A New Intermediate Representation for GCC based on the XARK Compiler Framework [⋆]

José M. Andión, Manuel Arenaz, and Juan Touriño

Computer Architecture Group
Department of Electronics and Systems
University of A Coruña, A Coruña, Spain
{jandion,arenaz,juan}@udc.es

Abstract. The automatic generation of efficient parallel code continues to be a challenge for compiler technology. In the past, only the developers of applications targeted for supercomputers were aware of exploiting the parallelism available in the hardware. Currently, the increase in the number of cores available in commodity processors has generalized the problem. Parallelizing compilers typically address automatic detection of parallelism through the analysis of a set of standard graphs that capture information about the statements of the program. Well-known examples are the Data Dependence Graph (DDG), the Control Flow Graph (CFG) and the Dominance Tree (DT). This paper outlines a new compiler intermediate representation (IR) intended to ease detection of parallelism in sequential programs. In constrast to standard compiler IRs that are based on the concept of program statement, the new IR hinges on the concept of *computational kernel* provided by the XARK compiler framework. Thus, a kernel represents a program construct that is used frequently by programmers. Well-known examples are inductions, reductions or recurrences. In addition, this paper presents several ideas for using such kernel-level IR for the automatic generation of parallel code for multi-core processors. An application from the SPEC CPU2000 benchmark suite is used as case study.

1 Introduction

Even for experienced programmers, the development and maintenance of programs that make an efficient use of the computer architecture is a complex time-consuming task. The actual emergence and widespread use of multi-core processors adds more complexity, as current compiler technology cannot generate efficient parallel code for multi-core processors, being the programmer responsible for exploiting the parallelism available in hardware.

Current optimizing and parallelizing compilers split programs into statements that are represented as abstract syntax trees. In order to capture program behavior, the compiler builds an intermediate representation that consists of a

[⋆] This research was supported by the Ministry of Education and Science of Spain and FEDER funds of the European Union (Project TIN2007-67537-C03) and by the Galician Government (Project PGIDIT06PXIB105228PR) set of standard graphs that connect program statements [1, 2, 10]. For instance, the GCC compiler establishes relationships between GIMPLE-SSA statements by building a control flow graph (CFG), a data dependence graph (DDG) or a dominance tree (DT). Although this approach was shown to be effective in practice for compilation of real codes for mono-core processors, the level of abstraction of statement-based IRs makes it difficult for the compiler to detect the parallelism available in real applications.

The recognition of program constructs that are frequently used by software developers (*computational kernels*, from now on) is a powerful mechanism to detect parallelism automatically [4, 6–9]. XARK [5] is an extensible compiler framework that provides a complete, robust and general solution to the problem of automatic recognition of computational kernels (inductions, reductions or array recurrences), even in the presence of complex control flows.

This paper presents preliminary work on the proposal of a new kernel-based IR and sketches an algorithm to guide automatic parallelization for multi-core processors. As an illustrative case study, the full-scale application EQUAKE of the SPEC CPU2000 benchmark suite is analyzed.

The rest of the paper is organized as follows. Section 2 describes the EQUAKE application. Section 3 presents the XARK compiler framework and the collection of kernels recognized in EQUAKE. Section 4 sketches a new kernel-based IR and describes the way it captures the behavior of EQUAKE as a whole. Section 5 outlines a preliminary algorithm for the automatic parallelization of EQUAKE targeting multi-core processors. Finally, Section 6 concludes the paper and presents future work.

2 The Application EQUAKE of SPEC CPU2000

One of the applications of the SPEC CPU2000 benchmark suite is *EQUAKE*. This program performs a simulation of seismic waves in large, highly heterogeneous valleys. EQUAKE is capable of recovering the time history of the ground motion due to a specific seismic event everywhere within a valley. Computations are carried out on an unstructured mesh which locally resolves wavelengths using a finite element method. The source code of EQUAKE consists of 1512 lines grouped into 16 procedures. This relatively-small size allows us to analyze its behavior manually in a reasonable amount of time. Our analysis of EQUAKE revealed that the code contains a significant variety of the kernels that are used frequently in practice.

For the sake of clarity, Figures 1 and 2 show only an excerpt of the `main()` and `smvp()` procedures, respectively. Note that `smvp()` is the most costly routine of the program. Approximately, it consumes more than 70% of the total execution time and, therefore, it is a likely target for an optimizing compiler. The loop $\texttt{for}_i$ of `main()` procedure (see Figure 1, lines 11–26) traverses the set of finite elements in order to assemble the global simulation variables `M` and `C` using the contribution of each finite element to the solution. In each $\texttt{for}_i$ iteration, the individual contribution is computed and stored in the element matrices, for instance, matrix `Ce`.

The output of EQUAKE is a report of the displacements at both the hypocenter and epicenter of the earthquake for a predetermined number of simulation timesteps. For this purpose, the loop nest $\texttt{for}_{iter}$ (Figure 1, lines 29–59) performs a time integration. In each timestep, the displacement (3D array `disp`) is computed using the values corresponding to the two previous timesteps and involving several procedure calls at run-time (e.g., `smvp()`). The arrays computed during the simulation phase (e.g., `M` and `C`) are used as input data. Once the displacement `disp` has been computed, each $\texttt{for}_{iter}$ iteration finishes by calculating the velocity (2D array `vel`). The three variables `disptminus`, `dispt` and `disptplus` are updated in a round-robin fashion (i.e., a circular swap) in each iteration of $\texttt{for}_{iter}$ (see lines 54–58). These three variables are used to access to the 3D array `disp`, which stores the results of EQUAKE in the current and the previous two timesteps.

3 The XARK Compiler Framework

The XARK compiler framework [5] builds a hierarchical representation of a program through the recognition of a comprehensive collection of kernels. It decomposes the program into a set of mutually dependent kernels that capture the behavior of a code fragment and provide the compiler with information about the computations performed at runtime on scalar and non-scalar variables (e.g., arrays, pointers). A detailed description of the collection of kernels can be consulted in [5]. Next, we present the kernels that appear in EQUAKE:

– *Regular assignment*: Assignment of a new value to a set of elements of an array variable following a regular access pattern, typically given by an affine expression of a loop index. The computation of array `vel` in the time integration phase (Figure 1, lines 49–52), assigns a new value to each element `vel[i][j]`, being `i` and `j` affine expressions of the indices of the enclosing loops $\texttt{for}_i$ and $\texttt{for}_j$. Another example is the computation of array `Ce` in each $\texttt{for}_i$ iteration during the simulation phase (Figure 1, lines 12–14).
– *Regular reduction*: The elements of an array variable are assigned a new value that depends on the value already stored in the array element. The elements updated in a regular reduction are determined by a regular access pattern. The computation of array `Ce` in lines 16–17 of Figure 1 is a regular reduction with a sum (+) reduction operation and a linear access pattern. Another example is the computation of array `disp[disptplus][*][*]` in the loop nest $\texttt{for}_i$ of lines 39–44 of Figure 1. In this case, the new value `disp[disptplus][i][j]` depends on itself and on two different array elements `disp[dispt][i][j]` and `disp[disptminus][i][j]`. Note that in the scope of $\texttt{for}_i$, the variables `disp[disptplus][*][*]`, `disp[dispt][*][*]` and `disp[disptminus][*][*]` represent three 2D arrays that do not overlap in memory.

```c
double **M, **C, **M23, **C23, **V23, **vel, ***disp, ***K;
int i, j, k, ii, jj, kk, iter, timesteps, disptplus, dispt, disptminus;
double time, Ke[12][12], Me[12], Ce[12], alpha;

/* Initialization phase */
disptplus = 0;
dispt = 1;
disptminus = 2;

/* Simulation phase */
for (i = 0; i < ARCHelems; i++) {
   for (j = 0; j < 12; j++) {
      Me[j] = 0.0;
      Ce[j] = 0.0;
   }
   for (j = 0; j < 12; j++)
      Ce[j] = Ce[j] + alpha * Me[j];
   for (j = 0; j < 4; j++) {
      M[ARCHvertex[i][j]][0] += Me[j * 3];
      M[ARCHvertex[i][j]][1] += Me[j * 3 + 1];
      M[ARCHvertex[i][j]][2] += Me[j * 3 + 2];
      C[ARCHvertex[i][j]][0] += Ce[j * 3];
      C[ARCHvertex[i][j]][1] += Ce[j * 3 + 1];
      C[ARCHvertex[i][j]][2] += Ce[j * 3 + 2];
   }
}

/* Time integration phase */
for (iter = 1; iter <= timesteps; iter++) {
   for (i = 0; i < ARCHnodes; i++)
      for (j = 0; j < 3; j++)
         disp[disptplus][i][j] = 0.0;
   smvp(ARCHnodes, K, ARCHmatrixcol, ARCHmatrixindex, disp[dispt],
      disp[disptplus]);
   time = iter * Exc.dt;
   for (i = 0; i < ARCHnodes; i++)
      for (j = 0; j < 3; j++)
         disp[disptplus][i][j] *= - Exc.dt * Exc.dt;
   for (i = 0; i < ARCHnodes; i++)
      for (j = 0; j < 3; j++)
         disp[disptplus][i][j] += 2.0 * M[i][j] * disp[dispt][i][j] -
               (M[i][j] - Exc.dt / 2.0 * C[i][j]) * disp[disptminus][i][j] -
               Exc.dt * Exc.dt * (M23[i][j] * phi2(time) / 2.0 +
               C23[i][j] * phi1(time) / 2.0 + V23[i][j] * phi0(time) / 2.0);
   for (i = 0; i < ARCHnodes; i++)
      for (j = 0; j < 3; j++)
         disp[disptplus][i][j] = disp[disptplus][i][j] /
               / (M[i][j] + Exc.dt / 2.0 * C[i][j]);
   for (i = 0; i < ARCHnodes; i++)
      for (j = 0; j < 3; j++)
         vel[i][j] = 0.5 / Exc.dt * (disp[disptplus][i][j] -
                  disp[disptminus][i][j]);

   /* Circular swap */
   i = disptminus;
   disptminus = dispt;
   dispt = disptplus;
   disptplus = i;
}
```

Fig. 1. Excerpt of the source code of the EQUAKE application.

```
void smvp(int nodes, double ***A, int *Acol, int *Aindex, double **v,
          double **w) {
   int i;
   int Anext, Alast, col;
   double sum0, sum1, sum2;

   for (i = 0; i < nodes; i++) {
      Anext = Aindex[i];
      Alast = Aindex[i + 1];
      sum0 = A[Anext][0][0]*v[i][0] + A[Anext][0][1]*v[i][1] +
           + A[Anext][0][2]*v[i][2];
      sum1 = A[Anext][1][0]*v[i][0] + A[Anext][1][1]*v[i][1] +
           + A[Anext][1][2]*v[i][2];
      sum2 = A[Anext][2][0]*v[i][0] + A[Anext][2][1]*v[i][1] +
           + A[Anext][2][2]*v[i][2];
      Anext++;
      while (Anext < Alast) {
         col = Acol[Anext];
         sum0 += A[Anext][0][0]*v[col][0] + A[Anext][0][1]*v[col][1] +
               + A[Anext][0][2]*v[col][2];
         sum1 += A[Anext][1][0]*v[col][0] + A[Anext][1][1]*v[col][1] +
               + A[Anext][1][2]*v[col][2];
         sum2 += A[Anext][2][0]*v[col][0] + A[Anext][2][1]*v[col][1] +
               + A[Anext][2][2]*v[col][2];
         w[col][0] += A[Anext][0][0]*v[i][0] + A[Anext][1][0]*v[i][1] +
                    + A[Anext][2][0]*v[i][2];
         w[col][1] += A[Anext][0][1]*v[i][0] + A[Anext][1][1]*v[i][1] +
                    + A[Anext][2][1]*v[i][2];
         w[col][2] += A[Anext][0][2]*v[i][0] + A[Anext][1][2]*v[i][1] +
                    + A[Anext][2][2]*v[i][2];
         Anext++;
      }
      w[i][0] += sum0;
      w[i][1] += sum1;
      w[i][2] += sum2;
   }
}
```

Fig. 2. Source code of the `smvp()` procedure of the EQUAKE application.

– *Irregular reduction*: A reduction where the updated array elements are determined by an indirection array. An example is the computation of `M` in the simulation phase (Figure 1, lines 18–21), which uses an indirection array `ARCHvertex` to determine the element of `M` updated in each iteration of $\texttt{for}_i$ and $\texttt{for}_j$.

The automatic recognition of computational kernels in full-scale real application requires the development of an inter-procedural kernel recognition engine. For this purpose, work in progress [3] focuses on the development of an efficient inter-procedural Gated Single Assignment (GSA) form on top of the GIMPLE-SSA infrastructure provided by GCC. GSA is an extension of the well-known Static Single Assignment (SSA) form where reaching definition information of scalar and array variables is represented syntactically. The construction of GSA involves two main tasks: first, placement of special operators (called ϕ generically) at the points of the program with multiple predecessors in the control flow graph; and second, renaming of program variables so that the left-hand sides of the assignment statements define distinct unique variables. Different kinds of ϕ operators are distinguished according to the point of the program where they are inserted:

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

As shown in Figure 2, `smvp()` carries out an irregular reduction and stores the result in the reduction array `w`, being `Acol` the indirection array and assuming that formal parameters `A`, `v` and `w` point to non-overlapping memory locations. Note that the inter-procedural GSA form enables the recognition of kernels through procedure calls. At the call site of lines 33–34 of Figure 1, the actual parameters of `smvp()` are known to point to disjoint memory locations. Thus, the recognition engine infers that `smvp()` computes an irregular reduction on array `disp[disptplus]` within the main program.

4 A New Kernel-Based Intermediate Representation

In this section we propose a new kernel-based IR intended to expose coarse-grain and fine-grain parallelism to the compiler. Standard IRs typically consist of statement-based DDG, CFG and DT. In a similar manner, our new kernel-based IR consists of a Kernel-based DDG (K-DDG) and a Kernel-based CFG (K-CFG).

As mentioned in Section 3, XARK builds a hierarchical representation that decomposes a program into a set of mutually dependent kernels. We call this representation the K-DDG, whose nodes represent kernels and whose edges represent dependences between kernels. For illustrative purposes consider the graph of EQUAKE presented in Figure 3. The nodes are depicted as ovals labeled with the program variable that stores the results of the computation of the kernel, as well as the type of kernel. On the one hand, the nodes `Me` and `M` of the simulation phase capture two kernels: regular assignment and irregular reduction, respectively. Note that kernels abstract the computations of a set of statements spread over the program. For instance, kernel `M` contains three statements of the body of loop $\texttt{for}_j$ (see Figure 1, lines 18–21). On the other hand, the edges capture the kernel-level dependences that connect statements of different kernels (see edge between kernels `Me` and `M` in Figure 3). Note that the remaining dependences between statements of a kernel are not exposed to the compiler in the K-DDG as they are represented in the type of kernel recognized by XARK.

The second graph of our new IR is the Kernel-based Control Flow Graph (K-CFG). We propose a two-phase construction algorithm that first groups the

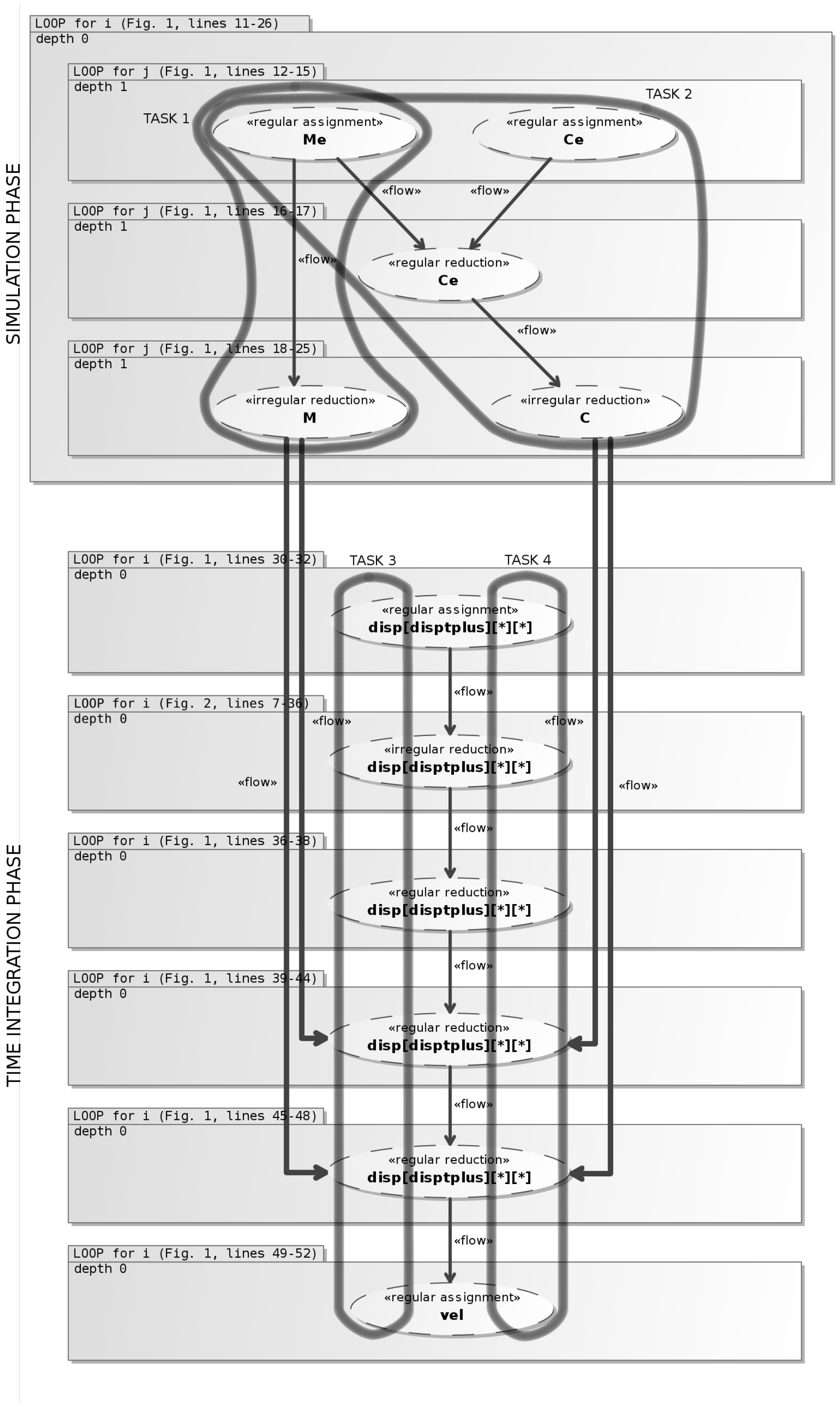

Fig. 3. Kernel-based IR of the source code of the EQUAKE application.

kernels (i.e., the nodes of the K-DDG) provided by XARK into *execution scopes* and later searches for *flow dependences* between the kernels associated with each execution scope. In the first phase, the execution scope of a kernel is computed using the concept of *region* [1]. A region of a flow graph is a collection of nodes N and edges E such that: (1) There is a header h in N that dominates all the nodes in N; (2) If some node m can reach a node n in N without going through h, then m is also in N; and (3) E is the set of all the control flow edges between nodes n_1 and n_2 in N, except (possibly) for some that enter h. Note that loop nests are often the most costly part of a program, thus being the most appropriate candidates for a compiler to extract parallelism. In order to build the K-CFG, the program is split into a hierarchy of loop regions that represent the execution scopes and kernels are attached to execution scopes by executing Algorithm 1. The inputs are the K-DDG provided by XARK and the CFG and the DT provided by the GCC compiler. The algorithm determines the set of basic blocks that contain each statement of a kernel K, excluding μ-statements associated to loop headers. Next, the basic block *bb_dom* that dominates the remaining basic blocks in the set is computed. The γ-statements of the GSA form assure that such basic block *bb_dom* exists. Finally, the execution scope of the kernel K is the innermost loop region that contains *bb_dom*.

In the second phase, the kernel-level dependences of the K-DDG are classified into *flow* or *not flow* dependences using Algorithm 2. For a kernel-level dependence $K_1 \rightarrow K_2$ to be a flow dependence, a dominate relationship between K_1 and K_2 must exist. Two cases are distinguished. In the first case, the execution scopes R_1 and R_2 of K_1 and K_2 are checked. If R_1 and R_2 have the same parent region and R_1 precedes R_2 in the hierarchy (see Algorithm 2, lines 4–5), then all the statements of K_1 are executed before any statement of K_2. Thus, $K_1 \rightarrow K_2$ is a flow dependence that assures that the results of K_1 have been computed before beginning the execution of K_2.

In the second case, we assume that K_1 and K_2 are attached to the same execution scope ($R_1 = R_2$), or that R_1 and R_2 do not have the same parent region. In order to establish a flow dependence between K_1 and K_2, we need to assure that all the statements of K_1 dominate all the statements of K_2 (see Algorithm 2, lines 6–10). An example of this situation is shown below:

```
if (c == 0) {
    a = 5;
    b = a + 3;
} else {
    a = 2;
    b = a + 1;
}
```

In this case, in each path of this if-then statement there is a pair-wise dominance relation between the sentence that writes `a` and the sentence that writes `b`. Thus, Algorithm 2 concludes that the kernel associated to `a` dominates the kernel associated to `b`.

The resulting kernel-based IR for the EQUAKE program is shown in Figure 3. In the K-CFG the nodes are grouped into execution scopes that represent the loops of the code. For the sake of clarity, the execution scopes of the inner loops

Algorithm 1 Computation of the execution scopes.

```
Input: K-DDG, CFG, DT
 1: foreach kernel $K$ in the K-DDG do
 2:     $bb\_dom$ = basic block of CFG that contains a stmt of $K$ (excluding $\mu$-stmt)
 3:     foreach statement $stmt$ in $K$ do
 4:         if $stmt$ is not a $\mu$-statement then
 5:             $bb\_stmt$ = basic block of CFG that contains $stmt$
 6:             if $bb\_stmt$ dominates $bb\_dom$ then
 7:                 $bb\_dom = bb\_stmt$
 8:             end if
 9:         end if
10:     end for
11:     $K$.execution_scope = innermost enclosing loop region of $bb\_dom$;
12: end for
```

Algorithm 2 Detection of kernel-level flow dependences.

```
Input: K-DDG, K-CFG, CFG, DT
 1: foreach kernel-level dependence $K_1 \rightarrow K_2$ of the K-DDG do
 2:     $R_1$ = execution_scope($K_1$)
 3:     $R_2$ = execution_scope($K_2$)
 4:     if ($R_1$.parent_reg = $R_2$.parent_reg) & ($R_1$ precedes $R_2$ in the hierarchy) then
 5:         mark $K_1 \rightarrow K_2$ as flow dependence
 6:     else if $\forall s_1 \in K_1 \; \exists s_2 \in K_2$ such that statements $s_1$ and $s_2$
 7:             belong to the same basic block in the CFG
 8:             and $s_1$ precedes $s_2$ in the DT then
 9:         mark $K_1 \rightarrow K_2$ as flow dependence
10:     end if
11: end for
```

of the time integration phase are not depicted in Fig. 3 (e.g., execution scope of inner loop `for`$_j$ — lines 40–44 — in loop nest `for`$_i$ — lines 39–44 —). As will be shown in the following section, this high-level representation abstracts the implementation details of the computational kernels, enabling the compiler to automatically detect the parallelism available in EQUAKE and to generate parallel code for a multi-core processor.

5 Automatic Detection of Parallelism for Multi-Core Processors

In general, the automatic parallelization of sequential codes involves two main problems: first, detection and decomposition of a program into a set of tasks that can be run in parallel; and, second, generation of parallel code for the underlying hardware architecture. The kernel-based IR presented in Section 4 exposes multiple levels of parallelism that range from parallelizable individual kernels (*intra-kernel parallelism*) up to a kernel-level dependence graph bounded to execution scopes (*inter-kernel parallelism*). Next, we outline a simple algorithm to

Algorithm 3 Task decomposition for multi-core processors.

```
Input: K-DDG, K-CFG
 1: merge execution scopes with one kernel and one cross-boundary edge
 2: $d = 0$
 3: foreach execution scope $R$ at depth $d$ in the K-CFG do
 4:     if $\forall$ kernel $K \in R$ such that $K$ is parallelizable then
 5:         $n\_drain\_kernels$ = number of kernels without outgoing edges in K-DDG
 6:                               that cross the execution scope boundaries
 7:         if $n\_drain\_kernels = P$ then
 8:             $tasks$ = set of $P$ drain kernels
 9:         else if $n\_drain\_kernels < P$ then
10:             $tasks$ = split parallelizable drain kernels to create $P$ tasks
11:         else
12:             $tasks$ = merge drain kernels to create $P$ tasks
13:         end if
14:         map $tasks$ to different cores
15:     end if
16:     $d$++
17: end for
```

decompose a program into tasks using current (dual-, quad-, eight-core) multi-core processors.

Algorithm 3 presents a pseudocode of the task decomposition strategy. The analysis focuses on those coarse-grain execution scopes where all the kernels are parallelizable, i.e, there exists a parallelizing code transformation targeted to each kernel [4, 6–9]. The types of kernels found in the K-CFG of EQUAKE are regular assignment, regular reduction and irregular reduction. Thus, all of these execution scopes are analyzed by the task decomposition algorithm.

The simulation phase contains two kernels that lead to the creation of two tasks: the first task corresponds to the execution of the kernels `Me` and `M` (see subgraph TASK1 in Figure 3); the second task involves kernels `Me`, `Ce` and `C` (see subgraph TASK2). If the target multi-core processor contains more than two cores, then intra-kernel parallelism is exploited by applying parallelizing transformations to the irregular reductions `M` and/or `C` as needed (see lines 9–10 in Algorithm 3). Note that, in general, this strategy replicates computations on different cores (see kernel `Me` belonging to TASK1 and TASK2 in Figure 3).

The time integration phase cannot be executed before the simulation phase finishes in order to prevent the violation of cross-boundary kernel-level dependences (appropriate synchronization is needed). The time integration phase is represented by six execution scopes, each containing one kernel. All of these six kernels are connected by one kernel-level flow dependence. As a result, these execution scopes are merged in order to expose a sequence of kernels to the compiler (see line 1 in Algorithm 3). Next, the kernel `vel` that lacks outgoing edges is transformed into parallel code in order to create as many tasks as needed. For illustrative purposes, assume that two tasks TASK3 and TASK4 are created (see Figure 3). Each task computes a subarray of `vel`. Thus, in order to minimize communication and synchronization, they must be assigned the computation of the corresponding subarrays of `disp`. As a result, the tasks work in parallel with memory locations that do not overlap. Finally, note that kernels with irregular access patterns need to be transformed using an inspector-executor approach to avoid communication and synchronization between the cores.

Overall, the strategy outlined in this section enables the detection of parallelism within full-scale applications. The kernel-based IR (K-DDG and K-CFG) naturally reflects the structure of the source code and, thus, avoids the violation of the data dependences specified by the programmer.

6 Conclusions and Future Work

This paper is a first step towards the definition of a kernel-based IR that exposes multiple levels of parallelism to the compiler. The new kernel-based IR is inspired by standard statement-based IRs used in current optimizing compilers. Thus, the K-DDG and the K-CFG are intended to provide a powerful framework for the development of new full-scale automatic parallelization techniques. The EQUAKE program of SPEC CPU2000 was used as case study to show the potential of this approach.

As future work we intend to improve the K-CFG construction algorithm and to run tests with well-known benchmark suites (e.g., SPEC, PERFECT). In addition, we will address the development of an algorithm for task decomposition that targets multi-core and many-core processors as well as GPU processors.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Addison-Wesley (2006)
2. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kaufmann (2001)
3. Arenaz, M., Amoedo, P., Touriño, J.: Efficiently Building the Gated Single Assignment Form in Codes with Pointers in Modern Optimizing Compilers. In: 14th International Euro-Par Conference (Euro-Par), Las Palmas de Gran Canaria, Spain. LNCS, vol. 5168, pp. 360–369. Springer (2008)
4. Arenaz, M., Touriño, J., Doallo, R.: Compiler Support for Parallel Code Generation through Kernel Recognition. In: 18th International Parallel and Distributed Processing Symposium (IPDPS), Santa Fe, NM, USA. IEEE Computer Society (2004)
5. Arenaz, M., Touriño, J., Doallo, R.: XARK: An eXtensible framework for Automatic Recognition of computational Kernels. ACM Trans. Program. Lang. Syst. 30(6) (2008)
6. Callahan, D.: Recognizing and Parallelizing Bounded Recurrences. In: 4th International Workshop on Languages and Compilers for Parallel Computing (LCPC), Santa Clara, CA, USA. LNCS, vol. 589, pp. 169–185. Springer (1991)
7. Lin, Y., Padua, D.A.: On the Automatic Parallelization of Sparse and Irregular Fortran Programs. In: 4th International Workshop on Languages, Compilers, and

Run-Time Systems for Scalable Computers (LCR), Pittsburgh, PA, USA. LNCS, vol. 1511, pp. 41–56. Springer (1998)

8. Pinter, S.S., Pinter, R.Y.: Program Optimization and Parallelization Using Idioms. ACM Trans. Program. Lang. Syst. 16(3), 305–327 (1994)
9. Setoain, J., Tenllado, C., Gómez, J.I., Arenaz, M., Prieto, M., Touriño, J.: Towards Automatic Code Generation for GPU Architectures. In: 9th International Workshop on State-of-the-Art in Scientific Computing on GPUs (PARA), Trondheim, Norway (2008)
10. Wolfe, M.: High performance compilers for parallel computing. Addison-Wesley (1996)