Program Behavior Characterization Through Advanced Kernel Recognition[*]

Manuel Arenaz, Juan Touriño, and Ramón Doallo

Computer Architecture Group
Department of Electronics and Systems
University of A Coruña, A Coruña, Spain
{arenaz,juan,doallo}@udc.es
http://gac.des.udc.es

Abstract. Understanding program behavior is at the foundation of program optimization. Techniques for automatic recognition of program constructs (from now on, computational kernels) characterize the behavior of program statements, providing compilers with valuable information to to guide code optimization. Our goal is to develop automatic techniques that summarize the behavior of full-scale real applications by building a high-level representation that hides the complexity of implementation details. The first step towards this goal is the description of applications in terms of computational kernels such as induction variables, reductions, and array recurrences. To this end we use XARK, a compiler framework that recognizes a comprehensive collection of frequently used kernels. This paper presents detailed experiments that describe several benchmarks from different application domains in terms of the kernels recognized by XARK. More specifically, 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 characterized in detail.

1 Introduction

Automatic code optimization hinges on advanced symbolic analysis to gather information about the behavior of programs. Compiler techniques for automatic kernel recognition carry out symbolic analysis in order to discover program constructs that are frequently used by software developers. Such techniques were shown to be a powerful mechanism to improve the performance of optimizing and parallelizing compilers. Well-known examples are the substitution of induction variables with closed-form expressions, the detection of reduction operations to raise the effectiveness of dependence analysis, the characterization of the access patterns of array references to predict program locality, or the automatic replacement of sequential algorithms with platform-optimized parallel versions.

[*] This research was supported by the Ministry of Education and Science of Spain and FEDER funds of the European Union (Project TIN2004-07797-C02), and by the Galician Government (Projects PGIDT05PXIC10504PN and PGIDIT06PXIB105228PR).

A.-M. Kermarrec, L. Bougé, and T. Priol (Eds.): Euro-Par 2007, LNCS 4641, pp. 237–247, 2007.

XARK [2] is an extensible compiler framework for automatic recognition of computational kernels. Unlike previous approaches that focus on specific and isolated kernels [6,8,11], XARK provides a general solution that detects a comprehensive collection of kernels that appear in real codes with regular and irregular computations. The recognition algorithm analyzes data dependences and control flow altogether, and handles scalar and array variables in a unified manner. The kernels are organized in families that share common syntactical properties. Some well-known examples are induction variables, scalar reductions, irregular reductions and array recurrences.

The rest of the paper is organized as follows. Section 2 gives a general overview of the XARK compiler and describes the families of computational kernels. Section 3 shows detailed experimental results for the benchmarks SparsKit-II, Perfect, SPEC CPU2000 and PLTMG. Finally, Section 4 concludes the paper and outlines future work.

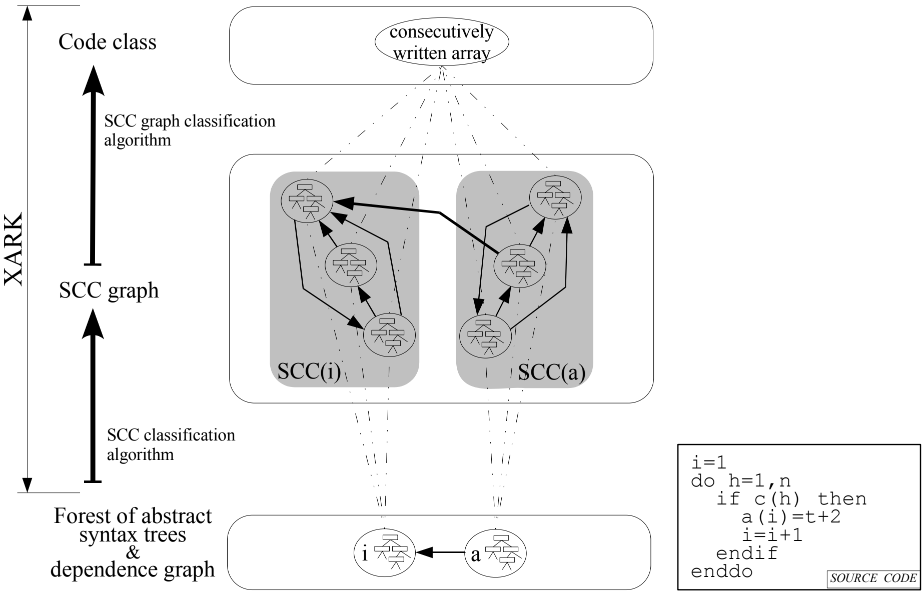
2 The XARK Compiler

2.1 Overview

XARK [2] is a compiler framework that provides a general solution to the problem of automatic kernel recognition. Three key characteristics distinguish XARK from previous approaches: (1) completeness, as it recognizes a comprehensive collection of computational 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 computational kernel; and (3) extensibility, as its design enables the addition of new recognition capabilities with little programming effort.

XARK internals consist of a two-phase demand-driven classification algorithm that analyzes the data dependences and the control flow graph of a program through its Gated Single Assignment (GSA) representation [12], which is an extension of the well-known Static Single Assignment (SSA) form where reaching definition information of scalar and array variables is represented syntactically. For illustrative purposes consider the example code presented in Figure 1. For the sake of clarity, the details about the GSA form and the loop index variable `h` have been omitted. The code consists of a loop $do_{\mathtt{h}}$ that computes a kernel called *consecutively written array* (see Section 2.2 later in this paper). 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. 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 framework is built on top of an intermediate representation where the source code statements are represented as abstract syntax trees, and the data dependences between statements are captured as use-def chains between the trees. In the first phase, XARK identifies the strongly connected components


XARK
Code class
SCC graph classification algorithm
SCC graph
SCC classification algorithm
Forest of abstract syntax trees & dependence graph
consecutively written array
SCC(i)
SCC(a)
i
a
SOURCE CODE

```
i=1
do h=1,n
  if c(h) then
    a(i)=t+2
    i=i+1
  endif
enddo
```

Fig. 1. Overview of the XARK compiler

(SCCs) of the dependence graph and carries out an intra-SCC analysis that determines the type of kernel computed during the execution of the statements of each SCC. As a result of this intra-SCC analysis, the code is decomposed into a set of mutually dependent kernels that capture the run-time behavior of each source code variable. In the example, the code is decomposed into two SCCs (namely, `SCC(i)` and `SCC(a)`) that enable the recognition of the induction variable and the array assignment (see Section 2.2) computed as a result of executing the statements `i=i+1` and `a(i)=t+2`, respectively. The dependence relationships between `a` and `i` are represented as use-def chains between the SCCs.

In the second phase, XARK focuses on the use-def chains between statements of different SCCs in order to recognize more complex kernels that result from combining simpler kernels in the same code. In the example of Figure 1, the isolated detection of the induction variable `i` and the array assignment `a` does not provide enough information to recognize the consecutively written array. Thus, during the inter-SCC analysis, XARK checks that array `a` is indexed with the induction variable `i`; it analyzes the control flow graph to prove that every time `a(i)=t+2` is executed, `i=i+1` is also executed; and checks that `i` is incremented in one unit in every loop iteration where `c(h)` is fulfilled. Under such conditions, the consecutively written array `a` is recognized successfully. The results of this stage provide the compiler with high level information about the behavior of the program, hiding the complexity of implementation details. This information is very useful for the construction of other passes of a parallelizing or optimizing compiler. Examples of successful application of XARK in the scopes of parallel code generation and prediction at compile-time of cache behavior have been presented in [3] and [1], respectively.

2.2 Collection of Kernels

The collection of computational kernels recognized by XARK is organized in the eight families described next: assignments, induction variables, maps, reductions, masks, array recurrences, reinitialized kernels, and complex written arrays.

Assignments. This is the simplest form of computational kernel. Given a variable of a program, it consists of setting the variable to a value that does not depend on the variable itself. The family is called *scalar assignment* or *array assignment* if the variable is a scalar or an array, respectively. Different classes of array assignments are distinguished according to the properties of the array index. If the index consists of a linear, polynomial or geometric function of the loop index, the family is called *regular array assignment* (e.g., `a(h)=f(h)` where `h` is the loop index). If the index is a loop-variant subscripted expression, it is called *irregular array assignment* (e.g., `a(b(h))=f(h)`).

Induction Variables. This family represents the type of scalar, integer-valued variables that are updated in the iterations of a loop, either in every iteration or in those iterations that fulfill a given condition. Different classes of induction variables (IVs) are distinguished: *linear*, if an integer-valued loop-invariant value is added to the value of the IV in each iteration (e.g., `i=i+1` is an IV of step one); *polynomial*, if it is the value of another IV that is added (e.g., `i=i+j` where `j=j+1`); and *geometric*, if the IV is multiplied by a loop-invariant (e.g., `k=2*k+1`).

Maps. A distinguishing characteristic of IVs is that there is a closed form function that allows the computation of the next value of the variable starting from its initial value or from its current value. A *map* represents a sequence of values that do not have such a closed form. In each loop iteration, the variable is assigned the value of an array reference whose subscript expression contains an occurrence of the variable (e.g., `i=next(i)`). When the variable is an array, different types of regular and irregular access patterns are considered (e.g., `a(h)=next(a(h))`, with loop index `h`, is an array map with a regular access pattern).

Reductions. A *scalar reduction* is a kernel with one scalar variable that is defined in terms of itself and at least one loop-variant subscripted expression (e.g., `r=r+a(h)`). An *array reduction* is defined in a similar manner, the reduction variable being an array (e.g., `r(h)=r(h)+a(h)`). The characteristics of the index expression lead to distinguish between regular and irregular array reductions. Well-known examples of this family of kernels are adding the elements of a vector, and finding the minimum/maximum element in each row of a matrix. A variant of a minimum/maximum reduction consists of gathering additional information about the reduction variable, for instance, the position of the minimum (or the maximum) value within each row.

```
flag=true
do h=1,n
   if flag then
      ...
      flag=false
   endif
...
enddo
```

Fig. 2. Example of loop that computes a mask kernel

```
do h=1,n
   i=0
   do hh=1,m
      i=i+1
   enddo
enddo
```

Fig. 3. Example of loop that computes a reintialized induction variable kernel

Masks. Masks are kernels that modify the value of a variable if its content fulfills a boolean condition. This family is called either *scalar find&set* or *array find&set*. A typical example is a loop that contains a set of statements that are executed only in the first loop iteration (see Figure 2). When the condition is *true* (`flag` in the example), such statements are executed and the condition is set to $false$ to avoid the execution in the subsequent loop iterations. Array masks with regular and irregular access patterns are also considered.

Array Recurrences. Array recurrences are kernels that compute the value of the elements of the array using the values of other elements of the array (e.g., `a(h)=a(h-1)+1`). Unlike array reductions, array recurrences use different index expressions to access to the elements of the array. Regular and irregular access patterns are also considered.

Reinitialized Kernels. Real codes may contain more elaborate program constructs built from the kernels described above. From a graphical point of view, they can be interpreted as a point in a multidimensional space where the syntactical kernels are the values represented in the axes. Thus, a reinitialized kernel is as follows: first, an assignment that sets a scalar/array variable to a given value at the beginning of every iteration of a loop; and second, an induction variable, a map, a reduction, a mask or an array recurrence that updates the value of the scalar/array variable during the execution of an inner loop. The example shown in Figure 3 contains a reinitialized IV `i`.

Complex Written Arrays. Another interesting family is called *complex written array* [7]. It consists of a scalar kernel (e.g., induction variable, reinitialized IV, 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 an IV of step one, the kernel is called *consecutively written array* (see the example code of Figure 1). When it is a reinitialized IV of step one, it is called *segmented consecutively written array*. Other variants of complex written arrays recognized by XARK involve an array reduction or an array recurrence instead of an array assignment. Then, they are called *(segmented) consecutively reduced array* and *(segmented) consecutively recurrenced array*, respectively.

3 Experimental Results

3.1 Benchmarks

Four benchmark suites have been used in the experiments: the Fortran routines included in SPEC CPU2000 [10], the Perfect benchmarks [5], the SparsKit-II library [9] and the PLTMG (Piecewise Linear Triangle Multi-Grid) code [4]. SPEC2000 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 1 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.

Table 1. Summary of characteristics of the benchmark suite

	SPEC2000	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%	77%	76%	77%	77%

SparsKit-II [9] contains routines for the manipulation of sparse matrices. It is organized in four modules: MATVEC, devoted to basic matrix-vector operations (e.g., matrix-vector products and triangular system solvers); BLASSM, which covers basic linear algebra operations (e.g., matrix-matrix products and sums); UNARY, to carry out unary operations with sparse matrices (e.g., extract a submatrix); and FORMATS, for the conversion of sparse matrices between different types of sparse storages.

SPEC CPU2000 [10] consists of six Fortran codes: a program in the area of quantum chromodynamics (WUPWISE), two weather prediction programs (SWIM and APSI), a very simple multi-grid solver for computing a three dimensional potential field (MGRID), coupled nonlinear partial differential equations solver in the scope of computational fluid dynamics and computational physics (APPLU), and a program in the area of high energy nuclear physics accelerator design (SIXTRACK).

The Perfect Benchmarks [5] are a collection of thirteen scientific and engineering Fortran programs that are representative of applications executed on high-performance computers and that have been used extensively in research on parallelizing and restructuring compilers. It covers different application areas: fluid dynamics (ADM, ARC2D, FLO52, OCEAN and SPEC77), chemical and physical modeling (BDNA, MDG, QCD and TRFD), engineering design (DYFESM and SPICE), and signal processing (MG3D and TRACK).

PLTMG (Piecewise Linear Triangle Multi-Grid) [4] is a Fortran-77 code that consists of an adaptive multi-grid solver for two dimensional problems in general domains.

Table 2. Kernel families recognized by XARK in Sparskit-II and PLTMG

Kernel family	MATVEC				BLASSM				UNARY				FORMATS				PLTMG			
	N	S	C	L	N	S	C	L	N	S	C	L	N	S	C	L	N	S	C	L
Assignments																				
Scalar assignment	16	1-1	0-0	2-4	37	1-1	1-1	1-3	61	1-1	0-4	1-3	82	1-2	1-4	1-4	503	1-4	1-9	1-4
Regular assignment	12	1-1	0-0	1-2	23	1-2	1-2	1-3	39	1-1	0-1	1-3	55	1-2	1-2	1-2	456	1-9	1-4	1-3
Irregular assignment					13	1-1	1-1	1-2	11	1-4	1-2	1-3	47	1-2	1-1	1-3	67	1-3	1-2	1-2
Induction variables																				
Linear	1	1-1	0-0	4-4	6	1-3	1-2	1-3	12	1-1	1-2	1-2	17	1-5	1-2	1-3	100	1-6	1-5	1-4
Polynomial																	5	1-1	0-0	1-2
Geometric									1	1-1	0-0	3-3	1	1-1	0-0	4-4	38	1-1	0-2	1-2
Maps																				
Linked list																	44	1-2	1-9	1-4
Regular map																	2	1-1	0-0	2-2
Reductions																				
Scalar reduction	5	1-1	0-0	1-1					12	1-1	0-1	1-3	4	1-1	0-0	1-1	105	1-2	1-3	1-2
Regular reduction	2	1-1	0-0	2-2	2	1-1	0-0	1-1	4	1-1	0-0	1-1	2	1-2	0-1	1-2	86	1-3	0-1	1-3
Irregular reduction	9	1-1	0-0	1-4	1	1-1	1-1	1-1	7	1-3	1-1	1-2	13	1-1	1-1	1-2	24	1-3	0-1	1-2
Scalar min/max									2	1-2	1-2	1-1	4	1-2	1-2	1-2	6	1-2	1-3	1-2
Masks																				
Scalar find-and-set					2	1-1	1-1	1-1									13	1-2	1-4	1-3
Regular find-and-set									4	1-2	1-1	1-1					6	1-2	1-2	1-2
Irregular find-and-set									2	1-1	1-1	1-2					5	2-4	1-3	1-3
Recurrences																				
Regular recurrences									7	1-1	0-0	1-1	18	1-1	0-0	1-2	51	1-3	0-1	1-2
Irregular recurrences									2	1-1	1-1	2-2	2	1-1	1-2	1-2	15	1-1	0-0	1-3
Reinitialized kernels																				
Induction variables									2	2-2	1-1	2-3	2	2-2	1-1	2-2				
Maps																	9	2-3	2-2	2-4
Reductions	1	2-2	0-0	2-2					5	2-2	1-1	2-2	2	2-2	0-0	2-3	4	2-2	0-0	2-2
Complex written arrays																				
C. written ar. (CwriA)					13	2-6	1-2	1-2	17	2-2	1-1	1-3	31	2-2	1-2	1-4	31	2-6	1-3	1-2
C. recurrenced ar. (CrecA)													1	2-2	0-0	2-2	5	2-2	1-1	2-2
Segmented CwriA									2	3-3	0-0	2-2								
TOTALS	46	1-2	0-0	1-4	97	1-6	0-2	1-3	190	1-4	0-4	1-3	281	1-5	0-4	1-4	1575	1-9	0-9	1-4
Kernels with irregular computations	21		46%		62		64%		96		51%		157		56%		642		41%	

3.2 Recognition Results

The first step towards the characterization of the behavior of programs is the recognition of the computational kernels that appear in the code. The last row of Table 1 measures the effectiveness of XARK in terms of the percentage of loops whose body has been decomposed into a set of kernels recognized by the compiler. The percentage of *recognized loops* is 77% on average, ranging from 76% in SparsKit-II up to 79% in SPEC2000.

The experiments revealed that the loops of the benchmarks can be described in terms of the eight kernel families recognized by XARK, which were introduced in Section 2.2. Tables 2-4 summarize the number of kernels N found in each module of SparsKit-II, PLTMG, Perfect and SPEC2000. The last rows show the totals for each module, including the percentage of computational kernels that contain irregular computations. Measurements of the complexity of the kernels are also presented: S is the range of statements that compose the kernels; C is the range of conditions checked in if-endif statements; and L is the range of nested loops that contain the statements of the kernels. The ranges are displayed in the format m-M, the numbers m and M being the minimum and the maximum, respectively.

Sparskit-II and PLTMG are codes that contain a high percentage of irregular computations, ranging from 41% in PLTMG up to 64% in the BLASSM module of Sparskit-II (see last row of Table 2). Irregularity is due to the presence of kernels with irregular access patterns (e.g., irregular assignment, irregular reduction, irregular find-and-set and irregular recurrence), array references in the conditions of if-endif constructs, or read-only subscripted array references

Table 3. Kernel families recognized by XARK in Perfect

Kernel family	ADM				SPICE				QCD				MDG				TRACK				BDNA			
	N	S	C	L	N	S	C	L	N	S	C	L	N	S	C	L	N	S	C	L	N	S	C	L
Assignments																								
Scalar assignment	278	1-3	1-6	1-3	42	1-1	0-0	1-2	14	1-2	1-2	1-2	20	1-1	0-0	1-3	2	1-1	0-0	1-1	360	1-3	0-2	1-4
Regular assignment	169	1-10	1-4	1-3	23	1-1	0-0	1-2	22	1-1	1-1	1-3	23	1-1	0-0	1-2	45	1-2	1-1	1-2	113	1-4	0-1	1-2
Irregular assignment																					1	1-1	0-0	2-2
Induction variables																								
Linear	62	1-2	2-2	1-3	20	1-1	0-0	1-2	2	1-1	0-0	4-4	23	1-1	1-1	1-3	6	1-1	1-1	1-2	43	1-1	0-0	1-3
Polynomial	4	1-1	0-0	2-3																				
Geometric	12	1-1	1-1	1-2	1	1-1	0-0	1-1					2	1-1	0-0	1-2	9	1-1	0-0	1-3	7	1-1	0-0	1-2
Reductions																								
Scalar reduction	17	1-1	0-0	1-3	6	1-1	0-0	1-2	2	1-1	0-0	1-1	22	1-2	0-0	1-2	4	1-1	0-0	1-2	33	1-1	0-0	1-2
Regular reduction	13	1-2	0-0	1-2	5	1-1	0-0	1-1	5	1-1	0-0	1-2	7	1-3	0-0	1-2	8	1-2	1-1	1-3	49	1-6	0-0	1-4
Irregular reduction									1	1-1	1-1	2-2									14	1-12	0-0	1-3
Masks																								
Regular find-and-set	1	2-2	1-1	1-1																				
Recurrences																								
Regular recurrences	7	1-1	0-0	1-3	5	1-1	0-0	1-1					1	1-1	0-0	1-1	6	1-1	0-0	1-1	20	1-3	0-0	1-2
Irregular recurrences					10	1-2	0-0	1-1																
Reinitialized kernels																								
Induction variables	7	2-2	0-0	2-3	1	2-2	0-0	2-2	3	2-2	0-0	4-4	10	2-2	0-0	2-4					1	2-2	0-0	3-3
Reductions					1	2-2	0-0	2-2					4	2-2	0-0	2-2					2	2-2	0-0	2-4
Complex written arrays																								
C. written ar. (CwriA)	4	2-2	0-0	1-2					45	2-28	1-1	1-4	10	2-2	0-0	1-4					23	2-2	0-0	1-1
C. reduced ar. (CredA)	8	2-2	0-0	2-3									4	2-2	0-0	2-2					45	2-2	0-0	1-2
Segmented CwriA													1	3-3	0-0	2-2								
Segmented CredA													1	3-3	0-0	2-2								
TOTALS	582	1-10	0-6	1-3	114	1-2	0-0	1-2	94	1-28	0-2	1-4	128	1-3	0-1	1-4	80	1-2	0-1	1-3	711	1-12	0-2	1-4
Kernels with irregular computations	29		5%		22		19%		6		6%		22		17%		8		10%		118		17%	

Kernel family	OCEAN				DYFESM				MG3D				ARC2D				FLO52				TRFD				SPEC77			
	N	S	C	L	N	S	C	L	N	S	C	L	N	S	C	L	N	S	C	L	N	S	C	L	N	S	C	L
Assignments																												
Scalar assignment	14	1-2	3-3	1-2	32	1-1	1-4	1-4	747	1-2	1-1	1-3	158	1-3	0-3	1-3	40	1-1	0-0	1-2	1	1-1	0-0	2-2	64	1-3	1-2	1-3
Regular assignment	29	1-2	0-0	1-2	55	1-3	0-1	1-4	65	1-5	0-0	1-3	65	1-4	0-0	1-3	56	1-4	0-0	1-3	5	1-2	0-0	1-2	120	1-2	0-1	1-2
Irregular assignment					6	1-1	1-1	2-2																				
Induction variables																												
Linear	27	1-1	0-0	1-2	10	1-1	0-0	1-2	194	1-2	0-0	1-3	24	1-2	1-1	1-3	11	1-1	1-1	1-2	10	1-2	1-1	1-4	62	1-2	1-1	1-3
Polynomial	8	1-1	0-0	1-2					52	1-1	0-0	2-3													2	1-1	1-1	2-2
Geometric	9	1-1	0-0	1-2					6	1-1	0-0	1-1					2	1-1	0-0	1-1					9	1-1	0-0	1-2
Reductions																												
Scalar reduction	4	1-1	0-0	1-1	18	1-2	0-1	1-4	2	1-1	0-0	1-1	1	1-1	0-0	2-2	4	1-1	0-0	1-1	4	1-1	0-0	2-4	33	1-1	1-1	1-3
Regular reduction	30	1-1	0-0	1-2	18	1-4	1-1	1-3	13	1-2	0-0	1-2	30	1-4	0-0	1-3	18	1-1	0-0	1-3	4	1-2	0-0	1-1	59	1-2	1-1	1-3
Irregular reduction					4	1-3	1-1	1-2																				
Scalar min/max																									3	1-1	1-1	1-2
Masks																												
Scalar find-and-set																									1	1-1	1-1	2-2
Recurrences																												
Regular recurrences	6	1-1	0-0	1-2	5	1-1	1-1	1-2	2				69	1-2	0-0	1-2	11	1-2	0-0	1-2					7	1-1	1-1	1-2
Reinitialized kernels																												
Induction variables	4	2-2	0-0	2-2	2	2-2	0-0	2-4	36	2-2	0-0	2-3					10	2-2	0-0	2-3	1	2-2	0-0	2-2	16	2-2	0-0	2-3
Reductions					11	2-2	0-0	2-4																	14	2-2	0-0	2-2
Complex written arrays																												
C. written ar. (CwriA)	2	2-2	0-0	1-1					2	2-2	0-0	1-1									2	2-2	0-0	1-1	2	2-2	0-0	1-1
C. recurrenced ar. (CrecA)																	2	2-2	0-0	1-1								
Segmented CwriA																									15	4-10	0-0	2-3
TOTALS	133	1-2	0-3	1-2	161	1-4	0-4	1-4	1119	1-5	0-1	1-3	347	1-4	0-3	1-3	154	1-4	0-1	1-3	27	1-2	0-1	1-4	407	1-10	0-2	1-3
Kernels with irregular computations	7		5%		30		19%		23		2%		29		8%		8		5%		0		0%		24		6%	

used in the computations of other kernels. Sparskit-II consists of small routines with complex computations. Apart from the high percentage of irregular computations, this complexity is reflected in the maximum number of statements (6 and 5 in BLASSM and FORMATS, respectively), conditions (4 in UNARY and FORMATS) and nested loops (4 in MATVEC, FORMATS and PLTMG). The experiments also reveal that PLTMG is a complex application. It consists of 1575 kernels that involve up to 9 statements, 9 conditions and 4 nested loops. Note that PLTMG is the unique benchmark that contains kernels of the eight families presented in Section 2.2. It contains 46 maps, which is a family that does not appear in any other benchmark. This fact shows that the recognition

Table 4. Kernel families recognized by XARK in SPEC CPU2000

Kernel family	WUPWISE				SWIM				MGRID				APPLU				SIXTRACK				APSI			
	N	S	C	L	N	S	C	L	N	S	C	L	N	S	C	L	N	S	C	L	N	S	C	L
Assignments																								
Scalar assignment	6	1-2	1-2	1-3					1	1-1	0-0	3-3	24	1-1	0-0	1-4	32	1-2	1-2	1-3	276	1-3	1-6	1-3
Regular assignment	2	1-1	0-0	1-2	23	1-1	0-0	2-2	4	1-1	0-0	3-3	26	1-25	0-1	1-4	432	1-29	1-4	1-4	167	1-10	1-4	1-3
Irregular assignment									1	2-2	0-0	1-1					3	1-1	1-1	1-1				
Induction variables																								
Linear	4	1-1	0-0	1-1													22	1-2	1-1	1-4	61	1-2	2-2	1-3
Polynomial																					4	1-1	0-0	2-3
Geometric									3	1-1	0-0	3-3	6	1-1	0-0	2-4	30	1-1	0-0	1-2	12	1-1	1-1	1-2
Reductions																								
Scalar reduction	2	1-1	0-0	1-1	3	1-1	0-0	2-2	1	1-1	0-0	3-3	3	1-1	0-0	2-2	8	1-1	0-0	1-1	17	1-1	0-0	1-3
Regular reduction	12	1-3	1-2	1-3	4	1-1	0-0	2-2	5	1-4	0-0	1-3	8	1-2	0-0	1-4	36	1-2	1-2	1-4	10	1-1	0-0	1-2
Irregular reduction																	4	1-1	1-1	1-1				
Scalar min/max									1	1-1	1-1	3-3												
Regular min/max													2	1-1	1-1	1-4								
Masks																								
Scalar find-and-set	1	2-2	4-4	1-1													1	1-1	1-1	2-2				
Regular find-and-set																	1	2-2	2-2	1-1	1	2-2	1-1	1-1
Recurrences																								
Regular recurrences					30	1-1	0-0	1-1	5	1-2	0-0	1-2	9	1-1	0-0	1-3	86	1-3	1-2	1-3	8	1-2	0-0	1-3
Reinitialized kernels																								
Induction variables																					7	2-2	0-0	2-3
Reductions	6	2-2	0-0	3-3													2	2-2	0-0	2-2				
Complex written arrays																								
C. written ar. (CwriA)	1	2-2	0-0	1-1													1	4-4	0-0	1-1	4	2-2	0-0	1-2
C. reduced ar. (CredA)	2	2-2	0-0	1-1																	6	2-2	0-0	2-3
TOTALS	36	1-2	0-4	1-3	60	1-1	0-0	1-2	21	1-4	0-1	1-3	78	1-25	0-1	1-4	658	1-29	0-4	1-4	573	1-10	0-6	1-3
Kernels with irregular computations	9		25%		7		12%		2		10%		8		10%		171		26%		27		5%	

of all the kernel families is essential in order to fully characterize the behavior of real applications.

Perfect and SPEC2000 are codes characterized by the regularity of their computations. On average, only 11% of the kernels contain some type of irregularity, ranging from 0% in TRFD up to 26% in SIXTRACK. The complexity of the analysis of the Perfect benchmarks is mainly due to the existence of plenty of linear, polynomial and geometric induction variables. SPEC2000 includes large applications such as APPLU, SIXTRACK and APSI. They contain complex implementations of the well-known kernels assignments, induction variables and reductions. Note that the maximum number of statements is 29 in SIXTRACK, the maximum number of conditions is 6 in APSI, and the maximum number of nested loops is 4 in APPLU and SIXTRACK. These results demonstrate that the representation of programs in terms of computational kernels hides the complexity of the implementation, and eases the understanding of program behavior by the compiler.

Overall, the three families assignments, induction variables and reductions cover, on average, 83% of the computations of the recognized loops. In MATVEC, ADM, TRACK, OCEAN, MG3D and APSI, they cover more than 90% of the computations, being almost 100% in MATVEC and MG3D. The family of complex written arrays covers 6% of the kernels on average. This percentage raises up to 13%, 14% and even 48% in MDG, BLASSM and QCD, respectively. Note that, implicitly, these kernels involve the computation of IVs and reinitialized IVs. Finally, it should be noted that the experiments enabled to identify new computational kernels not studied in the literature so far, in particular, array maps, irregular array recurrences, consecutively reduced (and recurrenced) arrays and segmented consecutively reduced (and recurrenced) arrays.

4 Conclusion

This article has demonstrated that a significant amount of the regular and irregular computations carried out in full-scale real applications can be characterized using the families of computational kernels recognized by the XARK compiler. The representation of programs in terms of kernels hides the complexity of implementation details, providing optimizing compilers with a promising tool to reason about programs and, thus, to guide program optimizations.

In addition, the experiments have shown that full-scale real applications require the recognition of all the kernel families detected by XARK. Finally, note that new kernel families that had not been studied in the literature have been found in the benchmarks.

As future work we intend to give a step forward by describing program behavior using a higher-level of abstraction that consists of dependence relationships between the computational kernels. It is also intended to use such high-level representation for code generation using a stream programming model.

References

1. Andrade, D., Arenaz, M., Fraguela, B.B., Touriño, J., Doallo, R.: Automated and Accurate Cache Behavior Analysis for Codes with Irregular Access Patterns. Concurrency and Computation: Practice and Experience (in press)
2. Arenaz, M., Touriño, J., Doallo, R.: A GSA-Based Compiler Infrastructure to Extract Parallelism from Complex Loops. In: 17th International Conference on Supercomputing, San Francisco, CA, 193–204 (2003)
3. Arenaz, M.: Compiler Support for Parallel Code Generation through Kernel Recognition. In: 18th International Parallel and Distributed Processing Symposium, Santa Fe, NM (2004)
4. Bank, R.E.: PLTMG Package. Available at `http://cam.ucsd.edu/~reb/software.html` [Last accessed May 2007]
5. 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., Martin, J.: The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. International Journal of Supercomputer Applications 3(3), 5–40 (1989)
6. Gerlek, M.P., Stoltz, E., Wolfe, M.: Beyond Induction Variables: Detecting and Classifying Sequences using a Demand-Driven SSA. ACM Transactions on Programming Languages and Systems 17(1), 85–122 (1995)
7. Lin, Y., Padua, D.A.: On the Automatic Parallelization of Sparse and Irregular Fortran Programs. In: O'Hallaron, D.R. (ed.) LCR 1998. LNCS, vol. 1511, pp. 41–56. Springer, Heidelberg (1998)
8. Pottenger, W.M., Eigenmann, R.: Idiom Recognition in the Polaris Parallelizing Compiler. In: 9th International Conference on Supercomputing, Barcelona, Spain, pp. 444–448 (1995)
9. Saad, Y.: SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations (Version 2). Available at `http://www.cs.umn.edu/~saad/software/SPARSKIT/sparskit.html` [Last accessed May 2007]

10. SPEC. SPEC CPU2000. Standard Performance Evaluation Corporation. Available at http://www.spec.org/cpu2000/ [Last accessed May 2007]
11. Suganuma, T., Komatsu, H., Nakatani, T.: Detection and Global Optimization of Reduction Operations for Distributed Parallel Machines. In: 10th International Conference on Supercomputing, Philadelphia, PA, pp. 18–25 (1996)
12. Tu, P., Padua, D.A.: Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers. In: 9th International Conference on Supercomputing, Barcelona, Spain, pp. 414–423 (1995)