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
Spain

Index

- Introduction: Motivation & Foundations
- New Kernel-based IR
  - Kernel-based Data Dependence Graph (K-DDG)
  - Kernel-based Control Flow Graph (K-CFG)
- Automatic Parallelization
  - Task Decomposition
- Conclusions & Future Work

Index

- **Introduction: Motivation & Foundations**
- New Kernel-based IR
  - Kernel-based Data Dependence Graph (K-DDG)
  - Kernel-based Control Flow Graph (K-CFG)
- Automatic Parallelization
  - Task Decomposition
- Conclusions & Future Work

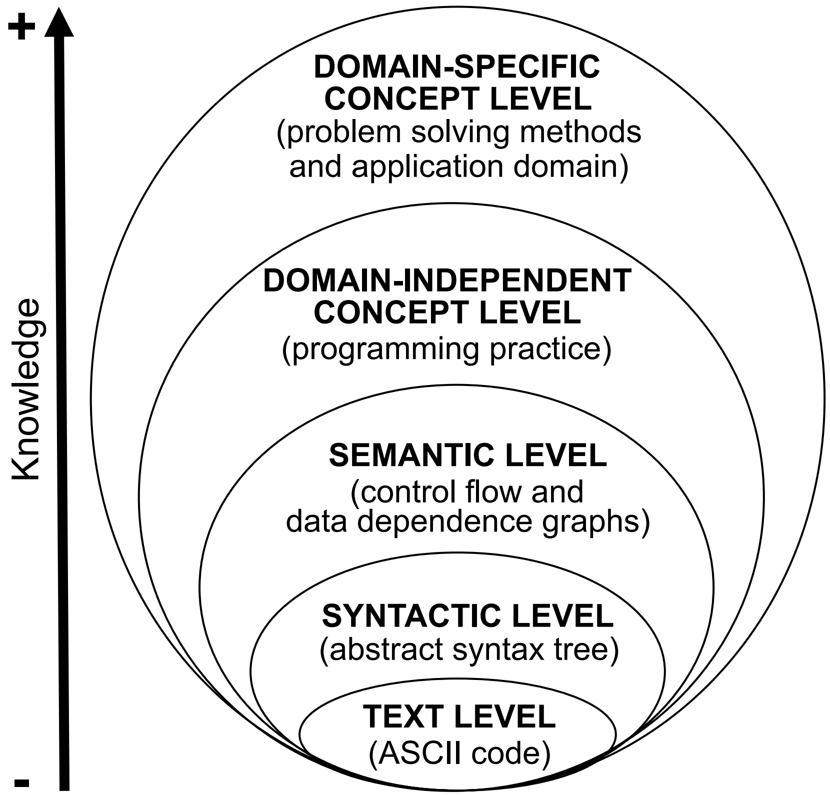
Motivation

- Increase in the number of cores available in commodity processors
- Make an efficient use of the computer architecture is a complex time-consuming task
- Current compiler technology based on ASTs does not expose the parallelism available in real applications
- We propose a new intermediate representation that exposes multiple levels of parallelism in whole programs

Case Study: EQUAKE

- An example of full-scale application is EQUAKE, from SPEC CPU2000
- Simulation of seismic waves in large, highly heterogeneous valleys
- Finite element method
  - Simulation phase
  - Time integration phase
- 70 % of execution time is consumed by smvp()

Domain Independent Computational Kernels



Reuse of platform-optimized parallel software libraries
Software re-engineering
Program synthesis

Induction variable substitution
Parallelizing transformations for inductions, reductions

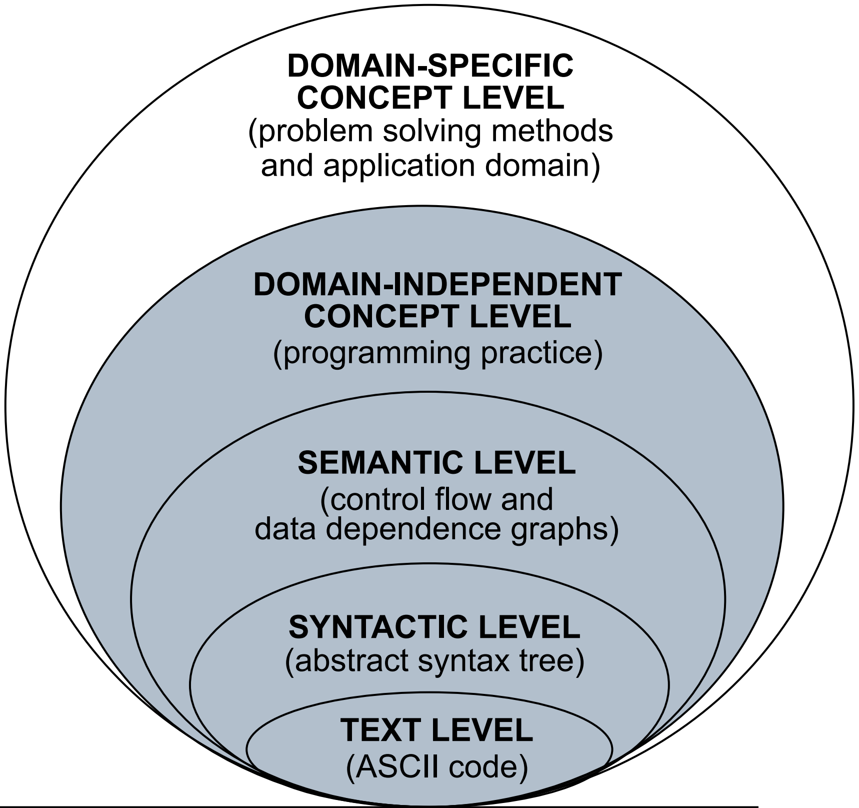
Constant propagation
Common subexpression elimination

Variable renaming

String matching & replacement

The XARK Compiler Framework

- General and extensible solution for automatic kernel recognition at the domain-independent concept level.
- Properties:
  - Completeness: scalars/arrays/pointers, ifs-endifs
  - Robustness: different versions of a kernel
  - Delocalization: statements spread over the source code
  - Uniqueness: one code, one kernel
  - Extensibility: user-defined kernels



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

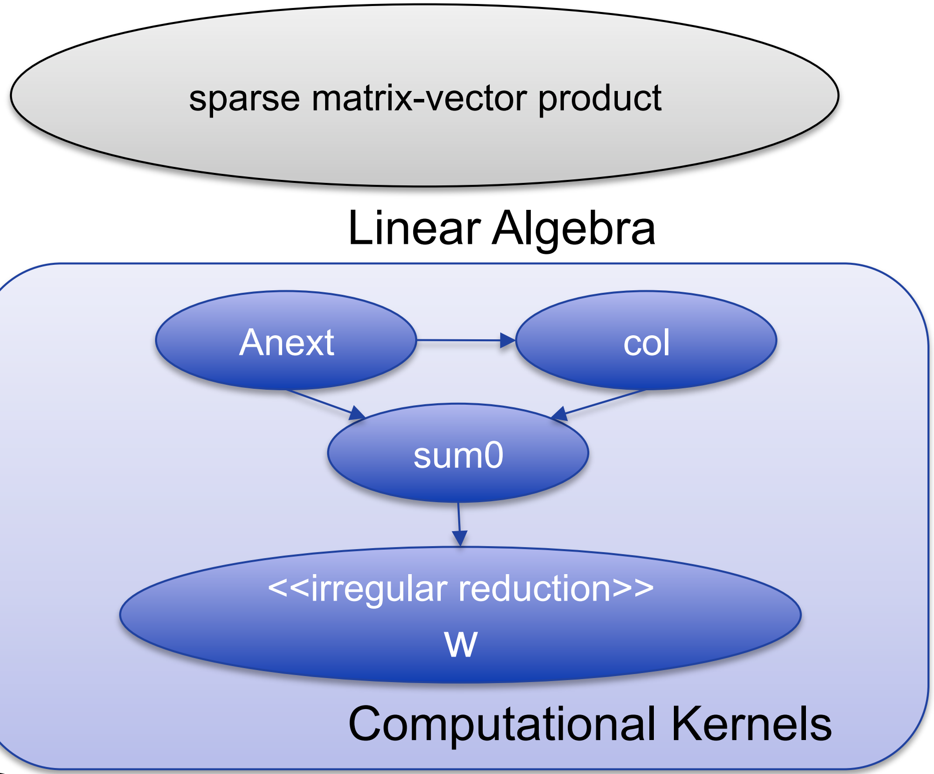
Recognition of smvp()

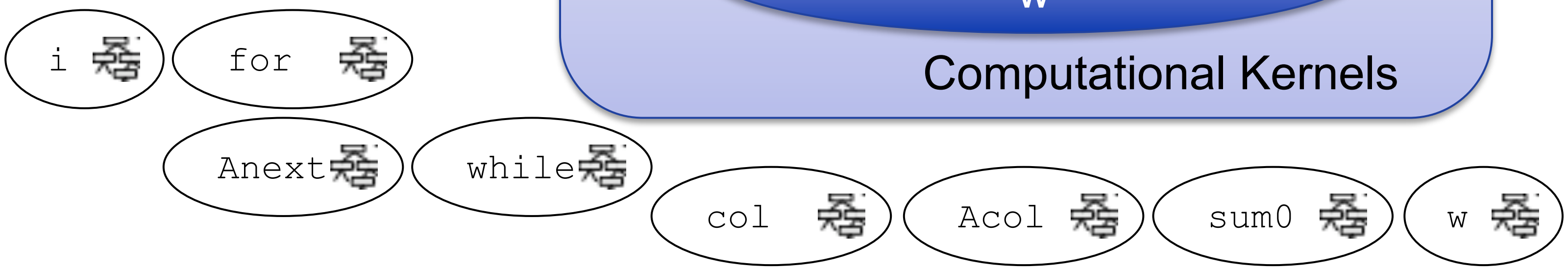
```
void smvp(int nodes, double ***A, int *Acol, int *Aindex, double **v, double **w) {
  int i, Anext, Alast, col; double sum0, sum1, sum2;

  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;
  }
}
```

Recognition of smvp()

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

Source code





Forest of ASTs + DDG + CFG

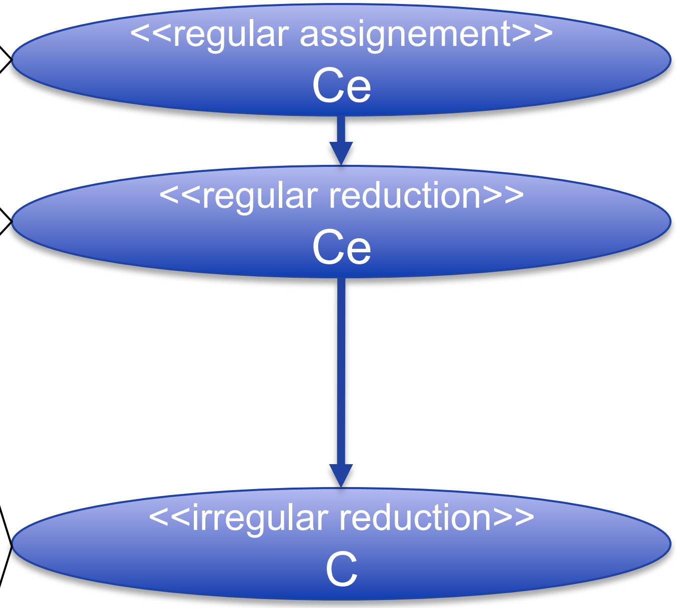
Index

- Introduction: Motivation & Foundations
- **New Kernel-based IR**
  - **Kernel-based Data Dependence Graph (K-DDG)**
  - **Kernel-based Control Flow Graph (K-CFG)**
- Automatic Parallelization
  - Task Decomposition
- Conclusions & Future Work

Kernel-based Data Dependence Graph (K-DDG)

- Provided by the XARK Compiler Framework

```
/* 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];
   }
}
```



Kernel-based Control Flow Graph (K-CFG)

- Similar to CFG
- Problem: establishment of flow dependences at the kernel level (dominance relationship)
- Two-phase construction
  - Group kernels into execution scopes
  - Search for flow dependences

Kernel-based Control Flow Graph (K-CFG)

- Execution scope
  - Similar to BB in CFG
  - Computed using the concept of region of a flow graph
  - The program is split into a hierarchy of loop regions that represent the execution scopes and kernels are attached to them
- All the sentences of a kernel belong to its execution scope

Algorithm 1 Computation of the execution scopes.

Input: K-DDG, CFG, DT

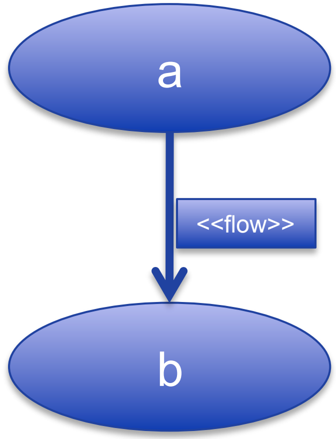
foreach kernel K in the K-DDG **do**
    bb_dom = basic block of CFG that contains a stmt of K (excluding μ-stmt)
    foreach statement $stmt$ in K **do**
        if $stmt$ is not a μ-statement **then**
            bb_stmt = basic block of CFG that contains $stmt$
            if bb_stmt dominates bb_dom **then**
                $bb\_dom = bb\_stmt$
            end if
        end if
    end for
    K.execution_scope = innermost enclosing loop region of bb_dom;
end for

Kernel-based Control Flow Graph (K-CFG)

Algorithm 2 Detection of kernel-level flow dependences.

Input: K-DDG, K-CFG, CFG, DT

foreach kernel-level dependence $K_1 \rightarrow K_2$ of the K-DDG **do**
  $R_1 = \text{execution\_scope}(K_1)$
  $R_2 = \text{execution\_scope}(K_2)$
  if (R_1.parent_reg $=$ R_2.parent_reg) & (R_1 precedes R_2 in the hierarchy) **then**
    mark $K_1 \rightarrow K_2$ as flow dependence
  else if $\forall s_1 \in K_1\ \exists s_2 \in K_2$ such that statements s_1 and s_2
      belong to the same basic block in the CFG
      and s_1 precedes s_2 in the DT **then**
    mark $K_1 \rightarrow K_2$ as flow dependence
  end if
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;
}
```

Index

- Introduction: Motivation & Foundations
- New Kernel-based IR
  - Kernel-based Data Dependence Graph (K-DDG)
  - Kernel-based Control Flow Graph (K-CFG)
- **Automatic Parallelization**
  - **Task Decomposition**
- Conclusions & Future Work

Automatic Parallelization

- Automatic parallelization
  - Detection of available parallelism
  - Decomposition and parallel code generation
- The kernel-based IR exposes multiple levels of parallelism
  - Intra-kernel parallelism: inside a kernel
    - Widely studied in compiler literature in 90s, specially irregular reductions
  - Inter-kernel parallelism: between kernels

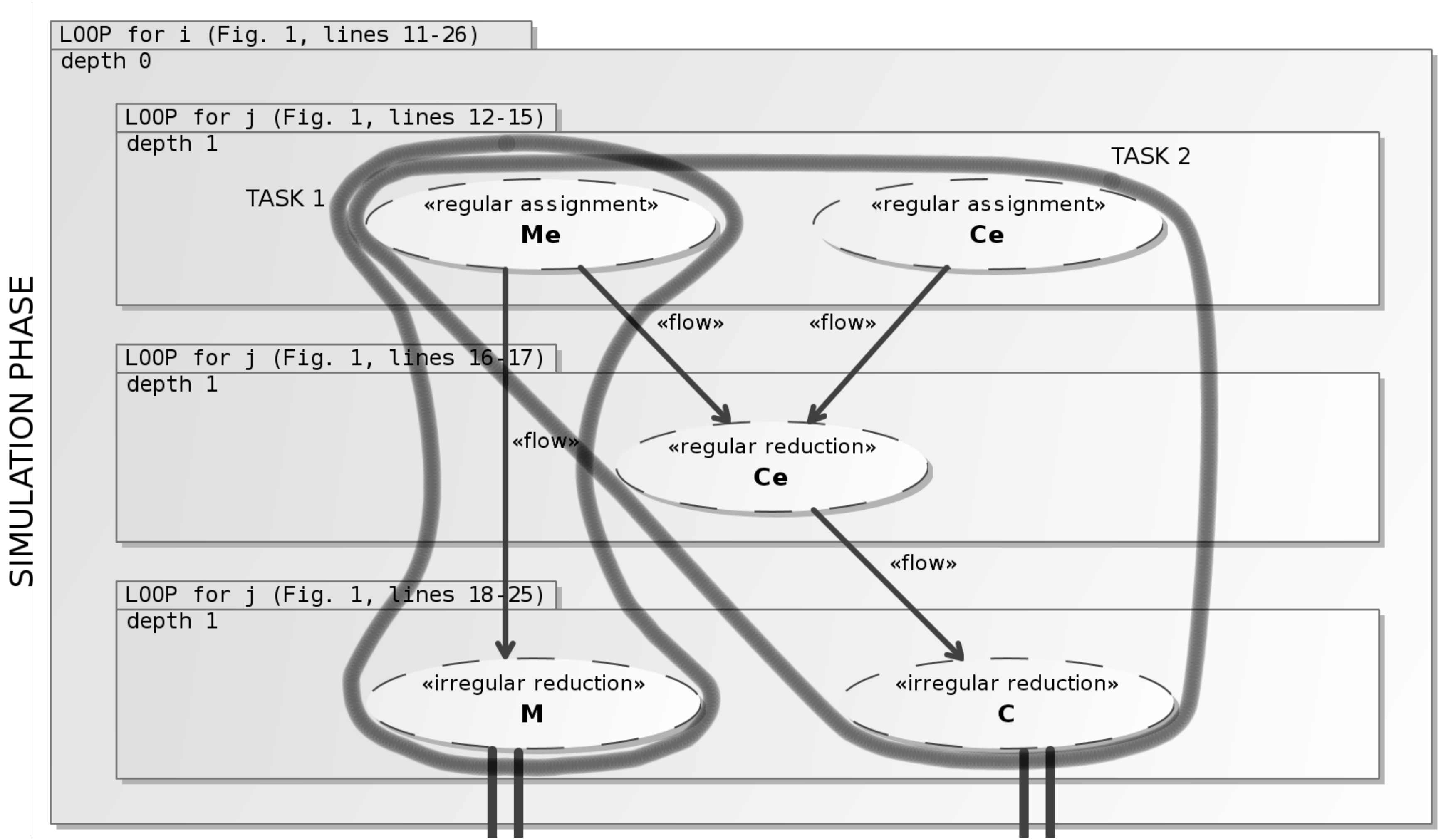
Task decomposition for multi-cores

Algorithm 3 Task decomposition for multi-core processors.

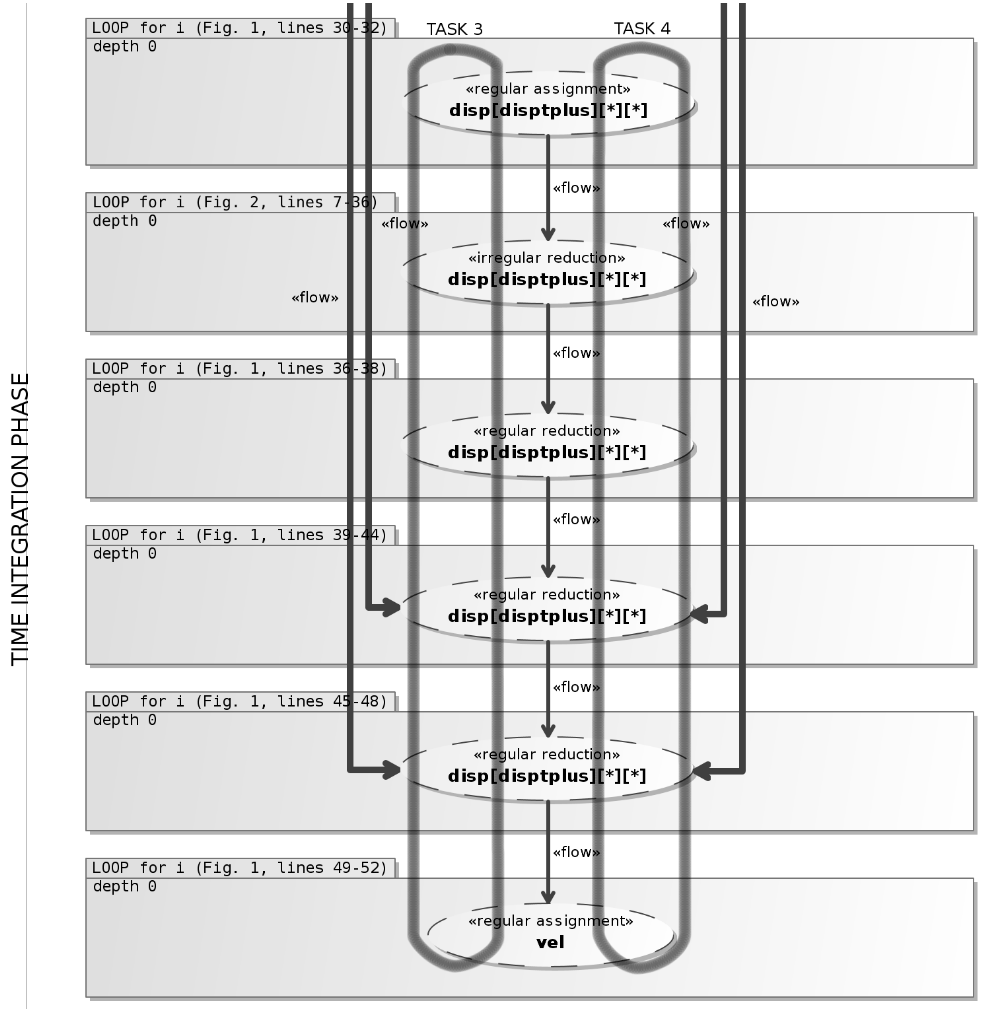
Input: K-DDG, K-CFG

merge execution scopes with one kernel and one cross-boundary edge
$d = 0$
foreach execution scope R at depth d in the K-CFG **do**
  if $\forall$ kernel $K \in R$ such that K is parallelizable **then**
    $n\_drain\_kernels$ = number of kernels without outgoing edges in K-DDG
      that cross the execution scope boundaries
    if $n\_drain\_kernels = P$ **then**
      $tasks$ = set of P drain kernels
    else if $n\_drain\_kernels < P$ **then**
      $tasks$ = split parallelizable drain kernels to create P tasks
    else
      $tasks$ = merge drain kernels to create P tasks
    end if
    map $tasks$ to different cores
  end if
  $d{+}{+}$
end for

Case Study: EQUAKE



Case Study: EQUAKE



Index

- Introduction: Motivation & Foundations
- New Kernel-based IR
  - Kernel-based Data Dependence Graph (K-DDG)
  - Kernel-based Control Flow Graph (K-CFG)
- Automatic Parallelization
  - Task Decomposition
- **Conclusions & Future Work**

Conclusions & Future Work

- Definition of a kernel-based IR
  - Exposes multiple levels of parallelism
  - Inspired by standard statement-based IRs
  - Framework for new whole program automatic parallelization techniques
- Work in progress
  - Port of XARK from Polaris to GCC
    - From F77 to C, C++, Fortran, Java…
  - XARK is built on top of GSA form
    - Inter-procedural GSA on top of GIMPLE-SSA

Conclusions & Future Work

- Work in progress
  - Give more knowledge to recognition engine
- Future Work
  - Improve the K-CFG construction algorithm
  - Run tests with well-known benchmark suites
  - Compare with existing auto-parallelization frameworks
  - Task decomposition for many-cores & GPUs

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
Spain