Automatic Partitioning of Sequential Applications Driven by Domain-Independent Kernels *

José M. Andión, Manuel Arenaz, and Juan Touriño

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

Abstract. The automatic parallelization of sequential applications is a great challenge for current compiler technology. The partitioning of a sequential application into parallel programs that can be executed concurrently on a given parallel architecture is a complex and time-consuming undertaking. In addition, the programmer is often responsible for defining a good partitioning that takes into account the properties of both the program and the architecture. This paper proposes a new fully automated partitioning algorithm driven by an intermediate representation of the sequential application in terms of the domain-independent concept-level kernels (e.g., induction, reduction, recurrence) recognized by the XARK compiler framework. Such kernel-centric view of the application hides the complexity of the implementation details (e.g., procedure calls, pointers, global variables, complex control flows) and provides robustness against different codification styles. For illustrative purposes, we use inter-procedural implementations of the Sobel edge filter and the EQUAKE application of SPEC CPU2000.

1 Introduction

The automatic partitioning of sequential applications into parallel programs to be executed concurrently on modern parallel architectures remains a great challenge for state-of-the-art parallelizing compilers. In general, this problem requires the intervention of the developer by using domain-specific programming languages that explicitly define a partition of the application. Unfortunately, this partitioning process is complex and time-consuming as it requires in-depth knowledge about both the application and the target parallel architecture, an skill that is unfamiliar to most programmers. In recent years, the emergence and widespread use of multicore and manycore architectures has exposed this situation beyond the high performance computing community.

* This research was supported by the Ministry of Science and Innovation of Spain and FEDER funds of the European Union (Project TIN2007-67537-C03) and by the FPU Program of the Ministry of Education of Spain (Reference AP2008-01012).

The main contribution of this paper is two-fold. First, a new intermediate representation (IR) of a sequential application in terms of the domain-independent concept-level kernels (*kernel-based IR*, from now on) recognized by the XARK compiler framework [5] is formally defined. XARK was shown to be effective to characterize a significant amount of the regular and irregular computations carried out in full-scale Fortran77 applications [4] from SPEC CPU2000, Perfect benchmarks, Sparskit-II and PLTMG. In order to widen the scope of application of XARK to other programming languages that use pointers, we have developed a simple and fast algorithm to build the Gated Single Assignment (GSA) form on top of the Static Single Assignment (SSA) form available in modern production compilers [2]. In addition, the design of an interprocedural GSA form to support automatic kernel recognition across procedure boundaries is work in progress.

The second contribution is a new automatic partitioning algorithm driven by the kernel-based IR. The algorithm takes advantage of the multiple levels of parallelism exposed in the kernel-based IR as well as of its robustness to different codifications of the same kernel. In the literature, the automatic detection of parallelism driven by domain-independent kernels [3, 7, 14, 16, 18] was shown to be an effective approach for non-interprocedural codes only. In contrast, we will use interprocedural C implementations of the Sobel edge filter and the EQUAKE application from SPEC CPU2000 to illustrate the behavior of our automatic partitioning approach.

The rest of the paper is organized as follows. Section 2 presents a formal definition of the new kernel-based IR. Section 3 sketches our algorithm for automatic partitioning of sequential applications for multicore systems. Sections 4 and 5 describe the case studies. Section 6 discusses related work. And, finally, Section 7 concludes the paper and presents future work.

2 Domain-Independent Kernel-Based IR

Parallelizing compilers typically use statement-based standard IRs (e.g., Data Dependence Graph –DDG–, Control Flow Graph –CFG–, Dominator Tree –DT–) that hinder discovering the parallelism available in sequential programs. In this section we propose a new kernel-based IR that hides the complexity of the implementation details, and exposes multiple levels of parallelism to the compiler. Inspired by standard IRs, our new kernel-based IR (first outlined in [1]) consists of a Kernel-based DDG (KDDG) and a Kernel-based CFG (KCFG).

2.1 The Kernel-based Data Dependence Graph (KDDG)

XARK [5] builds a hierarchical representation that decomposes a sequential application into a set of mutually dependent kernels that capture the behavior of the computations carried out on scalar and non-scalar variables (e.g., arrays, pointers). Such information is used to construct the KDDG as follows.

The KDDG is a graph $< N, E >$ having a set of nodes N representing the domain-independent kernels recognized by XARK, and a set of edges E

representing the data-dependence constraints among the kernels. The nodes and edges of the KDDG are constructed as follows:

- Each node represents a kernel $K(x_1 \dots x_n)$, which has a set of statements in GSA form, $s_1 ... s_n$, that define the output variables of the kernel $x_1 \dots x_n$. Each node is also labeled with the type of domain-independent kernel recognized by XARK (see example types later in Sections 4 and 5; all the details can be found in [5]). In addition, we define the *header* of $K(x_1 \dots x_n)$ as the statement s_h (with $h \in \{1 \dots n\}$) that dominates all of the remaining statements of the kernel. In a similar manner, we define the *latch* of $K(x_1 \dots x_n)$ as the statement s_l (with $l \in \{1 \dots n\}$) that posdominates all of the remaining statements of the kernel.
- Each edge $K(x_1 \dots x_n) \rightarrow K(y_1 \dots y_m)$ represents a kernel-level dependence that exposes a data dependence of the DDG that links statements of different kernels. Note that the data-dependences in the DDG between statements of the same kernel are not exposed to the compiler in the KDDG as they are represented in the type of kernel recognized by XARK.

2.2 The Kernel-Based Control Flow Graph (KCFG)

In order to reduce the complexity of program analyses, statement-based standard IRs group statements into basic blocks. Thus, we propose a two-phase KCFG construction algorithm that groups the kernels of the KDDG into execution scopes and that identifies kernel-level flow dependences among the edges of the KDDG.

Computation of execution scopes (Algorithm 1, lines 5–23). Parallelizing compilers typically focus on the loops of an application as they often consume most of the execution time and, thus, optimizations that improve the performance of loops can have a significant impact. The goal of the procedure COMPUTE_EXECUTION_SCOPES() is to attach each kernel to the innermost loop that contains its source code statements. Thus, the first step is to compute the hierarchy of loops of the application. Next, the algorithm computes the set of basic blocks that contain the statements in GSA form of the kernel $K(x_1 \dots x_n)$, excluding μ-statements associated to loop headers. Within this set, the algorithm uses the DT to select the basic block *bb_dom* that dominates the remaining basic blocks of the set. As a result, $K(x_1 \dots x_n)$ is attached to the innermost loop that contains *bb_dom* and that contains in its body all of the loop headers whose indices address the output variable of the kernel. Finally, the loops that are not attached any kernel are removed.

Detection of kernel-level flow dependences (Algorithm 1, lines 24–50). In order to obtain a good partitioning of a sequential application, producer-consumer relationships must be established at the kernel level. Let $K_1(x_1 \dots x_n)$ and $K_2(y_1 \dots y_m)$ be two kernels with sets of statements $s_1^{K_1} \dots s_n^{K_1}$ and $s_1^{K_2} \dots s_m^{K_2}$,

Algorithm 1 Construction of the KCFG.

Input: KDDG, CFG, DT

```
procedure BUILD_KCFG
    compute_execution_scopes()
    detect_flow_dependences()
end procedure

procedure COMPUTE_EXECUTION_SCOPES
    compute hierarchy of loops
    foreach kernel $K$ in the KDDG do
        $bb\_dom$ = basic block of CFG that contains a stmt of $K$ (excluding $\mu$-stmt)
        foreach statement $s^K$ in $K$ do
            if $s^K$ is not a $\mu$-statement then
                $bb\_s^K$ = basic block of CFG that contains $s^K$
                if $bb\_s^K$ dominates $bb\_dom$ then
                    $bb\_dom = bb\_s^K$
                end if
            end if
        end for
        $L$ = innermost enclosing loop of $bb\_dom$
        if $L$ includes loop indices that address the output variable of $K$ then
            $K$.execution_scope = $L$
        end if
    end for
    remove non-attached loops from hierarchy
end procedure

procedure DETECT_FLOW_DEPENDENCES
    foreach kernel-level dependence $K_1 \rightarrow K_2$ of the KDDG do
        $L_1$ = execution_scope($K_1$); $L_2$ = execution_scope($K_2$)
        if ($L_1$.parent == $L_2$.parent) && ($L_1$ precedes $L_2$ in the hierarchy) then
            mark $K_1 \rightarrow K_2$ as flow dependence
        else if $K_1$.latch dominates $K_2$.header then
            mark $K_1 \rightarrow K_2$ as flow dependence
        else
            mark = true
            foreach $s^{K_2} \in K_2$ (excluding $\mu$-stmt) do
                dom_stmt_found = false
                foreach $s^{K_1} \in K_1$ (excluding $\mu$-stmt) do
                    $BB_1$ = basic_block($s^{K_1}$); $BB_2$ = basic_block($s^{K_2}$)
                    if ($BB_1 == BB_2$) && ($s^{K_1}$ precedes $s^{K_2}$) then
                        dom_stmt_found = true; break
                    else if ($BB_1 \neq BB_2$) && ($BB_1$ dominates $BB_2$) then
                        dom_stmt_found = true; break
                    end if
                end for
                mark = mark && dom_stmt_found
            end for
            if mark == true then
                mark $K_1 \rightarrow K_2$ as flow dependence
            end if
        end if
    end for
end procedure
```

respectively. We say that there is a *kernel-level flow dependence*, $K_1 \Rightarrow K_2$, if a *kernel-level dominance relationship* exists. We say that K_1 *dominates* K_2 if and only if $\forall s^{K_2} \in K_2, \exists s^{K_1} \in K_1$ such that one of the following conditions hold:

1. If s^{K_1} and s^{K_2} are located in the same basic block in the CFG, then s^{K_1} precedes s^{K_2}.
2. If s^{K_1} and s^{K_2} belong to different basic blocks BB_1 and BB_2, respectively, then BB_1 dominates BB_2 in the DT.

Note that the computation of the kernel-level dominance relationship could be very expensive in full-scale real applications as they usually consist of a large set of kernels, each kernel being composed of a large set of statements. Thus, we propose two more efficient approaches to establish this relationship between two kernels K_1 and K_2, assuming that they are attached to the execution scopes of loop L_1 and L_2, respectively.

1. If L_1 and L_2 are located at the same depth in the hierarchy of loops and L_1 precedes L_2 in the hierarchy, then a flow dependence $K_1 \Rightarrow K_2$ exists.
2. If L_1 and L_2 are the same execution scope or they are located at different depths in the hierarchy of loops, then we take advantage of the header/latch information of K_1 and K_2 (see KDDG construction in Section 2.1). Let $K_1.header$ and $K_1.latch$ be the header and the latch statements of K_1. Analogously, let $K_2.header$ and $K_2.latch$ be the header and the latch of K_2. If $K_1.latch$ dominates $K_2.header$, then K_1 dominates K_2 and, as a result, a flow dependence $K_1 \Rightarrow K_2$ exists.

Algorithm 1 shows procedure DETECT_FLOW_DEPENDENCES() to identify the kernel-level flow dependences as described in this section. As will be shown in the rest of the paper, this kernel-based IR (KDDG and KCFG) abstracts the implementation details enabling the compiler to partition the sequential application automatically.

3 Automatic Partitioning Algorithm

The automatic partition of a sequential application into a set of concurrent programs requires in-depth knowledge about the code, but also about the target parallel architecture. On the one hand, the kernel-based IR presented in Section 2 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*). On the other hand, modern hardware architectures also expose multiple levels of parallelism that can be described as a graph. Thus, a multicore system may consist of a cluster of nodes with a Gigabit Ethernet or Infiniband interconnection network. Each node may have several multicore processors that commonly consists of 2-8 cores. Modern cores are designed to exploit instruction level parallelism as well as SIMD-like vector instructions such as Intel SSE or AMD 3DNow!. Such machine description of

Algorithm 2 Automatic partitioning.

Input: KCFG, ARCH

```
procedure SEQUENTIAL_PROGRAM_PARTITIONING
    initialization()
    search_best_partition([KCFG, 1, 0], ARCH)
end procedure

procedure INITIALIZATION(KCFG)
    mark kernels with low computational load as non-splittable
    merge consecutive execution scopes with one flow dependence
end procedure

function SEARCH_BEST_PARTITION([KG, A_{span_depth}, A_{depth}], ARCH)
    depth_{KG} = depth of the kernel-based subgraph KG
    depth_{ARCH} = depth of ARCH starting in level A_{depth}
    /* base case */
    if A_{depth} > depth_{ARCH} then
        return
    end if
    /* recursive case */
    KG_children = {}
    if depth_{KG} == depth_{ARCH} then
        map_kernels2arch([root nodes of KG, A_{span_depth}, A_{depth}], ARCH)
    else
        /* discard intermediate levels in ARCH */
        KG_children += [KG, 1, A_{depth} + 1]
        /* span KG across multiple levels of ARCH */
        for span = 1, depth_{ARCH} − A_{depth} − 1 do
            KG_children += [root nodes of KG, span, A_{depth}]
        end for
    end if
    KG_children += [subgraphs of KG with splittable root nodes, 1, A_{depth} + 1]
    foreach [KG_child, A_s, A_d] in KG_children do
        search_best_partition([KG_child, A_s, A_d], ARCH)
    end for
    KG_best_child = KG_child with minimun cost estimation
    return KG_best_child
end function

procedure MAP_KERNELS2ARCH([KG_roots, A_{span_depth}, A_{depth}], ARCH)
    #P = number of cores in A_{span_depth} levels from A_{depth} of ARCH
    #K = number of nodes in KG_roots
    if #K == #P then
        create a task for each root node in KG_roots
    else if #K < #P then
        split root nodes in KG_roots to create P tasks
    else
        merge root nodes in KG_roots to create P tasks
    end if
end procedure
```

the parallel architecture may be specified by the user or obtained automatically (e.g., Servet [12]).

Algorithm 2 presents the pseudocode of an automatic partitioning strategy driven by our kernel-based IR (KDDG and KCFG) that targets a multicore system with multiple levels of parallelism (ARCH). The algorithm starts with a call to procedure INITIALIZATION() that marks the kernels of the KCFG with low computational load as non-splittable (see line 6). Non-splittable kernels are good candidates to exploit SIMD-like vector instructions if the operations are supported by the processor. Otherwise, they will be executed sequentally. As for now, we assume that the computational load of a kernel is supplied by the programmer. In the future, we will incorporate a cost estimation model that will take into account the number of sentences of a kernel, the number of loop iterations, the cost of each operator, etc. Finally, the initialization stage attempts to reduce the complexity of the KCFG by clustering. Thus, consecutive execution scopes connected with one kernel-level flow dependence are merged into a unique execution scope (line 7).

The core of the automatic partitioning strategy is the recursive function SEARCH_BEST_PARTITIONING(). This function makes a top-down traversal of the KCFG looking for sets of splittable kernels to be mapped to each level of the hierarchy ARCH of the multicore system. For this purpose, we define the *depth of a subgraph of the KCFG* as the maximum number of splittable kernels in all the paths of the subgraph. Analogously, the *depth of a subgraph of ARCH* is the maximum number of levels of processing elements in all the paths of the subgraph. In addition, we define the *span depth* of a subgraph of the KCFG (see parameter A_{span_depth}) as the number of levels in ARCH that are devoted to map the root kernels of the subgraph. The basic idea is to compute the cost of every splittable kernel mapped to each combination of one or several levels of ARCH. The best partitioning will be that of minimal cost. Note that the algorithm considers mapping one kernel to several levels of ARCH (i.e., $A_{span\_depth} > 1$), as well as forcing splittable kernels to be executed sequentally if there are more levels of parallelism in the KCFG than in ARCH.

The first invocation of SEARCH_BEST_PARTITIONING() starts with KG being the whole KCFG, $A_{span\_depth} = 1$ and the highest coarse-grain level of ARCH (supposed to be $A_{depth} = 0$). In the recursive case, two possibilities are distinguished. First, if the number of non-mapped levels in KG is the same as in ARCH (i.e., if the condition $depth_{KG} == depth_{ARCH}$ in line 18 is fulfilled), then the root kernels of KG are mapped to the level A_{depth} of ARCH by executing MAP_KERNELS2ARCH(). Next, the algorithm builds the set KG_children of subgraphs of KG whose root nodes are splittable kernels. For each subgraph in KG_children, the best partitioning with $A_{span\_depth} = 1$ is computed through recursive calls to SEARCH_BEST_PARTITIONING() (lines 28–31). Finally, the best partitioning in KG is selected upon that of the child KG_best_child with minimum cost (line 32).

The recursive case of SEARCH_BEST_PARTITIONING() distinguishes a second possibility. The idea is to evaluate the cost of both discarding the assignment of computational load to a given architecture level (lines 21–22), and spanning the computational load into several levels of the computer architecture (lines 23–26). These possibilities will be evaluated by adding to KG_children the corresponding subgraphs of KG with A_{span_depth} from 1 up to $depth_{ARCH} - A_{depth} - 1$.

The goal of procedure MAP_KERNELS2ARCH() (lines 35–45) is to analyze the set of splittable kernels KG_roots in order to create as many tasks as needed to fill-in the cores of A_{span_depth} levels of the computer architecture ARCH, starting in level A_{depth}. The estimation of the cost of an application partition is a complex problem and will be addressed in future work. It depends on several factors such as the computational load of the kernels, the computational capacity of the processing elements, the amount of data that needs to be transferred, the synchronization between cores, etc.

Overall, the strategy outlined in this section enables the automatic partition of full-scale applications. The kernel-based IR (KDDG and KCFG) naturally reflects the structure of the source code and, thus, avoids the violation of the data dependences specified by the programmer. In the following sections we will show the behavior of this algorithm with the Sobel edge filter and the EQUAKE application of SPEC CPU 2000.

4 Case Study 1: Sobel Edge Filter

The Sobel edge filter is a well-known algorithm widely used in image processing and computer vision. This algorithm detects the edges of an image, that is, those pixels whose intensity is very different from the intensity of the neighbor pixels. For each pixel, the algorithm computes the gradient value that provides the largest increase from light to dark. For illustrative purposes, consider the interprocedural implementation shown in Figure 1. For each pixel of the original image (see loops in lines 19–20), the procedure `gradient_aprox` computes a convolution of the 3×3 matrix `GX` and the intensity of the pixel and its eight neighbors (lines 29–30). A similar convolution with the 3×3 matrix `GY` is also computed (lines 31–32). Finally, the sum of the absolute values of the two convolutions is truncated to the interval $[0, 255]$ (lines 35–36) before being stored in the output filtered image (lines 38–39). Note that, in order to compute the convolutions, the image boundaries are not processed (lines 24–27).

Figure 2 shows the kernel-based IR of the Sobel application, shaded nodes being the splittable kernels and thick solid lines being the kernel-level flow dependences. The types of kernels appearing in the IR are: *nc/inv* for initialization of variables to constant values (see $K(sumY_{23})$ in LOOP2); *nc/lin* for linear inductions ($K(Y_{2,92})$ in LOOP1); *nc/subs* for unpredictable values at compile-time (e.g., subscripted subscripts, pointer dereferences) (see $K(SUM_{77})$ in LOOP2); *nc/reduc* for scalar reductions ($K(sumY_{9,10,70,95})$ in LOOP4); and *nc/assig/lin:lin* for the initialization of a 2D array variable using a linear access pattern in both dimensions ($K(@edgeImage\_data_{101,128,129})$ in LOOP2).

The automatic partitioning algorithm presented in Section 3 proceeds as follows. The splittable kernels (shaded nodes) are $K(sumY_{9,10,70,95})$,

```
1  void gradient_aprox (long *sum, unsigned char **data,
2                       int cols, int Y, int X, int G[3][3])
3  {
4    int I, J;
5    for(I=-1; I<=1; I++)
6      for(J=-1; J<=1; J++)
7        (*sum) = (*sum) +
8            (int)((*((*data) + X + I + (Y + J)*cols)) * G[I+1][J+1]);
9  }
10
11 int main(void)
12 {
13   int             originalImage_rows, originalImage_cols,
14   int             edgeImage_rows, edgeImage_cols;
15   unsigned char*  originalImage_data, edgeImage_data;
16   int             X, Y, I, J, GX[3][3], GY[3][3];
17   long            sumX, sumY, SUM;
18
19   for(Y=0; Y<=(originalImage_rows-1); Y++)  {
20     for(X=0; X<=(originalImage_cols-1); X++)  {
21       sumX = 0;
22       sumY = 0;
23
24       if(Y==0 || Y==originalImage_rows-1)
25         SUM = 0;
26       else if(X==0 || X==originalImage_cols-1)
27         SUM = 0;
28       else {
29         gradient_aprox (&sumX, originalImage_data,
30                         originalImage_cols, Y, X, GX);
31         gradient_aprox (&sumY, originalImage_data,
32                         originalImage_cols, Y, X, GY);
33         SUM = abs(sumX) + abs(sumY);
34       }
35       if(SUM>255) SUM=255;
36       if(SUM<0) SUM=0;
37
38       *(edgeImage_data + X + Y*originalImage_cols) =
39           255 - (unsigned char)(SUM);
40     }
41   }
42 }
```

Fig. 1. Source code of the Sobel application.

$K(sumX_{7,8,48,93})$ and $K(@edgeImage\_data_{101,128,129})$. Thus, INITIALIZATION() marks $K(sumY_{9,10,70,95})$ and $K(sumX_{7,8,48,93})$ as not splittable because they are attached to execution scopes of 3 iterations only (see lines 5–6 in Figure 1). As a result, both kernels will be executed sequentally or accelerated with SIMD-like vector instructions.

Next, SEARCH_BEST_PARTITIONING() is invoked with KG being the whole KCFG with root node $K(@edgeImage\_data_{101,128,129})$, with $A_{span\_depth} = 1$ and $A_{depth} = 0$. For illustrative purposes, two multicore systems are considered. First, ARCH1 is an homogeneous multicore processor. As $KG_{depth} = ARCH_{depth} = 1$, the kernel *nc/assig/lin:lin* is parallelized by distributing the iteration space among the cores. Second, ARCH2 is a cluster of homogeneous multicore nodes. As $KG_{depth} < ARCH_{depth}$, the algorithm will evaluate the cost of parallelizing the kernel either on level 0 or on level 1 of ARCH2. It will also evaluate the

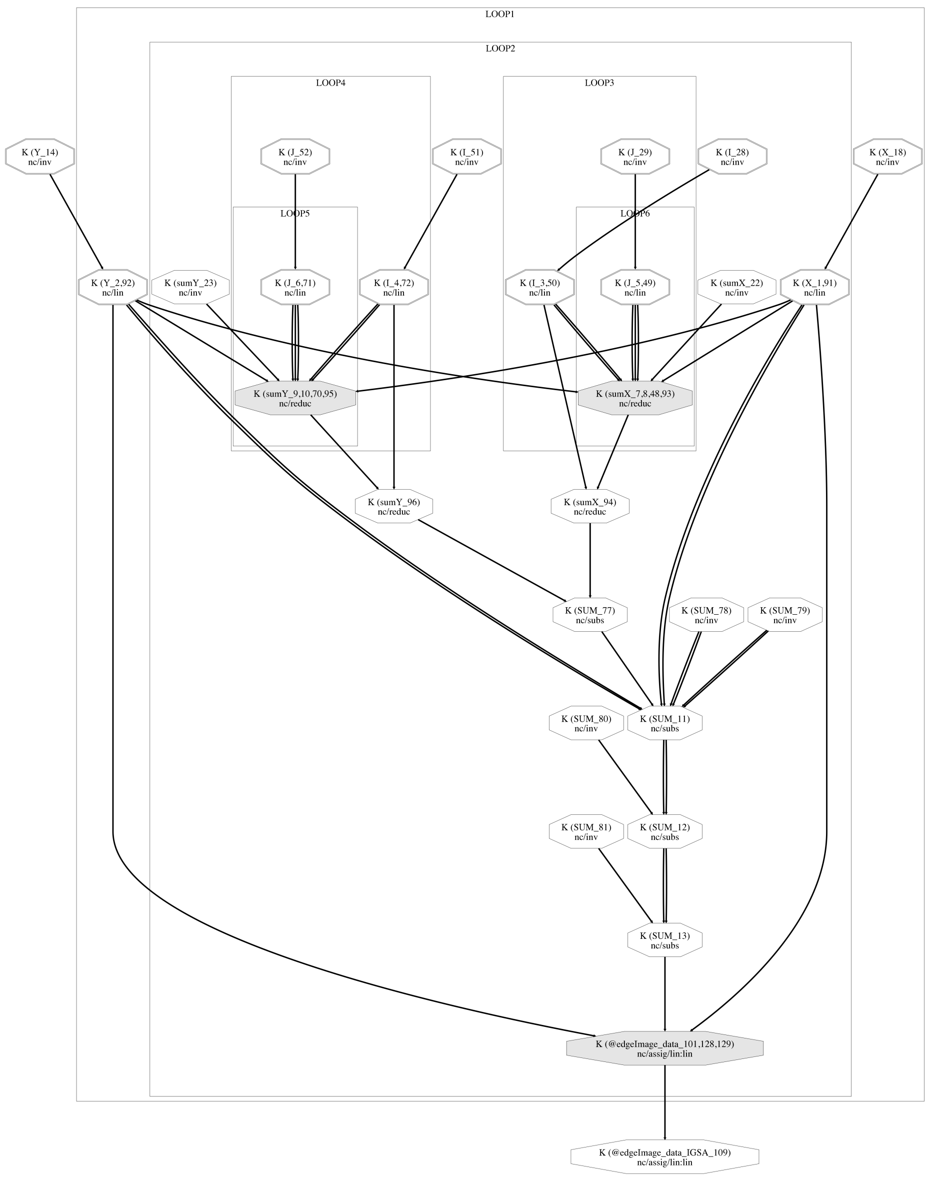

Fig. 2. Kernel-based Intermediate Representation of the Sobel application.

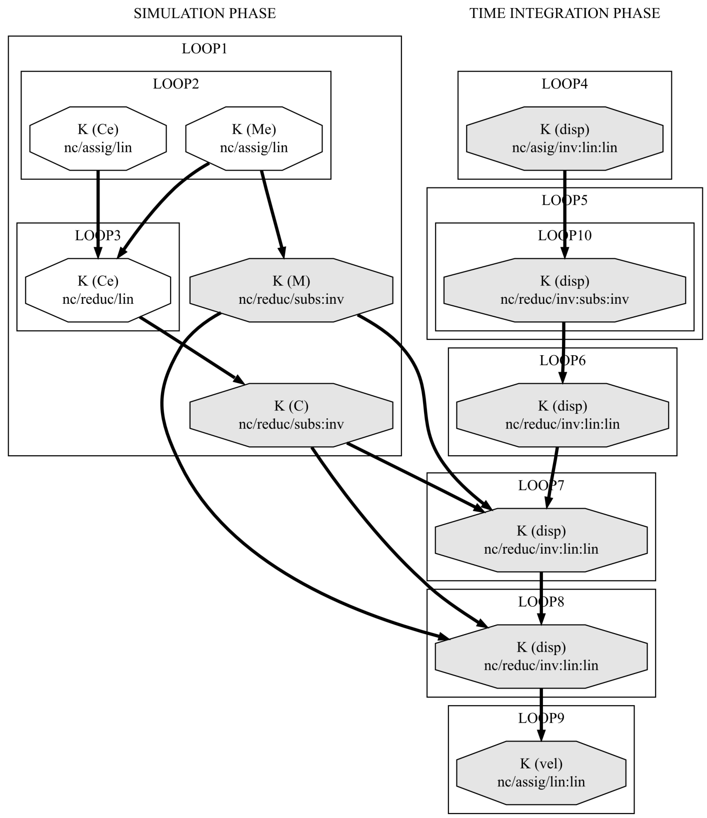

Fig. 3. Kernel-based IR of an excerpt of the EQUAKE application.

cost of spanning the kernel across levels 0 and 1 of ARCH2. The best mapping according to a cost estimation model will be selected.

5 Case Study 2: EQUAKE

One of the benchmarks that are part of the SPEC CPU2000 suite is *EQUAKE*. This application computes a simulation of 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 timesteps.

Figure 3 shows the kernel-based IR of an excerpt of the most time-consuming parts of EQUAKE. For the sake of clarity, the kernel notation omits the version numbers of the GSA variables. In addition, the kernels of read-only variables, temporary scalar variables and loop indices are not depicted. The EQUAKE application can be viewed as two separated phases. In the first phase, a simulation traverses the set of finite elements in order to compute the global simulation variables. In each iteration, the individual contribution is computed and stored in the element matrices represented by kernels $K(Me)$ and $K(Ce)$ attached to LOOP2 and LOOP3. These values are later assembled in kernels $K(M)$ and $K(C)$ that compute irregular array reductions (*nc/reduc/subs:inv*), which are attached to the outer LOOP1. In the second phase, a time integration loop computes the displacement (3D array `disp`) using the values corresponding to the two previous timesteps and involving several procedure calls at run-time. Note that the kernel $K(disp)$ in LOOP5 hides a procedure call to `smvp()`, which consumes more than 70% of the total execution time. The kernels $K(M)$ and $K(C)$ computed during the simulation phase are used as input data in the time integration phase. Once the displacement ($K(disp)$ in loops from LOOP4 to LOOP8) has been computed, each iteration finishes by calculating the velocity ($K(vel)$ in LOOP9).

The automatic partitioning algorithm starts with a search of the kernels with a low computational load. Thus, INITIALIZATION() marks $K(Me)$ and $K(Ce)$ in LOOP2 and LOOP3 as not splittable because arrays `Ce` and `Me` have only 12 elements. The remaining kernels work with arrays of `ARCHnodes` elements, which is unknown at compile time and is supposed to be a large value (in fact, `ARCHnodes` is the number of nodes of the finite element mesh). Next, INITIALIZATION() merges execution scopes from LOOP4 up to LOOP9 because they are connected with one kernel-level flow dependence only.

The kernel-based IR of Figure 3 represents the computation of one iteration of the time integration loop. Thus, the behavior of SEARCH_BEST_PARTITIONING() is as follows. First, we consider the multicore processor ARCH1. As $KG_{depth} > ARCH_{depth}$, the algorithm will explore different possibilities of mapping the splittable kernels to processor cores. When $K(vel)$ is split to create a set of tasks, each task is devoted to compute a subarray of `vel` in LOOP9. Thus, in order to minimize communication and synchronization, they must also be assigned the computation of the corresponding subarrays of `disp` in execution scopes from LOOP4 to LOOP8. As a result, the tasks work in parallel with memory locations that do not overlap. Finally, note that kernel $K(disp)$ in LOOP5 with an irregular access pattern needs to be transformed using an inspector-executor approach to avoid communication and synchronization between the cores. With a computer architecture with more levels of parallelism like ARCH2, the algorithm will also evaluate the cost of splitting $K(disp)$ in the inner LOOP10 simultaneously in order to minimize communication among nodes.

6 Related Work

Automatic partitioning of sequential applications is an important problem in many areas of computer science, rasing from maximizing performance for network processors up to compute-assisted design. There has been extensive research in partitioning multiple concurrent programs, called processes or tasks, among multiple processing elements [19, 8, 6, 13, 9–11, 15]. These approaches mainly focus on clustering and scheduling as they assume the sequential application is split into multiple concurrent programs by the compiler or the programmer, often using a domain-specific programming language.

Automatic partitioning of single sequential programs [20, 21, 17] is driven by an intermediate representation that captures the semantics of the program, typically, at the statement level and at the procedure level. Such intermediate representation consists of a program dependence graph (PDG) annotated with information about both the program and the target parallel architecture. The main limitation of PDG-centric approaches is that variations in the programming style may have a great impact on the quality of the partitioning. In contrast, our domain-independent kernel-centric approach hinges on the recognition engine of the XARK compiler framework, which hides the complexity of the implementation details to the partitioning algorithm.

7 Conclusions and Future Work

In this work we have formally defined a new compiler IR built on top of the domain-independent concept-level kernels recognized by the XARK compiler framework. We have also sketched a new partitioning algorithm for sequential applications based on the new kernel-based IR. We have illustrated the behavior of the approach with two interprocedural implementations of the Sobel edge filter and the EQUAKE application of SPEC CPU2000.

As future work we will define a cost model in order to estimate the cost of a partition considering factors as loop iterations, operation costs, synchronization costs, volume of data transferred among processors, etc. We will evaluate our approach with representative interprocedural implementations of well-known benchmarks.

References

1. Andión, J.M., Arenaz, M., Touriño, J.: A New Intermediate Representation for GCC based on the XARK Compiler Framework. In: Proceedings of 2nd International Workshop on GCC Research Opportunities (GROW) (in conjunction with the 5th International Conference on High-Performance Embedded Architectures and Compilers (HiPEAC)), pp. 89–100. Pisa, Italy (2010)
2. Arenaz, M., Amoedo, P., Touriño, J.: Efficiently Building the Gated Single Assignment Form in Codes with Pointers in Modern Optimizing Compilers. In: Proceedings of 14th International Euro-Par Conference (Euro-Par), Lecture Notes in Computer Science 5168:360–369. Las Palmas de Gran Canaria, Spain (2008)
3. Arenaz, M., Touriño, J., Doallo, R.: Compiler Support for Parallel Code Generation through Kernel Recognition. In: Proceedings of 18th International Parallel and Distributed Processing Symposium (IPDPS). Santa Fe, NM (2004)

4. Arenaz, M., Touriño, J., Doallo, R.: Program Behavior Characterization Through Advanced Kernel Recognition. In: Proceedings of 13th International Euro-Par Conference (Euro-Par), Lecture Notes in Computer Science 4641:237–247. Rennes, France (2007)
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. Ball, M., Cifuentes, C., Bairagi, D.: Partitioning of Code for a Massively Parallel Machine. In: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 225–236. Washington, DC (2004)
7. Callahan, D.: Recognizing and Parallelizing Bounded Recurrences. In: Proceedings of 4th International Workshop on Languages and Compilers for Parallel Computing (LCPC), Lecture Notes in Computer Science 589:169–185. Santa Clara, CA (1991)
8. Choudhary, A.N., Narahari, B., Nicol, D.M., Simha, R.: Optimal Processor Assignment for a Class of Pipelined Computations. IEEE Trans. Parallel Distrib. Syst. 5(4):439–445 (1994)
9. Dai, J., Huang, B., Li, L., Harrison, L.: Automatically Partitioning Packet Processing Applications for Pipelined Architectures. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 237–248. Chicago, IL (2005)
10. Ennals, R., Sharp, R., Mycroft, A.: Task Partitioning for Multi-Core Network Processors. In: Proceedings of 14th International Conference on Compiler Construction (CC), Lecture Notes in Computer Science 3443:76–90. Edinburgh, UK (2005)
11. Girkar, M., Polychronopoulos, C.D.: Partitioning Programs for Parallel Execution. In: Proceedings of 2nd International Conference on Supercomputing (ICS), pp. 216–229. Saint-Malo, France (1988)
12. González-Domínguez, J., Taboada, G.L., Fraguela, B.B., Martín, M.J., Touriño, J.: Servet: A Benchmark Suite for Autotuning on Multicore Clusters. In: Proceedings of 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS). Atlanta, GA (2010)
13. Li, L., Huang, B., Dai, J., Harrison, L.: Automatic Multithreading and Multiprocessing of C Programs for IXP. In: Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), pp. 132–141. Chicago, IL (2005)
14. Lin, Y., Padua, D.A.: On the Automatic Parallelization of Sparse and Irregular Fortran Programs. In: Proceedings of 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers (LCR), Lecture Notes in Computer Science, 1511:41–56. Pittsburgh, PA (1998)
15. Liu, L., Li, X.F., Chen, M., Ju, R.D.: A Throughput-Driven Task Creation and Mapping for Network Processors. In: Proceedings of International Conference on High-Performance Embedded Architectures and Compilers (HiPEAC), pp. 227–241. Ghent, Belgium (2007)
16. Pinter, S.S., Pinter, R.Y.: Program Optimization and Parallelization Using Idioms. ACM Trans. Program. Lang. Syst. 16(3):305–327 (1994)
17. Rul, S., Vandierendonck, H., De Bosschere, K.: Towards Automatic Program Partitioning. In: Proceedings of the 6th ACM Conference on Computing Frontiers (CF), pp. 89–98. New York, NY (2009)
18. Setoain, J., Tenllado, C., Gómez, J.I., Arenaz, M., Prieto, M., Touriño, J.: Towards Automatic Code Generation for GPU Architectures. In: Proceedings of 9th Inter-

national Workshop on State-of-the-Art in Scientific Computing on GPUs (PARA). Trondheim, Norway (2008)
19. Subhlok, J., Vondran, G.: Optimal Mapping of Sequences of Data Parallel Tasks. In: Proceedings of the ACM SIGPLAN on Principles and Practice of Parallel Programming (PPOPP), pp. 134–143. Santa Barbara, CA (1995)
20. Subramanian, R., Pande, S.: A Framework for Performance-based Program Partitioning. In book "Progress in Computer Research", Nova Science Publishers, Inc., pp. 151–169. Commack, NY (2001)
21. Vahid, F.: Partitioning Sequential Programs for CAD using a Three-Step Approach. ACM Trans. Design Autom. Electr. Syst. 7(3):413–429 (2002)