A Novel Compiler Support for Automatic Parallelization on Multicore Systems

José M. Andión*, Manuel Arenaz, Gabriel Rodríguez, Juan Touriño

Department of Electronics and Systems, University of A Coruña
Campus de Elviña, s/n, 15071 A Coruña, Spain

Abstract

The widespread use of multicore processors is not a consequence of significant advances in parallel programming. In contrast, multicore processors arise due to the complexity of building power-efficient, high-clock-rate, single-core chips. Automatic parallelization of sequential applications is the ideal solution for making parallel programming as easy as writing programs for sequential computers. However, automatic parallelization remains a grand challenge due to its need for complex program analysis and the existence of unknowns during compilation. This paper proposes a new method for converting a sequential application into a parallel counterpart that can be executed on current multicore processors. It hinges on an intermediate representation based on the concept of domain-independent kernel (e.g., assignment, reduction, recurrence). Such kernel-centric view hides the complexity of the implementation details, enabling the construction of the parallel version even when the source code of the sequential application contains different syntactic variations of the computations (e.g., pointers, arrays, complex control flows). Experiments that evaluate the effectiveness and performance of our approach with respect to state-of-the-art compilers are also presented. The benchmark suite consists of synthetic codes that represent common domain-independent kernels, dense/sparse linear algebra and image processing routines, and full-scale applications from SPEC CPU2000.

Keywords: automatic parallelization, parallelizing compiler, source-to-source compiler, compiler intermediate representation, domain-independent kernel, multicore processor

1. Introduction

Historically, the impressive advances in hardware technology have enabled to increase the performance of applications while preserving the sequential programming model. Nevertheless, the industry decided to replace single power-inefficient processors with many efficient processors on the same chip [1]. This decision has impacted dramatically on the software community, which is unfamiliar with parallel programming and thus has now been forced to develop productive tools for parallel programming.

*Corresponding author: Tel: +34 981 167 000 ext. 1376, Fax: +34 981 167 160
Email addresses: jandion@udc.es (José M. Andión), manuel.arenaz@udc.es (Manuel Arenaz), grodriguez@udc.es (Gabriel Rodríguez), juan@udc.es (Juan Touriño)

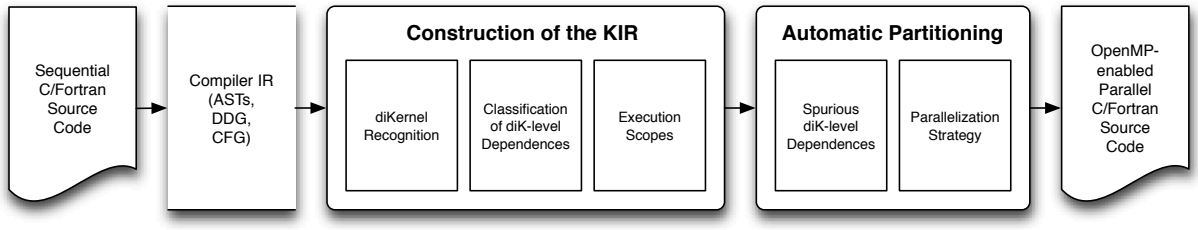


Figure 1: Workflow of the automatic parallelization approach driven by domain-independent kernels.

The parallel challenge has been addressed from different perspectives. Development efforts have mainly focused on using libraries to write parallel programs (e.g. MPI [2], CUDA [3]), on tools to specify parallel semantics within sequential programs (e.g. OpenMP [4], OpenACC [5]), on parallel programming languages (e.g. PGAS languages), and on parallelizing compilers that automatically rewrite sequential programs into a parallel counterpart [6, 7, 8, 9, 10]. It is clear that parallelizing compilers are a critical piece for the software community to meet the parallel challenge. However, despite great advances in compiler technology during the last decades [11, 12], current production compilers usually fail to parallelize even simple sequential programs. The main reason is that they hinge on classical dependence analysis to discover parallelism, and this analysis is extremely sensitive to syntactic variations in the source code. Thus, current compilers are usually unsuccessful in parallelizing codes with pointers and complex control flows.

This paper presents a new approach for the automatic parallelization of sequential programs based on the concept of domain-independent kernel (from now on, *diKernel*), term introduced in [13]. Figure 1 shows the workflow of the approach, which handles dependences of the compiler IR through the diKernels, in contrast to other approaches based on a classical dependence analysis between source code statements. The main contribution is a formal definition of a new compiler intermediate representation based on diKernels (from now on, *KIR*). Our KIR is insensitive to syntactic variations in the source code (e.g., use of arrays, pointers and/or complex control flow), and exposes multiple levels of parallelism to the compiler. Another contribution is an automatic partitioning strategy to map the parallelism exposed by the KIR to modern hardware based on multicore CPUs. The paper also evaluates the effectiveness and the performance of our new KIR-based approach against the GCC [9], Intel [10] and PLUTO [6] compilers. The benchmark suite consists of synthetic codes representative of frequently used diKernels, routines from dense/sparse linear algebra and image processing, and full-scale applications from SPEC CPU2000.

The rest of the paper is organized as follows. Section 2 describes the diKernels used in this paper (namely, assignment, reduction and recurrence diKernels). Section 3 presents our automatic parallelization approach driven by diKernels. More specifically, the new KIR is formally defined, and the automatic partitioning procedure targeting multicore processors is proposed. Section 4 details the behavior of our approach for the case studies of the benchmark suite. Section 5 presents the experimental results. Section 6 discusses related work. Finally, Section 7 concludes the paper and presents future work.

```
1 for (i = 0; i < n; i++) {
2   A[i] = 2;
3 }
```

(a) Regular assignment.

```
1 for (i = 0; i < n; i++) {
2   A[f[i]] = 3;
3 }
```

(b) Irregular assignment.

```
1 r = 0;
2 for (i = 0; i < n; i++) {
3   r = r + i;
4 }
```

(c) Scalar reduction with closed-form expression.

```
1 r = 0;
2 for (i = 0; i < n; i++) {
3   r = r + A[i];
4 }
```

(d) Scalar reduction with array reference.

```
1 r = A[0];
2 for (i = 1; i < n; i++) {
3   if (A[i] < r) r = A[i];
4 }
```

(e) Scalar reduction with conditional control flow.

```
1 for (i = 0; i < n-1; i++) {
2   A[i+1] = A[i+1] + 5;
3 }
```

(f) Regular reduction.

```
1 for (i = 0; i < n; i++) {
2   A[f[i]] = A[f[i]] + 3;
3 }
```

(g) Irregular reduction.

```
1 for (i = 1; i < n; i++) {
2   A[i] = A[i] + A[i-1];
3 }
```

(h) Regular recurrence.

Figure 2: Synthetic codes of representative assignment, reduction and recurrence diKernels.

2. Domain-Independent Kernels

Multiple definitions of the term computational kernel have been proposed in the literature in the context of automatic program analysis [13]. This work focuses on domain-independent computational kernels, named *diKernels*, which have proved to be a useful tool for program behavior characterization [14], automatic parallelization of complex loops [15], as well as for data locality optimization [16]. The diKernels do not represent domain-specific problem solvers. Instead, they characterize the computations carried out in a program from the point of view of the compiler IR. For example, a scalar reduction diKernel represents both the sum of a series of values as well as the dot product of two vectors; and a regular reduction diKernel can represent both the dense and sparse matrix-vector products. Note that diKernels exhibit the essential properties of a program from the point of view of automatic parallelization (reduction operations in the examples above). The rest of this section describes the diKernels used in this paper, namely assignment, reduction and recurrence.

2.1. Assignment diKernels

An *assignment diKernel* consists in storing a set of values in a set of memory addresses. Within a program, addresses are typically represented by scalar variables, memory pointers or indexed variables such as arrays. Thus, different types of assignment diKernels are distinguished. The simplest one is the *scalar assignment* $v = e$, which stores the value of the expression e in the memory address specified by the scalar variable v. The value e is not dependent on v, that is, neither e nor any function call within it contain occurrences of v.

A *regular assignment*, $A[i] = e(i)$ with $i \in \mathbb{N}$ taking values within the range of array A, stores the value of the expression $e(i)$ in the memory location $A[i]$ corresponding to the i^{th} entry of A. Similarly, $e(i)$ is not dependent on A.

As shown in Figure 2a, this diKernel typically represents a conflict-free loop where i is an affine expression of the loop indices.

An *irregular assignment* is represented as $A[f[i]] = e(i)$, where $i \in \mathbb{N}$, and $f[i] \in \mathbb{N}$ takes values within the range of A; f is an indirection array that introduces a compile-time unpredictable access pattern, and $e(i)$ is not dependent on A. As shown in the example of Figure 2b, this diKernel captures the output data dependences that will appear at run time (unless f is a permutation array). Irregular assignments are usually found in computer graphics, finite element applications or sparse matrix computations.

2.2. Reduction diKernels

In contrast to assignment diKernels, the *reduction diKernel* updates a memory location with a new value that depends on the current value. The most popular one is the *scalar reduction*, $v = v \oplus e(i)$ with $i \in \mathbb{N}$ varying over a certain interval of values, where the reduction variable v is a scalar, $\oplus$ is the reduction operator, and $e(i)$ is not dependent on v. Scalar reductions are of widespread use, thus parallel scalar reductions are typically supported by modern programming tools. Figures 2c–2e show typical examples of scalar reductions, ordered by increasing degree of complexity. In Figure 2c there exists a closed-form expression that computes the final value of r (for illustrative purposes, $r = (n^2 + n)/2$ in this case). In Figure 2d, $e(i)$ is an array reference $A[i]$ whose value is unknown at compile-time. Finally, Figure 2e introduces conditional control flows to compute a minimum reduction operation.

A *regular reduction*, $A[i] = A[i] \oplus e(i)$ with $i \in \mathbb{N}$, represents the calculation of the i^{th} entry of the array A, where i is an affine expression of the enclosing loop indices, and $e(i)$ is not dependent on A. In a similar manner, an *irregular reduction*, $A[f[i]] = A[f[i]] \oplus e(i)$, is characterized by the use of an indirection array f that selects the entries of the array A to be updated. Thus, this diKernel captures output and true data dependences that may appear at run time. Figures 2f and 2g show examples of regular and irregular reductions, respectively. Note that irregular reductions are very common in finite element applications and sparse matrix computations.

2.3. Recurrence diKernels

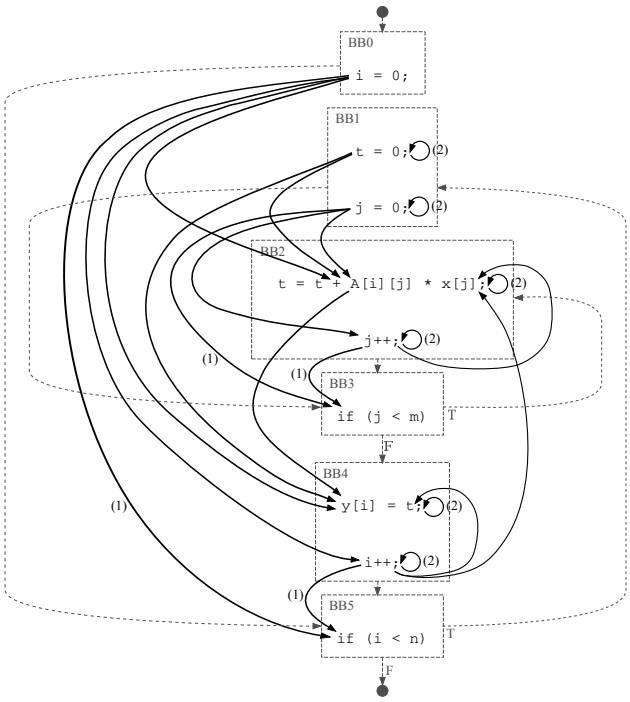
In contrast to reduction diKernels, a *regular recurrence*, $A[i] = A[i_1] \oplus A[i_2] \oplus \ldots \oplus e(i)$ computes a new value for $A[i]$, the indices $i, i_1, i_2 \ldots$ being affine expressions of the enclosing loop indices, and $e(i)$ not dependent on A. The distinguishing property of recurrence diKernels is that there is at least one index $i_x \in \{i_1, i_2 \ldots\}$ such that $i_x \neq i$. In a similar manner, the diKernel is an *irregular recurrence* if at least one index expression of $i, i_1, i_2 \ldots$ contains a reference to an indirection array f. Figure 2h shows an example of regular recurrence.

3. Automatic Parallelization Driven by diKernels

Current compilers typically address the automatic detection of parallelism by running classical dependence analyses on standard statement-based intermediate representations (e.g., Abstract Syntax Trees —ASTs—, Data Dependence Graph —DDG—, Control Flow Graph —CFG—). Such IRs are well suited for code generation, but not for the

```
1 for (i = 0; i < n; i++) {
2   t = 0;
3   for (j = 0; j < m; j++) {
4     t = t + A[i][j] * x[j];
5   }
6   y[i] = t;
7 }
```

(a) Source code.



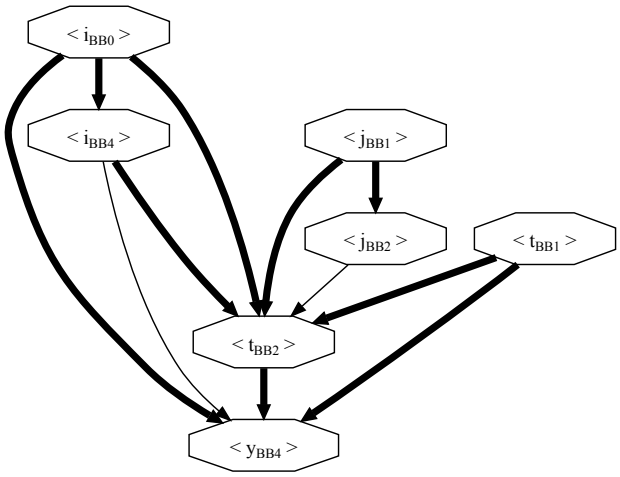
(b) Standard IR (ASTs, CFG and DDG) based on the statements of the program.

Figure 3: Dense matrix-vector multiplication.

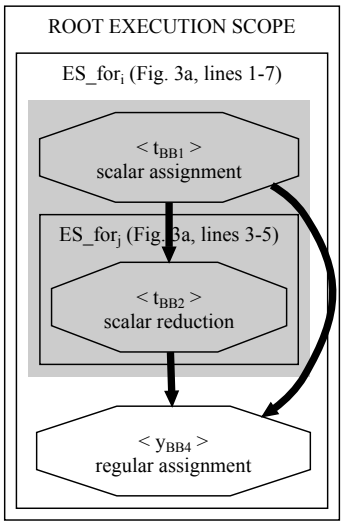
analysis of full-scale applications. This section presents a new approach for the automatic parallelization of sequential programs driven by diKernels. The first stage is the construction of a new IR built on top of diKernels, named KIR, that exposes multiple levels of parallelism in sequential programs. The second stage is an automatic partitioning technique that generates a parallel counterpart for a sequential application targeting multicore processors. Next, Section 3.1 formally defines the KIR and Section 3.2 sketches the KIR-driven automatic partitioning procedure.

3.1. KIR: A diKernel-based Intermediate Representation

Without loss of generality, assume that the source code of a program is represented by a statement-based IR that consists of a forest of ASTs, a DDG and a CFG. For illustrative purposes, Figure 3a shows an implementation of the dense matrix-vector multiplication. In each iteration i of the outer loop for_i, the dot product of the i^{th} row of matrix A and vector x is computed (see lines 2–5). Next, the result is stored in the i^{th} element of vector y (line 6). Figure 3b presents an excerpt of a typical IR where the ASTs represent source code statements. The CFG groups ASTs into basic blocks (dashed boxes) with precedence relationships (dashed edges). Loops are represented with preheader, header and latch basic blocks (BB) that initialize the loop index, check the loop exit condition and increment the loop index (see $BB0$, $BB5$ and $BB4$, respectively, for the loop for_i). Finally, the DDG exhibits data dependences between statements (solid edges).



(a) Steps 1 and 2: diKernel-level data dependence graph (→) with diKernel-level flow dependences (➡).



(b) Step 3: Hierarchy of execution scopes.

Figure 4: Construction of the KIR of the dense matrix-vector multiplication of Figure 3.

The construction of the KIR consists of three steps: first, the construction of the diKernels of the program and their dependence relationships (Definitions 3.1–3.2); second, the construction of flow dependences between diKernels (Definitions 3.3–3.4); and third, the construction of a hierarchy of execution scopes (Definitions 3.5–3.7) that reflects the computational stages of the sequential program and groups diKernels into these stages.

Definition 3.1. *A **diKernel** is a directed graph $K = (N, E)$ where E is the set of edges of a strongly connected component of the DDG, and N is the set of ASTs such that each AST $x_i \in N$ fulfills two conditions: first, x_i is an assignment-statement (thus, it is not a flow-of-control statement —e.g. branch, return, break, jump—); and second, there exist edges $x_i \rightarrow x_j$ or $x_j \rightarrow x_i$ in E where $x_j \in N$. The term $K<x_1 \dots x_n>$ denotes the ASTs $x_1 \dots x_n$ that belong to N.*

Definition 3.2. *Let SCC_x and SCC_y be two strongly connected components of the DDG associated with diKernels $K<x_1 \dots x_n>$ and $K<y_1 \dots y_m>$, respectively. A **diKernel-level data dependence** is an edge $x_i \rightarrow y_j$ of the DDG such that $SCC_x \neq SCC_y$, with $x_i \in \{x_1 \dots x_n\}$ and $y_j \in \{y_1 \dots y_m\}$. The term $K<x_1 \dots x_n> \rightarrow K<y_1 \dots y_m>$ denotes that DDG edge that crosses diKernel boundaries.*

For illustrative purposes, Figure 4a shows the diKernel-level data dependence graph of the dense matrix-vector multiplication of Figure 3. According to Definition 3.1, the branch statements of $BB3$ (`if (j<m)`) and $BB5$ (`if (i<n)`) are ignored for the construction of the diKernels. The computation of the for_i index i (line 1, Figure 3a) is represented by two diKernels: $K<i_{BB0}>$ for the index initialization (the term i_{BB0} denotes the statement `i=0` of the basic block $BB0$ in Figure 3b); and $K<i_{BB4}>$ for the index update. Note that the K of the diKernel notation

is omitted in the figures due to space limitations. In a similar manner, the for_j index j is represented by $K<j_{BB1}>$ and $K<j_{BB2}>$. The value of the dot product is stored in t. This temporary variable is initialized in $K<t_{BB1}>$ at the beginning of each for_i iteration (line 2, Figure 3a) and updated in $K<t_{BB2}>$ throughout the execution of the inner loop for_j (line 4, Figure 3a). Finally, the storage of the dot product value in the output array y is represented by the diKernel $K<y_{BB4}>$. By construction, the edges of the DDG are captured in the diKernel-level data dependence graph as follows: first, the incoming edges of branch statements are ignored (see edges with label (1) in Figure 3b); second, the edges whose source and target statements belong to the same diKernel are subsumed in the diKernel (see edges with label (2) in Figure 3b); and third, the edges that cross diKernel boundaries are exposed as diKernel-level data dependences in Figure 4a (see non-labeled forward and backward edges in Figure 3b).

The second step in the construction of the KIR is to determine flow dependences between diKernels. The diKernel-level data dependence graph does not reflect the order in which diKernels are executed. For this purpose we define dominance relationship between statements (Definition 3.3) before identifying flow dependences between diKernels (Definition 3.4). Note that these definitions take into account the CFG, the DDG, the Dominator Tree (DT) and the production and use of values throughout the program (both for scalar variables and ranges of non-scalar variables —e.g. arrays, structs—).

Definition 3.3. *Let x_i and x_j be ASTs that represent statements of a program. We say there is a* ***statement-level dominance relationship*** *in the following situations:*

- *Assume that x_i and x_j belong to the same basic block BB. If x_i precedes x_j within BB, then x_i dominates x_j.*
- *Assume that x_i and x_j belong to different basic blocks BBi and BBj. If BBi dominates BBj or BBi belongs to the body of a loop whose header BBh dominates BBj, then x_i dominates x_j.*

Definition 3.4. *Let K_x and K_y be diKernels connected by a diKernel-level data dependence $K<x_1 \dots x_n> \rightarrow K<y_1 \dots y_m>$. We say there is a* ***diKernel-level flow dependence****, $K_x \blacktriangleright K_y$, if it holds that statement x_i dominates statement y_j and DEF(x, x_i) $\supseteq$ USE(x, y_j); where $x_i \rightarrow y_j$ is the edge of the DDG associated with $K<x_1 \dots x_n> \rightarrow K<y_1 \dots y_m>$, and DEF(x, x_i)/USE(x, y_j) is the range of values produced/used throughout the execution of statement x_i/y_j.*

For illustrative purposes, Figure 4a highlights the diKernel-level flow dependences of the diKernel-level data dependence graph. As can be seen, $K<i_{BB0}> \blacktriangleright K<i_{BB4}>$ represents the flow between the initialization of the loop index i in the preheader of the loop ($BB0$) and its update in the latch of the loop ($BB4$). The two conditions hold as follows: first, the statement i_{BB0} dominates the statement i_{BB4} because $BB0$ dominates $BB4$; and second, i is a scalar variable, thus DEF(i, i_{BB0}) = USE(i, i_{BB4}) = {i}. The source code of the dense matrix-vector multiplication of Figure 3a does not contain diKernel-level flow dependences between non-scalar variables. Note that, in many programs, dependences are coded in very complex ways, for instance, through the usage of pointers. Our approach deals with pointers in

the algorithm for recognition of diKernels [13], which applies *array recovery* techniques to transform pointer-based programs into a semantically equivalent array-based form (similar to [17]). Illustrative examples of ranges of values of non-scalar variables (both array-based and pointer-based) produced/used in different statements will be described later in Section 4.

The third step in the construction of the KIR is to build the hierarchy of execution scopes. Typically, loops often consume most of the execution time and thus optimizations that improve their performance may have a significant impact on the overall efficiency. The goal of the hierarchy of execution scopes is to expose the computational stages of the program to the compiler. For this purpose, execution scopes are built upon loops (Definition 3.5) and organized in a hierarchy of execution scopes (Definition 3.6). In addition, diKernels are attached to execution scopes (Definition 3.7) to capture the computational stage of the sequential program where they are executed. Finally, diKernels are labeled with the type of assignment, reduction or recurrence diKernel that they carry out during the computation of their execution scope.

Definition 3.5. *Assume that a program is represented by a hierarchy of regions. An* ***execution scope*** *is a loop region R_L such that there exists a perfectly nested loop $L, L_1, \ldots, L_n$, being L the outermost loop.*

Definition 3.6. *The* ***hierarchy of execution scopes*** *is a tree whose set of nodes are the execution scopes of the program. The root node is a special execution scope that represents the program as a whole. The children of a node are built as follows. Let R_L be an execution scope, L its outermost loop, and L_{parent} the parent loop of L. If L_{parent} does not exist, then R_L is set as child of the root execution scope. Otherwise, R_L is set as child of R_{parent}, where R_{parent} is the execution scope of L_{parent}.*

Definition 3.7. *Let $x_1 \ldots x_n$ be the ASTs of a diKernel $K<x_1 \ldots x_n>$. Let $L_1, \ldots, L_n$ be the innermost loops that contain $x_1, \ldots, x_n$, respectively. We say that $K<x_1 \ldots x_n>$* ***belongs to the execution scope*** *R_L if and only if R_L is the execution scope of the innermost common loop for $L_1, \ldots, L_n$. By construction, if x_1 is the index of a loop L, and $K<x_1>$ is the diKernel that initializes this loop index, then $K<x_1>$ belongs to R_L.*

For illustrative purposes, Figure 4b shows the hierarchy of execution scopes. The dense matrix-vector multiplication of Figure 3a contains two loops for_i and for_j that are not perfectly nested. Thus, the execution scope of loop for_j (from now on, ES_for_j) is a child of ES_for_i, which is in turn a child of the root execution scope that represents the whole program. According to Definition 3.7, the diKernels $K<j_{BB1}>$ and $K<j_{BB2}>$ that capture the computation of loop index j are attached to ES_for_j (in a similar manner, $K<i_{BB0}>$ and $K<i_{BB4}>$ are attached to ES_for_i). Note that these diKernels and their incoming/outgoing diKernel-level dependences (e.g. $K<j_{BB1}> \rightarrow K<j_{BB2}>$) are not shown in the KIR of Figure 4b: their computation is not relevant to describe the operations carried out in the program, and loop indices are already represented in the notation of the execution scope and diKernel types. The remaining diKernels $K<t_{BB1}>$, $K<t_{BB2}>$ and $K<y_{BB4}>$ contain a unique assignment-statement, thus they are attached to the execution scope of the innermost loop that contains each statement: ES_for_i, ES_for_j and ES_for_i, respectively.

In summary, the new diKernel-based IR captures the whole semantics of the sequential program, but it only exposes to the compiler the program features that are key to find the parallelism implicit in the sequential code.

3.2. Automatic partitioning driven by the KIR

This section presents a new KIR-driven automatic partitioning technique to transform a sequential program into a parallel counterpart for multicore processors. As input, our approach takes the KIR presented in Section 3.1, which provides the program characteristics needed for the parallelization of the input sequential code. The method consists of two steps. The first step is to filter out the diKernel-level dependences of the KIR that do not prevent the parallelization of the sequential application (from now on, *spurious* diKernel-level dependences). The second step is the construction of an efficient OpenMP-enabled parallelization strategy for the sequential program as a whole.

Regarding the first step, one of the most important techniques to detect spurious diKernel-level dependences is the privatization of program variables, as it avoids memory-related dependences (i.e. anti- and output dependences) [18]. A scalar variable x defined within a loop is said to be privatizable [12] with respect to that loop if and only if every path from the beginning of the loop body to a use of x within that body must pass through a definition of x before reaching that use. Thus, given a set of privatizable scalar variables $x, y, z \ldots$ in a loop L, our technique searches for connected subgraphs of the KIR contained in the execution scope associated to L that represent the computations carried out on the variables $x, y, z \ldots$ Each subgraph, including children execution scopes with only diKernels referencing these variables, is *shaded* in order to omit these parts of the KIR in the discovering of parallelism. Thus, the diKernel-level dependences of the KIR are analyzed to detect the spurious ones.

Definition 3.8. *A diKernel-level dependence is* ***spurious*** *if one of the following conditions is fullfilled:*

1. *Let $K\langle x_i\rangle$ and $K\langle y_j\rangle$ be diKernels connected with a diKernel-level flow dependence $K\langle x_i\rangle \rightarrow K\langle y_j\rangle$. If $K\langle x_i\rangle$ belongs to a shaded subgraph, then $K\langle x_i\rangle \rightarrow K\langle y_j\rangle$ is spurious.*
2. *Let $K\langle x_i\rangle$ and $K\langle y_j\rangle$ be diKernels connected with a diKernel-level data dependence $K\langle x_i\rangle \rightarrow K\langle y_j\rangle$. If x_i dominates y_j and $DEF(x, x_i) \cap USE(x, y_j) = \emptyset$, then $K\langle x_i\rangle \rightarrow K\langle y_j\rangle$ is spurious.*
3. *Consider a sequence of three execution scopes, each one with an attached diKernel $K\langle x_i\rangle$, $K\langle x_j\rangle$ and $K\langle y_l\rangle$, respectively. Assume that the diKernels are connected with the diKernel-level flow dependences $K\langle x_i\rangle \rightarrow K\langle x_j\rangle$, $K\langle x_j\rangle \rightarrow K\langle y_l\rangle$, and $K\langle x_i\rangle \rightarrow K\langle y_l\rangle$. If $DEF(x, x_i) = USE(x, x_j) = DEF(x, x_j) = USE(x, y_l)$, then $K\langle x_i\rangle \rightarrow K\langle y_l\rangle$ is spurious.*

In the dense matrix-vector multiplication of Figure 3a, the scalar variable t is privatizable because every path from the beginning of for_i to the uses at lines 4 and 6 goes through the definition of line 2. Therefore, the KIR of Figure 4b contains a shaded subgraph composed of $K\langle t_{BB1}\rangle$, $K\langle t_{BB2}\rangle$, $K\langle t_{BB1}\rangle \rightarrow K\langle t_{BB2}\rangle$, and the execution scope ES_for_j of the inner loop for_j. Consequently, the diKernel-level dependences $K\langle t_{BB1}\rangle \rightarrow K\langle y_{BB4}\rangle$ and $K\langle t_{BB2}\rangle \rightarrow K\langle y_{BB4}\rangle$ are marked as spurious according to Definition 3.8, case 1.

The second step is to determine the OpenMP-enabled parallelization strategy that will drive the generation of parallel source code. The key idea is to find the critical path in the KIR and execute such computations within a unique parallel region in order to minimize thread creation/destruction. Our approach is based on the existence of parallelizing transformations designed for each type of diKernel. The procedure is as follows: (1) scalar reduction diKernels are executed as parallel reduction operations (using the `reduction` OpenMP clause); (2) regular assignment and regular reduction diKernels are converted into forall parallel loops [19]; (3) irregular assignment and irregular reduction diKernels are transformed via an array expansion technique [20, 21]; (4) in general, recurrence diKernels cannot be transformed in parallel code, but there exist parallelizing transformations for particular cases [22] (examples will be shown in Section 4). Thus, the critical path is the longest path that only contains diKernel-level flow dependences and parallelizable diKernels.

The target architecture addressed in this paper are multicore processors. In general, the parallelism available in parallelizable diKernels will suffice to generate a few coarse-grain threads to run on the multicore processor. As a consequence, when the KIR presents several critical paths that share computations, the non-shared parts are serialized in a unique critical path within the parallel region. Note that no synchronization between the non-shared computations is needed as they are not connected via diKernel-level dependences.

Given a parallel region of a critical path, our automatic partitioning strategy minimizes the synchronization overhead between diKernels. Thus, for each $K\langle x_i\rangle \rightarrow K\langle y_j\rangle$, the algorithm checks that: (1) $K\langle x_i\rangle$ and $K\langle y_j\rangle$ represent conflict-free computations that can be reordered arbitrarily; and (2) given DEF(x, x_i) for $K\langle x_i\rangle$ and USE(x, y_j) for $K\langle y_j\rangle$, then DEF(x, x_i) = USE(x, y_j). Under these conditions, the same workload distribution is scheduled for $K\langle x_i\rangle$ and $K\langle y_j\rangle$ in order to guarantee that the same thread is responsible for producing the value of $K\langle x_i\rangle$ that is consumed by $K\langle y_j\rangle$. As a result, no barrier is needed to preserve the diKernel-level flow dependence $K\langle x_i\rangle \rightarrow K\langle y_j\rangle$. The reordering to assign the same workload distribution is achieved by applying the same OpenMP scheduling clause.

Finally, the creation and destruction of OpenMP threads is minimized. If the parallel region is contained in a loop, OpenMP `parallel` directives are moved to enclose that loop. The critical path is confined between barriers, and the remaining computations in the loop are isolated into OpenMP `single` regions. This situation is very common in numerical simulations, as will be shown in Sections 4.4 and 4.5.

For illustrative purposes, the critical path of the KIR of the dense matrix-vector multiplication (see Figure 4b) consists of a single diKernel $K\langle y_{BB4}\rangle$ attached to ES_for_i. The type of diKernel is parallelizable. Thus, the conflict-free computations of the regular assignment are converted into a forall parallel loop. Note that the variables covered by the shaded subgraph are privatized within the parallel region.

Overall, our approach enables the automatic parallelization of full-scale applications for multicore processors minimizing the parallel overhead. The KIR naturally reflects the structure of the source code and thus avoids the violation of the data-flow constraints specified by the programmer. The next section details more complex examples extracted from both synthetic and real codes.

4. Automatic Parallelization of the Benchmark Suite

In this section, the potential of our KIR-driven automatic parallelization technique is evaluated with a set of benchmarks that are representative of important problems in computational science and engineering. Section 4.1 presents synthetic benchmarks that represent the main types of diKernels. Sections 4.2 and 4.3 describe important routines from dense/sparse linear algebra and image processing. Finally, Sections 4.4 and 4.5 focus on two full-scale applications from the SPEC CPU2000 benchmark suite.

4.1. Synthetic benchmarks

Some simple implementations of assignment, reduction and recurrence diKernels were shown in Figure 2. In all these cases, the KIR consists of one execution scope ES_for_i (apart from the root execution scope) that contains one diKernel (either $K<r_3>$ or $K<A_2>$). Note that the subindex refers to the line number (e.g. the term r_3 refers to the assignment-statement `r=r+i` in line 3 of Figure 2c). The most relevant difference between the examples is the type of diKernel (see the captions of Figure 2).

From the point of view of the automatic partitioning strategy, the examples of Figure 2 present a critical path composed of one diKernel. The parallelizing strategy hinges on the existence of parallelizing transformations specifically designed for each type of diKernel. As a result, the regular assignment of Figure 2a and the regular reduction of Figure 2f represent conflict-free loop iterations that are transformed into forall parallel loops [19]. The scalar reductions of Figures 2c–2e are executed as parallel reductions, which are usually supported in current parallel programming environments. The irregular assignment of Figure 2b and the irregular reduction of Figure 2g present cross-iteration dependences that are handled by parallelizing transformations based on array expansion [20, 21]. Finally, the regular recurrence of Figure 2h is recognized as a parallel prefix operation and an ad-hoc parallelizing transformation is applied [23].

4.2. Dense/sparse matrix-vector multiplication

Different versions of the matrix-vector multiplication have been proposed in the literature. In Section 3, the dense matrix-vector product (*DenseAMUX* from now on) was studied in detail from the point of view of our KIR-driven automatic parallelization approach. In this section, three additional sparse versions extracted from SparsKit-II [24] will be studied: *AMUX*, *AMUXMS* and *ATMUX*.

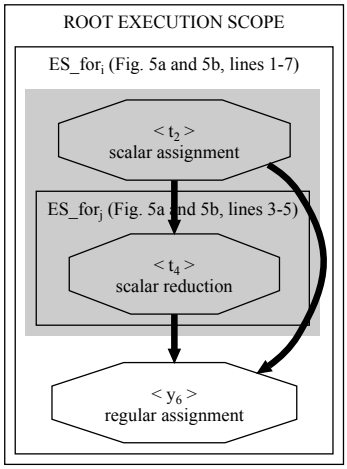
The benchmark AMUX (see Figure 5b) multiplies a matrix A stored in compressed sparse row (CSR) format by a vector x. The source code is very similar to DenseAMUX of Figure 5a, the differences being the subscripted bounds of the inner loop (see $ia[i]$ and $ia[i+1]$ in line 3) and the subscripted subscripts of arrays A, x and ja (see line 4). Thus, both DenseAMUX and AMUX are represented by the same KIR (Figure 5c). Note that the compile-time unknowns introduced in AMUX by the CSR format (subscripted loop bounds and subscripted subscripts) are not exposed in the KIR as they do not determine the implicit parallelism available in the program (they mainly impact on locality exploitation). Consequently, the partitioning strategy behaves as described in Section 3.2 and succeeds in generating

```
1 for (i = 0; i < n; i++) {
2   t = 0;
3   for (j = 0; j < m; j++) {
4     t = t + A[i][j] * x[j];
5   }
6   y[i] = t;
7 }
```

(a) Source code of DenseAMUX (dense matrix-vector multiplication).

```
1 for (i = 0; i < n; i++) {
2   t = 0;
3   for (j = ia[i]; j < ia[i+1]-1; j++) {
4     t = t + A[j] * x[ja[j]];
5   }
6   y[i] = t;
7 }
```

(b) Source code of routine AMUX from SparsKit-II.



(c) KIR for DenseAMUX and AMUX.

Figure 5: Dense and sparse matrix-vector multiplication.

parallel code by privatizing t computed in K<t_2> and K<t_4> and generating a forall loop for the regular assignment K<y_6> computed in for_i.

The benchmark AMUXMS (Figure 6a) multiplies a matrix A in modified sparse row (MSR) format by a vector x. The source code first initializes the output vector y with the product of the diagonal of matrix A (stored in the first n entries of A) and vector x. Next, the remaining operations are computed and accumulated in the result $y[j]$ (with $j \in \{0 \ldots n-1\}$), using a regular access pattern. Again, the key characteristics are the subscripted loop bounds (line 5) and the subscripted subscripts of arrays A, x and ja (line 6). The benchmark ATMUX (Figure 6b) multiplies the transpose of a matrix A in CSR format by a vector x. This routine is very similar to AMUXMS, the only differences being the initial value of y (line 2) and the use of an irregular access pattern to accumulate the results in y (see subscripted subscript $y[ja[l]]$ in line 6).

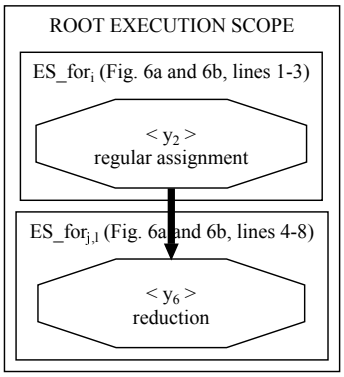
AMUXMS and ATMUX are also represented by a unique KIR (see Figure 6c) with two execution scopes ES_for_i and $ES\_for_{j,l}$. Note that for_j and for_l are two perfectly nested loops and, according to Definition 3.5, they are represented by a unique execution scope. In both routines, ES_for_i contains a regular assignment diKernel K<y_2> (note that the term K<y_2> refers to the assignment-statement *y[i]=...* in line 2 of Figures 6a and 6b). The main difference is the type of reduction diKernel that appears in $ES\_for_{j,l}$. AMUXMS contains a regular reduction K<y_6> that stores values in $y[j]$ (with $j \in \{0 \ldots n-1\}$) and all the subscripted subscripts affect read-only arrays (see line 6 in Figure 6a). In contrast, ATMUX contains an irregular reduction that writes values in $y[ja[l]]$ according to a subscripted access pattern (see line 6 in Figure 6b). Until now, we have illustrated the detection of diKernel-level flow dependences with scalar variables. However, the diKernel-level dependence between K<y_2> and K<y_6> involves a range of values

```
1 for (i = 0; i < n; i++) {
2   y[i] = A[i] * x[i];
3 }
4 for (j = 0; j < n; j++) {
5   for (l = ja[j]; l < ja[j+1]-1; l++) {
6     y[j] = y[j] + A[l] * x[ja[l]];
7   }
8 }
```

(a) Source code of routine AMUXMS from SparsKit-II.

```
1 for (i = 0; i < n; i++) {
2   y[i] = 0;
3 }
4 for (j = 0; j < n; j++) {
5   for (l = ia[j]; l < ia[j+1]-1; l++) {
6     y[ja[l]] = y[ja[l]] + x[j] * A[l];
7   }
8 }
```

(b) Source code of routine ATMUX from SparsKit-II.



(c) KIR for AMUXMS and ATMUX.

Figure 6: Variations of sparse matrix-vector multiplication.

of the non-scalar variable y and, according to Definition 3.4, it is marked as flow due to the following reasons. First, y_2 dominates y_6 because both statements belong to two different basic blocks that belong to loops for_i and for_j such that for_i (lines 1–3) precedes for_j (lines 4–8) and, consequently, all the basic blocks of for_i dominate all the basic blocks of for_j. And second, DEF(y, y_2) $\supseteq$ USE(y, y_6) because K<y_2> produces y[0:n − 1:1] and K<y_6> uses y[0:n − 1:1], the triplet F:L:S defining the range of values between the first position F and the last position L with a stride of S positions. As a result, in AMUXMS, DEF(y, y_2) = USE(y, y_6) = y[0:n − 1:1] and thus K<y_2> ➡ K<y_6> is a diKernel-level flow dependence. The subscripted subscript $y[ja[l]]$ (line 6 in Figure 6b) represents a potential access to any element of array y and the same holds for ATMUX.

The automatic partitioning strategy of Section 3.2 proceeds as follows. The critical path is a unique diKernel-level flow dependence K<y_2> ➡ K<y_6> with two parallelizable diKernels. Consequently, our technique generates a unique parallel region that covers ES_for_i and $ES\_for_{j,l}$. In AMUXMS, both diKernels represent conflict-free computations, their iteration spaces are equal, and DEF(y, y_2) = USE(y, y_6). As a result, for_i and $for_{j,l}$ are parallelized using the same workload distribution in order to avoid any synchronization between them. Regarding ATMUX, the parallelization of the irregular reduction K<y_6> with array expansion requires the initialization of the sections of the expanded array y to the value that reaches the irregular diKernel, 0 in this case. Thus, for_i is not parallelized but replicated in each thread. Finally, $for_{j,l}$ is parallelized using an OpenMP worksharing loop construct.

4.3. Sobel edge filter

The *Sobel edge filter* is a well-known algorithm widely used in image processing and computer vision. It detects the edges of an image, that is, those pixels whose intensity is very different from the intensity of the neighboring pixels. Consider the implementation shown in Figure 7. For each pixel of the original image (see loop nest in lines 8–9), the program computes a convolution *sumX* of the 3×3 matrix *GX* and the intensity of the pixel and its eight neighbors (lines 19–24). A similar convolution *sumY* with the 3×3 matrix *GY* is also computed (lines 25–30). Finally, the sum of the absolute values of *sumX* and *sumY* is truncated to the interval $[0, 255]$ (lines 34–35) and the resulting *SUM* is stored in the output *edgeImage_data* (lines 37–38). Note that, in order to compute the convolutions, *SUM* is set to zero for the pixels at image boundaries (see control flow at lines 13–16).

Figure 8 shows the KIR of the Sobel benchmark. The convolution loops are represented by two execution scopes $ES\_for_{I,J}$ and $ES\_for_{L,M}$ that contain one scalar reduction diKernel $K<sumX_{21}>$ and $K<sumY_{27}>$, respectively. For the sake of clarity, diKernels of the different execution paths of *SUM* are summarized in one diKernel $K<SUM_{35}>$. Note that we have selected a pointer-based implementation of the Sobel edge filter to demonstrate how our framework deals not only with array expressions but also with pointer-based accesses to non-scalar variables through array recovery techniques, as mentioned in Section 3.1. For instance, see the pointer dereference at line 37 of Figure 7. The pointer *edgeImage_data* remains unchanged in the body of the loop nest $for_{Y,X}$. The index of the outer loop *Y* ranges from 0 to *originalImage_rows-1* with step 1. The index of the inner loop *X* ranges from 0 to *originalImage_cols-1* with step 1, and *originalImage_cols* multiplies the index *Y* in the dereference expression. Thus, the loop nest is traversing the *edgeImage_data* memory region as an array, row-by-row, and the pointer-based expression can be rewritten as *edgeImage_data[Y][X]*.

With respect to the parallelization strategy, *SUM* is a privatizable scalar variable because every use within the loop body is dominated by a definition at the beginning of the loop nest (e.g., uses of *SUM* at lines 34–38 are dominated by definitions at lines 14, 16, 31, 34 and 35). Scalar variables *sumX* and *sumY* are also privatized. By ignoring the shaded subgraph, the critical path consists of a unique diKernel $K<edgeImage\_data_{37}>$ executed in the scope of $ES\_for_{Y,X}$. As the type of diKernel is a regular assignment, the automatic partitioning strategy succeeds as described for the dense matrix-vector multiplication (see Section 4.2).

4.4. SWIM from SPEC CPU2000

The SWIM application performs a weather prediction based on a numerical model of the shallow-water equations. It consists of an initialization phase and a time integration phase. In each time step, the subroutines `CALC1`, `CALC2` and `CALC3` (`CALC3Z` in the first time iteration) are called. For illustrative purposes, Figure 9a only shows part of the code of `CALC1` (remaining computations are very similar). Note that there exist data dependences between iterations that prevent the parallel execution of different time steps. Thus, the rest of this section focuses on the automatic parallelization of a time step loop iteration (see lines 3–19 of Figure 9a; contents of ES_for_{NCYCLE} in the KIR of Figure 9b) and such cross-iteration dependences are not shown either in the source code or in the KIR. Two loops in

```
void sobel(unsigned char *edgeImage_data, unsigned char *originalImage_data,
    int originalImage_rows, int originalImage_cols) {

  int                 GX[3][3], GY[3][3];
  int                 X, Y, I, J, L, M;
  long                sumX, sumY, SUM;

  for(Y = 0; Y <= (originalImage_rows - 1); Y++)  {
    for(X = 0; X <= (originalImage_cols - 1); X++)  {
      sumX = 0;
      sumY = 0;

      if(Y == 0 || Y == (originalImage_rows - 1))
        SUM = 0;
      else if(X == 0 || X == (originalImage_cols - 1))
        SUM = 0;

      else {
        for(I = -1; I <= 1; I++) {
          for(J = -1; J <= 1; J++) {
            sumX = sumX + (int)( (*(originalImage_data + X + I +
                        (Y + J) * originalImage_cols)) * GX[I+1][J+1]);
          }
        }
        for(L = -1; L <= 1; L++) {
          for(M = -1; M <= 1; M++) {
            sumY = sumY + (int)( (*(originalImage_data + X + L +
                        (Y + M) * originalImage_cols)) * GY[L+1][M+1]);
          }
        }
        SUM = abs(sumX) + abs(sumY);
      }

      if(SUM > 255) SUM = 255;
      if(SUM < 0) SUM = 0;

      *(edgeImage_data + X + Y * originalImage_cols) =
                  255 - (unsigned char)(SUM);
    }
  }
}
```

Figure 7: Source code of the Sobel application.

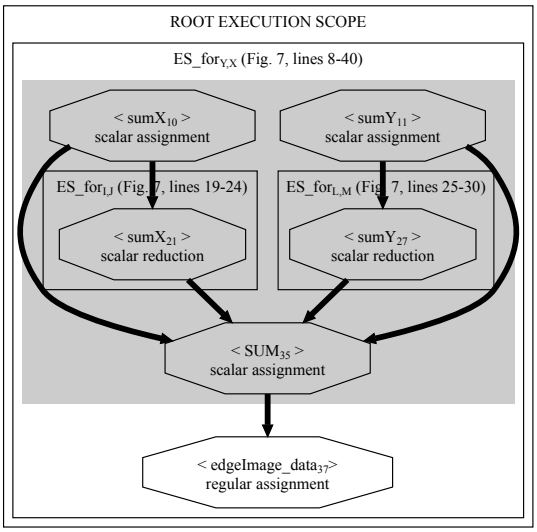
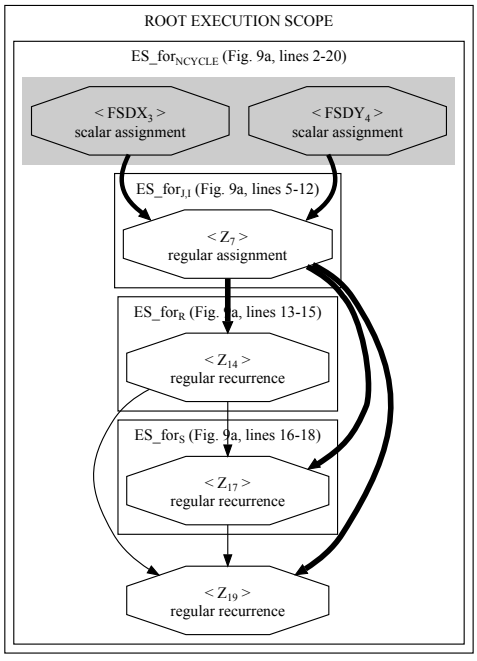


Figure 8: KIR of the Sobel application of Figure 7.

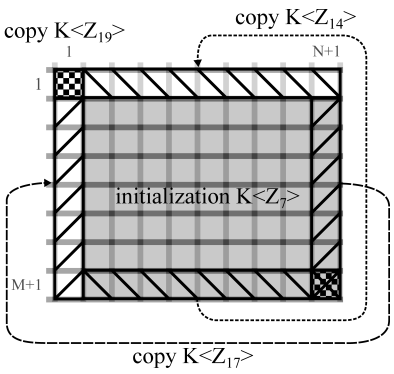
```
1  PROGRAM SHALOW
2    DO 90 NCYCLE=1,ITMAX
3      FSDX = 4.DO/DX
4      FSDY = 4.DO/DY
5      DO 100 J=1,N
6        DO 100 I=1,M
7          Z(I+1,J+1) =
8            (FSDX*(V(I+1,J+1)-V(I,J+1))
9            -FSDY*(U(I+1,J+1)-U(I+1,J)))
10           /(P(I,J)+P(I+1,J)
11           +P(I+1,J+1)+P(I,J+1))
12     100 CONTINUE
13     DO 110 R=1,N
14       Z(1,R+1) = Z(M+1,R+1)
15     110 CONTINUE
16     DO 115 S=1,M
17       Z(S+1,1) = Z(S+1,N+1)
18     115 CONTINUE
19     Z(1,1) = Z(M+1,N+1)
20 [...]
```

(a) Excerpt of the source code.



(b) KIR.

Figure 9: Excerpt of the source code and KIR of the SWIM application.



diKernel	Defined and used data
$K<Z_7>$	DEF(Z, Z_7)=Z[2:M+1:1][2:N+1:1] USE(Z, Z_7)=Z[::][::]
$K<Z_{14}>$	DEF(Z, Z_{14})=Z[1:1:1][2:N+1:1] USE(Z, Z_{14})=Z[M+1:M+1:1][2:N+1:1]
$K<Z_{17}>$	DEF(Z, Z_{17})=Z[2:M+1:1][1:1:1] USE(Z, Z_{17})=Z[2:M+1:1][N+1:N+1:1]
$K<Z_{19}>$	DEF(Z, Z_{19})=Z[1:1:1][1:1:1] USE(Z, Z_{19})=Z[M+1:M+1:1][N+1:N+1:1]

Figure 10: Data analysis of the source code and KIR of Figure 9 (SWIM).

$ES\_for_{J,I}$ (lines 5–12) compute a subset of the values of matrix Z using as input matrices U, V and P calculated in previous time steps. Next, the loop in ES_for_R (lines 13–15), the loop in ES_for_S (lines 16–18) and an assignment-statement (line 19) copy values of Z from the last row/column to the first row/column.

The KIR contains a sequence of execution scopes $ES\_for_{J,I}$, ES_for_R and ES_for_S one after another. First, a regular assignment $K<Z_7>$ represents the conflict-free computations of the first loop $for_{J,I}$ (lines 5–12). Second, the regular recurrence diKernel $K<Z_{14}>$ captures the copy of Z values to the first row of Z (in a similar manner, $K<Z_{17}>$ captures the copies to the first column of Z). And third, $K<Z_{19}>$ also copies the element $Z(M+1, N+1)$ to $Z(1, 1)$. Figure 10 illustrates the ranges of defined/used values of Z for each diKernel. Finally, note that the KIR contains a shaded subgraph with $K<FSDX_3>$ and $K<FSDY_4>$ that capture the privatizable scalar variables *FSDX* and *FSDY*.

The most relevant issue of this KIR is that $K<Z_7>$, $K<Z_{14}>$, $K<Z_{17}>$ and $K<Z_{19}>$ are connected with flow and data diKernel-level dependences. Thus, the automatic partitioning strategy starts by searching spurious diKernel-level dependences. First, $K<FSDX_3> \rightarrow K<Z_7>$ and $K<FSDY_4> \rightarrow K<Z_7>$ are spurious because $K<FSDX_3>$ and $K<FSDY_4>$ belong to the shaded subgraph (see Definition 3.8, case 1). In addition, $K<Z_{14}> \rightarrow K<Z_{17}>$ is spurious because the statement Z_{14} dominates Z_{17} but Z_{14} modifies a set of Z entries (DEF(Z, Z_{14}) = Z[1:1:1][2:N + 1:1]) that is not used in Z_{17} (USE(Z, Z_{17}) = Z[2:M + 1:1][N + 1:N + 1:1]) and thus DEF(Z, Z_{14}) ∩ USE(Z, Z_{17}) = ∅ (see Definition 3.8, case 2). In a similar manner, $K<Z_{14}> \rightarrow K<Z_{19}>$ and $K<Z_{17}> \rightarrow K<Z_{19}>$ are also spurious diKernel-level dependences. The rest of the KIR contains three critical paths $K<Z_7> \rightarrow K<Z_{14}>$, $K<Z_7> \rightarrow K<Z_{17}>$, and $K<Z_7> \rightarrow K<Z_{19}>$ that share the computations of $K<Z_7>$. Thus, a parallel region that encloses the three critical paths is created and the computations of $K<Z_{14}>$, $K<Z_{17}>$ and $K<Z_{19}>$ are executed in sequence after a barrier to avoid diKernel-level flow dependences violation. $K<Z_7>$ is a regular assignment and is transformed in a forall loop. $K<Z_{14}>$ is a regular recurrence but, as DEF(Z, Z_{14}) ∩ USE(Z, Z_{14})= ∅, it can be transformed in a forall loop. The same is true for $K<Z_{17}>$. $K<Z_{19}>$ is a diKernel with an only statement and is executed into an OpenMP `single` region. Due to their similarities, the same analysis of `CALC1` is applied to `CALC2`, `CALC3` and `CALC3Z`. Finally, the

location of the OpenMP `parallel` directive is optimized: the parallel region is enclosed in the time integration loop, thus directives are moved to enclose the loop for_{NCYCLE} and a barrier synchronizes the execution of each iteration (note that the critical path comprises the whole time iteration). In this manner, creation and destruction of threads is minimized.

4.5. EQUAKE from SPEC CPU2000

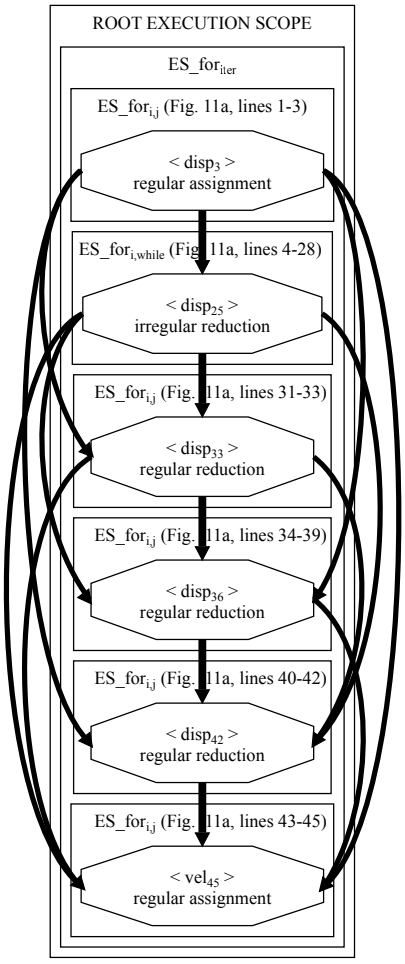
The *EQUAKE* application simulates seismic waves in large, highly heterogeneous valleys. EQUAKE is able to recover the time history of the ground motion caused by a seismic event in any place of a valley. An unstructured mesh is used to locally resolve wavelengths with a finite element method. As a result, EQUAKE reports the displacements at both the hypocenter and epicenter of the earthquake for a predetermined number of simulation time steps. The most time-consuming part of EQUAKE is a time integration loop that computes this displacement. Similarly to SWIM, there are dependences between consecutive time iterations. Thus, the rest of this section focuses on a time step loop iteration.

An excerpt of the source code of EQUAKE is shown in Figure 11a. For the sake of clarity, the KIR of Figure 11b does not show the diKernels of privatizable scalar variables. In each time step, a sequence of diKernels ($K<disp_3>$ to $K<disp_{42}>$) compute the displacement one after another, before computing the velocity ($K<vel_{45}>$). The analysis of one iteration of the time integration loop shows that the indices *disptminus*, *dispt* and *disptplus* are constant in each time iteration and they reference disjoint submatrices of the array *disp* (representing the displacement in the current and the two previous time steps). As a result, we can consider these submatrices as totally independent matrices and thus diKernels $K<disp_{25}>$ and $K<disp_{36}>$ are reductions and not recurrences.

The automatic partitioning procedure marks as spurious all diKernel-level flow dependences except the sequence of the six parallelizable diKernels $K<disp_3> \rightarrow K<disp_{25}> \rightarrow K<disp_{33}> \rightarrow K<disp_{36}> \rightarrow K<disp_{42}> \rightarrow K<vel_{45}>$ because all diKernels compute and use the whole *disp[disptplus]* matrix (see Definition 3.8, case 3). As a result, this is the critical path. The irregular reduction of $K<disp_{25}>$ is parallelized applying array expansion. As illustrated with AMUXMS (see the last paragraph of Section 4.2), this technique requires the initialization of the private arrays and $K<disp_3>$ is replicated in each thread. The remaining diKernels are transformed into forall loops. A barrier is inserted after the irregular reduction $K<disp_{25}>$ to consolidate the results of the private arrays in the original matrix. In contrast, the series of diKernels $K<disp_{33}> \rightarrow K<disp_{36}> \rightarrow K<disp_{42}> \rightarrow K<vel_{45}>$ can use the same OpenMP schedule clause and be parallelized without intermediate barriers. The creation of the parallel region is optimized enclosing the whole time integration loop as explained with SWIM (see Section 4.4). This example shows the potential of our approach to support not only the parallelization of isolated routines, but also the parallelization of full-scale applications. In addition, the loops of the application are not analyzed in isolation, which enables the generation of more efficient parallel code with the creation of a unique parallel region.

```
1  for (i = 0; i < ARCHnodes; i++)
2     for (j = 0; j < 3; j++)
3        disp[disptplus][i][j] = 0.0;
4  for (i = 0; i < ARCHnodes; i++) {
5     Anext = ARCHmatrixindex[i]; Alast = ARCHmatrixindex[i+1];
6     sum0 = K[Anext][0][0] * disp[dispt][i][0]
7        + K[Anext][0][1] * disp[dispt][i][1]
8        + K[Anext][0][2] * disp[dispt][i][2];
9     sum1 = K[Anext][1][0] * ...; sum2 = K[Anext][2][0] * ...;
10    Anext++;
11    while (Anext < Alast) {
12       col = ARCHmatrixcol[Anext];
13       sum0 += K[Anext][0][0] * disp[dispt][col][0]
14          + K[Anext][0][1]*disp[dispt][col][1]
15          + K[Anext][0][2]*disp[dispt][col][2];
16       sum1 += K[Anext][1][0]*...; sum2 += K[Anext][2][0]*...;
17       disp[disptplus][col][0] +=
18          K[Anext][0][0] * disp[dispt][i][0]
19          + K[Anext][1][0] * disp[dispt][i][1]
20          + K[Anext][2][0] * disp[dispt][i][2];
21       disp[disptplus][col][1] += K[Anext][0][1] ...
22       disp[disptplus][col][2] += K[Anext][0][2] ...
23       Anext++;
24    }
25    disp[disptplus][i][0] += sum0;
26    disp[disptplus][i][1] += sum1;
27    disp[disptplus][i][2] += sum2;
28 }
29
30 time = iter * Exc.dt;
31 for (i = 0; i < ARCHnodes; i++)
32    for (j = 0; j < 3; j++)
33       disp[disptplus][i][j] *= - Exc.dt * Exc.dt;
34 for (i = 0; i < ARCHnodes; i++)
35    for (j = 0; j < 3; j++)
36       disp[disptplus][i][j] +=
37          2.0 * M[i][j] * disp[dispt][i][j]
38          - (M[i][j] - Exc.dt / 2.0 * C[i][j])
39          * disp[disptminus][i][j] - ...
40 for (i = 0; i < ARCHnodes; i++)
41    for (j = 0; j < 3; j++)
42       disp[disptplus][i][j] /= (M[i][j] + Exc.dt / 2.0 * C[i][j]);
43 for (i = 0; i < ARCHnodes; i++)
44    for (j = 0; j < 3; j++)
45       vel[i][j] = 0.5 / Exc.dt * (disp[disptplus][i][j] - disp[disptminus][i][j]);
46
47 i = disptminus; disptminus = dispt; dispt = disptplus; disptplus = i;
```

(a) Excerpt of the source code.



(b) KIR.

Figure 11: Excerpt of the source code and KIR of the EQUAKE application.

5. Evaluation

As shown in the previous sections, our diKernel-based automatic parallelization strategy is effective to handle full-scale applications with arrays, pointers and complex control flows. In this section, it is compared with current parallelizing compilers using the benchmark suite presented in Section 4. Hereafter, Section 5.1 describes the experimental platform, Section 5.2 discusses the experimental results in terms of effectiveness and Section 5.3 shows the performance of the generated OpenMP-enabled parallel code in terms of execution times and speedups.

5.1. Experimental platform

The target multicore system consists of 2 Intel Xeon E5520 quad-core Westmere processors at 2.27GHz with 8MB of cache memory per processor and 8GB DDR3 1333 of shared memory. The operating system is CentOS release 5.3 GNU/Linux 2.6.18-128.1.14.el5 SMP x86_64.

Three state-of-the-art compilers have been selected to be compared with our proposal. The first one is the GNU Compiler Collection [9] (from now on, *GCC*) version 4.5.2. It supports automatic parallelization generating OpenMP code by means of the Graphite framework, based on a polyhedral representation [25]. The compilation options are `-march=core2 -msse4 -O2 -floop-parallelize-all -ftree-parallelize-loops=8`. The second one is the Intel C++/Fortran Compiler [10] (from now on, *ICC*) version 11.1 for the *intel64* architecture, that also supports automatic parallelization. The compilation flags are `-O2 -xSSE4.2 -parallel`. The third one is the *PLUTO* automatic parallelization research tool [6] version 0.6.0. It uses the polyhedral model to transform C programs into OpenMP code supporting efficient tiling and fusion. The compilation flags are `--tile --parallel`. Finally, our KIR-driven automatic partitioning approach (from now on, *KIR*) generates OpenMP source code and is built on top of GCC version 4.4.0.

5.2. Experimental results: Effectiveness

Table 1 shows a summary of the effectiveness of GCC, ICC, PLUTO and KIR for the benchmarks described in Section 4. The table summarizes for each benchmark some program characteristics that impact on the effectiveness of the compilers: type of the most representative diKernel (*diKernel*), existence of irregular computations in writes (*Irreg. writes*) and reads (*Irreg. reads*), existence of compile-time unknowns in loop bounds (*Unknown LB*), complex control flows (*Complex CF*) and the use of temporary variables to store intermediate results (*Temp. vars*). The effectiveness of each compiler for automatic parallelization is measured in terms of success ($\surd$), partial success ($\approx$), failure (blank table entries), or unsupported input programming language (*U*).

The synthetic benchmarks have been designed to expose different types of diKernels of increasing complexity. The *regular assignment* benchmark (see Figure 2a) is successfully parallelized by all the compilers because it is a simple array-based implementation with affine subscript expressions. The same holds for the *regular reduction* benchmark (Figure 2f). The introduction of an indirection array f that selects the locations to be updated in the *irregular assignment* (Figure 2b) causes the failure of GCC, ICC and PLUTO. GCC considers the data reference as

Table 1: Effectiveness of GCC, ICC, PLUTO and KIR for the benchmark suite.

Benchmark		Program Characteristics						Compilers			
		diKernel	Irreg. writes	Irreg. reads	Unknown LB	Complex CF	Temp. vars	GCC	ICC	PLUTO	KIR
Synthetic	regular assignment	regular assignment						√	√	√	√
	irregular assignment	irregular assignment	√		√						√
	scalar reduction 1	scalar reduction						≈	√		√
	scalar reduction 2	scalar reduction						≈	√		√
	scalar reduction 3	scalar reduction				√		≈	√		√
	regular reduction	regular reduction						√	√	√	√
	irregular reduction	irregular reduction	√	√	√						√
	regular recurrence	regular recurrence									√
Algebra	DenseAMUX	regular assignment					√		√	≈	√
	AMUX	regular assignment		√	√		√				√
	AMUXMS	regular reduction		√	√						√
	ATMUX	irregular reduction	√	√	√						√
Image	sobel1	regular assignment				√	√				√
	sobel2	regular assignment					√	√			√
Apps	SWIM	regular recurrence				√			√	*U*	√
	EQUAKE	irregular reduction	√	√	√				≈		√

not analyzable and ICC fails because it assumes output dependences. In contrast, KIR successfully handles diKernels with irregular computations in write operations. Note that the parallelization of the *irregular reduction* (Figure 2g) is also unsuccessful for GCC, ICC and PLUTO. Regarding the *regular recurrence* (Figure 2h), KIR detects that it is a parallel prefix sum and generates parallel code, while none of the contenders succeeds in parallelizing the benchmark. Finally, the analysis of the three scalar reductions (Figures 2c, 2d and 2e) provides more details about the behavior of the compilers: ICC and KIR parallelize the three implementations, GCC recognizes the scalar reductions, but no parallel code is generated, and PLUTO fails.

The linear algebra routines are variations of dense/sparse matrix-vector products. The *DenseAMUX* benchmark (see Figure 5a) is successfully handled by ICC: the outer loop is parallelized. PLUTO has partial success because it is very sensitive to syntactic variations in the source code. It does not parallelize DenseAMUX due to the use of the temporary variable *t* to store the dot product of a matrix row and the vector (see lines 3–5 in Figure 5a). If the code is rewritten without *t*, then PLUTO parallelizes the benchmark. GCC is not able to parallelize the outer loop.

As mentioned in Section 4.2, the sparse *AMUX* is very similar to *DenseAMUX* from the point of view of the KIR. However, ICC fails to parallelize, indicating that the loop structure is unsupported because the loop index variable requires complex computation (that is, there exist unknown loop bounds and irregular reads). GCC and PLUTO again fail to parallelize. Regarding *AMUXMS*, it consists of two separated loops. The first loop (see lines 1–3 in Figure 6a) is an example of regular assignment, and the compilers succeed with this type of diKernel, as mentioned above. The second loop (see lines 4–8 in Figure 6a) consists of a regular reduction that cannot be parallelized by the compilers, again due to the presence of irregular reads and unknown loop bounds. Finally, GCC, ICC and PLUTO also fail in *ATMUX* (see line 6 in Figure 6b) which includes an irregular reduction.

The benchmark *sobel1* is an implementation of the Sobel edge filter that contains a complex control flow for processing the pixels at the image boundaries (see lines 13–16 in Figure 7). GCC, ICC and PLUTO cannot parallelize *sobel1*. In contrast, the *sobel2* version removes the complex control flow by processing image boundaries in two separated loops. In this case, GCC parallelizes the application by unrolling the convolution loops (see lines 19–24 and 25–30); ICC and PLUTO fail because the convolution loops have complex subscripts. Rewriting the Sobel application using arrays instead of using pointers provides the same results. The Sobel benchmarks exhibit one of the main weaknesses of state-of-the-art compilers, that is, they are strongly dependent on implementation details such as subscripts, complex loop bounds or complex control flows.

SWIM is a full-scale application with regular computations only. Nevertheless, GCC is unable to parallelize any piece of code out of the initialization subroutine. Focusing on `CALC1`, ICC only parallelizes the first loop (see lines 5-12 in Figure 9a), discarding the recurrences because they are considered not to have enough workload. SWIM is written in Fortran, thus PLUTO cannot handle this code.

The last benchmark is the *EQUAKE* application. GCC only parallelizes some regular computations carried out in the auxiliary functions. In contrast, ICC is able to parallelize the computations of the regular reduction $K<disp_{33}>$ (see Figure 11b). Note that both GCC and ICC execute the irregular reduction $K<disp_{25}>$ sequentially. PLUTO

Table 2: Memory consumption in the parallelization of $K<disp_{25}>$ of EQUAKE.

#Threads	Pure array-expansion		Optimized array-expansion		
	Elements	KB	Elements	KB	Improvement (%)
2	181014	1414	83490	652	-54 %
4	362028	2828	253872	1983	-30 %
8	724056	5657	557031	4352	-23 %

cannot handle this benchmark.

Overall, we have demonstrated that GCC, ICC and PLUTO are, in general, effective compilers in parallelizing regular computations and scalar reductions, specially in synthetic benchmarks and routines from libraries. In contrast, these approaches have shown to be ineffective with irregular computations and full-scale applications. KIR overcomes these limitations handling a comprehensive set of codes in a unified manner.

5.3. Experimental results: Performance

This section presents a comparison in terms of performance. As representative example we have selected EQUAKE, a full-scale application that combines regular and irregular computations. The OpenMP-enabled parallel code generated by our KIR-driven automatic partitioning approach has been compiled with the Intel C++/Fortran Compiler (*KIR/ICC*) with the flags `-O2 -xSSE4.2 -openmp`, due to the fact that the contender is the same *ICC* compiler with the automatic parallelization support enabled.

As mentioned in Section 4.5, the irregular reduction $K<disp_{25}>$ (see Figure 11) is parallelized with an array expansion technique. Its main drawback is that memory consumption may be high because it grows proportionally to the number of threads: the number of elements of the expanded array is $ARCHnodes \times 3 \times \#threads$ (with $ARCHnodes = 30169$, the number of nodes of the unstructured grid topology that represents the valley where the simulation is performed). In order to reduce this overhead, our automatic approach fine-tunes the OpenMP parallel code. First, it imposes one of the threads to use the original array *disp* as its section of the expanded array. And second, it introduces an inspector before the time integration loop to determine the highest index of *disp* that is referenced by each thread. In this manner, our technique allocates less quantity of memory in each section of the expanded array. Table 2 shows a comparison between the memory needed by a pure array-expansion implementation and the optimized code generated by our approach.

Figure 12 shows the execution times and speedups of EQUAKE for a number of threads ranging from 1 to 8. Note that the total execution time is decomposed showing the time of the irregular reduction $K<disp_{25}>$ (*Irregular*), the overhead of the OpenMP parallelization of KIR (*Overhead*) and the remaining time (*Remaining*). The label $WL \times 1$ shows the results for the workload *ref* from SPEC CPU2000. The execution time of the sequential application has been taken as baseline (see horizontal line at 24.47 seconds) to calculate the speedups. Note that the *KIR/ICC*

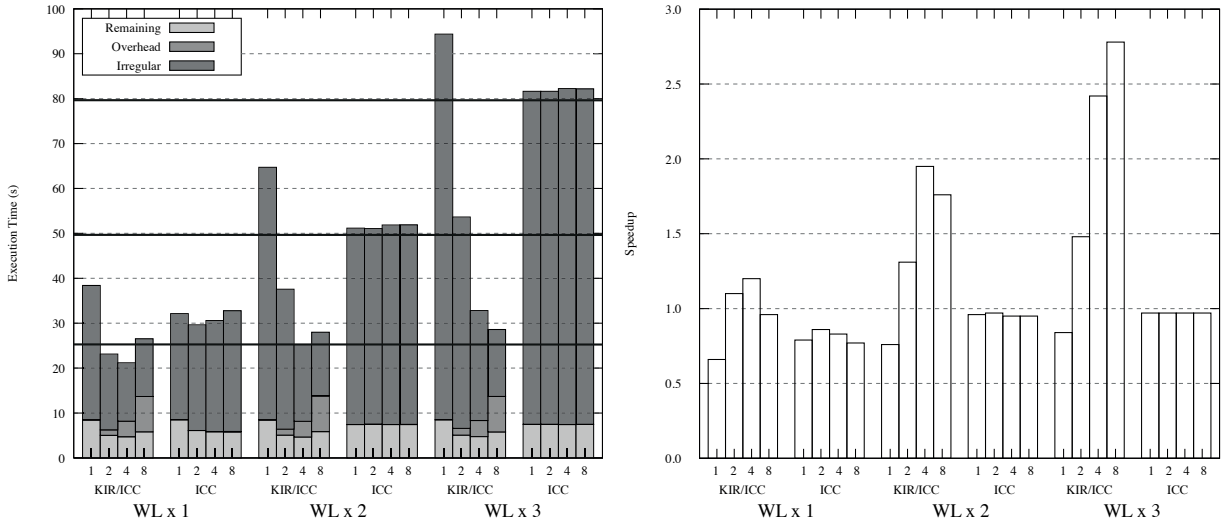


Figure 12: Execution times and speedups of EQUAKE.

execution time using one thread is increased due to the different optimizations applied by the Intel compiler into an OpenMP region and the inspector overhead. As can be observed, the KIR-driven approach outperforms ICC with 2, 4 and 8 threads up to a 30%. However, the increase in the number of threads does not have a significant impact on performance due to the low workload of the application. Performance results are also shown for higher workloads: $WL \times 2$ and $WL \times 3$ (twice and thrice the workload $WL \times 1$). The sequential execution times are shown in the graph at 49.32s and 79.49s, respectively. Note that ICC cannot reduce the execution time of the benchmark because it does not parallelize the irregular computations, which is the most costly part of the EQUAKE application. In contrast, KIR is able to reduce the sequential execution time up to a 64% (for $WL \times 3$ and 8 threads), revealing that addressing irregular computations is paramount for parallelizing full-scale applications.

6. Related Work

Many approaches have been explored to address the parallel challenge targeting current multicore processors. The automatic rewriting of sequential programs into a parallel counterpart is the ideal solution, but it remains as an open research subject due to the complexity of dealing with full-scale applications. Several IRs have been proposed in the literature to support automatic parallelization. Typically, the IR consists of forests of ASTs+DDG+CFG and it is analyzed with statement-based dependence analysis techniques. This approach is used in modern compilers, although it is unsuccessful handling syntactical variations in the source code. Next, we discuss alternative IRs for the automatic parallelization proposed in the literature.

The polyhedral model [25] is a mature technology that has reached production (e.g. GCC and IBM XL) and research compilers (e.g. PLUTO). It is a mathematical framework for loop nest parallelization and optimization. Its main drawback is that its scope of application is limited only to static-control, regular loop nests (see Section 5.2). A

recent extension [26] removes these limitations addressing general `while` loops and `if` conditions, although irregular data accesses are modeled conservatively (e.g., an array with a complex subscript is considered as a single variable).

Sato and Iwasaki [27] address the parallelization of complex reductions and scans. They transform the loop body into a matrix-multiplication form based on reduce and scan parallel primitives. In addition, their technique extracts max-operators from `if` statements automatically, enabling the parallelization of loops with complex control flows. However, this method does not address loop bodies with pointers or subscripted subscripts.

Liu et al. [28] target iteration-level parallelism as a graph optimization problem. They build a dependence graph for each loop, where nodes are the loop statements and edges represent dependence relations. The edges are weighted to indicate an intra-iteration dependence or the distance between inter-iteration dependences. Retiming is used to model dependence migration, maximizing the iterations that can be run in parallel. A new loop is generated from the optimized graph. This technique has not been integrated into a compiler to be evaluated on a multicore processor.

Decoupled Software Pipelining (DSWP) [29] proposes to divide a loop into critical path and off-critical path threads that run concurrently but communicate in a pipelined manner. First, this technique builds the Program Dependence Graph (PDG) of a loop and searches for strongly connected components (SCCs) on it. As a result, a directed acyclic graph (DAG) of SCCs is generated. Next, this DAG is partitioned into threads (1) maintaining all instructions of a SCC in the same thread and (2) balancing the estimated cycles necessary to execute each thread. Edges of the DAG that cross partition boundaries represent data values and control conditions that are communicated through produce/consume operations over a queue. This approach limits the performance improvement to the number and size of the SCCs. It has been extended by Huang et al. [30] introducing DSWP+. Instead of balancing the computational load between the pipelined threads, DSPW+ puts as much work as possible in the stages that can be subsequently parallelized with other techniques (e.g. forall, speculation, localwrite). This process was hand-made, although it has been automatized in the Nova compiler of the Parcae system by Raman et al. [31]. However, it targets only the hottest outermost loop nest and not the whole application.

The Paralax Infrastructure [32] proposes a combined approach for automatic extraction of parallelism in irregular codes. First, full-data structure SSA and use/def chains are used to compute the SCCs on the PDG of a loop and extract pipeline parallelism. A static performance model predicts the speedup of the parallelization and only loops with significant speedups are parallelized. Second, a light-weight programming model based on annotations helps the compiler to find thread-level parallelism. In this manner, they overcome the problem of determining the last definition of arrays. These annotations must be inserted by the programmer, although a tool has been developed to suggest these annotations based on profiling. Our approach also addresses irregular computations, but employs range-based analysis of arrays to overcome this problem. Moreover, OpenMP parallel code is generated instead of proposing a new programming environment that must be learnt by the user.

Canedo et al. [33] present a fully automatic parallelization approach based on a new IR called Concurrent PDG. This new IR models the whole application and has been implemented in a compiler. Nevertheless, it is only applicable to Simulink, a model-based design engineering tool that uses block diagram notation to describe mathematical models

of dynamic systems and controllers. Our approach targets general-purpose programming languages.

Tournavitis and Franke [34] propose IR Profiling, a hierarchical whole program representation focused on the extraction of pipeline parallelism. From the original sequential program, an instrumented executable is generated. The application is then executed with several input files, generating a set of trace files. The new IR, based on the PDG, is built upon these traces. Finally, a heuristic-guided partitioning algorithm produces the specification of the pipeline stages and parallel code is generated accordingly. The main drawback of this approach is that the dependences to build IR Profiling depend on the employed input files. The IR will be only correct for these concrete inputs, not for general ones. As the wauthors expect, the user must perform the final verification of the suggested partitioning scheme.

Overall, most of the techniques presented in the literature are partial approaches to automatic parallelization or they model simple loops individually. In contrast, our approach models sequential applications as a whole. In this way, KIR is able to generate a comprehensive parallelization strategy that minimizes the parallel overhead. In addition, our technique handles regular and irregular computations in a uniform manner, and addresses general-purpose languages. Finally, note that KIR is complementary to other techniques. For instance, the polyhedral model is strong in optimizing regular recurrence diKernels and may be used in conjunction with our approach.

7. Conclusions and Future Work

This paper has presented a new effective and efficient method to parallelize sequential applications automatically. It is based on the concept of domain-independent kernel to handle syntactical variations in the source code.

The first contribution is a new compiler intermediate representation called KIR. It is built on top of diKernels, which are connected with diKernel-level dependences and are grouped in execution scopes to recognize the stages of the input sequential application.

The second contribution is a new KIR-based automatic partitioning technique that builds a global OpenMP-enabled parallelization strategy targeting current multicore processors. The potential of our approach has been illustrated using a comprehensive benchmark suite that includes synthetic codes representative of frequently used diKernels, routines from dense/sparse linear algebra and image processing, and full-scale applications.

The third contribution is a comparative evaluation with the GCC, ICC and PLUTO compilers for the automatic parallelization of the benchmark suite in terms of effectiveness. In general, GCC, ICC and PLUTO fail to parallelize regular codes with complex control flows, and irregular computations. In contrast to our KIR-based approach, the evaluated compilers analyze loops in isolation and thus fail to optimize the joint parallelization of multiple loops.

Future research directions aim at increasing the performance of the generated OpenMP code, including locality exploitation techniques. Another important direction is the support of manycore architectures such as GPUs.

Acknowledgements

This research was supported by the Ministry of Economy and Competitiveness of Spain and FEDER funds of the European Union (Project ref. TIN2010-16735), and by the FPU Program of the Ministry of Education of Spain (ref. AP2008-01012).

References

[1] K. Asanovic, R. Bodík, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. A. Patterson, K. Sen, J. Wawrzynek, D. Wessel, K. A. Yelick, A View of the Parallel Computing Landscape, Commun. ACM 52 (2009) 56–67.

[2] MPI Forum, MPI: A Message-Passing Interface Standard (Version 3.0), `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`, [Last visited: Jan 2013].

[3] NVIDIA, CUDA Runtime API (Version 5.0), `http://docs.nvidia.com/cuda/cuda-runtime-api/`, [Last visited: Jan 2013].

[4] OpenMP Architecture Review Board, OpenMP Application Program Interface (Version 3.1), `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`, [Last visited: Jan 2013].

[5] OpenACC, The OpenACC Application Programming Interface (Version 1.0), `http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf`, [Last visited: Jan 2013].

[6] U. Bondhugula, A. Hartono, J. Ramanujam, P. Sadayappan, A Practical Automatic Polyhedral Parallelizer and Locality Optimizer, in: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Tucson, AZ, USA, ACM, 2008, pp. 101–113.

[7] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. A. Padua, Y. Paek, W. M. Pottenger, L. Rauchwerger, P. Tu, Parallel Programming with Polaris, IEEE Computer 29 (1996) 87–81.

[8] M. W. Hall, J.-A. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, M. S. Lam, Maximizing Multiprocessor Performance with the SUIF Compiler, IEEE Computer 29 (1996) 84–89.

[9] The Free Software Foundation, Inc., GCC, the GNU Compiler Collection, `http://gcc.gnu.org/`, [Last visited: Jan 2013].

[10] Intel Corporation, Intel C++ Composer XE for Windows, Linux, and Mac OS X, `http://software.intel.com/en-us/articles/intel-composer-xe/`, [Last visited: Jan 2013].

[11] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques and Tools, Addison-Wesley, 2006.

[12] R. Allen, K. Kennedy, Optimizing Compilers for Modern Architectures: A Dependence-based Approach, Morgan Kaufmann, 2001.

[13] M. Arenaz, J. Touriño, R. Doallo, XARK: An Extensible Framework for Automatic Recognition of Computational Kernels, ACM Trans. Program. Lang. Syst. 30 (2008) 32:1–32:56.

[14] M. Arenaz, J. Touriño, R. Doallo, Program Behavior Characterization Through Advanced Kernel Recognition, in: Proceedings of the 13th International Euro-Par Conference (Euro-Par), Rennes, France, volume 4641 of *LNCS*, Springer, 2007, pp. 237–247.

[15] M. Arenaz, J. Touriño, R. Doallo, Compiler Support for Parallel Code Generation Through Kernel Recognition, in: Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS), Santa Fe, NM, USA, IEEE Computer Society, 2004, p. 79.

[16] D. Andrade, M. Arenaz, B. B. Fraguela, J. Touriño, R. Doallo, Automated and Accurate Cache Behavior Analysis for Codes with Irregular Access Patterns, Concurr. Comput.: Pract. Exper. 19 (2007) 2407–2423.

[17] B. Franke, M. O'Boyle, Array Recovery and High-Level Transformations for DSP Applications, ACM Trans. Embed. Comput. Syst. 2 (2003) 132–162.

[18] R. Eigenmann, J. Hoeflinger, Z. Li, D. A. Padua, Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs, in: Proceedings of the 4th International Workshop on Languages and Compilers for Parallel Computing (LCPC), Santa Clara, CA, USA, volume 589 of *LNCS*, Springer, 1992, pp. 65–83.

[19] E. Albert, J. D. Lukas, G. L. Steele, Data Parallel Computers and the FORALL Statement, J. Parallel Distrib. Comput. 13 (1991) 185–192.

[20] P. Feautrier, Array Expansion, in: Proceedings of the 2nd International Conference on Supercomputing (ICS), St. Malo, France, ACM, 1988, pp. 429–441.

[21] E. Gutiérrez, O. Plata, E. L. Zapata, Data Partitioning-based Parallel Irregular Reductions, Concurr. Comput.: Pract. Exper. 16 (2004) 155–172.

[22] D. Callahan, Recognizing and Parallelizing Bounded Recurrences, in: Proceedings of the 4th International Workshop on Languages and Compilers for Parallel Computing (LCPC), Santa Clara, CA, USA, Springer, 1992, pp. 169–185.

[23] E. E. Santos, Optimal and Efficient Algorithms for Summing and Prefix Summing on Parallel Machines, J. Parallel Distrib. Comput. 62 (2002) 517–543.

[24] Y. Saad, SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations (Version 2), `http://www-users.cs.umn.edu/~saad/software/SPARSKIT/index.html`, [Last visited: Jan 2013].

[25] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, O. Temam, Semi-automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies, Int. J. Parallel Program. 34 (2006) 261–317.

[26] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, C. Bastoul, The Polyhedral Model Is More Widely Applicable Than You Think, in: Proceedings of the 19th International Conference on Compiler Construction (CC), Paphos, Cyprus, volume 6011 of *LNCS*, Springer, 2010, pp. 283–303.

[27] S. Sato, H. Iwasaki, Automatic Parallelization via Matrix Multiplication, in: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), San Jose, CA, USA, ACM, 2011, pp. 470–479.

[28] D. Liu, Y. Wang, Z. Shao, M. Guo, J. Xue, Optimally Maximizing Iteration-Level Loop Parallelism, IEEE Trans. Parallel Distrib. Syst. 23 (2012) 564–572.

[29] G. Ottoni, R. Rangan, A. Stoler, D. I. August, Automatic Thread Extraction with Decoupled Software Pipelining, in: 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Barcelona, Spain, IEEE Computer Society, 2005, pp. 105–118.

[30] J. Huang, A. Raman, T. B. Jablin, Y. Zhang, T.-H. Hung, D. I. August, Decoupled Software Pipelining Creates Parallelization Opportunities, in: Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Toronto, Ontario, Canada, ACM, 2010, pp. 121–130.

[31] A. Raman, A. Zaks, J. W. Lee, D. I. August, Parcae: a System for Flexible Parallel Execution, in: Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Beijing, China, ACM, 2012, pp. 133–144.

[32] H. Vandierendonck, S. Rul, K. De Bosschere, The Paralax Infrastructure: Automatic Parallelization with a Helping Hand, in: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT), Vienna, Austria, ACM, 2010, pp. 389–400.

[33] A. Canedo, T. Yoshizawa, H. Komatsu, Automatic Parallelization of Simulink Applications, in: Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Toronto, Ontario, Canada, ACM, 2010, pp. 151–159.

[34] G. Tournavitis, B. Franke, Semi-automatic Extraction and Exploitation of Hierarchical Pipeline Parallelism using Profiling Information, in: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT), Vienna, Austria, ACM, 2010, pp. 377–388.