A Parallelizing Compiler for Multicore Systems

José M. Andión, Manuel Arenaz, Gabriel Rodríguez and Juan Touriño
Departamento de Electrónica e Sistemas. Universidade da Coruña.
15071 A Coruña, Spain
{jandion,manuel.arenaz,grodriguez,juan}@udc.es

ABSTRACT

This manuscript summarizes the main ideas introduced in [1]. We propose a compiler that automatically transforms a sequential application into a parallel counterpart for multicore processors. It is based on an intermediate representation, named KIR, which exposes multiple levels of parallelism and hides the complexity of the implementation details thanks to the domain-independent kernels (e.g., assignment, reduction). The effectiveness and performance of our approach, built on top of GCC, has been tested with a large variety of codes.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processor—*Compilers, Optimization*

General Terms

Performance

Keywords

automatic parallelization, compiler intermediate representation, domain-independent kernel, multicore processor

1. INTRODUCTION

Traditionally, the hardware industry has made possible improving the performance of applications without changing the sequential programming model. However, this is no longer valid in the multicore era: a sequential program will only run on one of the processor cores, which will not become faster. Thus, developers have been forced to create new tools for productive parallel programming. This parallel challenge has been addressed from different sides: libraries (e.g. MPI, CUDA), compiler directives (e.g. OpenMP, OpenACC), programming languages (e.g. PGAS), and parallelizing compilers (e.g. GCC, ICC, PLUTO [4]). Automatic parallelization of applications is the ideal solution for making parallel programming easier. Nevertheless, current production compilers are not able to generate parallel code even for simple sequential programs because they rely on classical dependence analysis, which is extremely sensitive to syntactic variations and prevents the successful detection of parallelism in the presence of pointers and complex control flows.

SCOPES'14, June 10 - 11 2014, Sankt Goar, Germany

http://dx.doi.org/10.1145/2609248.2609254

This work presents a different approach for the automatic parallelization of sequential programs. It is based on the domain-independent kernels (from now on, *diKernels*), which characterize the computations carried out in a program without being affected by how they are coded. We describe the construction of a compiler intermediate representation (from now on, *KIR*) which represents different codifications of the same program in the same manner, and exposes multiple levels of parallelism. In addition, we present an automatic partitioning technique that exploits the coarse-grain parallelism exposed by the KIR targeting multicore processors.

The remainder of the manuscript is organized as follows. Section 2 describes the diKernels used in this work. Section 3 presents our automatic parallelization technique. Section 4 discusses related work. Finally, Section 5 concludes the paper and presents future research lines.

2. DOMAIN-INDEPENDENT KERNELS

The *computational kernels* have been extensively used in automatic program analysis. We work with diKernels which, instead of representing domain-specific problem solvers, describe the application features that are relevant to the compiler. The full collection of diKernels, including regular and irregular computations, can be consulted in [2]. The diKernels that appear in this work are:

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.

scalar reduction $v = v \oplus e(i)$ with $i \in \mathbb{N}$, where the reduction variable v is a scalar, $\oplus$ is an associative and commutative operator, and $e(i)$ is not dependent on v.

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

3. AUTOMATIC PARALLELIZATION DRIVEN BY DIKERNELS

Typical IRs (e.g., Abstract Syntax Trees –ASTs–, Data Dependence Graph –DDG–, Control Flow Graph –CFG–) are successful in generating optimum code for sequential programs. However, the detection of parallelism depends on the analysis of the whole application and such IRs become too complex under this situation. This section presents our approach, which builds an IR on top of diKernels named KIR and automatically partitions the KIR to exploit coarse-grain parallelism on multicores.

```
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 }
```

Figure 1: Source code of the matrix-vector multiplication.

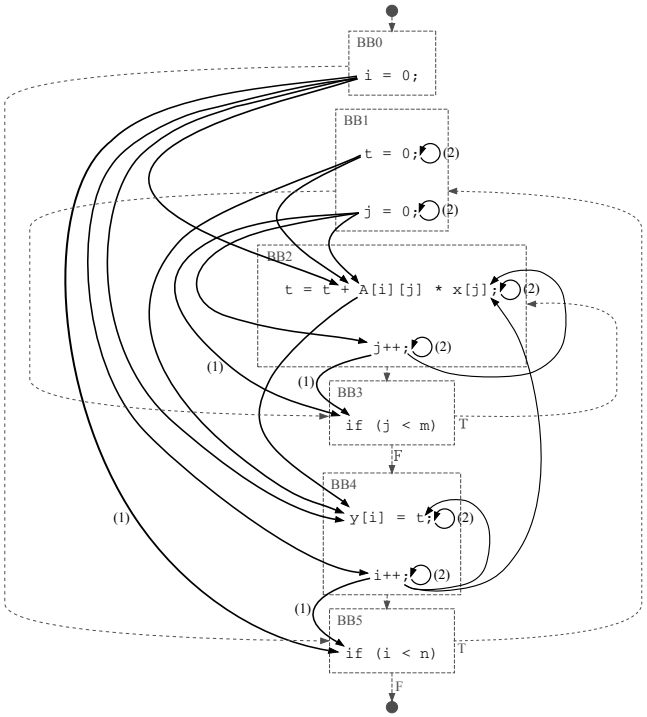

Figure 2: Standard statement-based IR of Fig. 1.

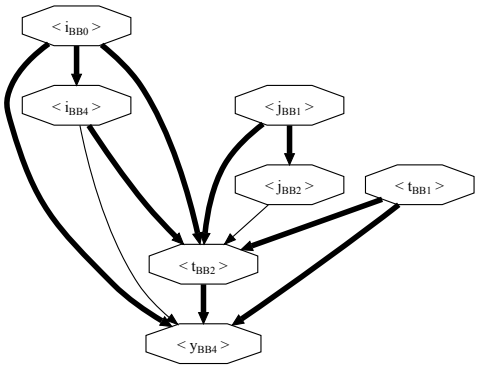

Figure 3: Steps 1 and 2 of the construction of the KIR of the matrix-vector multiplication: diKernel-level data dependences (→) and diKernel-level flow dependences (➡).

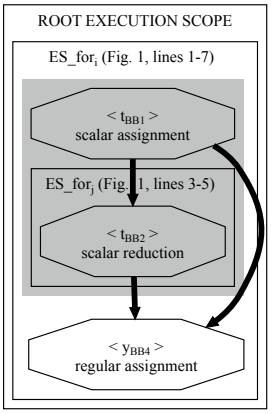

Figure 4: Step 3 of the construction of the KIR of the matrix-vector multiplication: hierarchy of execution scopes.

3.1 KIR: A diKernel-based IR

Consider the code of the matrix-vector multiplication shown in Fig. 1. Each iteration i of for_i computes the dot product between the i^{th} row of matrix A and vector x (see lines 2–5), and stores the result in the i^{th} position of vector y (line 6). An excerpt of a typical IR, where ASTs represent source code statements, is depicted in Fig. 2. ASTs are grouped into basic blocks (*BB*, dashed boxes) with precedence relationships (dashed edges) to build the CFG. Each loop (for instance, for_i) is represented by preheader (initializes the loop index, *BB*0), header (checks the loop exit condition, *BB*5), and latch (increments the loop index, *BB*4). Figure 2 also shows the data dependences between statements (solid edges) of the DDG.

The KIR is built on three steps: first, diKernels and their relationships (Def. 1–2); second, the identification of flow dependences (Def. 3–4); and third, the hierarchy of execution scopes (Def. 5–7).

Definition 1. A **diKernel** is a directed graph $K = (N, E)$ where E is the set of edges of a strongly connected component (SCC) 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–); and second, there exist edges $x_i \rightarrow x_j$ or $x_j \rightarrow x_i$ in E for some $x_j \in N$. The term $K\!<\!x_1 \ldots x_n\!>$ denotes the ASTs $x_1 \ldots x_n$ that belong to N.

Definition 2. Let SCC_x and SCC_y be two strongly connected components of the DDG associated with diKernels $K\!<\!x_1 \ldots x_n\!>$ and $K\!<\!y_1 \ldots y_m\!>$, 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 $x_i \in \{x_1 \ldots x_n\}$ and $y_j \in \{y_1 \ldots y_m\}$. The term $K_x \rightarrow K_y$ denotes that DDG edge which crosses diKernel boundaries.

The diKernel-level data dependence graph (see Fig. 3) is built from the IR of Fig. 2 as follows. Flow-of-control statements are ignored: both the branch statements of *BB*3 (`if(j<m)`) and *BB*5 (`if(i<n)`). Two diKernels capture the computation of the for_i index i (Fig. 1, line 1): the initialization in $K\!<\!i_{BB0}\!>$ (the term i_{BB0} denotes the statement `i=0` of the basic block *BB*0 in Fig. 2); and the update in $K\!<\!i_{BB4}\!>$. In the same way, $K\!<\!j_{BB1}\!>$ and $K\!<\!j_{BB2}\!>$ represent the computations over index j. The value of the dot product is stored in t: $K\!<\!t_{BB1}\!>$ initializes this temporary variable at the beginning of each for_i iteration (Fig. 1, line 2); and $K\!<\!t_{BB2}\!>$ updates its value throughout the execution of for_j (Fig. 1, line 4). Finally, $K\!<\!y_{BB4}\!>$ captures the storage of the dot product value in the output array y. Regarding DDG edges, the incoming edges of branch statements are ignored (see edges with label (1) in Fig. 2); the edges whose source and target statements belong to the same diKernel are subsumed in the diKernel (see edges with label (2)); and the edges that cross diKernel boundaries are exposed as diKernel-level data dependences in Fig. 3 (see non-labeled forward and backward edges in Fig. 2).

The second step is to determine flow dependences between diKernels to reflect the order in which they are executed.

Definition 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 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 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 of x produced/used throughout the execution of statement x_i/y_j.

The diKernel-level flow dependences have been highlighted in the graph of Fig. 3. $K<i_{BB0}> \blacktriangleright K<i_{BB4}>$ captures the flow between the initialization of i in the preheader of for_i ($BB0$) and its update in the corresponding latch ($BB4$). 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}) = $\{i\}$. Our approach handles pointers in the diKernel recognition [2], which applies array recovery techniques similar to [6]. Examples of range-based analysis of non-scalar variables (both for arrays and pointers) can be found in [1].

The third step is to build the hierarchy of execution scopes to expose the computational stages of the program to the compiler.

Definition 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 $L, L_1, \dots, L_n$, being L the outermost loop.

Definition 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 7. Let $x_1 \dots x_n$ be the ASTs of a diKernel $K<x_1 \dots x_n>$. Let $L_1, \dots, L_n$ be the innermost loops that contain $x_1, \dots, x_n$, respectively. We say that $K<x_1 \dots x_n>$ **belongs to the execution scope** R_L if and only if R_L is the execution scope of the innermost common loop for $L_1, \dots, L_n$. 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.

The hierarchy of execution scopes of the matrix-vector multiplication is depicted in Fig. 4. The two loops for_i and for_j (see Fig. 1) 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 a child of the root execution scope. $K<j_{BB1}>$ and $K<j_{BB2}>$ capture the computation of loop index j and thus belong to ES_for_j (in a similar manner, $K<i_{BB0}>$ and $K<i_{BB4}>$ belong to ES_for_i). Note that these diKernels and their incoming/outgoing diKernel-level dependences (e.g. $K<j_{BB1}> \blacktriangleright K<j_{BB2}>$) are not shown in the KIR of Fig. 4: computations on loop indices are already taken into account in the execution scope notation and diKernel types. The remaining diKernels consist of a unique assignment statement, thus they belong to the execution scope of the innermost loop that contains each statement. Hence, $K<t_{BB1}>$, $K<t_{BB2}>$ and $K<y_{BB4}>$ belong to ES_for_i, ES_for_j and ES_for_i, respectively.

3.2 Automatic partitioning driven by the KIR

Our technique consists of two steps: first, filtering out the diKernel-level dependences that do not prevent the parallelization (from now on, *spurious* diKernel-level dependences), and second, the construction of an efficient OpenMP parallelization for the whole application exploiting coarse-grain parallelism.

The privatization of program variables is helpful in the detection of spurious diKernel-level dependences. Hence, our technique *shades* connected subgraphs of the KIR that capture the computations carried out in the privatizable scalar variables of a loop L. These shaded subgraphs do not prevent program parallelization and are thus omitted in the discovering of parallelism.

Definition 8. A diKernel-level dependence is **spurious** if one of the following conditions is fulfilled:

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

Regarding the code of Fig. 1, t is a privatizable scalar variable because, before reaching uses at lines 4 and 6, it is necessary to go through the definition of line 2. Therefore, a shaded subgraph containing $K<t_{BB1}>$, $K<t_{BB2}>$, $K<t_{BB1}> \blacktriangleright K<t_{BB2}>$ and the execution scope ES_for_j is detected on the KIR of Fig. 4; and the diKernel-level dependences $K<t_{BB1}> \blacktriangleright K<y_{BB4}>$ and $K<t_{BB2}> \blacktriangleright K<y_{BB4}>$ are spurious (Def. 8, case 1).

The second step is the generation of OpenMP code. In order to reduce overhead, our technique minimizes thread creation/destruction by finding the critical path of the KIR and executing it within a unique parallel region. Our approach is based on the existence of parallelizing transformations for each type of diKernel: (1) scalar reduction diKernels are supported by the `reduction` OpenMP clause; (2) regular assignment and regular reduction diKernels are annotated with the `for` OpenMP pragma; (3) irregular assignment and irregular reduction diKernels are transformed via an array expansion technique [8]. Thus, the critical path of the KIR is the longest path of diKernel-level flow dependences connecting parallelizable diKernels.

Our technique minimizes the synchronization overhead scheduling the same workload distribution for each $K<x_i> \blacktriangleright K<y_j>$ if the following conditions hold: (1) computations of $K<x_i>$ and $K<y_j>$ can be reordered arbitrarily; and (2) given DEF(x, x_i) for $K<x_i>$ and USE(x, y_j) for $K<y_j>$, then DEF(x, x_i) = USE(x, y_j). In this way, the same thread produces the value of $K<x_i>$ that is consumed by $K<y_j>$ and no barrier is inserted.

Finally, when the parallel region is enclosed in a loop, OpenMP `parallel` directives are moved to confine that loop. The critical path is surrounded by barriers, and the remaining computations are annotated with OpenMP `single` pragmas. This optimization improves the performance of numerical simulations significantly.

Table 1: Speedups of EQUAKE from SPEC CPU2000.

	KIR			ICC		
Workload\Threads	2	4	8	2	4	8
$WL \times 1$	1.10	1.20	0.96	0.86	0.83	0.77
$WL \times 2$	1.31	1.95	1.76	0.97	0.95	0.95
$WL \times 3$	1.48	2.42	2.78	0.97	0.97	0.97

This work addresses multicore processors in both HPC and embedded systems. In general, the parallelism available in diKernels will suffice to generate a few coarse-grain threads. Our technique only requires OpenMP support in the target architecture and the developed optimizations do not need specific information about the underlying hardware, thus they have wide applicability.

In the matrix-vector multiplication of Fig. 4, the critical path of the KIR consists of the regular assignment $K{<}y_{BB4}{>}$ attached to ES_for_i. Hence, for_i is annotated with the `parallel for` OpenMP pragma and variables in the shaded subgraph are included into the `private` OpenMP clause.

Our technique has been implemented on top of GCC version 4.4.0. Its potential using a comprehensive benchmark suite that includes synthetic codes, routines from dense/sparse linear algebra and image processing, and full-scale applications from SPEC CPU2000 can be found in [1], along with a comparative evaluation with GCC, ICC and PLUTO in terms of effectiveness. In general, contenders fail to parallelize regular codes with complex control flows, and irregular computations. For instance, Table 1 presents the speedups with respect to the sequential version of the EQUAKE benchmark on a system with 2 Intel Xeon E5520 quad-core processors. As can be seen, ICC is unable to parallelize this case study properly while KIR reduces the execution time.

4. RELATED WORK

The polyhedral model [7] has reached production (GCC, IBM) and research (PLUTO) compilers. It is a mathematical framework for loop nest parallelization limited to static-control, regular loop nests. A recent extension [3] partially removes these limitations and models irregular data accesses conservatively (e.g., an array with a complex subscript is considered as a single variable).

Sato and Iwasaki [12] transform a loop body into a matrix-multiplication form based on reduce and scan parallel primitives. In addition, they extract max-operators from `if` statements, enabling the parallelization of loops with complex control flows.

Liu et al. [10] target iteration-level parallelism as a graph optimization problem: nodes are the statements of a loop, weighted edges represent dependence relationships.

Decoupled Software Pipelining (DSWP) [11] divides a loop into critical and off-critical path threads that run concurrently but communicate in a pipelined manner. Huang et al. [9] introduced DSWP+ which, instead of balancing the computational load, subsequently parallelizes the paths with other techniques (e.g. forall, localwrite).

The Paralax Infrastructure [13] uses full-data structure SSA and use/def chains to compute the SCCs on the Program Dependence Graph of a loop and extract pipeline parallelism. A lightweight programming model, which is based on annotations inserted by the programmer, helps the compiler to find thread-level parallelism.

Canedo et al. [5] present a fully automatic parallelization approach of whole Simulink applications.

Overall, most of the techniques are partial approaches, are not implemented on a compiler, or they model simple loops individually. In contrast, we model general-purpose sequential applications as a whole. In this way, KIR generates a comprehensive strategy that minimizes the parallel overhead. In addition, regular and irregular computations are jointly handled.

5. CONCLUSIONS AND FUTURE WORK

This work has presented a compiler devoted to parallelize the input sequential application automatically. It handles syntactical variations in the source code and regular and irregular computations jointly thanks to diKernels.

The first contribution is the KIR. This compiler intermediate representation consists of a set of diKernels, diKernel-level dependences which connect them, and execution scopes which represent the stages of the original program.

The second contribution is an automatic partitioning technique driven by the KIR. It exploits coarse-grain parallelism on multicore processors with a global OpenMP parallelization strategy for the whole application.

As future work, we will include locality exploitation techniques to improve the performance of the generated OpenMP code. In addition, our compiler will target fine-grain parallelism and support manycore architectures such as GPUs.

Acknowledgements

This research was supported by FEDER Funds of the European Union and the Ministry of Economy and Competitiveness of Spain (Project TIN2010-16735), by the Galician Government (Reference GRC2013-055), and by the FPU Program of the Ministry of Education of Spain (Reference AP2008-01012).

6. REFERENCES

[1] J. M. Andión et al. A Novel Compiler Support for Automatic Parallelization on Multicore Systems. *Parallel Comput.*, 39(9), 2013.

[2] M. Arenaz et al. XARK: An Extensible Framework for Automatic Recognition of Computational Kernels. *ACM Trans. Program. Lang. Syst.*, 30(6), 2008.

[3] M.-W. Benabderrahmane et al. The Polyhedral Model is more Widely Applicable than You Think. In *CC*, 2010.

[4] U. Bondhugula et al. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *PLDI*, 2008.

[5] A. Canedo et al. Automatic Parallelization of Simulink Applications. In *CGO*, 2010.

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

[7] S. Girbal et al. Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies. *Int. J. Parallel Program.*, 34(3), 2006.

[8] E. Gutiérrez et al. Data Partitioning-Based Parallel Irregular Reductions. *Concurr. Comput.: Pract. Exper.*, 16(2-3), 2004.

[9] J. Huang et al. Decoupled Software Pipelining Creates Parallelization Opportunities. In *CGO*, 2010.

[10] D. Liu et al. Optimally Maximizing Iteration-Level Loop Parallelism. *IEEE Trans. Parallel Distrib. Syst.*, 23(3), 2012.

[11] G. Ottoni et al. Automatic Thread Extraction with Decoupled Software Pipelining. In *MICRO*, 2005.

[12] S. Sato and H. Iwasaki. Automatic Parallelization via Matrix Multiplication. In *PLDI*, 2011.

[13] H. Vandierendonck et al. The Paralax Infrastructure: Automatic Parallelization with a Helping Hand. In *PACT*, 2010.