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
{jose.manuel.andion, manuel.arenaz, juan}@udc.es

Resumen

La generación automática de código paralelo eficiente sigue siendo un reto para los compiladores, especialmente ahora debido al aumento en el número de núcleos en los procesadores domésticos. Los compiladores paralelizadores típicamente tratan la detección automática de paralelismo a través del análisis de un conjunto de grafos estándar que capturan información sobre las sentencias del programa. Ejemplos bien conocidos son el grafo de dependencias de datos, el grafo de control de flujo y el árbol de dominación. Este artículo presenta una nueva representación intermedia *(intermediate representation –IR–)* diseñada para facilitar la detección de paralelismo en programas secuenciales. A diferencia de las IRs estándar, la nueva IR está basada en el concepto de *núcleo computacional (o kernel)* proporcionado por el entorno de compilación XARK [5]. Ejemplos de kernels son las inducciones, las reducciones o las recurrencias. Además, este artículo esboza varias ideas para usar esta nueva IR para la generación automática de código paralelo para procesadores multinúcleo. Como caso de estudio se ha utilizado una aplicación de la suite de benchmarks SPEC CPU2000.

1. Introducción

El desarrollo de programas que hagan un uso eficiente de la arquitectura del computador es una tarea compleja incluso para programadores experimentados. La popularización de los procesadores multinúcleo hace patente esta situación ya que la tecnología de compiladores actual no permite generar código paralelo eficiente para estos procesadores, siendo responsabilidad del programador el explotar el paralelismo disponible en el hardware.

Los compiladores optimizadores y paralelizadores dividen los programas en sentencias que se representan como árboles de sintaxis abstracta *(abstract syntax trees –ASTs–)*. Para capturar el comportamiento del programa, se construye una IR que consiste en un conjunto de grafos estándar que conectan los ASTs que representan sentencias del programa [1, 2, 11]. Por ejemplo, el compilador GCC [7] establece relaciones entre sentencias GIMPLE-SSA construyendo un grafo de control de flujo (CFG), un grafo de dependencias de datos (DDG) y un árbol de dominación (DT). Aunque esta aproximación se ha mostrado efectiva en la compilación de códigos reales para procesadores mononúcleo, su nivel de abstracción hace difícil detectar el paralelismo disponible en los programas.

El reconocimiento automático de construcciones de programa que son utilizadas frecuentemente por los desarrolladores de software (en adelante, *kernels*) es un poderoso mecanismo para detectar paralelismo automáticamente [4, 6, 8, 9, 10]. XARK [5] es un entorno de compilación extensible que proporciona una solución completa, robusta y general al problema del reconocimiento automático de kernels incluso en la presencia de flujos de control complejos. Este artículo describe una nueva IR basada en kernels y esboza un algoritmo para guiar la paralelización automática en procesa-

dores multinúcleo. Como caso de estudio se ha analizado la aplicación EQUAKE del benchmark SPEC CPU2000.

El resto del artículo se organiza del siguiente modo. La Sección 2 describe la aplicación EQUAKE. La Sección 3 presenta el entorno XARK y la colección de kernels reconocidos en EQUAKE. La Sección 4 muestra la nueva IR basada en kernels. La Sección 5 presenta un algoritmo de paralelización automática cuyo funcionamiento se ilustra con la aplicación EQUAKE. Finalmente, la Sección 6 presenta las conclusiones y el trabajo futuro.

2. Aplicación EQUAKE

EQUAKE es un programa de SPEC CPU2000 que realiza una simulación de ondas sísmicas en valles de gran tamaño y altamente heterogéneos. EQUAKE es capaz de recuperar el histórico del movimiento de tierra debido a un evento sísmico específico en cualquier lugar de un valle. Las computaciones se llevan a cabo en una malla no estructurada que localmente calcula las longitudes de onda utilizando un método de elementos finitos. Las Figuras 1 y 2 muestran un extracto de las 1512 líneas de código y los 16 procedimientos que componen EQUAKE. Concretamente, las Figuras 1 y 2 muestran los procedimientos `main()` y `smvp()` respectivamente. Nótese que `smvp()` es la rutina más costosa del programa. Aproximadamente consume más del 70 % del tiempo total de ejecución y, por tanto, es un objetivo adecuado para un compilador optimizador.

El bucle $\texttt{for}_i$ del procedimiento `main()` (Figura 1, líneas 8–20) recorre el conjunto de elementos finitos para calcular las variables de simulación globales `M` y `C` utilizando la contribución de cada elemento finito a la solución. En cada iteración $\texttt{for}_i$, la contribución individual es calculada y almacenada en las matrices de elementos, por ejemplo, la matriz `Ce`.

La salida de EQUAKE es un informe de los desplazamientos en el hipocentro y epicentro del terremoto para un número predeterminado de pasos de tiempo de simulación. Cada iteración de $\texttt{for}_{iter}$ (Figura 1, líneas 22–45) realiza una integración de un paso de tiempo que calcula el desplazamiento (array 3D `disp`) utilizando los valores correspondientes a los dos pasos de tiempo anteriores e involucrando varias llamadas a procedimiento en tiempo de ejecución (p.ej., `smvp()`). Los arrays calculados durante la fase de simulación (p.ej., `M` y `C`) se utilizan como datos de entrada. Una vez que el desplazamiento `disp` se ha calculado, cada iteración $\texttt{for}_{iter}$ finaliza calculando la velocidad (array 2D `vel`). Las tres variables `disptminus`, `dispt` y `disptplus` se actualizan siguiendo un modelo de round-robin o intercambio circular en cada iteración de $\texttt{for}_{iter}$ (Figura 1, líneas 43–44). Estas tres variables se utilizan para acceder al array 3D `disp`, que almacena los resultados de EQUAKE en el paso de tiempo actual y en los dos anteriores.

3. Entorno de Compilación XARK

XARK [5] construye una representación jerárquica de un programa a través del reconocimiento de una colección exhaustiva de kernels. XARK descompone el programa en un conjunto de kernels mutuamente dependientes que capturan el comportamiento de un fragmento de código y determina el tipo de cada uno de los kernels. El tipo de kernel proporciona al compilador información sobre el tipo de computación que se realiza en tiempo de ejecución con las variables escalares y no escalares (p.ej., arrays, punteros) del programa. Una descripción detallada de la colección de kernels puede consultarse en [5].

Los tipos de kernel que aparecen en EQUAKE son los tres siguientes. Primero, *asignación regular*, consistente en la asignación de un nuevo valor a un conjunto de elementos de una variable array siguiendo un patrón de acceso regular. El cálculo del array `vel` en la fase de integración en tiempo (Figura 1, líneas 40–42), asigna un nuevo valor a cada elemento `vel[i][j]`, siendo `i` y `j` expresiones afines de los índices de los bucles $\texttt{for}_i$ y $\texttt{for}_j$. Segundo, *reducción regular*, en el cual a los elementos de una variable array se les asigna un nuevo valor que depende del valor almacenado previamente en cada elemento del array. Los elementos actualizados en una reducción regular son de-

```
1  double **M, **C, **M23, **C23, **V23, **vel, ***disp, ***K;
2  int i, j, k, ii, jj, kk, iter, timesteps, disptplus, dispt, disptminus;
3  double time, Ke[12][12], Me[12], Ce[12], alpha;
4
5  /* Fase de inicializacion */
6  disptplus = 0; dispt = 1; disptminus = 2;
7  /* Fase de simulacion */
8  for (i = 0; i < ARCHelems; i++) {
9    for (j = 0; j < 12; j++) {
10     Me[j] = 0.0; Ce[j] = 0.0;
11   }
12   for (j = 0; j < 12; j++)
13     Ce[j] = Ce[j] + alpha * Me[j];
14   for (j = 0; j < 4; j++) {
15     M[ARCHvertex[i][j]][0] += Me[j*3]; M[ARCHvertex[i][j]][1] += Me[j*3+1];
16     M[ARCHvertex[i][j]][2] += Me[j*3+2];
17     C[ARCHvertex[i][j]][0] += Ce[j*3]; C[ARCHvertex[i][j]][1] += Ce[j*3+1];
18     C[ARCHvertex[i][j]][2] += Ce[j*3+2];
19   }
20 }
21 /* Fase de integracion en tiempo */
22 for (iter = 1; iter <= timesteps; iter++) {
23   for (i = 0; i < ARCHnodes; i++)
24     for (j = 0; j < 3; j++)
25       disp[disptplus][i][j] = 0.0;
26   smvp(ARCHnodes,K,ARCHmatrixcol,ARCHmatrixindex,disp[dispt],disp[disptplus]);
27   time = iter * Exc.dt;
28   for (i = 0; i < ARCHnodes; i++)
29     for (j = 0; j < 3; j++)
30       disp[disptplus][i][j] *= - Exc.dt * Exc.dt;
31   for (i = 0; i < ARCHnodes; i++)
32     for (j = 0; j < 3; j++)
33       disp[disptplus][i][j] += 2.0 * M[i][j] * disp[dispt][i][j] -
34             (M[i][j] - Exc.dt / 2.0 * C[i][j]) * disp[disptminus][i][j] -
35             Exc.dt * Exc.dt * (M23[i][j] * phi2(time) / 2.0 +
36             C23[i][j] * phi1(time) / 2.0 + V23[i][j] * phi0(time) / 2.0);
37   for (i = 0; i < ARCHnodes; i++)
38     for (j = 0; j < 3; j++)
39       disp[disptplus][i][j] = disp[disptplus][i][j]/(M[i][j]+Exc.dt/2.0*C[i][j]);
40   for (i = 0; i < ARCHnodes; i++)
41     for (j = 0; j < 3; j++)
42       vel[i][j] = 0.5/Exc.dt*(disp[disptplus][i][j]-disp[disptminus][i][j]);
43   /* Intercambio circular */
44   i = disptminus; disptminus = dispt; dispt = disptplus; disptplus = i;
45 }
```

Figura 1: Extracto del código fuente de la aplicación EQUAKE.

terminados por un patrón de acceso regular. El cálculo del array `Ce` en las líneas 12–13 de la Figura 1 es una reducción regular con una operación de reducción suma (+) y un patrón de acceso lineal. Tercero, *reducción irregular*, en la cual los elementos del array que se actualizan se determinan con un array de indirección. Un ejemplo es el cálculo de `M` en la fase de simulación (Figura 1, líneas 14–16), que utiliza el array de indirección `ARCHvertex` para determinar el elemento de `M` a actualizar en cada iteración de $\texttt{for}_i$ y $\texttt{for}_j$.

El reconocimiento automático de kernels en aplicaciones reales requiere el desarrollo de un motor de reconocimiento de kernels interprocedural. Con este fin estamos trabajando [3] en el desarrollo de una forma *Gated Single Assignment (GSA) interprocedural* eficiente encima de la infraestructura GIMPLE-SSA proporcionada por GCC. Como se muestra en la Figura 2, `smvp()` lleva a cabo una reducción irregular y almacena el resultado en el array de reducción `w`, siendo `Acol` el array de indirección y asumiendo que los parámetros formales `A`, `v` y `w` apuntan a localizaciones de memoria que no se solapan. La forma GSA interprocedural permite el reconocimiento de kernels a través de las llamadas a procedimiento. En la llamada de la línea 26 de la Figura 1, se sabe que los parámetros actuales de `smvp()` apun-

```
1  void smvp(int nodes, double ***A, int *Acol, int *Aindex, double **v, double **w) {
2    int i, Anext, Alast, col;
3    double sum0, sum1, sum2;
4
5    for (i = 0; i < nodes; i++) {
6      Anext = Aindex[i];
7      Alast = Aindex[i + 1];
8      sum0 = A[Anext][0][0]*v[i][0]+A[Anext][0][1]*v[i][1]+A[Anext][0][2]*v[i][2];
9      sum1 = A[Anext][1][0]*v[i][0]+A[Anext][1][1]*v[i][1]+A[Anext][1][2]*v[i][2];
10     sum2 = A[Anext][2][0]*v[i][0]+A[Anext][2][1]*v[i][1]+A[Anext][2][2]*v[i][2];
11     Anext++;
12     while (Anext < Alast) {
13       col = Acol[Anext];
14       sum0 += A[Anext][0][0]*v[col][0] + A[Anext][0][1]*v[col][1] +
15               + A[Anext][0][2]*v[col][2];
16       sum1 += A[Anext][1][0]*v[col][0] + A[Anext][1][1]*v[col][1] +
17               + A[Anext][1][2]*v[col][2];
18       sum2 += A[Anext][2][0]*v[col][0] + A[Anext][2][1]*v[col][1] +
19               + A[Anext][2][2]*v[col][2];
20       w[col][0] += A[Anext][0][0]*v[i][0] + A[Anext][1][0]*v[i][1] +
21                    + A[Anext][2][0]*v[i][2];
22       w[col][1] += A[Anext][0][1]*v[i][0] + A[Anext][1][1]*v[i][1] +
23                    + A[Anext][2][1]*v[i][2];
24       w[col][2] += A[Anext][0][2]*v[i][0] + A[Anext][1][2]*v[i][1] +
25                    + A[Anext][2][2]*v[i][2];
26       Anext++;
27     }
28     w[i][0] += sum0;
29     w[i][1] += sum1;
30     w[i][2] += sum2;
31   }
32 }
```

Figura 2: Código fuente del procedimiento `smvp()` de la aplicación EQUAKE.

tan a localizaciones de memoria disjuntas. De este modo, el motor de reconocimiento infiere que `smvp()` calcula una reducción irregular en el array `disp[disptplus]` en el ámbito del programa principal.

4. Nueva Representación Intermedia basada en Kernels

En esta sección proponemos una nueva IR basada en kernels diseñada para exponer al compilador el paralelismo de grano grueso y de grano fino disponible en el programa. Las IRs estándar constan típicamente de un DDG, un CFG y un DT que conectan los ASTs de las sentencias del programa. De un modo similar, nuestra nueva IR basada en kernels consta de un K-DDG *(Kernel-based DDG)* y un K-CFG *(Kernel-based CFG)*.

Como se ha mencionado en la Sección 3, XARK construye una representación jerárquica del programa. Llamamos a esta representación K-DDG, cuyos nodos representan kernels y cuyos arcos representan dependencias entre kernels. Por ejemplo, considérese en grafo de EQUAKE presentado en la Figura 3. Los nodos del K-DDG se dibujan como óvalos etiquetados con la variable del programa que almacena los resultados de las computaciones y con el tipo de kernel. Por una parte, los nodos `Me` y `M` de la fase de simulación capturan dos kernels: asignación regular y reducción irregular, respectivamente. Nótese que los kernels abstraen el cálculo de un conjunto de sentencias esparcidas por el programa. Por ejemplo, el kernel `M` contiene tres sentencias del cuerpo del bucle $\texttt{for}_j$ (Figura 1, líneas 14–16). Por otra parte, los arcos del K-DDG capturan las dependencias que conectan sentencias de diferentes kernels (véase el arco entre los kernels `Me` y `M` en la Figura 3). Nótese que las dependencias entre sentencias de un mismo kernel no aparecen en el K-DDG, pues están representadas por el tipo de kernel.

El segundo grafo de la nueva IR es el K-CFG. Proponemos un algoritmo de construcción en dos etapas cuya primera etapa agrupa a los kernels (es decir, los nodos del

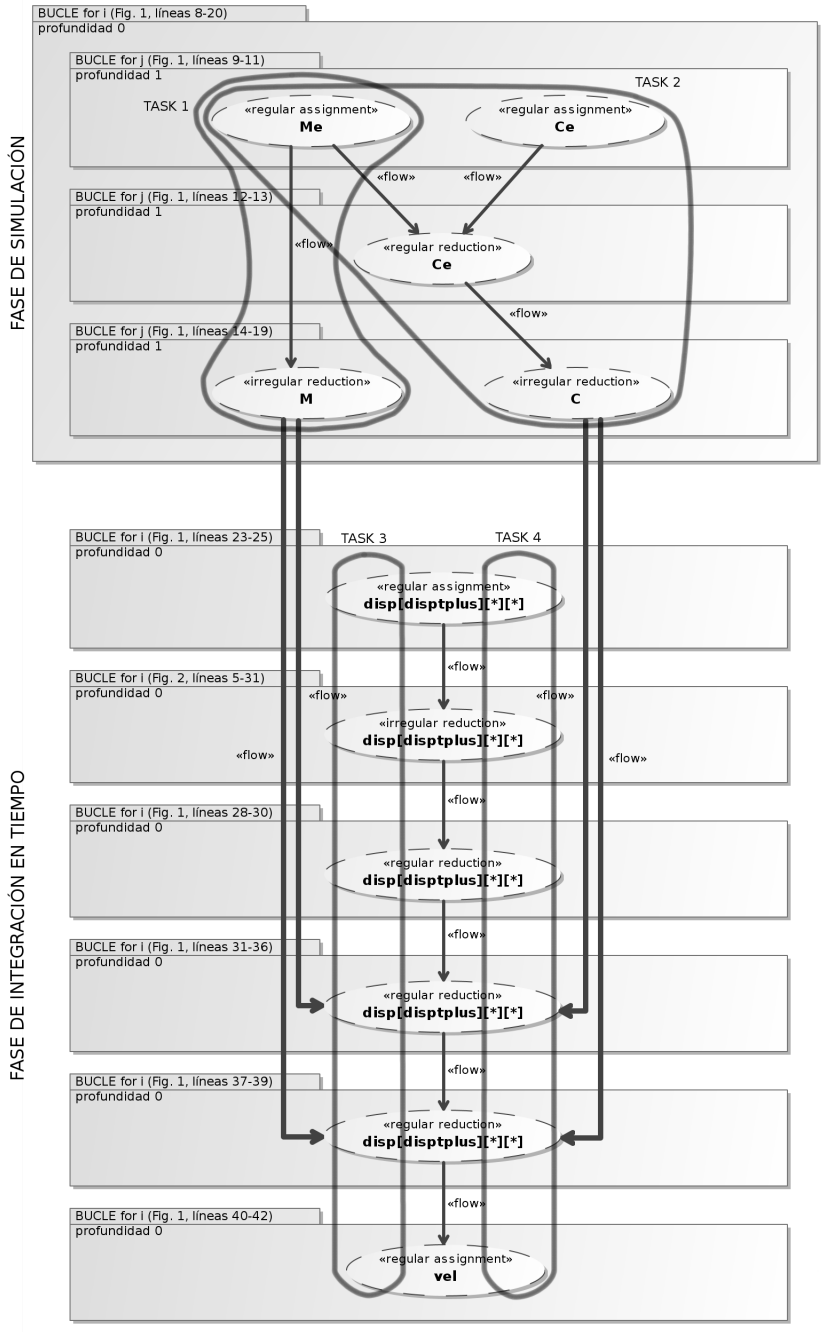

Figura 3: Representación intermedia basada en kernels del código fuente de la aplicación EQUAKE.

Algorithm 1 Cálculo de los ámbitos de ejecución.

```
Entrada: K-DDG, CFG, DT
 1: foreach kernel K en el K-DDG do
 2:     bb_dom = bloque básico del CFG que contiene una sentencia de K (excluyendo μ)
 3:     foreach sentencia stmt en K do
 4:         if stmt no es una sentencia μ then
 5:             bb_stmt = bloque básico del CFG que contiene a stmt
 6:             if bb_stmt domina a bb_dom then
 7:                 bb_dom = bb_stmt
 8:             end if
 9:         end if
10:     end for
11:     K.ámbito_de_ejecución = región bucle más interna para bb_dom;
12: end for
```

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 K1 → K2 del K-DDG do
 2:     R1 = ámbito_de_ejecución(K1)
 3:     R2 = ámbito_de_ejecución(K2)
 4:     if (R1.región_padre = R2.región_padre) & (R1 precede a R2 en la jerarquía) then
 5:         marcar K1 → K2 como dependencia de flujo
 6:     else if ∀s1 ∈ K1 ∃s2 ∈ K2  tal que las sentencias s1 y s2
 7:             pertenecen al mismo bloque básico en el CFG
 8:             y s1 precede a s2 en el DT then
 9:         marcar K1 → K2 como dependencia de flujo
10:     end if
11: end for
```

K-DDG) en *ámbitos de ejecución* los cuales se calculan utilizando el concepto de *región* [1]. Como los anidamientos de bucles son comúnmente la parte más costosa de un programa, son los candidatos más apropiados para que un compilador extraiga paralelismo. Para construir el K-CFG, el programa se rompe en una jerarquía de regiones bucle que representan los ámbitos de ejecución y los kernels se asocian a los mismos mediante el Algoritmo 1. Las entradas son el K-DDG proporcionado por XARK y el CFG y el DT proporcionados por el compilador GCC. El algoritmo determina el conjunto de bloques básicos que contienen cada sentencia de un kernel K, excluyendo las sentencias μ asociadas a las cabeceras de bucle. A continuación, se calcula el bloque básico *bb_dom* que domina a los restantes bloques básicos del conjunto. Las sentencias γ de la forma GSA aseguran que dicho bloque *bb_dom* existe. Finalmente, el ámbito de ejecución del kernel K es la región bucle más interna que contiene a *bb_dom*.

En la segunda etapa del algoritmo de construcción del K-CFG (Algoritmo 2), se buscan *dependencias de flujo* a nivel de kernel entre los arcos del K-DDG. Para que una dependencia a nivel de kernel $K_1 \rightarrow K_2$ sea una dependencia de flujo, debe existir una relación de dominación entre K_1 y K_2. Para determinar esta condición, se comparan los ámbitos de ejecución R_1 y R_2 de K_1 y K_2. Si R_1 y R_2 tienen la misma región padre y R_1 precede a R_2 en la jerarquía (Algoritmo 2, líneas 4–5), entonces todas las sentencias de K_1 se ejecutan antes que todas las de K_2 y podemos marcar dicha dependencia como de flujo.

En otro caso, para establecer una relación de flujo entre K_1 y K_2, tenemos que asegurar que todas las sentencias de K_1 dominan a todas las sentencias de K_2 (Algoritmo 2, líneas 6–10). Un ejemplo de esta situación es:

```
1 if (c == 0) {
2     a = 5;
3     b = a + 3;
4 } else {
5     a = 2;
6     b = a + 1;
7 }
```

donde en cada rama del if-then-else existe una relación de dominación entre la sentencia que escribe en `a` y la sentencia que escribe en `b`. Así, el Algoritmo 2 concluye que el kernel asociado a `a` domina al kernel asociado a `b`.

Algorithm 3 Descomposición en tareas para procesadores multinúcleo.

```
Entrada: K-DDG, K-CFG
 1: fusionar los ámbitos de ejecución con un kernel y un arco que cruce frontera
 2: d = 0
 3: foreach ámbito de ejecución R a profundidad d en el K-CFG do
 4:     if ∀ kernel K ∈ R tal que K es paralelizable then
 5:         n_drain_kernels = número de kernels sin arcos de salida en el K-DDG
 6:                           que crucen la frontera del ámbito de ejecución
 7:         if n_drain_kernels = P then
 8:             tasks = conjunto de P drain kernels
 9:         else if n_drain_kernels < P then
10:             tasks = romper los kernels paralelizables para crear P tareas
11:         else
12:             tasks = fusionar los drain kernels para crear P tareas
13:         end if
14:         mapear tasks a los diferentes núcleos
15:     end if
16:     d++
17: end for
```

5. Detección Automática de Paralelismo para Sistemas Multinúcleo

En esta sección se propone un algoritmo basado en la nueva IR descrita en la sección anterior que permite la detección del paralelismo existente en un programa y su descomposición en un conjunto de tareas que puedan ser ejecutadas en paralelo en un procesador multinúcleo (dual-, quad-, eight-core).

El pseudocódigo del Algoritmo 3 se centra en los ámbitos de ejecución de grano grueso que contienen kernels paralelizables, esto es, kernels para los que existen técnicas de transformación en código paralelo [4, 6, 8, 9, 10]. Los tipos de kernel encontrados en el K-CFG de EQUAKE son asignaciones regulares, reducciones regulares y reducciones irregulares. Todos estos kernels son paralelizables, por lo que todos los ámbitos de ejecución de EQUAKE serán objeto de análisis.

La fase de simulación de EQUAKE se representa mediante un ámbito de ejecución que contiene dos kernels sin arcos de salida que crucen la frontera. Así, el algoritmo creará dos tareas: una con los kernels `Me` y `M` (Figura 3, subgrafo TASK1), y otra con los kernels `Me`, `Ce` y `C` (subgrafo TASK2). Si el procesador multinúcleo destino contiene más de dos núcleos, entonces el algoritmo generará nuevas tareas mediante la paralelización de las reducciones irregulares `M` y/o `C` (Algoritmo 3, líneas 9–10). Téngase en cuenta que, en general, esta estrategia replica computaciones en diferentes núcleos (p.ej., `Me` pertenece a TASK1 y TASK2 en la Figura 3).

La fase de integración en tiempo no puede comenzar su ejecución antes de que la fase de simulación termine para prevenir la violación de las dependencias a nivel de kernel que cruzan dicha frontera (para ello se necesita de la sincronización apropiada). La fase de integración en tiempo se representa mediante seis ámbitos de ejecución, cada uno de los cuales contiene un kernel con una dependencia que cruza la frontera. Como resultado, estos ámbitos de ejecución se fusionan para exponer al compilador una secuencia de kernels (Algoritmo 3, línea 1). A continuación, el kernel `vel` que carece de arcos de salida es transformado en código paralelo creando tantas tareas como sea necesario. Sin pérdida de generalidad, asumamos que se crean dos tareas TASK3 y TASK4 (subgrafos en la Figura 3). Cada tarea calcula un subarray de `vel`. Así, para minimizar la comunicación y la sincronización, se debe asignar a las mismas tareas el cálculo de los correspondientes subarrays de `disp`. Como resultado, las tareas trabajan en paralelo con localizaciones de memoria que no se solapan. Finalmente, nótese que los kernels con patrones de acceso irregulares deben transformarse mediante técnicas inspector-ejecutor para evitar comunicación y sincronización entre núcleos.

En resumen, la estrategia esbozada en esta sección permite la detección de paralelismo en aplicaciones reales. La IR basada en ker-

nels (K-DDG y K-CFG) refleja naturalmente la estructura del código fuente y, de este modo, evita la violación de las dependencias de datos especificadas por el programador.

6. Conclusiones y Trabajo Futuro

Este artículo es un primer paso hacia la definición de una IR basada en kernels que exponga múltiples niveles de paralalelismo al compilador. De forma similar a las IRs estándar basadas en sentencias, el K-DDG y el K-CFG están diseñados para proporcionar un entorno potente para el desarrollo de nuevas técnicas de paralelización automática. La aplicación EQUAKE del SPEC CPU2000 se ha utilizado como caso de estudio para mostrar el potencial de esta aproximación.

Como trabajo futuro, formalizaremos los algoritmos de construcción de la nueva IR y desarrollaremos algoritmos de descomposición de tareas para sistemas multi-core, many-core y GPUs. Como casos de estudio abordaremos el análisis de benchmarks bien conocidos como SPEC o PERFECT.

Agradecimientos

Este trabajo ha sido financiado por el Ministerio de Ciencia e Innovación y fondos FEDER de la Unión Europea (Proyecto TIN2007-67537-C03) y por el Programa de FPU del Ministerio de Educación (Referencia AP2008-01012).

Referencias

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

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

[3] M. Arenaz, P. Amoedo, and J. Touriño. 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*, 2008.

[4] M. Arenaz, J. Touriño, and R. Doallo. Compiler Support for Parallel Code Generation through Kernel Recognition. In *18th International Parallel and Distributed Processing Symposium (IPDPS), Santa Fe, NM, USA*, 2004.

[5] M. Arenaz, J. Touriño, and R. Doallo. XARK: An eXtensible framework for Automatic Recognition of computational Kernels. *ACM Trans. Program. Lang. Syst.*, 30(6), 2008.

[6] D. Callahan. Recognizing and Parallelizing Bounded Recurrences. In *4th International Workshop on Languages and Compilers for Parallel Computing (LCPC), Santa Clara, CA, USA*, 1991.

[7] Free Software Foundation, Inc. GNU Compiler Collection (GCC) Internals. `http://gcc.gnu.org/onlinedocs/gccint/` [Último acceso: May 2010].

[8] Y. Lin and D. A. Padua. 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*, 1998.

[9] S. S. Pinter and R. Y. Pinter. Program Optimization and Parallelization Using Idioms. *ACM Trans. Program. Lang. Syst.*, 16(3), 1994.

[10] J. Setoain, C. Tenllado, J. I. Gómez, M. Arenaz, M. Prieto, and J. Touriño. 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.

[11] M. Wolfe. *High performance compilers for parallel computing*. Addison-Wesley, 1996.