Una Nueva Representación Intermedia para GCC basada en el Entorno de Compilación XARK

José M. Andión, Manuel Arenaz y Juan Touriño

Grupo de Arquitectura de Computadores
Departamento de Electrónica y Sistemas
Universidad de A Coruña
España

Índice

- Introducción: Motivación y Fundamentos
- Nueva IR basada en Kernels
  - Kernel-based Data Dependence Graph (K-DDG)
  - Kernel-based Control Flow Graph (K-CFG)
- Paralelización Automática
  - Descomposición en Tareas
- Conclusiones y Trabajo Futuro

Índice

- **Introducción: Motivación y Fundamentos**
- Nueva IR basada en Kernels
  - Kernel-based Data Dependence Graph (K-DDG)
  - Kernel-based Control Flow Graph (K-CFG)
- Paralelización Automática
  - Descomposición en Tareas
- Conclusiones y Trabajo Futuro

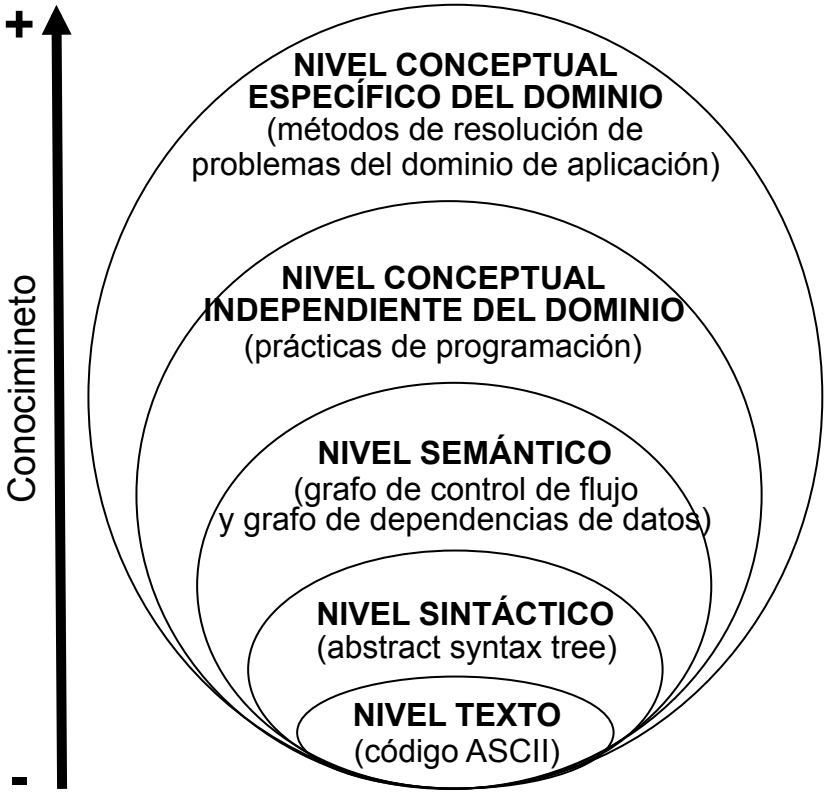
Motivación

- Aumento en el número de núcleos en los procesadores domésticos
- Hacer un uso eficiente de la arquitectura del computador es una tarea compleja
- La tecnología de compiladores actual basada en ASTs no expone el paralelismo existente en las aplicaciones reales
- Proponemos una nueva representación intermedia que expone múltiples niveles de paralelismo en los programas.

Caso de Estudio: EQUAKE

- Un ejemplo de aplicación real es EQUAKE, del SPEC CPU2000
- Simulación de ondas sísmicas en valles de gran tamaño y altamente heterogéneos
- Método de elementos finitos
  - Fase de simulación
  - Fase de integración en tiempo
- El 70 % del tiempo de ejecución es consumido por smvp()

Reconocimiento de Kernels



Reuso de librerías paralelas optimizadas
Reingeniería del software
Síntesis de programas

Substitución de variables de inducción
Paralelización de inducciones, reducciones

Propagación de constantes
Eliminación de subexpresiones comunes

Renombrado de variables

Búsqueda y reemplazo de strings

Entorno de Compilación XARK

- Solución general y extensible para el reconocimiento automático de kernels en el nivel conceptual independiente del dominio.
- Propiedades:
  - Completitud: escalares/arrays/punteros, ifs-endifs
  - Robustez: diferentes versiones de un kernel
  - Delocalización: sentencias esparcidas por el código fuente
  - Unicidad: un código, un kernel
  - Extensibilidad: kernels definibles por el usuario

M. Arenaz, J. Touriño and R. Doallo: "XARK: An eXtensible framework for Automatic Recognition of computational Kernels", ACM *Trans. Program. Lang. Syst.*, 30(6):1-56, October 2008

Reconocimiento de smvp()

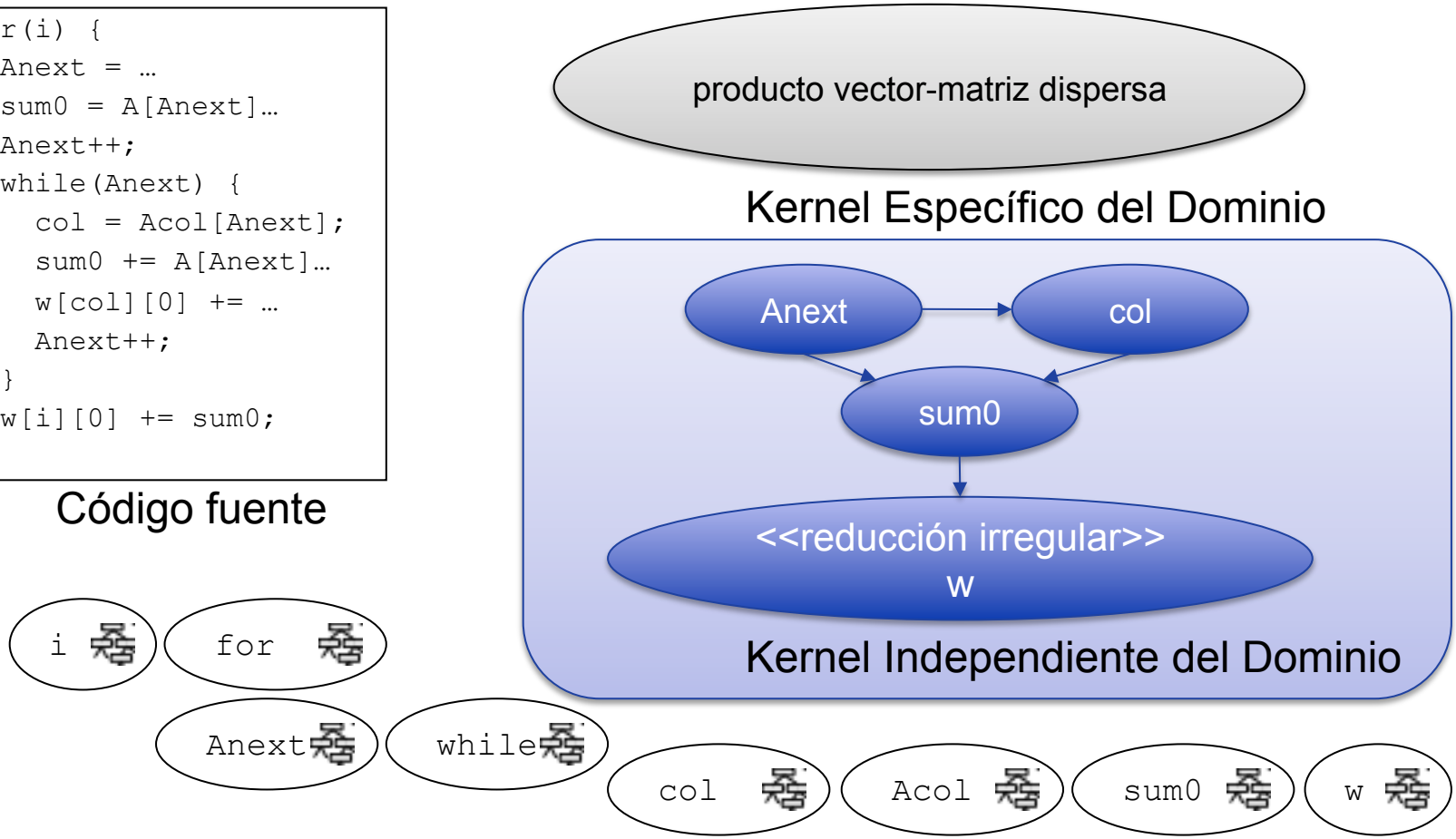
```
void smvp(int nodes, double ***A, int *Acol, int *Aindex, double **v, double **w) {
  int i, Anext, Alast, col; double sum0, sum1, sum2;

  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;
  }
}
```

Reconocimiento de smvp()

```
for(i) {
  Anext = …
  sum0 = A[Anext]…
  Anext++;
  while(Anext) {
    col = Acol[Anext];
    sum0 += A[Anext]…
    w[col][0] += …
    Anext++;
  }
  w[i][0] += sum0;
}
```

Código fuente



Bosque de ASTs + DDG + CFG

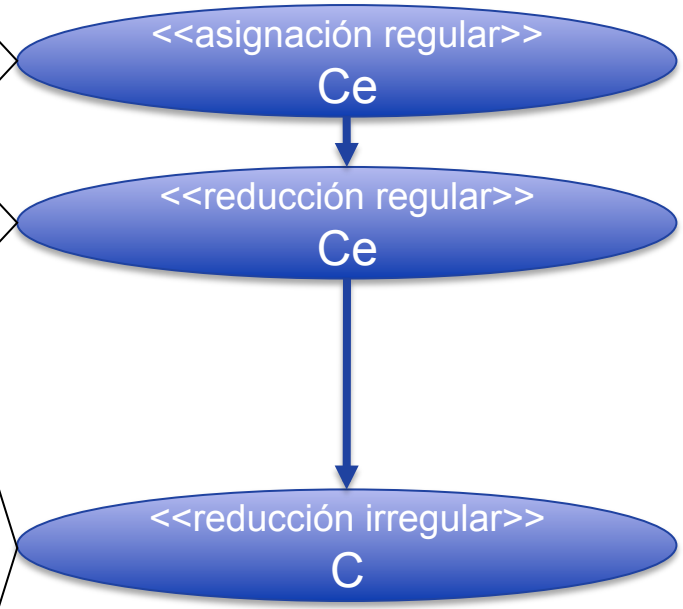
Índice

- Introducción: Motivación y Fundamentos
- **Nueva IR basada en Kernels**
  - **Kernel-based Data Dependence Graph (K-DDG)**
  - **Kernel-based Control Flow Graph (K-CFG)**
- Paralelización Automática
  - Descomposición en Tareas
- Conclusiones y Trabajo Futuro

Kernel-based Data Dependence Graph (K-DDG)

- Proporcionado por el entorno de compilación XARK

```
/* Fase de simulación */
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];
   }
}
```



Kernel-based Control Flow Graph (K-CFG)

- Similar al CFG
- Problema: estabecimiento de dependencias de flujo a nivel de kernels (relación de dominación)
- Construcción en dos etapas
  - Agrupar los kernels en ámbitos de ejecución
  - Buscar dependencias de flujo

Kernel-based Control Flow Graph (K-CFG)

- Ámbito de ejecución
  - Similar a BB en CFG
  - Calculado utilizando el concepto de region de un grafo de flujo
  - El programa se rompe en una jerarquía de regiones bucle que representan los ámbitos de ejecución y los kernels se asocian a ellos.
- Todas las sentencias de un kernel pertenecen a su ámbito de ejecución

Algorithm 1 Cálculo de los ámbitos de ejecución.

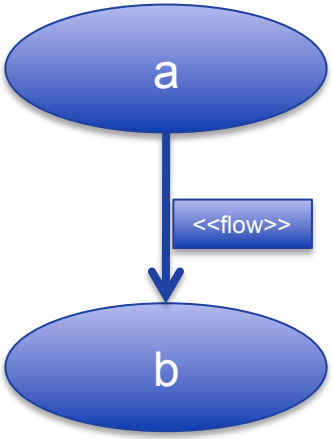
```
Entrada: K-DDG, CFG, DT
foreach kernel K en el K-DDG do
    bb_dom = bloque básico del CFG que contiene una sentencia de K (excluyendo μ)
    foreach sentencia stmt en K do
        if stmt no es una sentencia μ then
            bb_stmt = bloque básico del CFG que contiene a stmt
            if bb_stmt domina a bb_dom then
                bb_dom = bb_stmt
            end if
        end if
    end for
    K.ámbito_de_ejecución = región bucle más interna para bb_dom;
end for
```

Kernel-based Control Flow Graph (K-CFG)

Algorithm 2 Detección de dependencias de flujo a nivel de kernel.

Entrada: K-DDG, K-CFG, CFG, DT

1: **foreach** dependencia a nivel de kernel $K_1 \rightarrow K_2$ del K-DDG **do**
2: R_1 = ámbito_de_ejecución(K_1)
3: R_2 = ámbito_de_ejecución(K_2)
4: **if** (R_1.región_padre = R_2.región_padre) & (R_1 precede a R_2 en la jerarquía) **then**
5: marcar $K_1 \rightarrow K_2$ como dependencia de flujo
6: **else if** $\forall s_1 \in K_1\ \exists s_2 \in K_2$ tal que las sentencias s_1 y s_2
7: pertenecen al mismo bloque básico en el CFG
8: y s_1 precede a s_2 en el DT **then**
9: marcar $K_1 \rightarrow K_2$ como dependencia de flujo
10: **end if**
11: **end for**




```
a = 5;
b = a + 1;
```

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

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

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

Índice

- Introducción: Motivación y Fundamentos
- Nueva IR basada en Kernels
  - Kernel-based Data Dependence Graph (K-DDG)
  - Kernel-based Control Flow Graph (K-CFG)
- **Paralelización Automática**
  - **Descomposición en Tareas**
- Conclusiones y Trabajo Futuro

Paralelización Automática

- Paralelización Automática
  - Detección del paralelismo existente
  - Descomposición y generación de código paralelo
- La IR basada en kernels expone múltiples niveles de paralelismo
  - Paralelismo intra-kernel: dentro de un kernel
    - Ampliamente estudiados en la literatura de compiladores de los 90s, especialmente las reducciones irregulares
  - Paralelismo inter-kernel: entre kernels

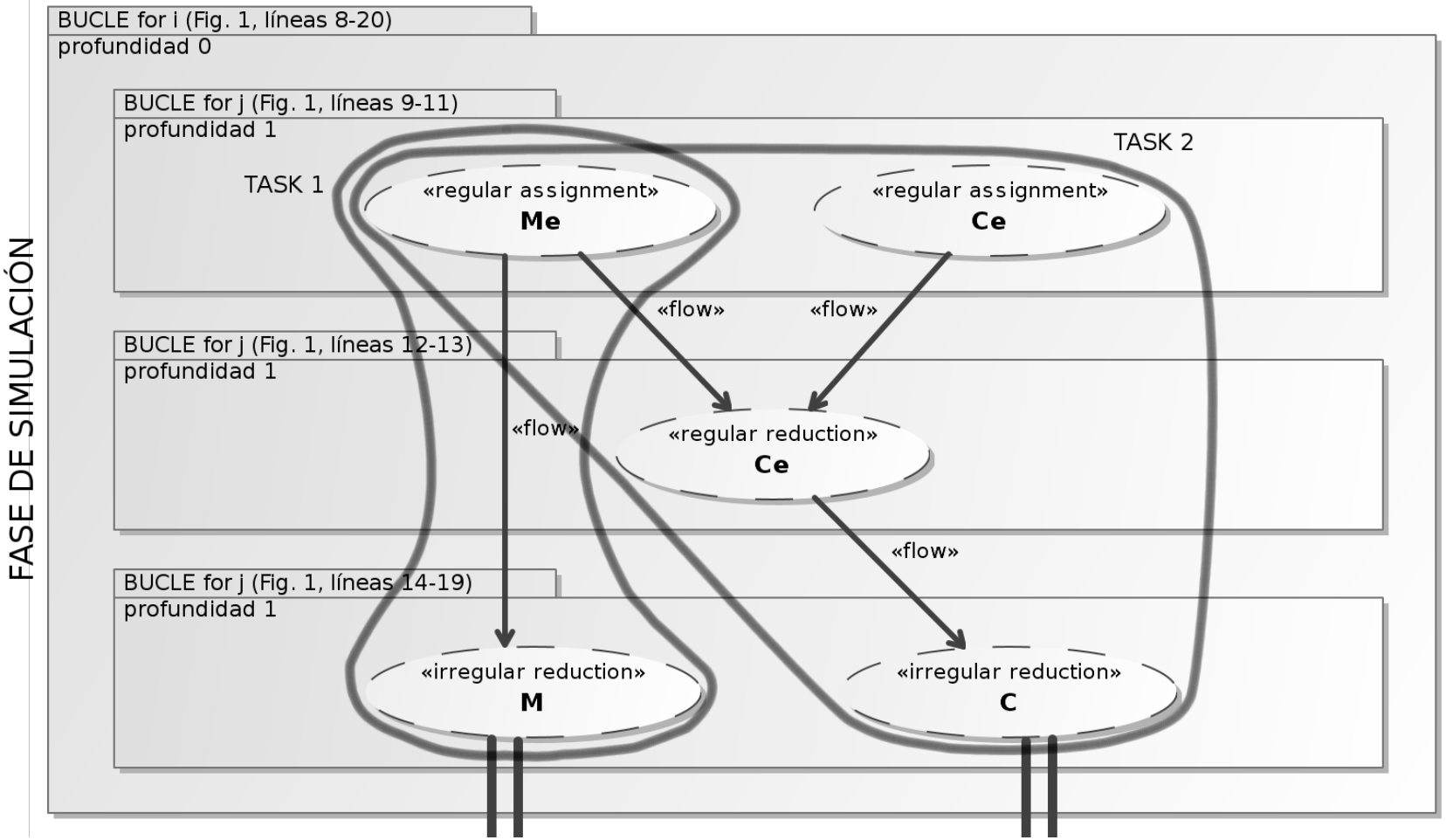
Descomposición en tareas para procesadores multi-núcleo

Algorithm 3 Descomposición en tareas para procesadores multinúcleo.

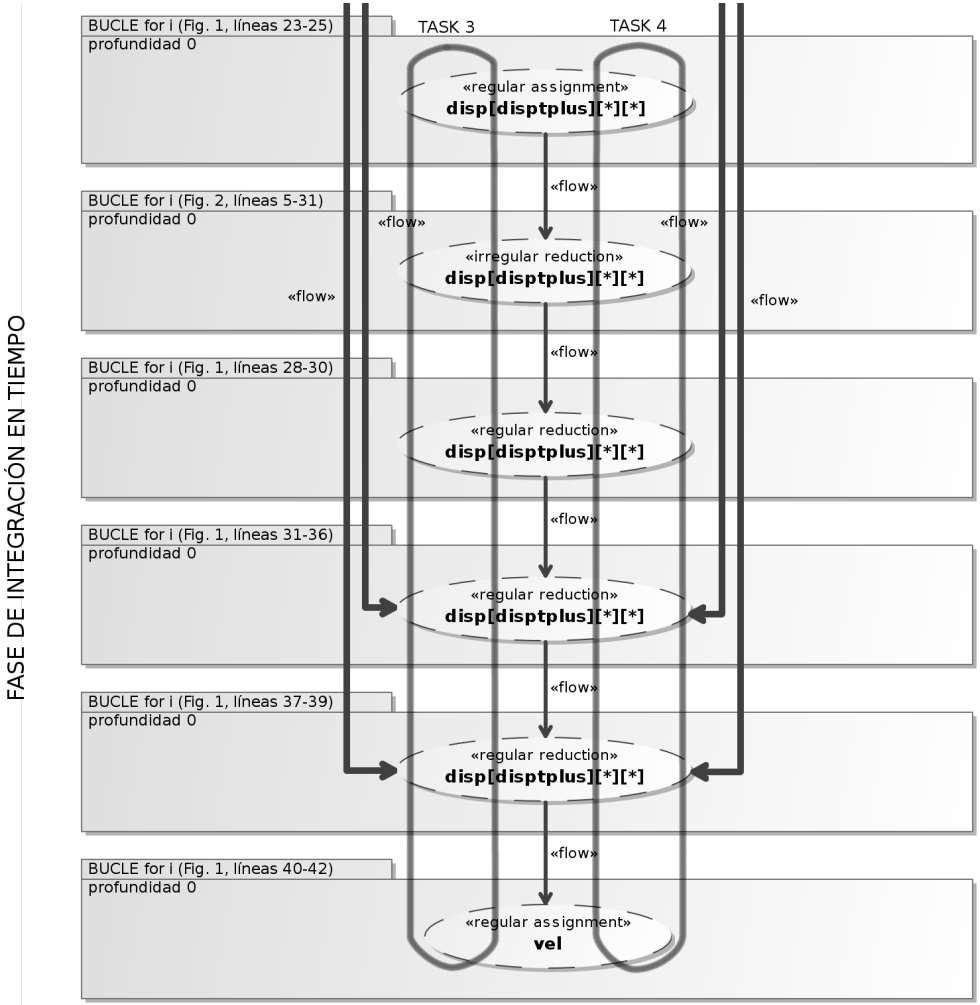
Entrada: K-DDG, K-CFG

```
fusionar los ámbitos de ejecución con un kernel y un arco que cruce frontera
d = 0
foreach ámbito de ejecución R a profundidad d en el K-CFG do
    if ∀ kernel K ∈ R tal que K es paralelizable then
        n_drain_kernels = número de kernels sin arcos de salida en el K-DDG
                                que crucen la frontera del ámbito de ejecución
        if n_drain_kernels = P then
            tasks = conjunto de P drain kernels
        else if n_drain_kernels < P then
            tasks = romper los kernels paralelizables para crear P tareas
        else
            tasks = fusionar los drain kernels para crear P tareas
        end if
        mapear tasks a los diferentes núcleos
    end if
    d++
end for
```

Caso de Estudio: EQUAKE



Caso de Estudio: EQUAKE



Índice

- Introducción: Motivación y Fundamentos
- Nueva IR basada en Kernels
  - Kernel-based Data Dependence Graph (K-DDG)
  - Kernel-based Control Flow Graph (K-CFG)
- Paralelización Automática
  - Descomposición en Tareas
- **Conclusiones y Trabajo Futuro**

Conclusiones y Trabajo Futuro

- Definición de una IR basada en kernels
  - Expone mútiples niveles de paralelismo
  - Inspirada en las IR estándar basadas en sentencias
  - Entorno para nuevas técnicas de paralelización automática de programas completos
- Trabajo en progreso
  - Port de XARK desde Polaris a GCC
    - De F77 a C, Fortran, e incluso Java, C++…
  - XARK se contruye sobre la forma GSA
    - Inter-procedural GSA sobre GIMPLE-SSA

Conclusiones y Trabajo Futuro

- Trabajo en progreso
  - Dar más conocimiento al motor de reconocimiento
- Trabajo futuro
  - Mejorar el algoritmo de construcción del K-CFG
  - Ejecutar tests con suites de benchmarks bien conocidos
  - Comparar con entornos de paralelizaxión automática existentes
  - Descomposición en tareas para many-cores y GPUs

Una Nueva Representación Intermedia para GCC basada en el Entorno de Compilación XARK

José M. Andión, Manuel Arenaz y Juan Touriño

Grupo de Arquitectura de Computadores
Departamento de Electrónica y Sistemas
Universidad de A Coruña
España