families

Kernel Family	Scalar	Array	Gated	Non-gated	Conditional	Nonconditional
Assignments (Section 3.1)						
scalar assignment	√			√	√	√
regular array assignment		√		√	√	√
irregular array assignment		√		√	√	√
Inductions (Section 3.2)						
linear induction	√			√	√	√
polynomial induction	√			√	√	√
geometric induction	√			√	√	√
Maps (Section 3.3)						
scalar map	√			√	√	√
regular array map		√		√	√	√
irregular array map		√		√	√	√
Reductions (Section 3.4)						
scalar reduction	√			√	√	√
regular array reduction		√		√	√	√
irregular array reduction		√		√	√	√
scalar minimum/maximum reduction	√		√		√	
regular array minimum/maximum reduction		√	√		√	
irregular array minimum/maximum reduction		√	√		√	
Masks (Section 3.5)						
scalar find&set	√		√		√	
regular array find&set		√	√		√	
irregular array find&set		√	√		√	
Array recurrences (Section 3.6)						
regular array recurrence		√		√	√	√
irregular array recurrence		√		√	√	√
Reinitialized kernels (Section 3.7)						
induction	√			√	√	√
map	√	√		√	√	√
reduction	√	√	√	√	√	√
mask	√	√	√		√	
array recurrence		√		√	√	√
Complex written arrays (Section 3.8)						
consecutively written array		√		√	√	√
consecutively reduced array		√		√	√	√
consecutively recurrenced array		√		√	√	√
segmented consecutively written array		√		√	√	√
segmented consecutively reduced array		√		√	√	√
segmented consecutively recurrenced array		√		√	√	√

conditional scalar assignment kernels can be found in real codes. For brevity, the notation *(non-)conditional* will be used throughout the article to refer to both the conditional and the non-conditional kernels. The computation of `irow` in the loop do_{h} of Figure 3(a) is an example of conditional scalar assignment.

Let the statement $a(s) = e$ represent an array assignment that stores the value of expression e in the memory location given by the sth entry of the array

```
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
   endif
enddo
```

(a) Computation of the length of the smallest row and the row number of a sparse CRS matrix (from SparsKit-II, routine *rperm*). Conditional scalar assignment (variable `irow`) and scalar minimum with index reduction (variables `minlen` and `irow`).

```
do h=1,n
   t=0
   do k=ia(h),ia(h+1)-1
      t=t+a(k)*x(ja(k))
   enddo
   y(h)=t
enddo
```

(b) Product of a sparse CRS matrix and a vector (from SparsKit-II, routine *amux*). Within the loop $do_{\mathtt{k}}$, nonconditional scalar reduction (variable `t`). Within the loop $do_{\mathtt{h}}$, nonconditional regular array assignment with linear access pattern (variable `y`), and reinitialized nonconditional scalar reduction (variable `t`).

```
do j=1,nrow
   i=perm(j)
   iao(i+1)=ia(j+1)-ia(j)
enddo
```

(c) Permutation of the rows of a sparse CRS matrix (from SparsKit-II, routine *rperm*). Nonconditional irregular array assignment with linear access pattern (variable `iao`).

```
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)
      endif
      ko=ko+1
   enddo
enddo
```

(d) Loop nest from SparsKit-II, routine *rperm*. Within the loop $do_{\mathtt{k}}$, nonconditional induction variable `ko` and (non)conditional consecutively written arrays `jao` and `ao`. Within $do_{\mathtt{ii}}$, reinitialized nonconditional induction variable `ko` and (non)conditional segmented consecutively written arrays `jao` and `ao`.

Fig. 3. Examples of kernels from real codes.

variable a, where there is not any occurrence of variable a in e. Different types of array assignments will be distinguished according to the properties of the subscript s. Parallelizing compilers have traditionally focused on array assignments that appear in regular codes, where s can be rewritten as a linear, polynomial or geometric function of the loop index. These kernels are called *regular array assignments*. The computation of array `y` in the loop $do_{\mathtt{h}}$ of Figure 3(b) is a nonconditional regular array assignment. In particular, the regular access pattern `h` is a linear induction (see Section 3.2) given by the loop index of $do_{\mathtt{h}}$. Irregular codes contain *irregular array assignments* where s is a loop-variant subscripted expression whose value cannot be determined at compile-time. The computation of array `iao` in $do_{\mathtt{j}}$ of Figure 3(c) is a nonconditional irregular array assignment. Note that `i+1` is a subscripted expression because the scalar variable `i` takes the value of a different array entry, `perm(j)`, at the beginning of each $do_{\mathtt{j}}$ iteration.

3.2 Inductions

In the literature, the term *induction* is used to represent the type of scalar, integer-valued variables that are updated in all the iterations of a loop, and for

which a well-defined closed form expression can be calculated. In this article, this term will be used in a more general sense. If the variable is updated in every loop iteration, it is a *nonconditional induction*. In real programs, however, the variable can be updated conditionally during the execution of the loop. In this case, it is a *conditional induction*.

The simplest form of induction is *(non)conditional linear induction*, 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. Note that loop-invariants usually involve array references, which may even contain subscripted subscripts. The recognition of more complex inductions was shown to be of interest for the automatic analysis of real programs [Gerlek et al. 1995]. Thus, *(non)conditional polynomial inductions* are characterized by the addition of another linear induction variable to the induction variable, and *(non)conditional geometric induction* by the product of the induction variable and an invariant. In the literature, these kernels are generically called basic inductions because the variable is defined in terms of itself. When the variable is defined in terms of other linear, polynomial or geometric induction variables, they are called derived inductions. The computation of the scalar variable `ko` in the inner loop do_{k} of Figure 3(d) is a basic non-conditional linear induction.

3.3 Maps

A distinguishing characteristic of inductions is that there is a closed form function that allows the computation of the next value of the variable starting from its initial value (nonconditional inductions), or from its current value (conditional inductions). In real codes, sequences of values that do not have such a closed form are often used. This kind of computations will be referred to as *maps*.

A map is a kernel where a variable is assigned the value of an array reference whose subscript expression contains an occurrence of the variable. Like derived inductions, derived maps are defined in terms of the variable of another map kernel. The kernel is called *(non)conditional scalar map* if the variable is a scalar, and either *(non)conditional regular array map* or *(non)conditional irregular array map* if it is an array. Different types of regular (e.g., linear, polynomial, geometric) and irregular access patterns are allowed. An example of non-conditional regular array map is shown in Figure 4(a).

3.4 Reductions

A reduction is usually defined as the process of obtaining a single element by combining the elements of a vector [Allen and Kennedy 2002]. Well-known examples are adding the elements of a vector (sum reduction), and finding the minimum/maximum element in a vector (minimum/maximum reduction). In this paper a definition that only takes into account the syntactical properties of the program constructs is presented.

```
do i=1,nbf
   do j=1,2
      ibndry(j,i)=mark(ibndry(j,i))
   enddo
enddo
```

(a) Loop nest from routine *cleanup* of the PLTMG code. Nonconditional regular array map (variable `ibndry`).

```
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
      endif
   enddo
enddo
```

(b) Conversion of a sparse matrix into CRS format (SparsKit-II, routine *ssrcsr*). Conditional irregular array reduction (variable `iwk`).

```
do h=1,n
   if diag(h)≠0 then
      diag(h)=1/diag(h)
   else
      diag(h)=1
   endif
enddo
```

(c) Loop from SparsKit-II, routine *dscaldg*. Regular array find&set `diag` with linear access pattern.

```
iao(1)=1
do j=1,nrow
   iao(j+1)=iao(j+1)+iao(j)
enddo
```

(d) Loop from SparsKit-II, routine *rperm*. Nonconditional regular array recurrence (variable `iao`).

Fig. 4. Examples of kernels from real codes (cont.).

3.4.1 *Scalar/Array Nongated Reductions.* A *(non-)conditional scalar reduction* is a kernel with one scalar variable that is defined in terms of itself and at least one loop-variant subscripted expression. The scalar variable may be either integer-valued or floating-point-valued. Derived scalar reductions are also defined in terms of the variable of a different scalar reduction. An abstract representation of a scalar reduction is $v = v \oplus e$, where v is the reduction variable, $\oplus$ is the reduction operator and e is an expression with zero occurrences of v. The inner loop do_{k} of Figure 3(b) contains a nonconditional scalar reduction that involves subscripted subscripts of several indirection levels.

The kernel *(non-)conditional array reduction* is defined in a similar manner. Let the statement $a(s) = a(s) \oplus e$ represent the computation of the sth entry of the array variable a, where e does not contain any occurrence of a. The characteristics of s lead to distinguish between regular array reductions and irregular array reductions. The loop nest do_{i} of Figure 4(b) calculates a conditional irregular array reduction (array variable `iwk`).

3.4.2 *Scalar/Array Gated Reductions.* Another well-known reduction operation is the computation of the minimum (or 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. In comparison with non-gated reductions, the distinguishing characteristic is the implementation of the reduction operator using if-endif constructs that check the value of the reduction variable. This type of kernel is called either *scalar minimum reduction* or *scalar maximum reduction*. The code of Figure 3(a) calculates a scalar minimum reduction where `minlen` is the reduction variable and {`ia(h+1)-ia(h)`; `h=2...nrow`} is the set of values. A variant of a minimum/maximum reduction that is frequently used in real codes consists of gathering additional

information about the reduction variable, for instance, the position of the minimum (or the maximum) value within the set. This variant of the kernel will be termed *scalar minimum/maximum with index reduction*. An example involving the scalars `minlen` and `irow` is shown in Figure 3(a). It should be noted that *array minimum/maximum reductions*, with and without index, appear in real codes as well. Regular and irregular access patterns to the array variable are also allowed. An example is the computation of the minimum with index of each row of a matrix.

3.5 Masks

Masks are conditional gated kernels that modify the value at a memory location if its content fulfills a Boolean condition. When a scalar variable is involved, the kernel is called *scalar find&set*. A typical example is a loop that contains a set of statements that are executed only in the first loop iteration. When the condition is *true*, such statements are executed and the condition is set to *false* to avoid the execution in the subsequent loop iterations. The masks that involve array variables are called *regular array find&set* or *irregular array find&set*. The loop do_{h} of Figure 4(c) contains a regular array find&set where the array variable `diag` presents a linear access pattern.

3.6 Array Recurrences

The non-gated kernels described in Sections 3.1–3.4 represent assignment and reduction operations. Given the computation $a(s) = e$, they cover the cases where there are zero and one occurrences of $a(s)$ in e, respectively. There is a remaining case where e contains a set of occurrences $a(s_1), \ldots, a(s_m)$ so that, in the general case, the subscripts $s, s_1, \ldots, s_m$ are different. This kernel is called *(non-)conditional array recurrence*. Access patterns to the array variable may be either regular or irregular. Note that in the classical sense, an array recurrence satisfies the additional constraint that at least one subscript is symbolically less than s, which is an important property in the scope of application domains such as parallelizing compilers. The computation of the array `iao` in do_{j} of Figure 4(d) is a nonconditional regular array recurrence with a linear access pattern.

3.7 Reinitialized Kernels

Real codes may contain more elaborate program constructs built from kernels described in the previous sections. From a graphical point of view, they can be interpreted as a point in a multidimensional space where the kernels are the values represented in the axes. Thus, a reinitialized kernel is as follows: first, an assignment (Section 3.1) that sets a scalar/array variable to a given value at the beginning of every iteration of a loop; and second, an induction, a map, a reduction, a mask or an array recurrence (Sections 3.2–3.6) that updates the value of the scalar/array variable during the execution of an inner loop. A *reinitialized non-conditional linear induction* is presented in the loop $do_{\mathtt{ii}}$ of Figure 3(d) using `ko` as scalar variable. Note that, at run-time, `ko` is set to the value of the subscripted expression `iao(perm(ii))` at the beginning of

each $do_{\texttt{ii}}$ iteration, the variables `iao` and `perm` being loop-invariant arrays. A *reinitialized non-conditional scalar reduction* to compute a temporary sum `t` is shown in Figure 3(b).

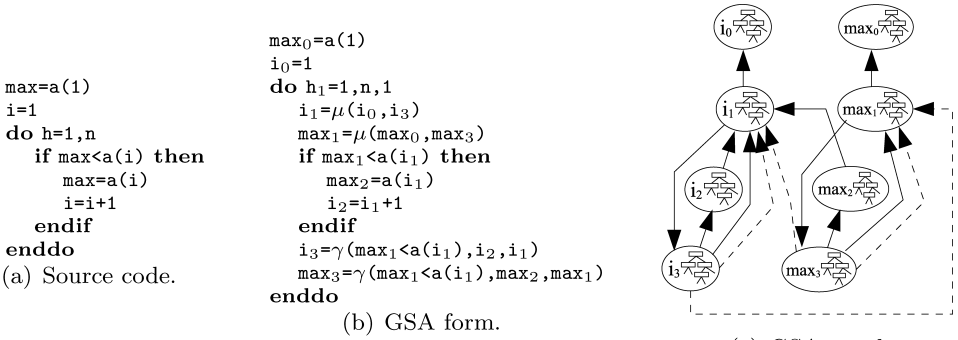
3.8 Complex Written Arrays

Another interesting family is called *complex written array*. It consists of a scalar kernel (e.g., induction, reinitialized induction, scalar reduction) that defines the array entries to be modified during the execution of the code, and an array assignment whose left-hand side subscript is a linear function of the scalar variable. When the scalar kernel is a (non-)conditional linear induction of step one, the kernel is called *(non-)conditional consecutively written array* [Lin and Padua 1998]. Examples are shown in loop $do_{\texttt{k}}$ of Figure 3(d) involving the linear induction variable `ko` as the subscript of the arrays `ao` and `jao`. When the scalar kernel is a reinitialized linear induction of step one, it is called *(non) conditional segmented consecutively written array*. Note that in the scope of the outer loop $do_{\texttt{ii}}$ of Figure 3(d), `ao` and `jao` fit into this category as `ko` is a reinitialized linear induction variable of step one. The variety of complex written arrays considered in this work is shown in Table I, including kernels that involve an array reduction or an array recurrence instead of an array assignment.

4. RECOGNITION OF SIMPLE KERNELS

In general, the computations carried out during the execution of a code can be represented as a set of mutually dependent simple kernels. This issue was illustrated in the overview of Section 2 using as a guide a loop that computes a consecutively written array kernel. In that section, the recognition of simple kernels through the analysis of the strongly connected components (SCCs) that appear in the GSA graph was introduced. From the great variety of kernels shown in Table I, the SCC classification algorithm addresses the recognition of assignments, inductions, maps, non-gated reductions, some array gated reductions, array masks, array recurrences and reinitialized array kernels. Note that, for instance, the recognition of scalar minimum/maximum reductions, scalar masks, reinitialized scalar kernels and complex written arrays cannot be accomplished in this stage because the SCCs do not contain enough information. The recognition of these kernels is carried out by the kernel graph classification algorithm presented later in Section 5.

The rest of this section is organized as follows. Definitions and notations for the characterization of the simple kernels represented by the SCCs of GSA graphs as well as the dependences between them are introduced in Section 4.1. A taxonomy of SCC classes that capture the properties of the different types of simple kernels is formally defined in Section 4.2. The SCC classification algorithm for the construction of the kernel graph is described in detail in Section 4.3. Finally, a case study extracted from the example of Figure 2 is presented in Section 4.4.



(a) Source code. (b) GSA form. (c) GSA graph.

Fig. 5. Search of SCCs in GSA graphs considering conditional use-def chains.

4.1 Definitions and Notations

The following terminology will be used throughout this article: *source code statement* will refer to a statement of the source code; *GSA statement* (or simply *statement*) will be used for a statement of the GSA form; finally, μ-statement, γ-statement and α-statement will denote GSA statements that contain μ, γ and α operators, respectively.

An important property of the techniques for automatic kernel recognition is the ability to recognize the variations of a kernel, even in the presence of complex control flows or in those implementations where the source code statements of the kernel are spread over the program. Techniques that hinge on the identification of SCCs in SSA forms were shown to be effective for this purpose because they use the data-dependence information to drive the recognition process. The SCC classification algorithm that will be presented in this section is based on the following definition of SCC in the context of GSA graphs.

Definition 4.1. Let $x \rightarrow y$ be a use-def chain of the GSA graph that represents a reaching definition of the GSA variable y for a use in a GSA statement that defines the variable x. It is a *conditional use-def chain* if x is defined in a γ-statement whose conditional expression contains the use of the reaching definition y.

Definition 4.2. A *strongly connected component*, $SCC(x_1, \ldots, x_n)$, is a maximal subgraph of the GSA graph where every node in the subgraph can be reached by every other node in the subgraph following nonconditional use-def chains only, that is, ignoring conditional use-def chains.

In order to illustrate how Definition 4.2 enables the recognition of simple kernels, consider the example of Figure 5, where the conditional use-def chains introduced by the condition $\texttt{max}_1\texttt{<a(i}_1\texttt{)}$ are depicted as dashed edges in the GSA graph. Considering conditional use-def chains, one $SCC(\texttt{i}_1, \texttt{i}_2, \texttt{i}_3, \texttt{max}_1, \texttt{max}_2, \texttt{max}_3)$ that represents both a scalar maximum reduction (variable max) and a conditional induction variable i that influences the control flow arises. However, if the conditional use-def chain is ignored following Definition 4.2, two separate components $SCC(\texttt{i}_1, \texttt{i}_2, \texttt{i}_3)$

and $SCC(\texttt{max}_1, \texttt{max}_2, \texttt{max}_3)$ that distinguish the two simple kernels are identified.

In order to increase the robustness of the SCC classification algorithm, the following property of the SCCs is defined (see Section 7.1 for a discussion about the robustness of the recognition algorithms).

Definition 4.3. Let $y_1, \ldots, y_m$ be a subset of the GSA variables of a strongly connected component $SCC(x_1, \ldots, x_n)$ such that $y_i \in \{x_1, \ldots, x_n\}$ and y_i is not defined in a μ-statement or γ-statement. The *cardinality* of $SCC(x_1, \ldots, x_n)$, $|SCC(x_1, \ldots, x_n)|$, is the number of different source code variables represented by $y_1, \ldots, y_m$.

In Section 3, some interesting properties of the kernels were introduced (see Definitions 3.1–3.3). Accordingly, the same properties are defined in the scope of an SCC in order to distinguish the different types of simple kernels that can be recognized by the SCC classification algorithm. This article restricts the cardinality of the SCCs to zero and one, as experiments conducted on real programs demonstrated that they enable the recognition of most of the kernels (see Section 6 for the details).

Definition 4.4. Let x be a source code variable represented by a cardinality-0 or cardinality-1 $SCC(x_1, \ldots, x_n)$. The component is a *scalar SCC* if x is a scalar variable, and it is an *array SCC* if x is an array variable. The notations $SCC_C^S(x_1, \ldots, x_n)$ and $SCC_C^A(x_1, \ldots, x_n)$ will represent a scalar and an array SCC with cardinality C, respectively.

Definition 4.5. Let $c_1, \ldots, c_m$ be the set of conditional expressions associated with the γ-statements of a cardinality-0 or cardinality-1 $SCC(x_1, \ldots, x_n)$. It is a *gated SCC* if $\exists x_k$ $(k \in \{1 \ldots n\})$ such that there is an occurrence of x_k in $c_1, \ldots, c_m$, that is, if x_k influences the control flow of the statements of $SCC(x_1, \ldots, x_n)$. Otherwise, it is a *non-gated SCC*.

Definition 4.6. Let $SCC(x_1, \ldots, x_n)$ be a cardinality-0 or cardinality-1 SCC. It is a *conditional SCC* if $SCC(x_1, \ldots, x_n)$ contains at least one γ-statement, that is, if $SCC(x_1, \ldots, x_n)$ contains at least one assignment statement enclosed within an if-endif construct. Otherwise, it is a *non-conditional SCC*.

Other types of SCCs that are useful for automatic program analysis in XARK are distinguished. The criteria are related to the number of μ, γ and α statements that the SCC contains.

Definition 4.7. An $SCC(x_1, \ldots, x_n)$ is *trivial* if it consists of exactly one GSA statement ($n = 1$). Otherwise, it is *non-trivial* ($n > 1$).

Definition 4.8. An $SCC(x_1, \ldots, x_n)$ is *virtual* if it is composed of μ or γ statements only. Otherwise, it is *non-virtual*.

As mentioned in Section 2.2, the SCC classification algorithm addresses the recognition of simple kernels through the analysis of the forest of ASTs of the GSA statements jointly with the GSA graph and the control flow graph. In an AST, a tree node represents an operator (e.g., assignment, fetch, array reference,

plus, product), and the children of a node are the operands. Leaf tree nodes capture either invariant values (e.g., constants), or fetches of variables whose values are calculated in a different GSA statement.

Definition 4.9. Given a node of the AST that represents a GSA statement of a program, the *level of the node* is the number of nodes contained in the path from the root of the AST to that node. The *indirection level of the node* is the number of nodes in the path from the root of the AST to that node that represent array references in the source code of the program.

Definition 4.10. Let $\oplus$ be an operator represented by a tree node of the AST of a GSA statement. The *operator class*, $[\![\oplus]\!]$, is the type of kernel that is calculated as a result of executing the operator.

Definition 4.11. Let $SCC(x_1, \ldots, x_n)$ be a strongly connected component. The *SCC class*, $[\![SCC(x_1, \ldots, x_n)]\!]$, is the type of kernel computed by executing the source code statements represented by the GSA statements where x_1,. . .,x_n are defined.

Different abbreviations are used for the SCC classes of scalar and array SCCs. For the class of a scalar nongated component $[\![SCC_C^S(x_1, \ldots, x_n)]\!]$, a pair χ/θ is used, where $\chi \in \{c, nc\}$ indicates the conditionality (see Definition 4.6) and θ is the type of scalar kernel listed in the taxonomy of SCCs presented later in Figure 6. For instance, nc/lin denotes a non-conditional linear induction. Let $a(s_1, \ldots, s_d) = e$ represent the array kernel computed by an array component $SCC_C^A(x_1, \ldots, x_n)$. The class of an array nongated component $[\![SCC_C^A(x_1, \ldots, x_n)]\!]$ is denoted by a triplet $\chi/\tau/\theta_1 : \cdots : \theta_d$ where $\chi \in \{c, nc\}$ is the conditionality, τ is the type of array operation listed in Figure 6 (see array assignment, array reduction, array recurrence or array map in Section 3), and $\theta_i = [\![s_i]\!]$ with $i \in \{1, \ldots, d\}$ is the type of scalar kernel that represents the access pattern in the ith dimension of the d-dimensional array a. An example of array non-gated SCC class is $c/reduc/subs$, which corresponds to a conditional irregular array reduction of a unidimensional array. Similar notations are used for gated SCC classes, the difference being that conditionality is not specified because all gated SCCs are conditional. All the possibilities for both scalar and array SCC classes will be introduced in Section 4.2.

The results of the SCC classification algorithm are summarized in the kernel graph IR (see the overview of Figure 2), which exhibits both the simple kernels captured by the SCCs and the dependences between them. In the following, the concept of dependence between statements is generalized to the concept of dependence between SCCs, different types of dependences between SCCs are introduced, and the kernel graph is formally defined. Unless otherwise stated, for the purpose of the recognition of simple kernels, dependences will be represented and referred to as use-def chains in order to emphasize the demand-driven nature of the SCC classification algorithm.

Definition 4.12. Let $SCC(x_1, \ldots, x_n)$ and $SCC(y_1, \ldots, y_m)$ be two SCCs. A *SCC use-def chain*, $SCC(x_1, \ldots, x_n) \rightarrow SCC(y_1, \ldots, y_m)$, exists if the GSA graph contains at least one use-def chain $x_j \rightarrow y_k$ (with $j \in \{1 \ldots n\}$

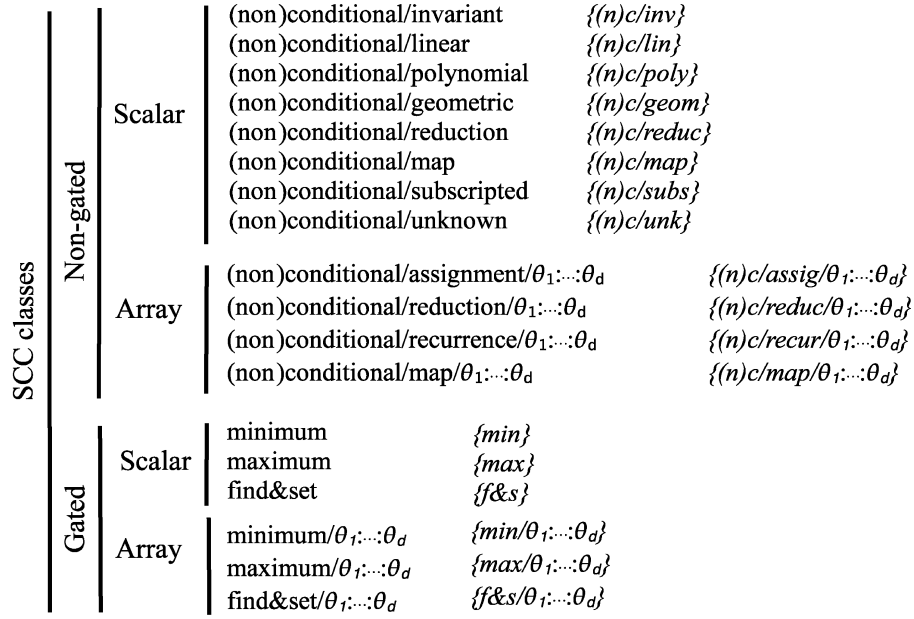


Fig. 6. Taxonomy of SCCs in GSA graphs. Abbreviations of SCC classes are written in italic within braces. The notation $\theta_1 : \cdots : \theta_d$ represents the classes of the subscripts of a d-dimensional array variable of an array SCC, the subscript classes being *invariant*, *linear*, *polynomial*, *geometric*, *reduction*, *map*, *subscripted* or *unknown*.

and $k \in \{1 \ldots m\}$) between two GSA statements of $SCC(x_1, \ldots, x_n)$ and $SCC(y_1, \ldots, y_m)$, respectively.

The terms, *use statement*, *definition statement*, *use-SCC*, and *definition-SCC* (abbreviated as *def-SCC*), will denote the statement where x_j is defined, the statement where y_k is defined, $SCC(x_1, \ldots, x_n)$ and $SCC(y_1, \ldots, y_m)$, respectively.

Several classes of SCC use-def chains will be distinguished in the kernel graph. The goal of these classes is to exhibit those dependences that will provide the kernel graph classification algorithm with scenarios for the recognition of compound kernels (see Section 5 for the details).

Definition 4.13. Let $SCC(x_1, \ldots, x_n) \rightarrow SCC(y_1, \ldots, y_m)$ be an SCC use-def chain between two cardinality-0 or cardinality-1 SCCs, and let $x_j \rightarrow x_k$ (with $j \in \{1 \ldots n\}$ and $k \in \{1 \ldots m\}$) be the corresponding use-def chain between GSA statements. A *conditional SCC use-def chain* $SCC(x_1, \ldots, x_n) \rightsquigarrow SCC(y_1, \ldots, y_m)$ exists if $x_j \rightarrow y_k$ is a conditional use-def chain (see Definition 4.1). Otherwise, other two classes of chains are distinguished. It is a *structural SCC use-def chain* $SCC(x_1, \ldots, x_n) \Rightarrow SCC(y_1, \ldots, y_m)$ if one of the following properties is fulfilled:

(1) The variables $x_1, \ldots, x_n, y_1, \ldots, y_m$ are definitions of one source code variable.

(2) $SCC(x_1, \ldots, x_n)$ is an array SCC with $\theta_1 : \cdots : \theta_d$ as the classes of the subscripts; $SCC(y_1, \ldots, y_m)$ is a scalar SCC of class χ/θ; x_j is defined in an α-statement whose left-hand side subscript contains a use of the reaching definition y_k; and $\exists r \in \{1 \ldots d\}$ such that $\theta = \theta_r$.

Otherwise, it is a *non-structural SCC use-def chain*, represented by the notation $SCC(x_1, \ldots, x_n) \nRightarrow SCC(y_1, \ldots, y_m)$.

Definition 4.14. Given a program in GSA form, the *kernel graph* is defined as the graph whose nodes are the SCCs that appear in the GSA graph (in accordance with Definition 4.2), and whose edges are the conditional, structural and nonstructural SCC use-def chains.

In the kernel graph, the SCC use-def chains are attached a pointer to their corresponding use-def chains between GSA statements. In addition, they are annotated with information that enables the identification of scenarios in the kernel graph classification algorithm presented later in Section 5. Note that the properties (1) and (2) of Definition 4.13 will enable the identification of the scenarios that characterize reinitialized scalar kernels and complex written arrays, respectively.

4.2 Taxonomy of SCCs

The SCCs that appear in GSA graphs can be classified according to the characteristics of the statements that compose each SCC. A taxonomy of SCC classes is presented in Figure 6. The following criteria are used: scalar/array (Definition 4.4), gated/non-gated (Definition 4.5) and conditionality (Definition 4.6).

4.2.1 *Scalar Non-Gated SCCs.* These SCCs may be trivial or non-trivial (Definition 4.7). Nontrivial SCCs capture basic inductions, scalar non-gated reductions and scalar maps as follows:

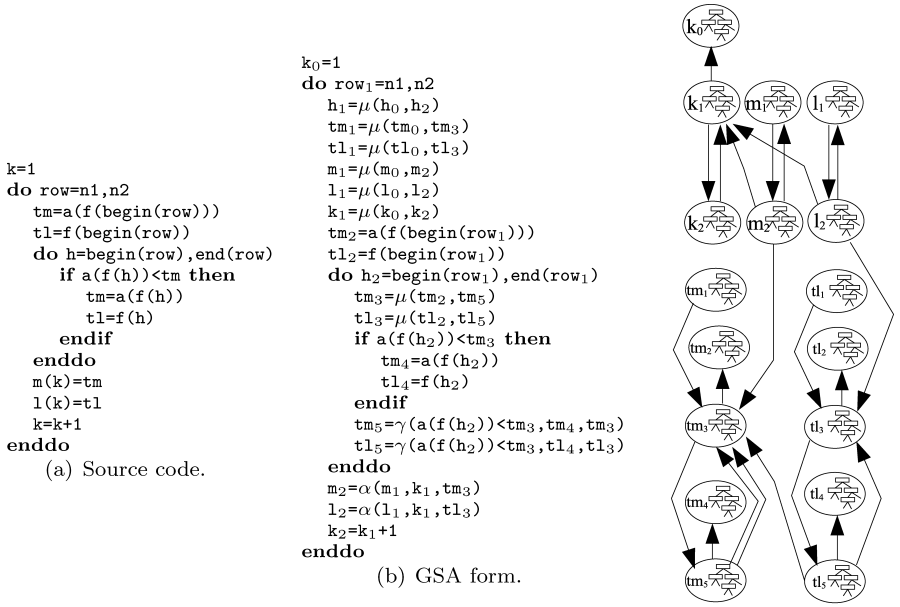
—*(non)conditional / linear* (see Section 3.2). The operations within the SCC are a linear combination of the source code variable and invariant expressions. The loop $do_{\texttt{row}}$ of Figure 7 contains a non-conditional linear induction that is represented by a non-gated component $SCC_1^S(\texttt{k}_1, \texttt{k}_2)$ of class *non-conditional / linear*.

—*(non)conditional / polynomial* (see Section 3.2). The operations are a linear combination of the source code variable, invariants and one linear induction variable represented by a different SCC.

—*(non)conditional / geometric* (see Section 3.2). The source code variable is multiplied by an invariant value.

—*(non)conditional / reduction* (see Section 3.4). The operations within the SCC involve the source code variable and at least one non-invariant array reference.

—*(non)conditional / map* (see Section 3.3). The right-hand sides of scalar statements consist of one array reference only. Besides, the subscripts of such array reference consist of a fetch of the variable defined in the SCC. An example



(a) Source code.

(b) GSA form.

(c) GSA graph.

Fig. 7. Computation of the minimum with index of each row of a sparse matrix.

code is the traversal of a linked list implemented by means of an array, that is, `i=next(i)` within a loop body.

—*(non)conditional / unknown*, which captures those SCCs whose kernel does not match any other scalar non-gated SCC class.

Regarding trivial scalar non-gated SCCs, they represent derived inductions (linear, polynomial and geometric), derived scalar reductions and derived scalar maps. The operations performed in trivial SCCs are not defined in terms of the source code variable of the SCC, but in terms of the variables of other SCCs. The classification rules described above for non-trivial SCC classes also apply to trivial ones. Furthermore, two additional trivial SCC classes are distinguished:

—*(non)conditional / invariant*. Let $\{v_1, \ldots, v_n\}$ be the set of scalar/array variables that are referenced in the statements of the SCC. The SCC is invariant if $\forall v_k$ ($k \in \{1 \ldots n\}$), v_k is not defined within the fragment of code (e.g., a loop body).

—*(non)conditional / subscripted*. Let $\{v_1, \ldots, v_n\}$ be the set of scalar/array variables that are referenced in the statements of the SCC. The SCC is subscripted if $\exists v_k$ ($k \in \{1 \ldots n\}$) such that it is a fetch of an array variable whose subscript is not invariant (e.g., a loop-variant expression that changes its value in each loop iteration).

Some examples of cardinality-0 and cardinality-1 scalar non-gated SCCs are presented in Figure 7. The computation of the scalar `tl` in the scope of the inner loop do_{h} is an example of conditional scalar assignment (see

Section 3.1). Its representation in terms of SCCs is as follows. On the one hand, there is a cardinality-1 trivial component $SCC_1^S(\texttt{tl}_4)$ of class *non-conditional*/*subscripted* that captures the loop-variant nature of the values assigned to `tl`. On the other hand, as `tl` is modified within an if-endif construct, a cardinality-0 virtual component $SCC_0^S(\texttt{tl}_3, \texttt{tl}_5)$ that captures the conditionality of the kernel appears in the GSA graph as well.

4.2.2 *Array Non-Gated SCCs.* Let $a(s_1, \ldots, s_d) = e$ be an array assignment statement that represents the computations carried out in an array SCC. Different classes are distinguished according to the number of occurrences of array a that appear in the right-hand side expression e at an indirection level 0 (see Definition 4.9):

—*(non)conditional*/*assignment*/$\theta_1 : \cdots : \theta_d$ (see Section 3.1), if there are zero occurrences; where $\theta_i = [\![s_i]\!], i \in \{1, \ldots, d\}$.
—*(non)conditional*/*reduction*/$\theta_1 : \cdots : \theta_d$ (see Section 3.4), if there is one occurrence matching $a(s_1, \ldots, s_d)$.
—*(non)conditional*/*recurrence*/$\theta_1 : \cdots : \theta_d$ (see Section 3.6), if e contains a set of array references $\{a(s_1^1, \ldots, s_d^1), \ldots, a(s_1^r, \ldots, s_d^r)\}$ such that at least one of them, $a(s_1^k, \ldots, s_d^k)$ with $k \in \{1 \ldots r\}$, fulfills that $s_1 \neq s_1^k$ or $s_2 \neq s_2^k$ or ... or $s_d \neq s_d^k$.

An additional class where the occurrences of a appear in the right-hand side expression e at indirection level greater than 0 (i.e., within the subscript of an array reference) is distinguished:

—*(non)conditional*/*map*/$\theta_1 : \cdots : \theta_d$ (see Section 3.3), if e consists of an array reference that matches $b(\ldots, a(s_1, \ldots, s_d), \ldots)$, where b is a multidimensional array different from a.

The notation captures the access pattern corresponding to each dimension of the left-hand side array reference $a(s_1, \ldots, s_d)$, which are determined by applying the rules of trivial scalar non-gated SCCs. Thus, the following types of access patterns are distinguished: *invariant*, *linear*, *polynomial*, *geometric*, *reduction*, *map*, *subscripted* and *unknown*. The computations of array `l` in Figure 7 are represented by means of an array non-gated $SCC_1^A(\texttt{l}_1, \texttt{l}_2)$ of class *non-conditional*/*assignment*/*linear*. The example also contains array references whose subscripts fit into the classes *linear* and *subscripted*, for instance, `begin(row)` and `a(f(h))`, respectively.

4.2.3 *Scalar Gated SCCs.* They appear in loops where a scalar variable is defined inside an if-endif construct whose condition contains occurrences of that variable. Gated SCCs are all conditional as they contain at least one γ-statement. Let $v = e$ represent the source code assignment statement of a scalar gated SCC. Let c be the condition defined by the γ-statements of the SCC. The following classes are distinguished:

—*minimum* or *maximum* (see Section 3.4.2). The condition c matches $v < e$, $v \leq e$, $e > v$ or $e \geq v$ for minimum, where e does not contain occurrences of v. For maximum, it matches $e < v$, $e \leq v$, $v > e$ or $v \geq e$.

—*find&set* (see Section 3.5). The condition does not fulfill the properties of a minimum or a maximum.

It should be noted that scalars defined within if-endif constructs are represented by a trivial SCC that captures the scalar assignment statement, and a virtual gated SCC that contains the information about the condition. Thus, the SCC classification algorithm classifies the SCC as a *minimum*, *maximum* or *find&set* kernel, and annotates the SCC as a *candidate* in order to carry the information to the subsequent recognition phase. As will be shown in Section 5, all the information will be available during the execution of the kernel graph classification algorithm, where the candidate classes will be confirmed or discarded. The example of Figure 7 computes the minimum with index of each row of a sparse matrix. In each $do_{\texttt{row}}$ iteration, the minimum is stored in the scalar `tm`, whose representation in GSA form includes a candidate scalar gated $SCC_0^S(\texttt{tm}_3, \texttt{tm}_5)$. Note that the corresponding scalar assignment statement is captured in a different $SCC_1^S(\texttt{tm}_4)$.

4.2.4 *Array Gated SCCs.* Three classes are distinguished: *minimum* / $\theta_1 : \cdots : \theta_d$, *maximum* / $\theta_1 : \cdots : \theta_d$ and *find&set* / $\theta_1 : \cdots : \theta_d$, where $\theta_i = [\![s_i]\!]$ denote the classes of the subscripts of a d-dimensional array (with $i \in \{1, \ldots, d\}$). They verify the same conditions as the corresponding scalar gated SCC classes, the difference being that an array reference instead of a scalar variable is involved. Furthermore, the information about both the condition and the array assignment statement is available in the γ and α statements of array gated SCCs. In Figure 4(c) a kernel *find&set* / *linear* involving the one-dimensional array `diag` was presented.

4.3 SCC Classification Algorithm

The pseudocode of the algorithm for the recognition of simple kernels is presented in Figure 8 and will be explained in several steps in the following pages. The top-level procedure `Build_kernel_graph()` constructs the kernel graph in two stages. First, the SCCs of the GSA graph are identified in `Find_SCCs_in_GSA_graph()` using the Tarjan algorithm [Tarjan 1972]. This algorithm provides support for the demand-driven classification of SCCs as it ensures that an SCC is not found until all the SCCs associated with the variables referenced in the SCC have been found and, therefore, classified. Second, the SCCs are classified according to the SCC taxonomy of Figure 6 by means of a demand-driven algorithm implemented through three procedures with indirect recursion: `Classify_SCC()`, `Classify()` and TF_{id}. During the classification process, the SCC use-def chains are identified and annotated with the information needed by the kernel graph classification algorithm in order to distinguish the scenarios that characterize compound kernels. The kernel graph is completed with the classification of SCC use-def chains into conditional, structural or non-structural according to Definition 4.13. This task is accomplished in the procedure `Classify_SCC_use_def_chains()`.

```
procedure Build_kernel_graph()
input:
  Forest of ASTs and GSA graph
output:
  Kernel graph
{
  Find_SCCs_in_GSA_graph()
  for each SCC(x_1, ..., x_n) {
    Classify_SCC(SCC(x_1, ..., x_n))
    Classify_SCC_use_def_chains(SCC(x_1, ..., x_n))
  }
}

procedure Classify_SCC()
input:
  SCC(x_1, ..., x_n)
output:
  [[SCC(x_1, ..., x_n)]]: SCC class
{
  if SCC(x_1, ..., x_n) not visited {
    if SCC(x_1, ..., x_n) is trivial {
      // Start at root of the AST of x_1 = ⊗
      [[SCC(x_1, ..., x_n)]]=Classify(=, <> )
    }
    else {
      // Start at root of the AST of x_1 = μ(x_0, x_n)
      [[SCC(x_1, ..., x_n)]]=Classify(=, <> )
    }
  }
}

// Transfer function for id
procedure TF_id()
input:
  y: Fetch operator
  C: Classification context < ε, β, l, il >
output:
  [[id]]: Operator class
{
  // 1st branch: AST outside the code fragment
  if ∄ def-SCC for y.gsa_link {
    [[y]]=nc/inv
  }
  // 2nd branch: AST of the SCC(x_1, ..., x_n) under
  //             classification
  else if y.gsa_link in SCC(x_1, ..., x_n) {
    if y.gsa_link in stack_of_ASTs {
      if y and ε fetch the same scalar variable {
        [[y]]=nc/lin // Detect a cycle in a scalar SCC
      }
      else {
        [[y]]=nc/none // Detect a cycle in an array SCC
      }
    }
    else {
      [[y]]=Classify(y.gsa_link, <> )
    }
  }
  // 3rd branch: AST of a different SCC(y_1, ..., y_m)
  else if y.gsa_link in SCC(y_1, ..., y_m) {
    if ∃ AST such that
       AST∈SCC(y_1, ..., y_m) && AST∈stack_of_ASTs {
      [[y]]=nc/unk // Detect a cycle in the kernel graph
    }
    else {
      Classify_SCC(SCC(y_1, ..., y_m))
      [[y]]=[[SCC(y_1, ..., y_m)]]
    }
    establish SCC(x_1, ..., x_n) → SCC(y_1, ..., y_m)
    annotate SCC(x_1, ..., x_n) → SCC(y_1, ..., y_m)
  }
}

procedure Classify()
input:
  ⊕: Operator of a tree node
  C: Classification context < ε, β, l, il >
output:
  [[⊕]]^ε_{β,l,il}: Operator class
{
  case ⊕ of
  // Assignment operator =
  // of a GSA statement x = ⊗
  // (root of the corresponding AST)
  = (x, ⊗):
    if( AST not visited ) {
      push AST of = (x, ⊗)
      if( x is a compiler temporary ) {
        [[⊗]]=Classify(⊗,C)
      }
      else {
        [[⊗]]=Classify(⊗, < x, ►, 2, 0 >)
      }
      [[⊕]]^ε_{β,l,il} = TF_=([[⊗]],C)
      annotate the statement with [[⊕]]^ε_{β,l,il}
      pop AST of = (x, ⊗)
    }
    else {
      retrieve [[⊕]]^ε_{β,l,il} from the annotation
    }
  // Fetch of a scalar/array variable
  id:
    [[⊕]]^ε_{β,l,il} = TF_id(id,C)
  // Array reference
  a(s):
    [[a]]=Classify(a,< ε, β, l + 1, il >)
    [[s]]=Classify(s,< ε, β, l + 1, il + 1 >)
    [[⊕]]^ε_{β,l,il} = TF_a(s)([[a]],[[s]],C)
  // Special operators in GSA form
  μ(x_out, x_in):
    [[x_in]]=Classify(x_in,< ε, β, l + 1, il >)
    [[x_out]]=Classify(x_out,< ε, β, l + 1, il >)
    [[⊕]]^ε_{β,l,il} = TF_μ([[x_out]],[[x_in]],C)
  α(a, s, ⊗):
    [[a]]=Classify(a, < a(s), β, l + 1, il >)
    [[s]]=Classify(s, < a(s), ◄, l + 1, 1 >)
    [[⊗]]=Classify(⊗, < a(s), ►, l + 1, 1 >)
    [[⊕]]^ε_{β,l,il} = TF_α([[a]],[[s]],[[⊗]],C)
  γ(c, x_in, x_out):
    [[c]]=Classify(c,< ε, ?, l + 1, il >)
    [[x_in]]=Classify(x_in,< ε, β, l + 1, il >)
    [[x_out]]=Classify(x_out,< ε, β, l + 1, il >)
    [[⊕]]^ε_{β,l,il} = TF_γ([[c]],[[x_in]],[[x_out]],C)
  // Arithmetic (e.g., +, *),
  // logical(e.g., and, or, not)
  // and relational operators (e.g., ≠, ≥)
  ⊗(⊗_1, ⊗_2):
    [[⊗_1]]=Classify(⊗_1,< ε, β, l + 1, il >)
    [[⊗_2]]=Classify(⊗_2,< ε, β, l + 1, il >)
    [[⊕]]^ε_{β,l,il} = TF_⊗([[⊗_1]],[[⊗_2]],C)
  // Constant value
  K:
    [[⊕]]^ε_{β,l,il} = TF_K([[K]],C)
  // Otherwise
  default:
    [[⊕]]^ε_{β,l,il} = unk // Default TF
  }
}
```

Fig. 8. Pseudocode of the SCC classification algorithm for the recognition of simple kernels.

4.3.1 *Procedures.* `Classify_SCC()`, `Classify()` *and* TF_{id}. In `Classify_SCC()`, the class of an $SCC(x_1, \ldots, x_n)$ is computed as the class of the assignment operator of the μ-statement inserted after the header of the outermost loop of a loop nest (the exceptions are trivial SCCs, for which the unique GSA statement is used). The operator class $[\![=]\!]$ is determined by calling the recursive procedure `Classify()`, which launches a post-order traversal of the AST of the statement. When the leaf nodes that represent fetch operators of variables defined in other ASTs are reached, TF_{id} is executed. TF_{id} distinguishes three situations for the classification of a fetch operator y. Let *y.gsa_link* be a pointer to the AST of the definition statement of variable y (i.e., the use-def chains of the GSA graph are implemented as pointers *y.gsa_link*). First, if *y.gsa_link* is located outside the code fragment analyzed by the compiler (see 1st branch of TF_{id} in Figure 8), then there is not any SCC that contains *y.gsa_link* and $[\![y]\!]$ is set to class *nc*/*inv* to indicate that y is recognized as an invariant operator. Note that the *gsa_link* improves the performance of the SCC classification algorithm as searches for definition statements within programs are avoided. The second situation copes with target ASTs *y.gsa_link* that belong to the $SCC(x_1, \ldots, x_n)$ under classification. As shown at the end of the 2nd branch of TF_{id}, *y.gsa_link* is classified by calling `Classify()` and the corresponding operator class is assigned to y. The third situation distinguished in TF_{id} (3rd branch of TF_{id} in Figure 8) handles target ASTs *y.gsa_link* that belong to a different $SCC(y_1, \ldots, y_m)$. In this case, the classification of $SCC(x_1, \ldots, x_n)$ is deferred, the class $[\![SCC(y_1, \ldots, y_m)]\!]$ is computed, the classification of $SCC(x_1, \ldots, x_n)$ is resumed by setting $[\![y]\!]=[\![SCC(y_1, \ldots, y_m)]\!]$, and an SCC use-def chain $SCC(x_1, \ldots, x_n) \rightarrow SCC(y_1, \ldots, y_m)$ (see Definition 4.12) is established in the kernel graph. Finally, when all the fetch operators have been processed, $[\![SCC(x_1, \ldots, x_n)]\!]$ is successfully determined. It should be noted that ASTs are analyzed only once during the execution of the SCC classification algorithm in order to avoid redundant computations. Thus, in `Classify()`, the annotation of the ASTs with the operator class $[\![=]\!]$ is required in order to compute the class of several fetch operators of the same variable. The algorithm uses a *stack_of_ASTs* that contains the ASTs whose classification is in progress in order to detect cycles in the GSA graph and thus assure the termination of the recognition process. Two termination conditions are distinguished. First, the detection of fetch operators whose target AST *y.gsa_link* belongs to the $SCC(x_1, \ldots, x_n)$ under classification, which captures the cycles included in scalar and array SCCs (2nd branch of TF_{id} in Figure 8). In both cases, $[\![y]\!]$ is set to an SCC class that enables the recognition of the correct simple kernel. And second, the detection of fetch operators whose *y.gsa_link* is included in a different $SCC(y_1, \ldots, y_m)$ that is also under classification (3rd branch of TF_{id} in Figure 8). This situation captures mutually dependent simple kernels that arise as a result of ignoring conditional use-def chains during SCC search and that introduce cycles in the kernel graph. In this case, $[\![y]\!]$ is set to the class *nc*/*unk* to indicate that the simple kernels cannot be recognized by the SCC classification algorithm.

```
do h=1,g_size
   ...
   i=g(i)
enddo
```
(a) Computation of a scalar map.

```
do h=1,g_size
   ...
   i=h+1
   r=r+g(i)
enddo
```
(b) Computation of a scalar reduction.

Fig. 9. Source codes to illustrate the contextual classification of operators.

During the execution of `Classify()`, a post-order traversal of an AST is performed. At each node of the AST, the operator class is calculated by a *transfer function* that merges the classes derived for the operands. In Arenaz [2003], the transfer functions of the most common operators are described in detail: assignment ($TF_{=}$), fetch (TF_{id}), array reference ($TF_{a(s)}$), special GSA operators (TF_{μ}, TF_{γ} and TF_{α}), and arithmetic, logical and relational operators (e.g., TF_{+}, TF_{*}, $TF_{\neq}$, $TF_{<}$, $TF_{>}$, TF_{not}, TF_{and}, TF_{or}). For illustrative purposes, only TF_{id} is presented in the pseudocode of Figure 8. Unlike approaches to automatic kernel recognition that define a library that captures the properties of each kernel, the XARK compiler encodes such properties in the transfer functions. In order to distinguish the different situations in which an operator is used during the execution of a program, the concept of *classification context* is introduced.

Definition 4.15. Let $\oplus$ be an operator represented by a node of the AST of a GSA statement. The *classification context of the operator* is a tuple $< \epsilon, \beta, l, il >$ built as follows:

(1) Suppose that the GSA statement is an α-statement $a_k = \alpha(a_{prev}, s, e)$ that captures a source code array assignment statement $a(s) = e$. Then, $\epsilon = a_k(s)$, and either $\beta = \blacktriangleright$ (if the node is a part of e), or $\beta = \blacktriangleleft$ (if the node is a part of s);
(2) Suppose that the GSA statement is a γ-statement $x_k = \gamma(c, x_{true}, x_{false})$ representing an if-endif construct in the source code. Then, $\epsilon = x_k$, and either $\beta =?$ (if the node is a part of c) or $\beta = \blacktriangleright$ (if the node is a part of x_{true} or x_{false});
(3) Otherwise, $\epsilon = x$ and $\beta = \blacktriangleright$;

In all cases, l and il are the level and the indirection level of the node, respectively (see Definition 4.9). The notation $[\![\oplus]\!]^{\epsilon}_{\beta,l,il}$ represents the class of the operator $\oplus$ with respect to the classification context $< \epsilon, \beta, l, il >$.

In order to illustrate why contextual classification is needed, focus on the array reference `g(i)` of Figure 9. In Figure 9(a), the loop-carried dependence introduced by the scalar variable `i` is represented by an SCC in the GSA graph. The classification of the AST of the statement `i=g(i)` begins with an empty classification context (represented in Figure 8 as <> in the calls to `Classify()` from procedures `Classify_SCC()` and TF_{id}) that is later updated as the post-order traversal proceeds. Thus, the class $[\![\texttt{g(i)}]\!]^{\texttt{i}}_{\blacktriangleright,2,0}$ is determined to be a non-conditional scalar map (denoted by the class *nc / map*) because (1) the array reference `g(i)` is the right-hand side of the statement (given by $\beta = \blacktriangleright$ and $l = 2$ in the classification context), and (2) the subscript of the array reference and

the ϵ operator of the classification context are fetch operators of the variable i of the left-hand side of i=g(i). Note that subscripts are characterized by indirection levels $il \geq 1$ in the classification context (il is incremented in Classify() only when an array reference operator or the subscript of an α operator are analyzed, while l is updated in each call to Classify()). Now consider the computation of the scalar reduction using the variable r in the loop of Figure 9(b). In this case, g(i) is an operand of the sum operator that appears in the statement r=r+g(i), i being a scalar variable that takes a different value in each loop iteration. Thus, $[\![\mathtt{g(i)}]\!]^{\mathtt{r}}_{\blacktriangleright,3,0}$ represents the computation of a loop-variant expression (class *nc*/*subs*) because (1) the array reference is an operand of the + operator in the right-hand side of the statement (given by $\beta = \blacktriangleright$ and $l > 2$), and (2) the subscript i does not match the fetch operator of the variable r (given by ϵ). The example illustrates how the classification context provides transfer functions with enough information to distinguish the different situations that may arise.

4.3.2 *Procedure* Classify_SCC_use_def_chains(). The top level procedure of the SCC classification algorithm, Build_kernel_graph(), completes the construction of the kernel graph by classifying the use-def chains between SCCs into the categories defined in Definition 4.13 (namely, structural, non-structural and conditional). During the execution of the SCC classification algorithm the SCC use-def chains are annotated in TF_{id} with the information available in the classification context $< \epsilon, \beta, l, il >$ (see 3rd branch of TF_{id} in Figure 8). As a result, the classification of the SCC use-def chains is as follows. First, conditional SCC use-def chains are recognized when $\beta =$?. Second, structural SCC use-def chains are detected by checking properties (1) and (2) of Definition 4.13. For property (1), by checking that the def-SCC and use-SCC represent the same source code variable. For property (2), by checking the following five conditions: the def-SCC captures a scalar variable, the use-SCC captures an array variable, the class of the def-SCC matches the class that describes the access pattern in one dimension of the class of the use-SCC, $\beta = \blacktriangleleft$ and $il \geq 1$. And third, non-structural SCC use-def chains are recognized if the above conditions are not fulfilled. The usefulness of this information was pointed out in Section 2.3, where it was used for the identification of the scenarios that enable the recognition of compound kernels by the kernel graph classification algorithm.

4.4 Case Study

The consecutively written array kernel computed in the loop of Figure 2 will be used to illustrate the behavior of the SCC classification algorithm. For clarity, the classification context will be omitted if it is not relevant for the explanations. The first step is the identification of the SCCs that appear in the GSA graph. It contains the following components: $SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)$ and $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)$, that represent the conditional induction (variable i) and the conditional array assignment (variable a), respectively; $SCC_1^S(\mathtt{h}_1)$, which captures the loop index h; and $SCC_1^S(\mathtt{t}_2)$ and $SCC_0^S(\mathtt{t}_1, \mathtt{t}_3)$ that represent the flow of values of the scalar variable t computed inside the if-endif construct. Focus on the classification of

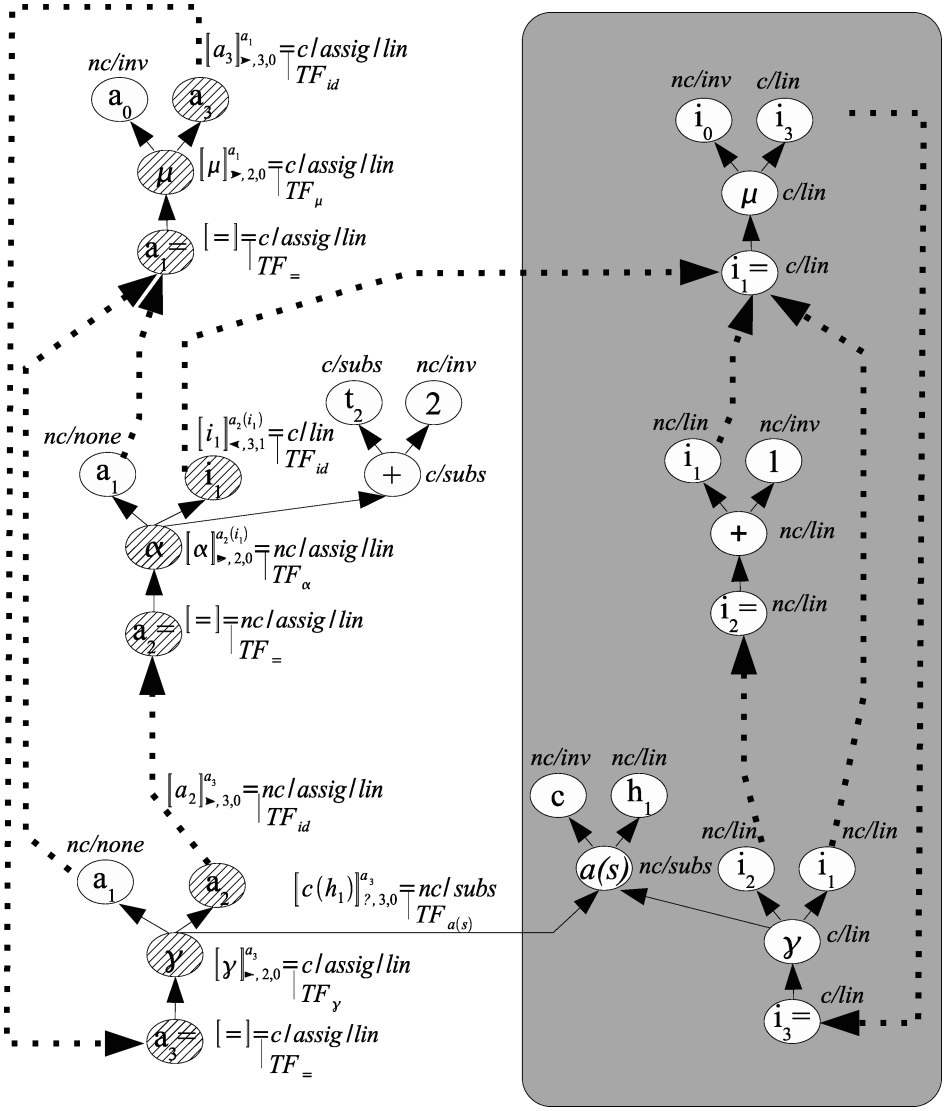


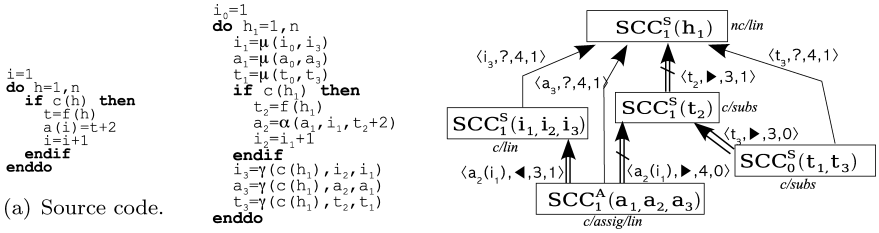
Fig. 10. Classification of $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)$ and $SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)$ from the example of Figure 2. The nodes of the ASTs are annotated with the operator classes computed by the demand-driven SCC classification algorithm. Hatched nodes in the ASTs of $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)$ highlight those operators whose classification is deferred until the classification of $SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)$ is finished.

$SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)$ and $SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)$. The IR used in this stage is the forest of ASTs of the statements of the loop body combined with the use-def chains of the GSA graph and the control flow graph. In Figure 10, ASTs are depicted as directed graphs with solid arrows. For clarity, the control flow graph is not drawn and the root nodes of the ASTs are labeled with the unique left-hand side GSA variable of the statement followed by the assignment operator. The gsa_links are drawn as dotted arrows that establish use-def chains between

different ASTs. The operator classes are displayed above or to the right of the nodes of the AST.

Without loss of generality, suppose that the SCC classification algorithm begins with the analysis of $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)$. The procedure `Classify_SCC()` derives $[\![SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)]\!]$ by calling `Classify()` with the assignment operator of the AST of $\mathtt{a}_1{=}\mu(\mathtt{a}_0,\mathtt{a}_3)$. At the beginning of the post-order traversal, the AST is pushed onto the *stack_of_ASTs* and the classification context for the $=$ operator is initialized to $<>$ (see calls to `Classify()` from `Classify_SCC()` and TF_{id} in Figure 8, and the noncontextual operator classes $[\![=]\!]$ in Figure 10). The first leaf node that is found corresponds to the fetch operator of the variable $\mathtt{a}_0$. Thus, the transfer function TF_{id} uses $\mathtt{a}_0.gsa\_link$ to determine the statement where $\mathtt{a}_0$ is defined. As it is located outside the loop body ($\mathtt{a}_0.gsa\_link$ is not displayed in the figure), $\mathtt{a}_0$ is recognized as a loop invariant and it is assigned the class *nc/inv* (see 1st branch of TF_{id} in Figure 8).

The analysis of the leaf node $\mathtt{a}_3$ is rather different because the AST of the definition statement $\mathtt{a}_3{=}\gamma(\mathtt{c}(\mathtt{h}_1),\mathtt{a}_2,\mathtt{a}_1)$ belongs to the $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)$ under classification. In this case, TF_{id} follows $\mathtt{a}_3.gsa\_link$ to locate the AST of the γ-statement and calls `Classify()` with the corresponding assignment operator (see 2nd branch of TF_{id}). The analysis of $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)$ continues in a similar manner until the classification of the α operator in $\mathtt{a}_2{=}\alpha(\mathtt{a}_1,\mathtt{i}_1,\mathtt{t}_2{+}2)$ is addressed. First, as the ϵ operator of the classification context is a fetch of the array variable `a`, the class of $\mathtt{a}_1$ is set to *nc/none* to indicate that $\mathtt{a}_1$ is defined in a μ-statement of $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)$ whose classification is still in progress (see the termination condition for cycles captured in array SCCs in the 2nd branch of TF_{id} in Figure 8). Second, the operator class $[\![\mathtt{i}_1]\!]^{\mathtt{a}_2(\mathtt{i}_1)}_{\blacktriangleleft,3,1}$ of the left-hand side subscript of the source code statement `a(i)=t+2` is carried out. Note that the classification context is modified to reflect such situation: $\epsilon = \mathtt{a}_2(\mathtt{i}_1)$, $\beta = \blacktriangleleft$ and $il = 1$ indicate that i is the subscript of the array reference in the left-hand side of the statement. The classification of this leaf node $\mathtt{i}_1$ deserves special attention because $\mathtt{i}_1.gsa\_link$ points to the AST of a statement $\mathtt{i}_1{=}\mu(\mathtt{i}_0,\mathtt{i}_3)$ that belongs to a different $SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)$ whose classification is not in progress (i.e., $\mathtt{i}_1{=}\mu(\mathtt{i}_0,\mathtt{i}_3) \notin stack\_of\_ASTs$). In this situation, TF_{id} defers the classification of $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)$ and starts the computation of $[\![SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)]\!]$. The classification of $SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)$ finishes because the variable i is not defined in terms of variables defined in other SCCs (see the shaded region of Figure 10). As a result, the SCC classification algorithm sets $[\![SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)]\!]$ to the class *c/lin* of the $=$ operator of $\mathtt{i}_1{=}\mu(\mathtt{i}_0,\mathtt{i}_3)$, which indicates that a conditional linear induction (see Section 3.2) has been recognized successfully. The initialization and updating of the induction variable is represented using a basic chain of recurrences [Zima 1986] that consists of the initial value of the induction variable, and the function to compute the increment in each loop iteration. During the execution of the SCC classification algorithm, the initial value is gathered from the assignment statement $\mathtt{i}_0{=}1$ where the invariant $\mathtt{i}_0$ of the $\mu(\mathtt{i}_0,\mathtt{i}_3)$ operator is defined (see Figure 2). The increment is determined in TF_+ during the analysis of the operator $\mathtt{i}_1{+}1$, which adds the constant `1` to the value of the induction variable in each iteration where the condition is fulfilled.



(a) Source code.

(b) GSA code.

(c) Kernel graph.

Fig. 11. Kernel graph of the example code of Figure 2.

When the classification of $SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)$ finishes, none of the operator classes in the path from the root of $\mathtt{a}_1{=}\mu(\mathtt{a}_0,\mathtt{a}_3)$ to the leaf node $\mathtt{i}_1$ of $\mathtt{a}_2{=}\alpha(\mathtt{a}_1,\mathtt{i}_1,\mathtt{t}_2{+}2)$ have been computed yet (see hatched nodes in the ASTs of Figure 10). Next, the classification of $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)$ is resumed with the computation of the contextual class $[\![\mathtt{i}_1]\!]^{\mathtt{a}_2(\mathtt{i}_1)}_{\blacktriangleleft,3,1}$. As shown at the end of the 3rd branch of TF_{id} in Figure 8, this contextual class is assigned the class c/lin of $SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)$ and the SCC use-def chain is annotated with the classification context $< \mathtt{a}_2(\mathtt{i}_1), \blacktriangleleft, 3, 1 >$. Once all the children of the α operator have been classified, the transfer function TF_α computes $[\![\alpha]\!]^{\mathtt{a}_2(\mathtt{i}_1)}_{\blacktriangleright,2,0}$ as follows. On the one hand, the class $c/subs$ indicates that the right-hand side $\mathtt{t}_2{+}2$ contains zero references to the array a given by the ϵ operator of the classification context. On the other hand, the class $[\![\mathtt{i}_1]\!]^{\mathtt{a}_2(\mathtt{i}_1)}_{\blacktriangleleft,3,1}{=}c/lin$ characterizes the access pattern of the left-hand side subscript of the source code statement $\mathtt{a(i)=t+2}$. Using this information, TF_α derives the class $nc/assig/lin$ to indicate that the α-statement computes a regular array assignment with linear access pattern (see Section 3.1). Finally, $TF_=$ transfers this class to the root of the AST of $\mathtt{a}_2{=}\alpha(\mathtt{a}_1,\mathtt{i}_1,\mathtt{t}_2{+}2)$. The execution of the SCC classification algorithm continues until the operator classes of all the nodes have been determined. As the conditionality refers to the presence of γ-statements in the SCC (see Definition 4.6), it is TF_γ that changes the SCC class from $nc/assig/lin$ to $c/assig/lin$. At the end, the SCC class $[\![SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)]\!]$ is set to the class $c/assig/lin$ of $\mathtt{a}_1{=}\mu(\mathtt{a}_0,\mathtt{a}_3)$, which represents a *conditional regular array assignment* with linear access pattern.

In order to build the kernel graph, both the SCCs and the SCC use-def chains have to be classified. The kernel graph of the example code of Figure 2 is presented in Figure 11(c), where the source code and the GSA form are also included in order to improve readability. When TF_{id} classifies the fetch operator of a variable defined in a different SCC (3rd branch of TF_{id} in Figure 8), a SCC use-def chain is set in the kernel graph, and it is annotated with the classification context of the fetch operator. Later, once the analysis of the SCCs has finished, `Build_kernel_graph()` executes `Classify_SCC_use_def_chains()`. Three categories are distinguished (see Definition 4.13): conditional, structural and non-structural (denoted as $\rightsquigarrow$, $\Rightarrow$ and $\not\Rightarrow$, respectively). Consider the SCC use-def chain $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3) \rightarrow SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)$ found during the analysis of the case study of Figure 10. As described above, the demand-driven nature of the SCC classification algorithm ensures that when the computation

of $[\![SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)]\!]$ finishes, the class $[\![SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)]\!]$ is already known by the compiler. According to Definition 4.13, $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3) \rightarrow SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)$ is a structural use-def chain (denoted as $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3) \Rightarrow SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)$ in Figure 11(c)) because the access pattern of the class of the array use-SCC (*c*/*assig*/*lin*) and the class of the scalar def-SCC (*c*/*lin*) are both equal to *lin*, and because the annotations of the SCC use-def chain are $\beta = \blacktriangleleft$ and $il \geq 1$ (see Section 4.3.2). The remaining SCC use-def chains with $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)$ as use-SCC are classified as follows: $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3) \rightsquigarrow SCC_1^S(\mathtt{h}_1)$ is conditional because the fetch operator of the loop index $\mathtt{h}_1$ is a part of the condition $\mathtt{c}(\mathtt{h}_1)$ of an if-endif construct (see $\beta =?$ in Figure 11(c)); and $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3) \not\Rightarrow SCC_1^S(\mathtt{t}_2)$ is non-structural as it fulfills neither the properties of structural SCC use-def chains nor of conditional ones.

The execution of the SCC classification algorithm results in the construction of the kernel graph of Figure 11(c), which summarizes the dependences between the simple kernels computed in the loop nest: a *conditional linear induction* (variable i and $[\![SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)]\!]$=*c*/*lin*), a *conditional regular array assignment* with linear access pattern (variable a and $[\![SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)]\!]$=*c*/*assig*/*lin*), and a *conditional scalar assignment* (variable t and $[\![SCC_1^S(\mathtt{t}_2)]\!]$=*c*/*subs*) that, in each loop iteration, sets t to a different value that is not known at compile-time. In addition, the kernel graph contains SCCs (e.g., $SCC_0^S(\mathtt{t}_1, \mathtt{t}_3)$) that arise as a result of building the GSA form of the source code and that are not relevant from the point of view of kernel recognition. The organization of this information in more elaborate kernels is described in detail in the following section.

5. RECOGNITION OF COMPOUND KERNELS

The final goal of the XARK compiler is the recognition of the kernel families included in Table I, which requires the analysis of the dependences between the simple kernels identified by the SCC classification algorithm. As shown in the overview of Figure 2, XARK constructs several high-level IRs on top of the GSA form. On the one hand, the kernel graph, which captures the information retrieved by the SCC classification algorithm, namely, the kernel class that represents the computations of each SCC (i.e., the SCC class) and the classes of SCC use-def chains. And, on the other hand, the *code class*, which summarizes the results of the kernel graph classification algorithm.

The code class consists of a set of kernel classes that represent either the simple kernels captured by the SCC classes of the taxonomy of Figure 6, or the compound kernels recognized through the analysis of the dependences between simple kernels (basically, scalar gated reductions, scalar masks, reinitialized scalar kernels and complex written arrays of Table I). In addition, the code class contains the dependences between kernel classes, which are represented as sets of SCC use-def chains. For illustrative purposes, consider the code class of Figure 2, which consists of the kernel class *c*/*cwa* of the conditional consecutively written array (associated with $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)$ and $SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)$); the kernel class *c*/*subs* of the computation of t (related to $SCC_0^S(\mathtt{t}_1, \mathtt{t}_3)$ and $SCC_1^S(\mathtt{t}_2)$); and the use-def chain *c*/*cwa* → *c*/*subs* given by

```
procedure Classify_kernel_graph()
input:
  Kernel graph
output:
  [[code]]:  Code class
{
  [[code]] = {}
  foreach SCC(x_1,...,x_n) with zero incoming SCC use-def chains {
    [[code]] = Classify_node(SCC(x_1,...,x_n))
  }
}

procedure Classify_node()
input:
  SCC(x_1,...,x_n):  Node of the kernel graph
input/output:
  [[code]]:  Code class
{
  actions = {}
  if( SCC(x_1,...,x_n)∈stack_of_SCCs ) { /* 1st branch:  Detect cycles in the kernel graph */
    [[code]] = Execute_actions(insert_kernel([[SCC(x_1,...,x_n)]],{SCC(x_1,...,x_n)}))
  }
  else {
    push SCC(x_1,...,x_n) onto stack_of_SCCs
    if( SCC(x_1,...,x_n) not visited ) { /* 2nd branch */
      mark SCC(x_1,...,x_n) as visited
      if( SCC(x_1,...,x_n) has zero outgoing SCC use-def chains ) { /* 3rd branch */
        [[code]] = Execute_actions(insert_kernel([[SCC(x_1,...,x_n)]],{SCC(x_1,...,x_n)}))
      }
      else { /* 4th branch */
        for each structural SCC(x_1,...,x_n) ⇒ SCC(y_1,...,y_m) {
          [[code]] = Classify_node(SCC(y_1,...,y_m))
          [[code]] = TF⇒(SCC(x_1,...,x_n) ⇒ SCC(y_1,...,y_m))
        }
        for each conditional SCC(x_1,...,x_n) ⇝ SCC(y_1,...,y_m) {
          [[code]] = Classify_node(SCC(y_1,...,y_m))
          [[code]] = TF⇝(SCC(x_1,...,x_n) ⇝ SCC(y_1,...,y_m))
        }
        for each non-structural SCC(x_1,...,x_n) ⇏ SCC(y_1,...,y_m) {
          [[code]] = Classify_node(SCC(y_1,...,y_m))
          [[code]] = TF⇏(SCC(x_1,...,x_n) ⇏ SCC(y_1,...,y_m))
        }
      }
    }
    pop SCC(x_1,...,x_n) off stack_of_SCCs
  }
}
```

Fig. 12. Pseudocode of the kernel graph classification algorithm.

$SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3) \rightarrow SCC_1^S(\mathtt{t}_2)$. Note that the code class provides a hierarchical description of the source code by capturing the set of SCCs and the SCC use-def chains that lead to the recognition of each kernel class. The rest of this section describes the kernel graph classification algorithm for building the code class and details the case study of Figure 2. More examples of codes can be found in Arenaz [2003].

5.1 Kernel Graph Classification Algorithm

The pseudocodes of Figures 12–14 describe the behavior of the kernel graph classification algorithm. At the top level, `Classify_kernel_graph()` starts a post-order traversal from each SCC with zero incoming SCC use-def chains by calling the recursive procedure `Classify_node()`. When a node $SCC(x_1, \ldots, x_n)$ is visited, the successors $SCC(y_1, \ldots, y_m)$ in the kernel graph that are reached through structural, conditional or non-structural SCC use-def chains are processed in that order (see 4th branch of `Classify_node()` in Figure 12). This

```
procedure TF⇒()
input:
  SCC(x1,...,xn) ⇒ SCC(y1,...,ym): SCC use-def chain of the kernel graph
input/output:
  [[code]]: Code class
{
  [[use_krnl]] = find_kernel_class_in_code_class([[SCC(x1,...,xn)]])
  [[def_krnl]] = find_kernel_class_in_code_class([[SCC(y1,...,ym)]])
  actions = insert_kernel([[SCC(x1,...,xn)]],{SCC(x1,...,xn)})
          && insert_chain(SCC(x1,...,xn) ⇒ SCC(y1,...,ym))
  if( SCC(y1,...,ym) is trivial ) {
    // 1st scenario: Reinitialized scalar kernels
    if( SCC_1^S(x1,...,xn) && [[use_krnl]] is (n)c/lin, (n)c/poly, (n)c/geom, (n)c/reduc or (n)c/map ) {
       || SCC_0^S(x1,...,xn) && [[use_krnl]] is min or max ) {
      if( Test_reinitialized_scalar_kernel() succeeds )
        actions = insert_kernel(reinitialized [[use_krnl]],{SCC(x1,...,xn),SCC(y1,...,ym)})
    }
    // 2nd scenario: Scalar gated kernels
    else if( SCC_0^S(x1,...,xn) && [[use_krnl]] is a candidate min, max or f&s ) {
      if( Test_scalar_gated_kernel() succeeds )
        actions = insert_kernel([[use_krnl]],{SCC(x1,...,xn),SCC(y1,...,ym)})
    }
    else {
      ...
    }
  }
  else { /* Nontrivial SCC(y1,...,ym) */
    // 3rd scenario: Consecutively written arrays
    if( SCC_1^S(y1,...,ym) && [[def_krnl]]=(n)c/lin &&
        SCC_1^A(x1,...,xn) && [[use_krnl]]=(n)c/assig/lin &&
        SCC use-def chain annotation is β = ◄ and il ≥ 1 ) {
      if( Test_consecutively_written_array() succeeds )
        actions = insert_kernel((n)c/cwa,{SCC(x1,...,xn),SCC(y1,...,ym)})
    }
    // 4th scenario: Segmented consecutively written arrays
    else if( SCC_1^S(y1,...,ym) && [[def_krnl]] is reinitialized (n)c/lin &&
             SCC_1^A(x1,...,xn) && [[use_krnl]]=(n)c/assig/lin &&
             SCC use-def chain annotation is β = ◄ and il ≥ 1 ) {
      if( Test_segmented_consecutively_written_array() succeeds )
        actions = insert_kernel(segmented (n)c/cwa,{SCC(x1,...,xn),SCC(y1,...,ym)})
    }
    // 5th scenario: Virtual SCCs
    else if( SCC_0^S(x1,...,xn) consists of μ-statements only )
      find kernel_class of SCC(y1,...,ym) in [[code]]
      actions = attach_SCC(SCC_0^S(x1,...,xn),kernel_class)
    }
    else {
      ...
    }
  }
  [[code]] = Execute_actions(actions)
}

procedure TF⇝()
input:
  SCC(x1,...,xn) ⇝ SCC(y1,...,ym): SCC use-def chain of the kernel graph
input/output:
  [[code]]: Code class
{
  [[use_krnl]] = find_kernel_class_in_code_class([[SCC_1^S(x1,...,xn)]])
  [[def_krnl]] = find_kernel_class_in_code_class([[SCC_1^S(y1,...,ym)]])
  actions = insert_kernel([[SCC(x1,...,xn)]],{SCC(x1,...,xn)})
          && insert_chain(SCC(x1,...,xn) ⇝ SCC(y1,...,ym))
  // 1st scenario: Scalar minimum/maximum with index reduction
  if( SCC_0^S(y1,...,ym) && [[def_krnl]] is min or max ) {
    /* The analysis of structural, conditional and nonstructural SCC use-def chains in that */
    /* order guarantees that the scalar min/max reduction has already been recognized in TF⇒ */
    if( Test_scalar_minimum/maximum_with_index_reduction() )
      actions = insert_kernel(min/max with index,{SCC(x1,...,xn),SCC(y1,...,ym)})
    }
    else {
      ...
    }
  }
  [[code]] = Execute_actions(actions)
}
```

Fig. 13. Pseudocode of transfer functions for structural and conditional SCC use-def chains.

```
procedure Execute_actions()
input:
  actions: List of actions
input/output:
  [[code]]: Code class
{
  for each action in actions {
    switch( action ) {
    insert_kernel(new_kernel_class,set_of_SCCs):
      /* Check that kernels that are more complex than the new one have not been recognized */
      if( ∄ kernel classes whose set of SCCs ⊃ set_of_SCCs ) {
        /* Remove kernel classes that are simpler than the new one */
        for each kernel_class whose set of SCCs ⊂ set_of_SCCs {
          for each SCC(x_1,...,x_n) in the set of SCCs of kernel_class {
            remove dependences of [[code]] with SCC(x_1,...,x_n) as use-SCC or def-SCC
          }
          remove kernel_class from [[code]]
        }
        /* Insert the new kernel class */
        if( ∄ kernel class whose set of SCCs = set_of_SCCs ) {
          insert new_kernel_class in [[code]]
        }
      }
      break
    insert_chain(SCC(x_1,...,x_n) → SCC(y_1,...,y_m)):
      if( SCC(x_1,...,x_n) and SCC(y_1,...,y_m) are associated with different kernel classes ) {
        if( ∄ dependence between such kernels classes ) {
          insert new dependence between the kernel classes in [[code]]
        }
        add SCC(x_1,...,x_n) → SCC(y_1,...,y_m) to the corresponding set of SCC use-def chains
      }
      break
    attach_SCC(SCC,kernel_class):
      insert SCC in the set of SCCs of kernel_class
      break
    }
  }
}
```

Fig. 14. Pseudocode of the algorithm for the management of the code class.

is because structural and conditional chains have been defined to capture the scenarios that typically represent the computation of compound kernels (see the description of an example scenario in Section 2.3). When the classification of a successor $SCC(y_1, \ldots, y_m)$ finishes, the SCC use-def chain $SCC(x_1, \ldots, x_n) \rightarrow SCC(y_1, \ldots, y_m)$ is analyzed by means of the corresponding transfer function: $TF_{\Rightarrow}$, $TF_{\rightsquigarrow}$ and $TF_{\not\Rightarrow}$ for structural, conditional and non-structural chains, respectively. Like in the SCC classification algorithm, the recognition capabilities of the compiler are mainly encoded in the transfer functions. As shown in the pseudocode of Figure 13, the transfer functions analyze the information attached to the SCC use-def chain of the kernel graph, compute the actions to be carried out in order to determine the code class $[\![code]\!]$, and call `Execute_actions()` to actually modify $[\![code]\!]$. The procedure `Execute_actions()` of Figure 14 distinguishes three types of actions (insert a kernel class, insert a chain between kernel classes, and attach an SCC to a kernel class), which will be described later in Section 5.1.1. Note that the nodes of the kernel graph are visited only once in order to avoid redundant computations (see 2nd branch of `Classify_node()`). Furthermore, the termination of the kernel graph classification algorithm is assured by means of a *stack_of_SCCs* that enables the detection of mutually dependent simple kernels, which may arise because the SCC classification algorithm ignores

conditional use-def chains during SCC search (see Definition 4.2 and 1st branch of `Classify_node()` in Figure 12).

The recognition of the scenarios that characterize compound kernels is carried out in $TF_{\Rightarrow}$ and $TF_{\rightsquigarrow}$. Figure 13 presents fragments of both transfer functions that include some scenarios that are frequently found in real codes. The transfer function $TF_{\nRightarrow}$ is not shown because it just inserts in the code class two simple kernels $[\![SCC(x_1, \ldots, x_n)]\!]$ and $[\![SCC(y_1, \ldots, y_m)]\!]$, as well as the corresponding dependence $SCC(x_1, \ldots, x_n) \rightarrow SCC(y_1, \ldots, y_m)$. Given an SCC use-def chain $SCC(x_1, \ldots, x_n) \rightarrow SCC(y_1, \ldots, y_m)$, the information available in the kernel graph that defines a scenario is: first, the properties of $SCC(x_1, \ldots, x_n)$ and $SCC(y_1, \ldots, y_m)$ (see Definitions 4.3–4.8; for example, cardinality, scalar/array); second, the kernel classes $[\![use\_krnl]\!]$ and $[\![def\_krnl]\!]$ computed at the beginning of each transfer function for $SCC(x_1, \ldots, x_n)$ and $SCC(y_1, \ldots, y_m)$, respectively; third, the class of the SCC use-def chain (see Definition 4.13, namely, conditional, structural or non-structural); and fourth, the classification context annotated in the SCC use-def chain. The computation of $[\![use\_krnl]\!]$ (or $[\![def\_krnl]\!]$) is as follows. In `find_kernel_class_in_code_class()`, $[\![code]\!]$ is searched for a kernel class that represents the computations captured by $SCC(x_1, \ldots, x_n)$ (or $SCC(y_1, \ldots, y_m)$). In case of success, $[\![use\_krnl]\!]$ is set to such kernel class to reflect that, during the analysis of the kernel graph, a compound kernel that includes the computation of $SCC(x_1, \ldots, x_n)$ has already been recognized. In case of failure, $[\![use\_krnl]\!]$ is set to $[\![SCC(x_1, \ldots, x_n)]\!]$ to attempt the detection of a new compound kernel. The behavior of $TF_{\Rightarrow}$ and $TF_{\rightsquigarrow}$ is as follows. When a scenario is found, an appropriate compile-time test that proves the existence of a compound kernel is performed. If the test succeeds, an action to insert the class of the compound kernel in $[\![code]\!]$ is carried out. Otherwise, two default actions are executed: insert the class of the simple kernel represented by $SCC(x_1, \ldots, x_n)$, and insert the chain $SCC(x_1, \ldots, x_n) \rightarrow SCC(y_1, \ldots, y_m)$. The post-order traversal of the kernel graph preserves the correctness of the code class by assuring that a kernel class associated with $SCC(y_1, \ldots, y_m)$ has been inserted before.

For illustrative purposes, consider the code of Figure 7 for the computation of the minimum with index of each row of a sparse matrix. As described in Section 4.2.3, the SCC classification algorithm checks the condition `a(f(h))<tm` and recognizes a *minimum* kernel that is annotated as a *candidate* because the condition `a(f(h))<tm` and the definition source code statement `tm=a(f(h))` are available in different scalar SCCs ($SCC_0^S(\texttt{tm}_3, \texttt{tm}_5)$ and $SCC_1^S(\texttt{tm}_4)$, respectively). This situation would be reflected in the kernel graph as a structural SCC use-def chain $SCC_0^S(\texttt{tm}_3, \texttt{tm}_5) \Rightarrow SCC_1^S(\texttt{tm}_4)$ where the def-SCC is trivial and the class of the use-SCC is a *candidate minimum*. As these properties match the 2nd scenario of $TF_{\Rightarrow}$ in Figure 13, the procedure `Test_scalar_gated_kernel()` is executed in order to assure that the condition matches a `<` operator where the operands on the left and on the right are, respectively, the right-hand side `a(f(h))` and the left-hand side `tm` of the definition scalar statement. As a result, a *minimum* kernel class associated with $SCC_0^S(\texttt{tm}_3, \texttt{tm}_5)$ and $SCC_1^S(\texttt{tm}_4)$ would be inserted in the code class. Later in the analysis of the kernel graph, a post-order traversal started from the node

$SCC_0^S(\mathtt{tl}_3, \mathtt{tl}_5)$ with zero incoming SCC use-def chains would find a conditional use-def chain $SCC_0^S(\mathtt{tl}_3, \mathtt{tl}_5) \rightsquigarrow SCC_0^S(\mathtt{tm}_3, \mathtt{tm}_5)$. In order to recognize the minimum with index kernel (see 1st scenario of $TF_{\rightsquigarrow}$ in Figure 13), the compiler must be aware that a minimum kernel has already been recognized. Thus, searching the code class for a kernel class associated to $SCC_0^S(\mathtt{tm}_3, \mathtt{tm}_5)$ would result in setting $[\![def\_krnl]\!]$=*minimum* and preventing the use of the incorrect class $[\![def\_krnl]\!]$=*candidate minimum*, which would lead the compiler to fail in the recognition of the scenario of the minimum with index reduction.

5.1.1 *Management of the Code Class.* The procedure `Execute_actions()` presented in Figure 14 distinguishes three actions: *insert_kernel* to insert a kernel class, *insert_chain* to insert a chain between kernel classes, and $attach\_SCC$ to associate an SCC with an existing kernel class.

The first action is as follows. Two parameters are considered: the new kernel class and the corresponding set of SCCs of the kernel graph (*new_kernel_class* and *set_of_SCCs* in Figure 14, respectively). The *new_kernel_class* is ignored in two cases. First, when $[\![code]\!]$ contains another kernel class whose set of SCCs is a superset of *set_of_SCCs*, which indicates that the new kernel is simpler than another one that has already been recognized. And second, when there is a kernel class in $[\![code]\!]$ attached to the same set of SCCs, meaning that *new_kernel_class* has already been recognized through the analysis of another structural or conditional SCC use-def chain. Except in these two cases, the *new_kernel_class*, and its corresponding *set_of_SCCs*, is inserted in $[\![code]\!]$ after removing the kernel classes whose set of SCCs is a subset of *set_of_SCCs*, as they capture kernels that are simpler than the new one.

The second action is *insert_chain*. The dependences between kernel classes are represented as sets of SCC use-def chains that summarize all the use-def chains between SCCs of the kernel classes as one dependence in the code class. Let $SCC(x_1, \ldots, x_n) \rightarrow SCC(y_1, \ldots, y_m)$ be an SCC use-def chain that links the kernel classes associated with $SCC(x_1, \ldots, x_n)$ and $SCC(y_1, \ldots, y_m)$, respectively. A new dependence is inserted in $[\![code]\!]$ if two conditions are fulfilled: first, $SCC(x_1, \ldots, x_n)$ and $SCC(y_1, \ldots, y_m)$ are included in the sets of SCCs of two different kernel classes; and second, $[\![code]\!]$ does not contain any dependence between such kernel classes. Note that $SCC(x_1, \ldots, x_n) \rightarrow SCC(y_1, \ldots, y_m)$ is added to the set of SCC use-def chains of the new dependence or of the corresponding existing dependence. Furthermore, SCC use-def chains that characterize scenarios of compound kernels are implicitly represented in the kernel classes of compound kernels.

The actions described above are the core of the algorithm for the manipulation of the code class. The third action, $attach\_SCC$, is devoted to handle the virtual SCCs (see Definition 4.8) that appear in the GSA graph. Consider the example of Figure 7. As the minimum `tm` is computed in the inner loop $do_{\mathtt{h}}$, a virtual $SCC_0^S(\mathtt{tm}_1)$ represents the flow of values of `tm` in the outermost loop $do_{\mathtt{row}}$. Thus, during the post-order traversal started from $SCC_0^S(\mathtt{tm}_1)$, the structural SCC use-def chain $SCC_0^S(\mathtt{tm}_1) \Rightarrow SCC_1^S(\mathtt{tm}_3, \mathtt{tm}_5)$ leads the compiler to attach $SCC_0^S(\mathtt{tm}_1)$ to an existing kernel, the scalar minimum reduction represented by

Procedure call	Actions	Code class
Post-order traversal starting from $SCC_1^A(a_1, a_2, a_3)$		
1. Classify_node($SCC_1^A(a_1, a_2, a_3)$)		
2. Classify_node($SCC_1^S(i_1, i_2, i_3)$)		
3. Classify_node($SCC_1^S(h_1)$)		
4. Execute_actions()	I:*nc/lin*:{$SCC_1^S(h_1)$}:✔	⟦*nc/lin*⟧
5. $TF_{\rightsquigarrow}(SCC_1^S(i_1, i_2, i_3) \rightsquigarrow SCC_1^S(h_1))$		
6. Execute_actions()	I:*c/lin*:{$SCC_1^S(i_1, i_2, i_3)$}:✔	⟦*c/lin,nc/lin*⟧
	I:$SCC_1^S(i_1, i_2, i_3) \rightsquigarrow SCC_1^S(h_1)$:✔	⟦*c/lin* → *nc/lin*⟧
7. $TF_{\Rightarrow}(SCC_1^A(a_1, a_2, a_3) \Rightarrow SCC_1^S(i_1, i_2, i_3))$		
8. Execute_actions()	I:*c/cwa*:{$SCC_1^A(a_1, a_2, a_3)$, $SCC_1^S(i_1, i_2, i_3)$}:✔	⟦*c/cwa* → *nc/lin*⟧
9. Classify_node($SCC_1^S(h_1)$)		
10. $TF_{\rightsquigarrow}(SCC_1^A(a_1, a_2, a_3) \rightsquigarrow SCC_1^S(h_1))$		
11. Execute_actions()	I:*c/assig/lin*:{$SCC_1^A(a_1, a_2, a_3)$}:✘	⟦*c/cwa* → *nc/lin*⟧
	I:$SCC_1^A(a_1, a_2, a_3) \rightsquigarrow SCC_1^S(h_1)$:✔	⟦*c/cwa* → *nc/lin*⟧
12. Classify_node($SCC_1^S(t_2)$)		
13. Classify_node($SCC_1^S(h_1)$)		
14. $TF_{\not\Rightarrow}(SCC_1^S(t_2) \not\Rightarrow SCC_1^S(h_1))$		
15. Execute_actions()	I:*c/subs*:{$SCC_1^S(t_2)$}:✔	⟦*c/cwa* → *nc/lin*, *c/subs*⟧
	I:$SCC_1^S(t_2) \not\Rightarrow SCC_1^S(h_1)$:✔	⟦*c/cwa* → *nc/lin*, *c/subs* → *nc/lin*⟧
16. $TF_{\not\Rightarrow}(SCC_1^A(a_1, a_2, a_3) \not\Rightarrow SCC_1^S(t_2))$		
17. Execute_actions()	I:*c/assig/lin*:{$SCC_1^A(a_1, a_2, a_3)$}:✘	⟦*c/cwa* → *nc/lin*, *c/subs* → *nc/lin*⟧
	I:$SCC_1^A(a_1, a_2, a_3) \not\Rightarrow SCC_1^S(t_2)$:✔	⟦*c/cwa* → *nc/lin*, *c/subs* → *nc/lin*, *c/cwa* → *c/subs*⟧
Post-order traversal starting from $SCC_0^S(t_1, t_3)$		
18. Classify_node($SCC_0^S(t_1, t_3)$)		
19. Classify_node($SCC_1^S(t_2)$)		
20. $TF_{\Rightarrow}(SCC_0^S(t_1, t_3) \Rightarrow SCC_1^S(t_2))$		
21. Execute_actions()	A:$SCC_0^S(t_1, t_3)$:*c/subs*:✔	⟦*c/cwa* → *nc/lin*, *c/subs* → *nc/lin*, *c/cwa* → *c/subs*⟧
22. Classify_node($SCC_1^S(h_1)$)		
23. $TF_{\rightsquigarrow}(SCC_0^S(t_1, t_3) \rightsquigarrow SCC_1^S(h_1))$		
24. Execute_actions()	I:*c/subs*:{$SCC_1^S(h_1)$}:✘	⟦*c/cwa* → *nc/lin*, *c/subs* → *nc/lin*, *c/cwa* → *c/subs*⟧
	I:$SCC_0^S(t_1, t_3) \rightsquigarrow SCC_1^S(h_1)$:✔	⟦*c/cwa* → *nc/lin*, *c/subs* → *nc/lin*, *c/cwa* → *c/subs*⟧

Fig. 15. Computation of the code class of the kernel graph of Figure 11(c). Step-by-step execution of the kernel graph classification algorithm.

$SCC_0^S(tm_1)$ and $SCC_1^S(tm_3, tm_5)$ that was previously recognized through the 5th scenario of $TF_{\Rightarrow}$ (see Figure 13).

The next section presents a case study that describes the kernel graph classification algorithm in detail. The computation of the code class of the example of Figure 2 is analyzed step by step, focusing on the recognition of the scenarios and on the execution of the transfer functions.

5.2 Case Study

The kernel graph of Figure 11(c) exhibits the simple kernels found in the source code of our running example, as well as the dependence relationships between those simple kernels. Figure 15 summarizes the step-by-step execution of the post-order traversal of the two SCCs with zero incoming SCC use-def chains: $SCC_1^A(a_1, a_2, a_3)$ from steps 1 to 17, and $SCC_0^S(t_1, t_3)$ from steps 18 to 24. For each step, the procedure call under execution, the actions carried out on the code class, and the contents of the code class are presented. Procedure calls are indented to exhibit the call trace of the algorithm. Due to space limitations, the actions are encoded in three pieces of information (separated by colons)

as follows. On the one hand, the symbol I represents both *insert_kernel* and *insert_chain* actions. In the first case, the new kernel class and the set of SCCs are indicated. In the second case, the SCC use-def chain that links two kernel classes is specified. On the other hand, the symbol A denotes *attach_SCC*, for which the new SCC and the existing kernel class are detailed. The success (✔) or the failure (✘) in the execution of an action is also indicated.

Without loss of generality, suppose that `Classify_kernel_graph()` initializes the code class, and starts a post-order traversal from $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)$ by calling `Classify_node(`$SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)$`)`. In order to recognize compound kernels as soon as possible, the structural chain $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3) \Rightarrow SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)$ is followed and `Classify_node(`$SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)$`)` is executed. The post-order traversal continues and the node $SCC_1^S(\mathtt{h}_1)$ is reached through the conditional SCC use-def chain $SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3) \rightsquigarrow SCC_1^S(\mathtt{h}_1)$. As $SCC_1^S(\mathtt{h}_1)$ has zero outgoing SCC use-def chains, `Classify_node(`$SCC_1^S(\mathtt{h}_1)$`)` inserts the kernel class *nc/lin* that captures the loop index `h` by calling `Execute_actions(insert_kernel(`*nc/lin*`,{`$SCC_1^S(\mathtt{h}_1)$`}))` (see 3rd branch of `Classify_node()` in Figure 12, and step 4 in Figure 15). The last step carried out by the procedure call `Classify_node(`$SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)$`)` is the analysis of $SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3) \rightsquigarrow SCC_1^S(\mathtt{h}_1)$. $TF_{\rightsquigarrow}$ will execute the default actions because there is not any scenario that matches the properties of this conditional SCC use-def chain. Thus, the class of the use $SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)$ is inserted in $[\![code]\!]$ (see step 6 in Figure 15). In addition, a dependence between the conditional induction variable `i` represented by $SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)$ and the loop index `h` captured by $SCC_1^S(\mathtt{h}_1)$ is inserted in $[\![code]\!]$, and $SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3) \rightsquigarrow SCC_1^S(\mathtt{h}_1)$ is added to the corresponding set of SCC use-def chains. The resulting code class is $[\![c/lin \rightarrow nc/lin]\!]$.

Next, the analysis of the structural chain $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3) \Rightarrow SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)$ is addressed. This SCC use-def chain has the properties of the 3rd scenario of $TF_{\Rightarrow}$ in Figure 13: the class of the scalar def-SCC is *c/lin*, the class of the array use-SCC is *c/assig/lin*, and the annotations $\beta = \blacktriangleleft$ and $il \geq 1$ indicate that the induction variable `i` is referenced in the subscript of the left-hand side of the statement `a(i)=t+2` represented by $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)$. In order to confirm the existence of this compound kernel, `Test_consecutively_written_array()` runs the following compile-time test proposed in Lin and Padua [1998]: first, it checks that all the operations on the induction variable `i` are increments (or decrements) of one unit; and second, it checks that every time an array entry `a(i)` is written, the induction variable `i` is updated. Within the XARK compiler, the first constraint is easily checked using the step size of the induction (step size `1` in the example), which is represented in the basic chain of recurrences computed during the execution of the SCC classification algorithm (see Section 4.4). The second constraint is also fulfilled as the statements $\mathtt{i}_2=\mathtt{i}_1+1$ and $\mathtt{a}_2=\alpha(\mathtt{a}_1,\mathtt{i}_1,\mathtt{t}_2+2)$ belong to the same basic block of the control flow graph of the loop. Consequently, the procedure `Test_consecutively_written_array()` succeeds and the compiler executes the action `insert_kernel(`*c/cwa*`,{`$SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)$`,`$SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)$`})` (see step 8 in Figure 15). Thus, the procedure `Execute_actions()` proceeds as follows. As the code class $[\![c/lin \rightarrow nc/lin]\!]$ contains the conditional induction `i` that is a

part of the conditional consecutively written array, c/lin is removed and c/cwa is inserted in $[\![code]\!]$. Such situation is detected by checking that the SCCs attached to the new class, $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)$ and $SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)$, are a superset of the $SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)$ associated with the conditional induction. In addition, the use-def chain $c/lin \rightarrow nc/lin$ introduced by $SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3) \rightsquigarrow SCC_1^S(\mathtt{h}_1)$ now defines a dependence relationship $c/cwa \rightarrow nc/lin$. As a result, the code class is $[\![c/cwa \rightarrow nc/lin]\!]$. Note that other detection techniques such as monotonic evolution [Wu et al. 2001] can also be used to recognize consecutively written arrays. Consequently, the scenarios are a mechanism that enables the use of XARK as a unified framework where compiler techniques with different goals can be integrated and executed selectively based on the characteristics of the source code.

The post-order traversal started from $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)$ continues, and the conditional SCC use-def chain $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3) \rightsquigarrow SCC_1^S(\mathtt{h}_1)$ is analyzed. On the one hand, $\mathtt{Classify\_node}(SCC_1^S(\mathtt{h}_1))$ does not change the contents of $[\![code]\!]$ because the node $SCC_1^S(\mathtt{h}_1)$ has already been visited (see 2nd branch in Figure 12). On the other hand, the properties of the SCC use-def chain do not match any scenario of $TF_{\rightsquigarrow}$ and thus the default actions $\mathtt{insert\_kernel}(c/assig/lin, \{SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)\})$ and $\mathtt{insert\_chain}(SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3) \not\Rightarrow SCC_1^S(\mathtt{h}_1))$ are executed. The first action ignores the new class $c/assig/lin$ because it represents a kernel that is a part of the consecutively written array c/cwa already included in $[\![code]\!]$. The compiler detects this situation by checking that the SCCs attached to c/cwa, $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)$ and $SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)$, are a superset of the $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)$ associated with the new kernel class. Finally, the SCC use-def chain of the second action is added to the set of SCC use-def chains between the kernel classes of the consecutively written array and the loop index (see step 11 of Figure 15).

The classification of the remaining chain $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3) \not\Rightarrow SCC_1^S(\mathtt{t}_2)$ is summarized in steps 12–17 of Figure 15. Thus, the analysis of $SCC_1^S(\mathtt{t}_2) \not\Rightarrow SCC_1^S(\mathtt{h}_1)$ leads to the recognition of a new kernel class by running the default actions of $TF_{\not\Rightarrow}$: insert the kernel class $c/subs$ of the use $SCC_1^S(\mathtt{t}_2)$ and its dependence with the loop index $\mathtt{h}$. The resulting code class is $[\![c/cwa \rightarrow nc/lin, c/subs \rightarrow nc/lin]\!]$ (see step 15 of Figure 15). The last step of the post-order traversal started from $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)$ is the execution of $TF_{\not\Rightarrow}$ for $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3) \not\Rightarrow SCC_1^S(\mathtt{t}_2)$. As shown in step 17 of Figure 15, the compiler does not find any known scenario and executes the default actions of $TF_{\not\Rightarrow}$. First, the compiler attempts to insert the classes of the use-SCC and the def-SCC in the code class. However, in this case the action fails because the regular array assignment represented by $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)$ is a part of the consecutively written array included in the the code class. And second, a dependence relationship between the kernel classes c/cwa and $c/subs$ defined by $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3) \rightarrow SCC_1^S(\mathtt{t}_2)$ is inserted in $[\![code]\!]$. The resulting code class is $[\![c/cwa \rightarrow nc/lin, c/subs \rightarrow nc/lin, c/cwa \rightarrow c/subs]\!]$.

In order to complete the analysis of the whole kernel graph, the kernel graph classification algorithm starts a post-order traversal from $SCC_0^S(\mathtt{t}_1, \mathtt{t}_3)$. Two SCC use-def chains are found. On the one hand, $SCC_0^S(\mathtt{t}_1, \mathtt{t}_3) \Rightarrow SCC_1^S(\mathtt{t}_2)$

Table II. Summary of Characteristics of the Benchmark Suite

	SPEC	Perfect	SparsKit-II	PLTMG	Totals
#Routines	273	608	103	258	1242
#Code lines	53173	60136	8286	27530	149125
#Loops analyzed	769	1245	293	651	2958
#Loops recognized	609	955	224	502	2290
%Loops recognized	79%	82%	76%	77%	77%

points to a node that has already been visited. Thus, $TF_{\Rightarrow}$ is applied and the virtual $SCC_0^S(\mathtt{t}_1, \mathtt{t}_3)$ is attached to the kernel class $c/subs$ of $[\![code]\!]$ (see 5th scenario of $TF_{\Rightarrow}$ in Figure 13 and step 21 in Figure 15). The last step of the algorithm addresses the analysis of the conditional SCC use-def chain $SCC_0^S(\mathtt{t}_1, \mathtt{t}_3) \rightsquigarrow SCC_1^S(\mathtt{h}_1)$, which executes the default actions and adds the SCC use-def chain to the corresponding dependence in $[\![code]\!]$ (see step 24 in Figure 15).

The code class $[\![c/cwa \rightarrow nc/lin, c/subs \rightarrow nc/lin, c/cwa \rightarrow c/subs]\!]$ computed by the kernel graph classification algorithm concisely represents that the temporary variable `t` (captured by $SCC_1^S(\mathtt{t}_2)$ and $SCC_0^S(\mathtt{t}_1, \mathtt{t}_3)$) is used in the consecutively written array `a` (captured by $SCC_1^A(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)$ and $SCC_1^S(\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)$) during the execution of the loop $do_{\mathtt{h}}$ of Figure 2. Note that the structural SCC use-def chains of the kernel graph are intrinsically represented in the kernel classes c/cwa and $c/subs$, while the non-structural and the conditional SCC use-def chain exhibit the dependence relationships between `a`, `t` and `h`. Overall, the code class provides a hierarchical description of the loop body as a set of kernels and a set of dependences between these kernels that abstracts the implementation details of the source code. Thus, other compiler techniques can benefit from the information captured in the code class. As an example of the potential of this representation, Arenaz et al. [2004] describe how the code class meets the information requirements of several source-to-source code transformations in the scope of a parallelizing compiler.

6. EXPERIMENTAL RESULTS

The XARK compiler framework has been developed on top of the Polaris intermediate representation [Blume et al. 1996] and includes all the stages shown in Figure 2 (Polaris provides a translator of Fortran77 code into GSA form). Four benchmark suites have been used in the experiments: the Fortran routines included in SPEC CPU2000 [SPEC], the Perfect benchmarks [Berry et al. 1989], the SparsKit-II library [Saad 1994] and the PLTMG (Piecewise Linear Triangle Multi-Grid) code [Bank 2007]. SPEC and Perfect are well-known benchmarks that have been extensively used in the literature. SparsKit-II and PLTMG have been selected because their source codes contain plenty of irregular computations that cover the typical kernels found in full-scale applications. Table II shows the size of the benchmarks in terms of number of routines and number of code lines, and presents the percentage of loops recognized successfully by the XARK compiler.

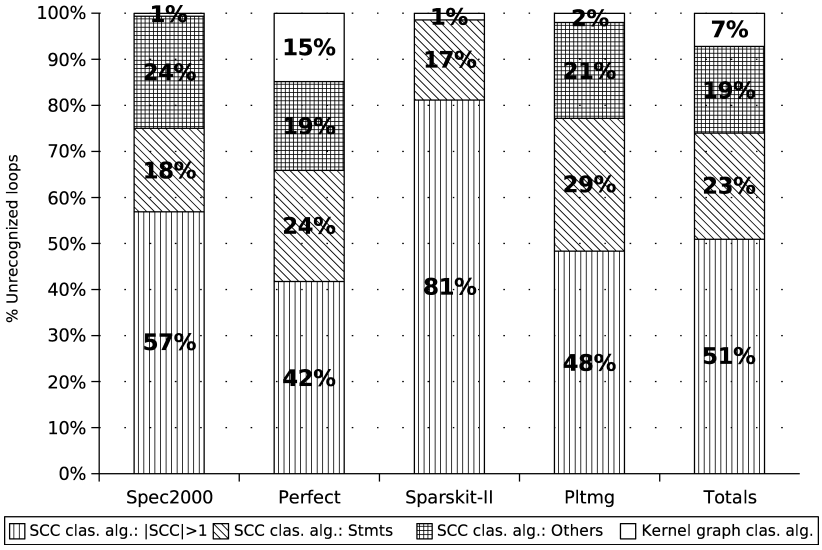


Fig. 16. Failure reasons of the XARK compiler.

The automatic recognition of kernels is the basis for a variety of techniques used in restructuring, parallelizing and optimizing compilers. Thus, Section 6.1 shows statistics about the different kernel families found in SPEC, Perfect, SparsKit-II and PLTMG. Typical applications of this information are induction variable substitution and the replacement of the code represented by a kernel with a platform-optimized version of that code, for instance, the parallel computation of an irregular reduction or a consecutively written array. Section 6.2 presents a characterization of array accesses at several levels of indirection as this information is needed by different compiler techniques. Thus, many data-dependence tests fail in the presence of nonlinear and subscripted subscripts at indirection level one. Another example is cache behavior analysis, which requires the characterization of the subscripts of all the references to an array variable.

6.1 Recognition Results

The effectiveness of XARK has been measured in terms of the percentage of recognized loops, that is, loops whose body is represented by a kernel graph that does not contain any unknown kernel classes. The last row of Table II shows that the percentage of success is 77% on average, ranging from 76% in SparsKit-II up to 79% in SPEC.

A breakdown of the reasons why the recognition process fails is presented in Figure 16. From a total of 668 unrecognized loops, 93% contain simple kernels that are not recognized by the SCC classification algorithm because of (1) the existence of SCCs of cardinality greater than one (*SCC clas. alg.:* $|SCC| > 1$ in Figure 16), (2) the presence of SCCs with statements that belong to different kernel classes (*SCC clas. alg.: Stmts*), and (3) several failure reasons that are not

very frequent in the benchmarks (*SCC clas. alg.: Others*). The remaining 7% are loops where the kernel graph classification algorithm either detects unknown scenarios or it runs compile-time tests that do not lead to the recognition of known compound kernels (*Kernel graph clas. alg.* in Figure 16).

The SCC classification algorithm focuses on the analysis of SCCs with cardinality zero and one. Thus, the primary failure reason is *SCC clas. alg.:* $|SCC| > 1$, which prevents the characterization of the computations carried out in 51% of the 668 unrecognized loops. Note that this limitation of the recognition algorithm affects 81% of the unrecognized loops in SparsKit-II. A preliminary manual analysis revealed that SCCs with cardinality greater than one represent, for instance, swap operations between scalars and between array elements, mutual induction or mutual array recurrences [Redon and Feautrier 1993; Zhang and D'Hollander 1994; Pinter and Pinter 1994]. From the loops that only contain SCCs of cardinality zero and one, *SCC clas. alg.: Stmts* indicates that 23% of loops are not recognized successfully because at least one SCC is composed of statements (i.e., assignment operator = in procedure `Classify()` of Figure 8) that are assigned different classes by the SCC classification algorithm. For instance, such SCCs capture a combination of array assignments and array reductions with regular and irregular access patterns, as well as a complex control flow. Finally, the remaining 19% corresponds to loops that contain, for instance, infrequently used Fortran operators or variations in the implementation of known kernels whose recognition would require the tuning of the transfer functions of the SCC and kernel graph classification algorithms.

The analysis of the code classes that characterize the loops of the benchmark suites revealed that, although real codes contain a great variety of different code classes, all of them are built on top of a small set of kernels. Table III summarizes the number of kernels found in SPEC, Perfect, SparsKit-II and PLTMG, the last column showing the results for the four benchmark suites. The eight kernel families distinguished in Table I are considered: assignments (covering 66% of the kernels), inductions (9%), maps (0, 5%), reductions (10%), masks (0, 5%), array recurrences (3%), reinitialized kernels (1%), and complex written arrays (10%). The vast majority of kernels (65%) are scalar assignments and regular array assignments. Inductions and reductions, which have been the focus of an intensive research activity during the last decade, cover up to 19% of the recognized kernels. More specifically, the benchmarks contain 730 linear inductions, 76 polynomial inductions, 162 geometric inductions, 1047 non-gated reductions and 18 gated reductions (16 scalar minimum/maximum reductions - 5 of them with index -, and 2 regular array reductions). Note that these numbers do not include the inductions and reductions that appear as a part of the 11% of reinitialized kernels (102 linear inductions, 9 scalar maps, 51 non-gated reductions and 1 scalar minimum/maximum reduction) and complex written arrays. Finally, the remaining 5% of recognized kernels corresponds to irregular array assignments, maps, masks and array recurrences. Despite this low percentage, the recognition of these kernels is necessary in order to fully characterize real applications. It should be noted that the experiments have led to find new kernels that have not been studied in the literature, in particular, array maps, irregular array recurrences, consecutively reduced (and

Table III. Collection of Kernel Families Recognized by XARK

Kernel Family	SPEC	Perfect	SparsKit-II	PLTMG	Total
Assignments	3213	2568	396	1026	7203
scalar assignment	2008	1772	196	503	4479
regular array assignment	1201	789	129	456	2575
irregular array assignment	4	7	71	67	149
Inductions	170	617	38	143	968
linear induction	100	494	36	100	730
polynomial induction	5	66	0	5	76
geometric induction	65	57	2	38	162
Maps	0	0	0	46	46
scalar map	0	0	0	44	44
regular array map	0	0	0	2	2
Reductions	350	427	67	221	1065
scalar reduction	34	146	21	105	306
regular array reduction	309	259	10	86	664
irregular array reduction	4	19	30	24	77
scalar minimum/maximum reduction	1	3	6	6	16
regular array minimum/maximum reduction	2	0	0	0	2
Masks	6	2	8	24	40
scalar find&set	2	1	2	13	18
regular array find&set	4	1	4	6	15
irregular array find&set	0	0	2	5	7
Array recurrences	142	149	29	66	386
regular array recurrence	142	139	25	51	357
irregular array recurrence	0	10	4	15	29
Reinitialized kernels	15	123	12	13	163
induction	7	91	4	0	102
map	0	0	0	9	9
reduction	8	32	8	4	52
Complex written arrays	46	628	240	133	1047
consecutively written array	22	322	229	109	682
consecutively reduced array	24	241	0	0	265
consecutively recurrenced array	0	8	3	24	35
segmented consecutively written array	0	57	8	0	65
segmented consecutively reduced array	0	3	0	0	3

recurrenced) arrays and segmented consecutively reduced (and recurrenced) arrays (see collection of kernels in Section 3).

Experiments about the classes of kernels that appear in loops with regular and irregular computations have also been conducted. Hereafter, an *irregular loop* is a loop whose body fulfills at least one of the following constraints: first, the assignment statements contain array references at indirection levels greater than one; and second, the condition of the if-endif constructs contain array references at indirection level greater than zero. Note that these constraints introduce data dependences and control dependences that cannot be determined at compile-time, respectively. A loop that does not fulfill any of these constraints is a *regular loop*. From 2958 analyzed loops, 438 (15%) contain irregular computations. More specifically, in 262 loops (9%), irregularity is due to the presence of array assignments, array reductions, array masks and array recurrences with irregular access patterns. The irregularity of the

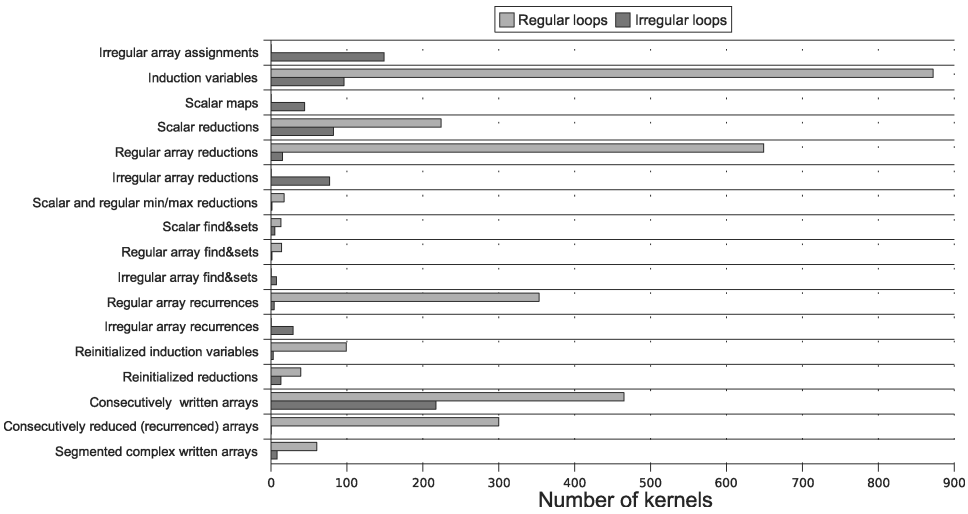


Fig. 17. Distribution of kernel families (excluding scalar assignments and regular array assignments) in regular and irregular loops.

remaining 176 loops (6%) is due to the existence of either array references in the conditions of if-endif constructs, or read-only subscripted array references used in the computations of other kernels. Figure 17 exhibits the differences between the classes of kernels that appear in regular and irregular loops. For clarity, scalar assignments and regular array assignments have not been included because they cover 65% of the kernels on average. Inductions are by far the most frequent kernels in both regular and irregular loops. Note that they appear as simple kernels as well as a part of reinitialized kernels and complex written arrays. Furthermore, the figure shows which is the set of kernels with irregular access patterns that appear only in irregular loops, namely, irregular array assignments, scalar maps, irregular array reductions, irregular array find&sets and irregular array recurrences. This is because, in regular loops, the subscripts of the array references do not depend on other array references and can be rewritten in terms of the index variables of the enclosing loops.

6.2 Array Access Analysis

The goal of these experiments is to find out which are the types of subscript expressions that are most often used in real applications. For each indirection level, the subscripts in each array dimension were classified into six categories that capture the structure of the subscripting functions. Finally, the number of times each category of access pattern occurs in the benchmarks was counted. Figure 18 shows the frequency of each category with respect to the total number of subscripts. The six categories are related to the classes of scalar non-gated SCCs of the taxonomy of Figure 6 as follows: *invariant* (*(n)c*/*inv*), *linear* (*nc*/*lin*), *polynomial or geometric* (*nc*/*poly*, *nc*/*geom*), *monotonic* (*c*/*lin*, *c*/*poly*, *c*/*geom*), *subscripted* (*(n)c*/*reduc*, *(n)c*/*map*, *(n)c*/*subs*) and *unknown* (*(n)c*/*unk*). Subtotals for indirection levels one ($il = 1$) and greater than one ($il > 1$), and totals for all indirection levels (last column *Totals*) are presented.

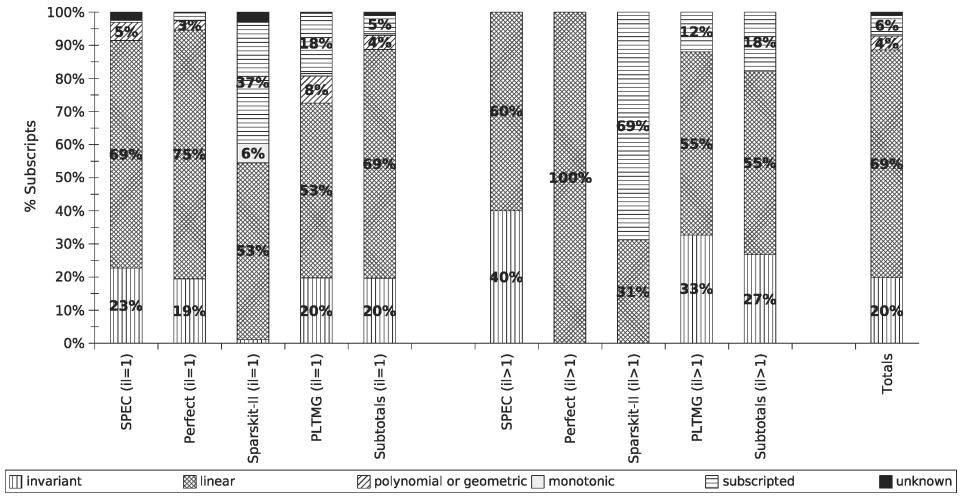


Fig. 18. Distribution of access patterns at indirection level (il) one and greater than one. Totals for all indirection levels are also presented.

The results for indirection level one reveal the different nature of the computations involved in the benchmarks. On the one hand, SPEC and Perfect contain less than 5% of subscripted subscripts distributed among a small number of irregular loops (only 13 of 769 and 56 of 1245, respectively; see Table II). In contrast, SparsKit-II and PLTMG contain more than 18% of subscripted subscripts distributed among a significant number of loops (the number of irregular loops is 145 of 293 and 224 of 627, respectively). In addition, SparsKit-II contains 6% of monotonic access patterns, which introduces an additional complexity from the point of view of automatic program analysis. On the other hand, SPEC and Perfect benchmarks contain more than 95% (756 of 769 and 1189 of 1245, respectively) of loops with regular computations. The results of Figure 18 show that, on average, 89% of array accesses are invariant (20%) or linear (69%). Although the access patterns found in SparsKit-II and PLTMG are invariant and linear as well, the amount of regular loops is 48% only.

Regarding the access patterns at indirection levels greater than one, the experiments show that 82% are regular. More specifically, all of them fit into three categories only: invariant (27%), linear (55%) and subscripted (8%). Note that 100% of access patterns are either invariant or linear in SPEC and Perfect, while SparsKit-II and PLTMG contain 69% and 12% of subscripted subscripts, respectively. The existence of array references with multiple levels of indirection remarks the irregular nature of the computations carried out in SparsKit-II and PLTMG.

7. DISCUSSION

7.1 Robustness

The detection of SCCs in SSA-like representations was shown to be an effective approach for the recognition of kernels in codes with complex control and data

flows. Such approach captures the flow of values of a program and thus provides robustness against different codifications of a given kernel. XARK introduces a higher degree of robustness through the definition of the cardinality of an SCC. For illustrative purposes, consider the two intermediate representations of an array recurrence shown in Figure 19. The Polaris compiler translates each source code statement into one AST with tree nodes that represent n-ary operations. In contrast, GCC breaks each source code statement into a set of ASTs where operations are split into 3-address form, using temporary variables to hold intermediate values. In both cases, the array recurrence is represented by one SCC of cardinality one associated to the source code variable `a`. In Polaris, the SCC classification algorithm traverses one AST and classifies the right-hand side `a(wa)+1` in the classification context $<$ `a(i)`, ►, 2, 0 $>$ given by the left-hand side `a(i)`, the position ►, the level 2 and the indirection level 0. However, for the SCC classification algorithm to have success in GIMPLE, the classification context has to be propagated from the use of the temporary `T` in `a(i)=T+1` to the right-hand side of its definition statement `T=a(wa)`. This issue is handled in the SCC classification algorithm whenever the analysis of an AST that has not been visited is launched (see Figure 8, procedure `Classify()`, case $=$ in the `switch` statement).

Overall, the kernel recognition is robust against source code transformations that introduce temporaries and other structural changes that do not change the SCC graph conditional dependences and individual SCC cardinalities. As a result, the XARK compiler framework can be used at different phases of the compilation process.

7.2 Time Complexity

The XARK compiler addresses kernel recognition in two phases. In the first phase, the SCC classification algorithm traverses the forest of ASTs in a demand-driven manner so that the operator represented by each tree node is classified only once and its kernel class is reused subsequently. Thus, the construction of the kernel graph takes $O(N + E)$ time, where N is the number of tree nodes in the forest of ASTs and E is the number of use-def chains in the GSA graph. In the second phase, the kernel graph classification algorithm traverses the kernel graph in a demand-driven manner analyzing all SCCs and all SCC use-def chains only once. As a result, the construction of the code class takes $O(N_{scc} + E_{scc})$ time, where N_{scc} and E_{scc} are the number of SCCs and SCC use-def chains, respectively. For illustrative purposes, approximations of size of the intermediate representations built by XARK are as follows. Regarding the forest of ASTs combined with GSA graph, $N + E$ is 137223, 412065, 1645682 and 1762616 in SparsKit-II, PLTMG, SPEC and Perfect, respectively. Regarding the kernel graph, $N_{scc} + E_{scc}$ is 35163, 106845, 530316 and 587993 in SparsKit-II, PLTMG, SPEC and Perfect, respectively. Approximations of the running time (in seconds) of the XARK compiler for the benchmarks SparsKit-II, PLTMG, SPEC and Perfect are 11s, 103s, 170s and 266s, respectively. Note that the increase of the running time is proportional to the total number of nodes and edges of the intermediate representations built by XARK.

```
wa=n
i=0
do h=1,n
   i=i+1
   a(i)=a(wa)+1
   wa=i
enddo
```

(a) Intermediate representation used in the Polaris compiler.

```
wa=n
i=0
do h=1,n
   i=i+1
   T=a(wa)
   a(i)=T+1
   wa=i
enddo
```

(b) GIMPLE intermediate representation used in the GCC compiler.

Fig. 19. Example of loop that contains a linear induction (variable `i`), an array recurrence (variable `a`), and a scalar wrap-around kernel (variable `wa`).

7.3 Extensibility

The classification algorithms of XARK hinge on a demand-driven post-order traversal of the forest of ASTs and the kernel graph, as well as on the application of a transfer function at each tree node. A transfer function derives a kernel class for an operation taking into account not only the kernel classes of the operands, but also the classification context of the corresponding operator. This design makes kernel recognition easier to implement and maintain, as well as more powerful.

For illustrative purposes, consider the recognition of wrap-around variables. In this kernel, a scalar variable is assigned a value from outside a loop in the first iteration, and then takes the value of an induction variable for the remainder of the iterations. The detection of wrap-around variables is important for optimizing compilers because the first loop iteration may be peeled off, and the wrap-around variable may be treated as an induction variable. In Figure 19, there is a use of a wrap-around variable `wa` at the array assignment statement `a(i)=a(wa)+1`. Within `a(wa)`, the reference to `wa` has the value `n` in the first iteration and the value of the linear induction variable `i` on subsequent iterations. Typically, recognition is carried out in a separate pattern-matching phase run after induction variable detection that checks that the wrap-around variable is used in the loop before being assigned. Within the XARK compiler, the scalar assignment `wa=i` is represented by a scalar trivial SCC. The demand-driven nature of the SCC classification algorithm assures that the linear induction variable `i` is recognized before classifying its use at the right-hand side of `wa=i`. Thus, the use of `i` is classified as nc/lin. In order to recognize `wa` properly, the following extensions must be performed. First, the transfer function TF_{id} checks that the use of `i` occurs at level 2 at the right-hand side of the statement `wa=i`. If the test is successful, the trivial SCC is annotated as a candidate wrap-around. Second, TF_{id} must assure that the assignment statement of the wrap-around variable post-dominates all the statements where it is used, which assures that the wrap-around variable is set to a new value before proceeding to the next loop iteration. This is accomplished as follows. When a use of `wa` is found during the execution of the SCC classification algorithm and a dependence relationship between the SCCs is established, TF_{id} must check the post-dominance relationship between the use and the definition statements. If the test succeeds in all cases, the annotation of the class of the trivial SCC as candidate is cleared.

Overall, the extension of the XARK compiler is accomplished by adding new transfer functions for any new operator represented as a tree node in the compiler intermediate representation, and by adding new rules to the transfer functions that will enable the recognition of user-defined kernels. Finally, note that appropriate compile-time tests must be implemented when needed.

8. RELATED WORK

Following the five-level classification of kernel recognition techniques presented in Figure 1, the algorithms of the XARK compiler framework are compared with other approaches that work at the domain-independent concept level (Section 8.1) and at the domain-specific concept level (Section 8.2). In addition, XARK is compared with some frameworks proposed in the literature that carry out advanced symbolic analysis (Section 8.3).

8.1 Kernel Recognition at the Domain-Independent Concept Level

There is an extensive literature about kernel recognition. A pattern-matching technique oriented to find parallel loops in the scope of a parallelizing compiler is proposed in Pottenger and Eigenmann [1995]. The method matches the statements of a loop body to a set of predefined patterns that characterize linear inductions as well as scalar and array non-gated reductions. For the recognition of array reductions, a data-dependence test analyzes the matched reduction variables in order to prove that all loop iterations reference different elements of the reduction array. The main limitation is that neither other types of kernels nor complex control flows can be handled. In addition, variations in the programming style have a great impact on the effectiveness of the approach because source code statements are matched directly.

Jouvelot and Dehbonei [1989] and Ammarguellat and Harrison [1990] use abstract interpretation to derive symbolic expressions that summarize the effect of a loop on each variable assigned in the loop body. A set of grammar rules determines the symbolic value associated with each operator of the source code (i.e., expressions and statements). Next, such symbolic expressions are pattern-matched to a data-base of known kernels. These methods are able to recognize some forms of inductions as well as scalar and array reductions. In addition, Ammarguellat and Harrison [1990] detect array recurrences, including some complex forms such as mutual array recurrences. These kernels are only a small fraction of the collection of kernels recognized by the XARK compiler. In addition, these works do not address the analysis of the dependences between the kernels in order to recognize compound kernels.

Callahan [1991] presents a framework for the parallelization of loops whose computations can be represented as a generalization of a parallel prefix operation. The recognition algorithm uses the SCCs of the data-dependence graph to compute composable symbolic functions that represent the computations carried out in a loop. The approach handles loops without if-endif constructs, and only recognizes kernels from the family of array recurrences, including mutual array recurrences. Fisher and Ghuloum [1994] propose a similar framework that is able to handle complex control flows and that recognizes the kernels

included in the families of inductions, reductions and array recurrences, including mutual inductions and mutual array recurrences. However, the frameworks require aggressive symbolic analysis and were not evaluated extensively using well-known benchmark suites. Furthermore, their ability to discover kernels characterized by irregular access patterns (e.g., irregular array reductions) has not been demonstrated in practice.

Suganuma et al. [1996] present a reduction detection algorithm based on the analysis of equations that are built using the information in the data-dependence graph of the source code. First, for each assignment statement in a loop body, an equation that captures the left-hand side, the right-hand side, and the set of predicates that guard the execution of a statement is computed. Second, the equations of the statements of each candidate SCC are merged and the result is matched against a set of templates that characterize reductions. Candidate SCCs are selected by preprocessing the loop with scalar privatization techniques. Although complex control flows are handled, the scope of application is limited to the family of reductions.

The symbolic differencing method proposed by Haghighat and Polychronopoulos [1996] consists of executing a few iterations of a loop body symbolically and saving the symbolic value of each expression at each iteration in a difference table. Next, polynomial and geometric progressions are recognized by interpolating the sequence of symbolic values of each expression. The method requires extensive symbolic expression manipulation, but it is the most powerful technique for the recognition of the family of inductions.

van Engelen [2001] addressed the recognition of conditional and non-conditional inductions (and even factorials and exponentials) by building chains of recurrences that represent the value of the induction variable across the iterations of a loop. Symbolic manipulation is used to rewrite the chains of recurrences as the algorithm proceeds. Unlike the symbolic differencing method, periodic sequences cannot be implicitly handled.

In contrast to techniques based on the analysis of the SCCs of the data-dependence graph of the source code, other methods address recognition through the SCCs of a graph that captures data flow information about the program. Thus, Redon and Feautrier [1993] compute an ad-hoc graph that captures the single assignment information of the source code. Such graph is used to build a system of linear recurrence equations, which is later simplified by computing closed-form expressions through pattern-matching techniques. Due to its high computational cost, this approach is not applicable in practice. In addition, only non-conditional kernels and array references with linear access patterns are handled. Pinter and Pinter [1994] propose a general algorithm based on the construction of the computation graph, which is built from three copies of a dependence graph that represents the initial, a middle and the final iteration of a loop. Recognition is based on matching and replacing graph patterns in order to simplify the computation graph until the most complex kernels can be detected. The simplification process is governed by a grammar that defines the replacement rules as well as the order in which these rules are applied. The approach recognizes reductions, array recurrences and even some compound kernels such as reinitialized scalar and

array reductions. In contrast to XARK, this method requires a preprocessing stage that applies standard loop-level compiler optimizations (e.g., dead-code and dead-store elimination, loop distribution and unrolling) in order to expose the kernels in the computation graph. In addition, its main drawback is the limited support for the analysis of if-endif constructs and arrays (e.g., irregular access patterns are not handled).

Gerlek et al. [1995] present a demand-driven classification method mainly devoted to discover generalized inductions in loop bodies. Recognition is addressed through the analysis of the SCCs of the data-dependence graph of the Static Single Assignment (SSA) form. Like the GSA-based SCC classification algorithm included in XARK, transfer functions that encode the kernels are defined for each operator of the code in SSA form. Next, the kernels are discovered as a result of applying the transfer functions during the execution of a post-order traversal of the abstract syntax trees of the SSA statements. The main drawback is that array references cannot be handled because the SSA form only captures reaching definition information of scalar variables.

Overall, the XARK framework is able to recognize most of the kernels addressed in the other techniques, even in codes with complex control flows. Unlike previous works, XARK has been extensively evaluated using well-known benchmarks suites, not a small set of isolated loops. It should be noted that XARK has been designed so that it can be easily extended to recognize new kernels. This characteristic enabled the recognition of kernels that had not been studied in the literature so far (e.g., segmented consecutively written/reduced array) during the evaluation process.

8.2 Kernel Recognition at the Domain-Specific Concept Level

There is a vast literature about the automatic recognition of kernels that capture the knowledge and the problem solving methods of specific application domains. The approaches vary considerably in their application domain, goal, methodology and status of implementation. In the scope of automatic parallelization of scientific codes, Sabot and Wholey [1993], Bhansali and Hagemeister [1995], Metzger [1995], di Martino and Iannello [1996], Keßler [1996], and Keßler and Smith [1999] discover linear algebra operations (e.g., equation system solvers, FFT, matrix-matrix product) that are later replaced with equivalent efficient parallel programs, possibly from a machine-specific library. Other techniques [Harandi and Ning 1990; Wills 1990; Kozaczynski et al. 1992; Paul and Prakash 1994] focus on software re-engineering, where searching through large amounts of source code to locate relevant information is a critical task. In order to emphasize the difference between domain-independent and domain-specific automatic recognition, in the following, the terms *kernel* and *concept* will be used, respectively.

According to the five-level classification presented in Figure 1, concept recognition is the highest abstraction level in automatic recognition, and thus it enables the most aggressive program transformations and optimizations. However, in order to handle all the complexity of real-world application domains, two main issues must be addressed: the definition of a concept classification hierarchy and the definition of an efficient search strategy. The hierarchy

captures the information about the subconcepts that compose a concept and about the constraints on and between the subconcepts. Hierarchies are usually implemented as large libraries that guide the recognition process (e.g., the number of concepts is 91 in Keßler [1996] and 492 in Metzger [1995]). Thus, it is desirable to define an efficient search strategy that supports partial recognition of programs in domains where knowledge is incomplete.

The XARK compiler provides an extensible, general-purpose kernel recognition technique that can be used as a basis for the recognition of concepts in the scope of different application domains. This article has presented experimental evidence that a small hierarchy of kernels suffices to represent a wide spectrum of applications. In addition, it shows that encoding the properties of the kernels as transfer functions, enables the implementation of an efficient and robust classification algorithm. Note that the code class built by XARK supports partial recognition by classifying into unknown kernel classes those parts of the program whose computations could not be recognized successfully.

8.3 Compiler Frameworks for Advanced Symbolic Analysis

In van Engelen et al. [2004] a unified framework for nonlinear dependence testing and symbolic analysis (e.g., value range analysis and array region analysis) is proposed. The core of the framework is the computation of chains of recurrences that represent the updating of the variables assigned in a loop body. An algorithm that uses this information to recognize generalized inductions even in the presence of complex if-endif constructs is described. The XARK compiler has been successfully applied to predict and understand the behavior of memory hierarchy [Andrade et al. 2007] by building and manipulating chains of recurrences that represent the memory accesses of array references. Thus, extending XARK to fully support the chains of recurrences formalism would widen its scope of application by providing a standard interface with other compiler techniques. Furthermore, XARK builds a hierarchical representation of the source code as kernels and dependence relationships between kernels, which enables optimizing and parallelizing compilers to apply restructuring techniques that go far beyond induction variable substitution. An example application was presented in Arenaz et al. [2004], where XARK was used as a powerful information-gathering infrastructure to generate parallel code based on the recognition of kernels (e.g., irregular array assignment and irregular array reduction).

Fahringer and Scholz [2000] presented a unified framework that collects symbolic information about program variables at arbitrary program points. The information includes variable values, assumptions and constraints about such variable values, and conditions under which control flow reaches a program statement. In addition, the framework recognizes inductions by solving a system of equations that represent as first-order logic formulae the initial value of the induction variable, its updating and the loop exit condition. In contrast, the XARK compiler extracts program information at the kernel level, which enables the characterization of the computations carried out in a set of statements, not in a single statement only. Furthermore, XARK recognizes many important

kernels that are not addressed in Fahringer and Scholz [2000], specially array kernels such as irregular reductions, array recurrences or consecutively written arrays. Overall, XARK provides parallelizing and optimizing techniques with a unified framework that enables more aggressive program transformations and optimizations.

9. CONCLUSIONS

This article has presented and evaluated the XARK compiler framework for automatic recognition of frequently used program constructs. XARK builds a hierarchical representation of the program in terms of kernels and dependences between kernels. The internals consist of two GSA-based demand-driven algorithms that use the strongly connected components of the GSA graph and the dependences between the SCCs as a guide for the analysis of the data dependences and the control flow of a program. This form of analysis is clearly a more general solution than previous approaches, which focus on the detection of specific and isolated kernels and do not provide a general-purpose recognition algorithm that covers scalar and array kernels in a unified manner.

The key characteristics of the demand-driven compiler framework contributed in this paper are: (1) completeness, as the GSA-based algorithms recognize kernels that involve integer-valued and floating-point-valued scalar and array variables, as well as if-endif constructs that introduce complex control flows; (2) robustness against different versions of a kernel; and (3) extensibility, as the addition of new recognition capabilities is accomplished through the modification of a set of transfer functions that encode the characteristics of the kernels.

The extensibility and the demand-driven nature of XARK widen its scope of application beyond automatic kernel recognition. On the one hand, the transfer functions of the GSA-based algorithms can be used in an optimizing compiler as a common information-gathering phase to build an interface that meets the specific information requirements of other compiler passes. Examples of interfaces that have already been integrated in XARK are some parallelizing code transformations and an analytical model for the prediction of cache memory behavior. On the other hand, XARK supports the chains of recurrences formalism. This characteristic provides a standard interface with other widely used compiler techniques, such as induction variable substitution, value range analysis, array region analysis or data-dependence testing.

Another relevant contribution of this paper is the definition of a comprehensive collection of kernels that cover both regular and irregular computations, their organization into sets of families, and the identification of new kernels that appear in real codes but whose study has not been addressed so far. Overall, this work has shown that a significant amount of the regular and irregular computations carried out in full-scale real applications can be characterized using a small set of kernels.

REFERENCES

AHO, A. V., LAM, M. S., SETHI, R., AND ULLMAN, J. D. 2006. *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison-Wesley, Reading, MA.

ALLEN, R. AND KENNEDY, K. 2002. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan-Kaufmann, San Francisco, CA.

AMMARGUELLAT, Z. AND HARRISON, W. L. 1990. Automatic recognition of induction and recurrence relations by abstract interpretation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 283–295.

ANDRADE, D., ARENAZ, M., FRAGUELA, B. B., TOURIÑO, J., AND DOALLO, R. 2007. Automated and accurate cache behavior analysis for codes with irregular access patterns. *Concur. Comput. Pract. Exper. 19,* 18 (Dec.), 2407–2423.

ARENAZ, M. 2003. Compiler framework for the automatic detection of loop-level parallelism. Ph.D. dissertation, Department of Electronics and Systems, University of A Coruña. (Available at `http://www.des.udc.es/~arenaz/papers/phdthesis_arenaz.pdf`.

ARENAZ, M., TOURIÑO, J., AND DOALLO, R. 2003. A GSA-based compiler infrastructure to extract parallelism from complex loops. In *Proceedings of the 17th International Conference on Supercomputing*. (San Francisco, CA). ACM, New York, 193–204.

ARENAZ, M., TOURIÑO, J., AND DOALLO, R. 2004. Compiler support for parallel code generation through kernel recognition. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium* (Santa Fe, NM). IEEE Computer Society Press, Los Alamitos, CA.

BALLANCE, R. A., MACCABE, A. B., AND OTTENSTEIN, K. J. 1990. The program dependence web: A representation supporting control, data, and demand-driven interpretation of imperative languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 257–271.

BANK, R. E. 2007. *PLTMG package*. Available at `http://cam.ucsd.edu/~reb/software.html`.

BERRY, M., CHEN, D., KOSS, P., KUCK, D., POINTER, L., LO, S., PANG, Y., ROLOFF, R., SAMEH, A., CLEMENTI, E., CHIN, S., SCHNEIDER, D., FOX, G., MESSINA, P., WALKER, D., HSIUNG, C., SCHWARZMEIER, J., LUE, K., ORZAG, S., SEIDL, F., JOHNSON, O., SWANSON, G., GOODRUM, R., AND MARTIN, J. 1989. The Perfect Club benchmarks: Effective performance evaluation of supercomputers. *Int. J. Supercomput. Apps. 3,* 3, 5–40.

BHANSALI, S. AND HAGEMEISTER, J. R. 1995. A pattern matching approach for reusing software libraries in parallel systems. In *Proceedings of Workshop on Knowledge Based Systems for the Reuse of Program Libraries* (Sophia Anthipolis, France).

BLUME, W., DOALLO, R., EIGENMANN, R., GROUT, J., HOEFLINGER, J., LAWRENCE, T., LEE, J., PADUA, D. A., PAEK, Y., POTTENGER, W. M., RAUCHWERGER, L., AND TU, P. 1996. Parallel programming with Polaris. *IEEE Computer 29,* 12 (Dec.), 78–82.

CALLAHAN, D. 1991. Recognizing and parallelizing bounded recurrences. In *Proceedings of the 4th International Workshop on Languages and Compilers for Parallel Computing* (Santa Clara, CA). Lecture Notes in Computer Science, vol. 589. Springer-Verlag, New York, 169–185.

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13,* 4 (Oct.), 451–490.

DI MARTINO, B. AND IANNELLO, G. 1996. PAP recognizer: A tool for automatic recognition of parallelizable patterns. In *Proceedings of the 4th International Workshop on Program Comprehension* (Berlin, Germany). IEEE Computer Society Press, Los Alamitos, CA, 164–174.

FAHRINGER, T. AND SCHOLZ, B. 2000. A unified symbolic evaluation framework for parallelizing compilers. *IEEE Trans. Parallel Dist. Syst. 11,* 11 (Nov.), 1105–1125.

FAIGIN, K. A., WEATHERFORD, S. A., HOEFLINGER, J. P., PADUA, D. A., AND PETERSEN, P. M. 1994. The Polaris internal representation. *Int. J. Parall. Prog. 22,* 5 (Oct.), 553–586.

FISHER, A. L. AND GHULOUM, A. M. 1994. Parallelizing complex scans and reductions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (Orlando, FL). ACM, New York, 135–146.

GCC INTERNALS. *GNU Compiler Collection Internals (GCC)*. Available at `http://gcc.gnu.org/onlinedocs/gccint.pdf`.

GERLEK, M. P., STOLTZ, E., AND WOLFE, M. 1995. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA. *ACM Trans. Program. Lang. Syst. 17,* 1 (Jan.), 85–122.

HAGHIGHAT, M. R. AND POLYCHRONOPOULOS, C. D. 1996. Symbolic analysis for parallelizing compilers. *ACM Trans. Program. Lang. Syst. 18,* 4 (July), 477–518.

HARANDI, M. T. AND NING, J. Q. 1990. Knowledge-based program analysis. *IEEE Softw. 7,* 1, 74–81.

JOUVELOT, P. AND DEHBONEI, B. 1989. A unified semantic approach for the vectorization and parallelization of generalized reductions. In *Proceedings of the 3rd International Conference on Supercomputing* (Heraklion, Crete). ACM, New York, 186–194.

KEßLER, C. W. 1996. Pattern-driven automatic parallelization. *Scient. Progr. 5,* 3, 251–274.

KEßLER, C. W. AND SMITH, C. 1999. The SPARAMAT approach to automatic comprehension of sparse matrix computations. In *Proceedings of the 7th International Workshop on Program Comprehension* (Pittsburgh, PA). IEEE Computer Society Press, Los Alamitos, CA, 200–207.

KNOBE, K. AND SARKAR, V. 1998. Array SSA form and its use in parallelization. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, CA). ACM, New York, 107–120.

KNOBE, K. AND SARKAR, V. 2000. Enhanced parallelization via analyses and transformations on Array SSA form. In *Proceedings of the 8th International Workshop on Compilers for Parallel Computers* (Aussois, France).

KOZACZYNSKI, W., NING, J. Q., AND ENGBERTS, A. 1992. Program concept recognition and transformation. *IEEE Trans. Softw. Eng. 18,* 12, 1065–1075.

LIN, Y. AND PADUA, D. A. 1998. On the automatic parallelization of sparse and irregular Fortran programs. In *Proceedings of the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers* (Pittsburgh, PA). Lecture Notes in Computer Science, vol. 1511. Springer-Verlag, New York, 41–56.

MERRILL, J. 2003. GENERIC and GIMPLE: A new tree representation for entire functions. In *Proceedings of the 2003 GCC Developers Summit*. 171–180. (Available at `http://www.gccsummit.org/2003`.

METZGER, R. 1995. Automated recognition of parallel algorithms in scientific applications. In *IJCAI-95 Workshop Program Working Notes: "The Next Generation of Plan Recognition Systems"*.

MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan-Kaufmann, San Francisco, CA.

PAUL, S. AND PRAKASH, A. 1994. A framework for source code search using program patterns. *IEEE Trans. Softw. Eng. 20,* 6, 463–475.

PINTER, S. S. AND PINTER, R. Y. 1994. Program optimization and parallelization using idioms. *ACM Trans. Program. Lang. Syst. 16,* 3 (May), 305–327.

POTTENGER, W. M. AND EIGENMANN, R. 1995. Idiom recognition in the Polaris parallelizing compiler. In *Proceedings of the 9th International Conference on Supercomputing* (Barcelona, Spain). ACM, New York, 444–448.

REDON, X. AND FEAUTRIER, P. 1993. Detection of recurrences in sequential programs with loops. In *Proceedings of the 5th International Parallel Architectures and Languages Europe Conference* (Munich, Germany). Lecture Notes in Computer Science, vol. 694. Springer-Verlag, New York, 132–145.

SAAD, Y. 1994. *SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations (Version 2)*. (Available at `http://www.cs.umn.edu/~saad/software/SPARSKIT/sparskit.html`.

SABOT, G. AND WHOLEY, S. 1993. CMAX: A Fortran translator for the connection machine system. In *Proceedings of the 7th International Conference on Supercomputing* (Tokyo, Japan). ACM, New York, 147–156.

SPEC. *SPEC CPU2000*. Standard Performance Evaluation Corporation. Available at `http://www.spec.org/cpu2000/`.

SUGANUMA, T., KOMATSU, H., AND NAKATANI, T. 1996. Detection and global optimization of reduction operations for distributed parallel machines. In *Proceedings of the 10th International Conference on Supercomputing* (Philadelphia, PA). ACM, New York, 18–25.

TARJAN, R. E. 1972. Depth-first search and linear graph algorithms. *SIAM J. Comput. 1,* 2 (June), 146–160.

TU, P. AND PADUA, D. A. 1995. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of the 9th International Conference on Supercomputing* (Barcelona, Spain). ACM, New York, 414–423.

VAN ENGELEN, R. 2001. Efficient symbolic analysis for optimizing compilers. In *Proceedings of the 10th International Conference on Compiler Construction* (Genova, Italy). Lecture Notes in Computer Science, vol. 2027. Springer-Verlag, New York, 118–132.

VAN ENGELEN, R., BIRCH, J., SHOU, Y., WALSH, B., AND GALLIVAN, K. 2004. A unified framework for nonlinear dependence testing and symbolic analysis. In *Proceedings of the 18th International Conference on Supercomputing* (Saint Malo, France). ACM, New York, 106–115.

WILLS, L. M. 1990. Automated program recognition: A feasibility demonstration. *Artif. Intell. 45*, 1-2, 113–171.

WOLFE, M. 1996. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Reading, MA.

WU, P., COHEN, A., HOEFLINGER, J., AND PADUA, D. A. 2001. Monotonic evolution: An alternative to induction variable substitution for dependence analysis. In *Proceedings of the 15th International Conference on Supercomputing* (Sorrento, Italy). ACM, New York, 78–91.

ZHANG, F. AND D'HOLLANDER, E. H. 1994. Enhancing parallelism by removing cyclic data dependencies. In *Proceedings of the 6th International Parallel Architectures and Languages Europe Conference* (Athens, Greece). Lecture Notes in Computer Science, vol. 817. Springer-Verlag, New York, 387–397.

ZIMA, E. V. 1986. Automatic construction of systems of recurrence relations. *USSR Comput. Math. Math. Phys. 24*, 11-12, 193–197.

ZIMA, E. V. 1995. Simplification and optimization transformations of chains of recurrences. In *Proceedings of the 8th International Symposium on Symbolic and Algebraic Computation* (Montreal, Ont., Canada). ACM New York, 42–50.

Received December 2006; revised July 2007 and December 2007; accepted January 2008