Kimble: a Hierarchical Intermediate Representation for Multi-Grain Parallelism

Nicolas BENOIT Stéphane LOUISE

CEA, LIST, Embedded Real Time Systems Laboratory,
Point Courrier 94, Gif-sur-Yvette, F-91191 France
{firstname.lastname}@cea.fr

Abstract

Because modern computer architectures expose multiple levels of parallelism, the problem of mapping program parts to specific resources raises several trade-offs. Those are easier to arbitrate at a level of abstraction where a wide range of software granularities and information are still available. The role of the intermediate representation in this process is two-fold. On one hand, it has to support the characterization and expression of parallelism at the relevant granularities. On the other hand, it has to facilitate code transformation and generation for possibly heterogeneous processing elements.

This paper introduces Kimble, an intermediate representation where program constructs are organized in a hierarchy of directed graphs. After a presentation of its structure and its building process, its applications and implementation in a source-to-source transforming tool built on top of GCC are described.

Keywords hierarchical, dependency, parallelism, granularity

1. Introduction

Emerging massively parallel architectures [4, 7, 8, 18] feature numerous and possibly heterogeneous Processing Elements (PEs). It has become common to draw multiple levels of parallelism within a single chip: the PEs clusters level, the PEs level, the thread level (e.g. SMT), the data level (e.g. SIMD), the instruction level (e.g. VLIW or EPIC), etc.

On the programming side, portability concerns and code maintenance costs often make the development of target-specific implementations an unsatisfactory option. Hence, high-level abstractions to express parallelism have flourished, stressing compilers abilities to bridge the gap between software and hardware levels.

Consequently, when a compiler maps a parallel program onto a parallel architecture, it encounters several trade-offs, e.g. the task- versus data-parallelism trade-off [16]. To arbitrate among them, compilers must gather information and perform code transformation at a wide range of granularities. That is why the aggregation of local information must be centralized to a single point in the compilation flow where global mapping decisions can be taken.

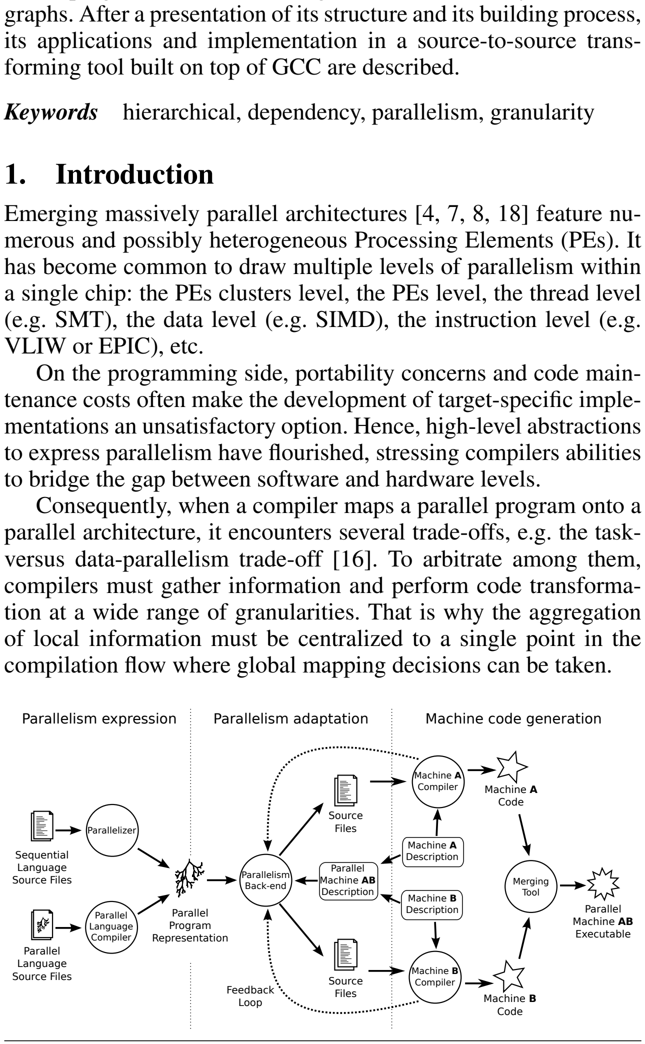


Figure 1. Integration of parallelism within compilation flow.

Figure 1 shows an abstract compilation flow where the high-level software parallelism is brought to the hardware level by a compiler's component named *parallelism back-end*. The parallelism expressed by the programmer and detected by the compiler are merged within a single Intermediate Representation. This representation is used as the raw material for code transformation and adaptation to a parallel architecture named AB. A part of the input program is mapped onto a subset A of the PEs, and another part onto a subset B.

In this paper, we present Kimble, a high-level inter-procedural Intermediate Representation designed to meet the following requirements:

1. Expose multiple granularities of code.
2. Describe dependencies among program constructs.
3. Describe data parallelism.
4. Enable source-to-source compilation.
5. Ease program characterization and analysis.

Kimble comes in the form of a hierarchy of Directed-Acyclic-Graphs (DAGs) which describes dependence relationships among program constructs (first and second criteria). Kimble objects map high-level program constructs such as loops, branches, functions, etc. (fourth criteria). Loops are annotated with the type of dependence exposed among iterations (third criteria). Aggregation of information is allowed with the design of methods to aggregate individual characteristics along the hierarchy (fifth criteria).

Kimble has been implemented in a source-to-source transforming tool built on top of GCC that has the role of the parallelism back-end of 1.

The remainder of the paper is organized as follow: section 2 presents the related work, section 3 describes Kimble structure, section 4 explains its building process, and section 5 gives potential applications and presents its usage in our GCC extension.

2. Related Work

In order to support parallelism expression and adaptation, several proposals of IRs have already been formulated.

Hierarchical dependence graphs were introduced in [17], where loop nests and their data-dependencies were hierarchically represented. The authors highlighted the benefits of this representation regarding reordering transformations. However, control dependence were not considered.

The Program Dependence Graph described in [5] is a proposal of data and control dependence unification into a program representation. Within a PDG, nodes are statements or code regions, and edges are annotated with the data values and control conditions on which a node's execution depends. The authors show how this representation simplifies the writing of optimization passes and parallel program partitioning. In this regard, it is mentioned to be used for vectorization and a number of code transformations such as code motion and loop fusion. The authors propose to extend the

PDG with a hierarchical organization, it is not described in-depth though.

The Hierarchical Task Graph presented in [6, 14] aims at providing parallelizing compilers with an IR that describes parallelism at all levels of granularity. An HTG contains three types of nodes: simple tasks with no subtasks, compound tasks which embed subtasks in an HTG and looping tasks which embed the iteration body as an HTG. Kimble graphs differ mostly from HTGs on the following points:

- Two Kimble constructs (i.e. function calls) are allowed to share the same subgraph (i.e. the function called).
- Conditions are detached from basic-blocks which become regions (aggregate of control-neutral code).
- The parallelism within those regions is captured, thus addressing near instruction-level parallelism.

Those differences impact program transformation, source code generation and scheduling, and are believed to simplify them.

A hierarchical graph representation for DSP programs is proposed in [9]. This structure is exploited during the scheduling phase, as it allows to decompose bottlenecks into nodes with a finer granularity. A similar representation is used within MAGELLAN [3], a partitioning and scheduling heuristic for hardware-software codesign. Both proposals are scheduling-oriented, and thus do not address program transformation nor source code generation.

The OSCAR Compiler [10] integrates a multi-grain parallelizer based on the macro-dataflow [12] program representation. The macro-dataflow is a directed hierarchical graph which describes macro-tasks and their data dependencies at various levels of granularity: subroutine calls, repetitive blocks and statements. The compiler schedules the graph on the target architecture and uses a high level API to abstract hardware mechanisms. To the contrary of Kimble, the macro-dataflow graph does not map directly to program constructs. Code transformation and source code generation are therefore not as much facilitated.

3. Presentation of Kimble IR

This section presents the structure of the Kimble IR and the information it contains.

3.1 Structure

The structure of the Kimble IR can be described as a hierarchy of Directed-Acyclic-Graphs (DAGs). More formally, it is a directed graph G defined by an ordered triple

$$G = (N, D, E) \quad (1)$$

where N is a set of nodes corresponding to program constructs, D is a set of ordered pairs of nodes expressing a dependency between two nodes and E is a set of ordered pairs of nodes expressing a nesting (or hierarchical) relationship.

Considered without E, the pair (N, D) defines a graph G' containing multiple disconnected DAGs, where edges express a control or data dependence. It is assumed that each of those DAGs has a single source node, i.e. a node n such as $indegree(n) = 0$.

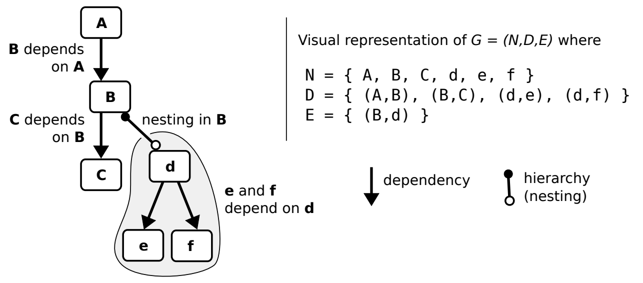
The elements of E are used to connect the DAGs of G' and build a hierarchy, so that multiple levels of granularity can be described. For example, a pair (n, m) of E expresses that the DAG with source node m is nested within n. While dependence edges form a DAG, nesting relationships can form cycles in order to express recursive flows.

Figure 2 is a visual representation of a Kimble graph. Two DAGs respectively containing nodes A, B, C and d, e, f are hierarchically connected.

For convenience, the source node of the DAG at the coarsest level of granularity is named the *root* of the Kimble graph.

3.2 Supported Programs

A program and its CFG must satisfy the following constraints in order to be describable under Kimble IR:



Visual representation of G = *(N,D,E)* where

```
N = { A, B, C, d, e, f }
D = { (A,B), (B,C), (d,e), (d,f) }
E = { (B,d) }
```

Figure 2. Kimble graph example.

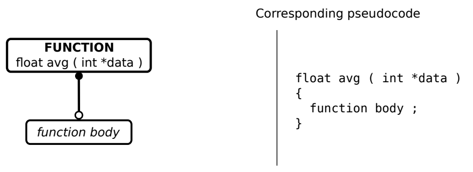
1. no indirect function calls, e.g. by means of pointers;
2. no gotos and labels unless they satisfy CFG constraints;
3. within a CFG cycle, the edges exiting the cycle have the same target (workaround described in Figure 1 of [6]);
4. basic-blocks with no successor contain a return or a call to a never returning function such as *exit*;

3.3 Program Constructs

Each node in a Kimble graph has a type which corresponds to the kind of program construct it expresses. It can be one of the following: function, loop, region, cluster, statement, guard or function call.

3.3.1 Function

When defined, a function is described with a node which embeds a subgraph made of loops, regions and guards. If the function body is not accessible, e.g. if it is defined in an external library, the node has no subgraph. Figure 3 shows a representation of a function node.



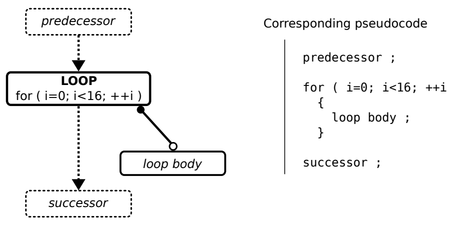
Corresponding pseudocode

```
float avg ( int *data )
{
  function body ;
}
```

Figure 3. Function node with its subgraph node.

3.3.2 Loop

A loop node always embeds a subgraph of loops, regions and guards. It can be annotated with the type of parallelism it supports: *undividable*, *map* (all iterations are independent) or *reduce* (iterations expose a reduction dependence scheme). Figure 4 shows a representation of a loop node.



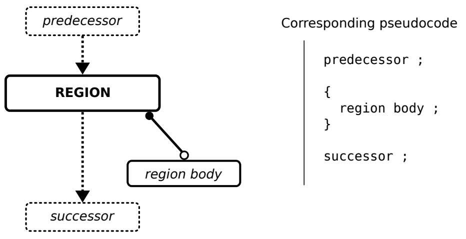
Corresponding pseudocode

```
predecessor ;

for ( i=0; i<16; ++i )
  {
    loop body ;
  }

successor ;
```

Figure 4. Loop node with its subgraph node.

3.3.3 Region

A region embeds a subgraph made of clusters, statements and function calls. In other words, it is a basic-block from which the branch has been extracted, as described in [5]. Figure 5 shows a representation of a region node.

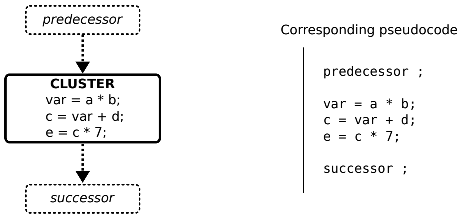


```
predecessor ;

{
  region body ;
}

successor ;
```

Figure 5. Region node with its subgraph node.

3.3.4 Cluster

A cluster corresponds to a sequence of statements that can not be broken apart. It is allowed to be empty, so that it can be used to as a source or sink node in the directed graph structure. A cluster node is never linked to a subgraph. Figure 6 shows a representation of a cluster node.

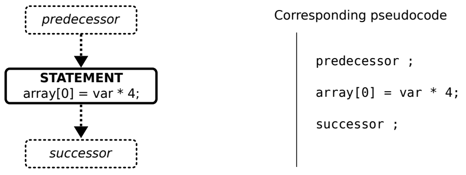


```
predecessor ;

var = a * b;
c = var + d;
e = c * 7;

successor ;
```

Figure 6. Cluster node.

3.3.5 Statement

A statement is a three-address code instruction, an operation with a reduced number of operands: one destination and two sources at most. It is never linked to a subgraph. Figure 7 shows a representation of a statement node.

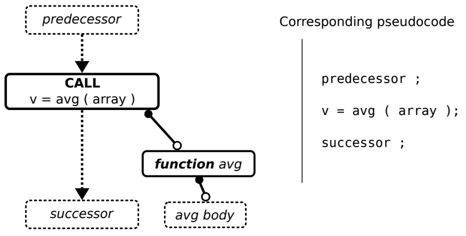


```
predecessor ;

array[0] = var * 4;

successor ;
```

Figure 7. Statement node.

3.3.6 Function Call

A function call is a node which is linked to a subgraph made of its target function node. Function calls are excluded from clusters in order to ease their manipulation and the access to their subgraph. It should be noted that a call can be linked to the function node it belongs to, thus expressing recursion. Figure 8 shows a representation of a function call node.

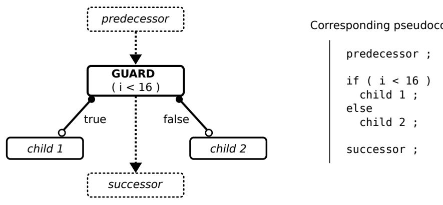


```
predecessor ;

v = avg ( array );

successor ;
```

Figure 8. Call node with its target function node.

3.3.7 Guard

A guard describes a conditional branch, its subgraphs correspond to the case where its evaluation returns *true* or *false*. Complex conditional paths are built with this elementary construct. Figure 9 shows a representation of a guard node with children branches.



```
predecessor ;

if ( i < 16 )
  child 1 ;
else
  child 2 ;

successor ;
```

Figure 9. Guard node with two children branches.

3.4 Program Constructs Hierarchy

In order to be consistent, the hierarchy of program constructs is constrained. For example, a loop node can not have a subgraph which contains a function node. Figure 10 summarizes the types of nodes allowed in each node's subgraph(s).

$$\begin{array}{lll}
Function & \rightarrow & \{\emptyset|Loop|Region|Guard\} \\
Loop & \rightarrow & \{Loop|Region|Guard\} \\
Region & \rightarrow & \{Cluster|Statement|Call\} \\
Cluster & \rightarrow & \{\emptyset\} \\
Statement & \rightarrow & \{\emptyset\} \\
Call & \rightarrow & \{Function\} \\
Guard & \rightarrow & True\{Loop|Region|\emptyset\}; False\{Loop|Region|\emptyset\}
\end{array}$$

Figure 10. Hierarchy of program constructs.

3.5 Kimble Graph Traversal

Traversing a Kimble graph requires to select two orderings. First, a granularity-related ordering to specify if the coarser levels should be visited first or last. Second, a dependency-related ordering to specify if the traversal of a DAG should happen in a topological order or reversed.

To illustrate the traversal, we propose a recursive method summarized by algorithm 1 which applies a function *f* to each node. This method operates in a bottom-up fashion on the granularity side, and in a topological order on the dependency side. In other words, a node is not processed while its predecessors have not been visited, it is never visited twice, and its subgraphs are visited first. The set *V* is used to store the visited nodes.

Algorithm 1 Method traverse (node *n*, function *f*, set *V*)

```
1: /* do not process a node twice */
2: if $n \in V$ then
3:     return
4: end if
5: /* constraint topological order */
6: for all $m$ in $n$.predecessors do
7:     if $m \notin V$ then
8:         return
9:     end if
10: end for
11: /* process subgraphs */
12: for all $sg$ in $n$.subgraphs do
13:     traverse ( $sg, f, V$ )
14: end for
15: /* process current node */
16: $f$ ( $n$ )
17: add $n$ to $V$
18: /* process successors */
19: for all $m$ in $n$.successors do
20:     traverse ( $m, f, V$ )
21: end for
```

4. Construction of Kimble IR

In the proposed compilation flow of Figure 1, the Kimble IR is shown to be available between the parallelism expression and extraction tools and the parallelism back-end. This section presents how it can be built and annotated at this stage of the compilation flow.

4.1 Step 1: Identifying Program Constructs

The first step towards the building of a Kimble graph is the identification of program constructs. It is highly dependent of the level of abstraction at which the construction process occurs.

4.1.1 From an Abstract Syntax Tree

The program constructs available in Kimble can easily be matched with high-level programming language constructs as long as the constraints of subsection 3.2 are satisfied. However, statements are required to be in a three-address code representation. Transforming AST statements to three-address code is straightforward and only requires the insertion of intermediate variables.

In the case where the language offers program constructs which are not directly available in Kimble, they must be either converted to available constructs or implemented as extensions.

To illustrate the conversion, we may consider the *switch* construct available in C. Within Kimble, such construct can be mapped with a set of guards. In case the control flow is complicated by multiple-entries cases (e.g. some cases have no *break* statement), the code can either be duplicated for each distinct case, or the guards expressions extended to cover multiple cases.

4.1.2 From a Three-Address Code IR with CFG

In the case where the AST has already been converted to a three-address code representation (e.g. SSA form) and a CFG computed, the required tasks consist in checking the constraints described in subsection 3.2, identifying loops, and then identifying regions and guards.

The identification of loops can be eased if the compiler has already built a list of basic-blocks for each loop. This is the case in GCC for example. Otherwise, a natural loop discovery pass from the CFG is required. The construction of a loop node in Kimble requires the identification of the exit conditions. Extracted from the CFG, loops have a *do* { ... } *while ( );* meaning. It is possible to convert them to a *for ( )* { ... } meaning, by first identifying an induction variable and a single exit condition, and second by checking that moving the exit condition before the loop body does not break the original execution flow.

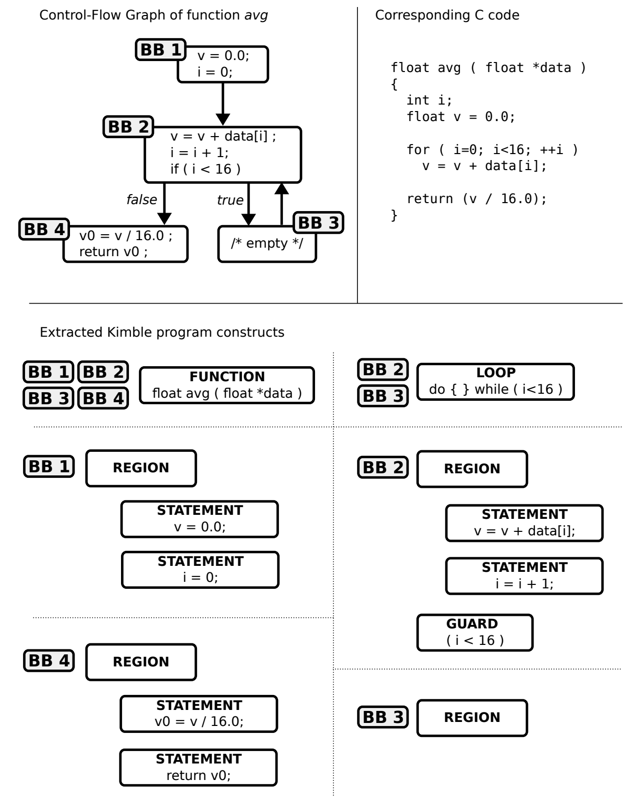
Once the list of basic-blocks belonging to a loop have been discovered, regions and guards can be built. Regions correspond to basic-blocks from which the condition has been extracted. Guards are built for basic-blocks with multiple exits. Dominance information are used to tell which blocks are guarded and which blocks are executed no matter what conditional path is taken.

Figure 11 shows an example of a set of program constructs extracted from a CFG.

4.2 Step 2: Building of a Hierarchy

Once program constructs have been identified, they can be hierarchically linked together and the content of dependency graphs established. In the case where the Kimble IR is built from an AST, the hierarchical structure of the AST can be directly used. Below, we describe how the hierarchy can be built from a CFG. This process occurs from finest levels of granularity to coarser ones.

- Function calls subgraphs correspond to the function node of their target.
- Guards subgraphs are reconstructed using dominance information of the CFG.
- Regions subgraphs contain the statements of the basic-blocks it maps, without the conditional.
- Loops subgraphs contain the inner-loops, the regions and the guards built from the basic-blocks.
- Functions subgraphs contain the regions that do not belong to a loop and the loops which have no outer loop. They are obtained by traversing their respective CFG.



```
float avg ( float *data )
{
  int i;
  float v = 0.0;

  for ( i=0; i<16; ++i )
    v = v + data[i];

  return (v / 16.0);
}
```

Figure 11. Program constructs identification.

4.3 Step 3: Taking Dependencies into Account

At this point of the Kimble graph building, a set of program constructs have been extracted and assigned to multiple DAGs. Now, for each DAG, program constructs must be linked to each other according to their dependencies.

4.3.1 Initial State of the DAGs

When operating from an AST, the control information is given by the ordering of the nodes in the AST.

When operating from a CFG, a topological sort is performed to extract the ordering of the basic-blocks. This allows to sequence loops, regions and guards, and to restore part of the control-dependencies. Within regions, statements are sequenced according to their order in the enclosing basic-block.

Figure 12 shows the state of the Kimble graph of the *avg* function described previously at this point of the process.

4.3.2 Dependencies Matrix

For each DAG, a matrix named *dependencies matrix* is built and used to keep track of the relationship between program constructs. It is also used to transform and annotate the DAGs in a process operating from the finer levels of granularities to the coarsest.

A dependencies matrix assigns one line and one row to each node, e.g. a 4×4 matrix is used for a DAG with four nodes. The ordering of the nodes corresponds to their current order in the DAG. Each element of the matrix points to a dependence description structure.

Data-dependencies information may come from different sources: data-flow analysis, parallelism extraction tool, annotations at the source-code level.

Control-dependencies which have not been captured by the guarding constructs are also represented in dependencies matrices. It mainly concerns program constructs which affect the execution flow such as *return* statements, which are always the last to be executed at a given level of hierarchy.

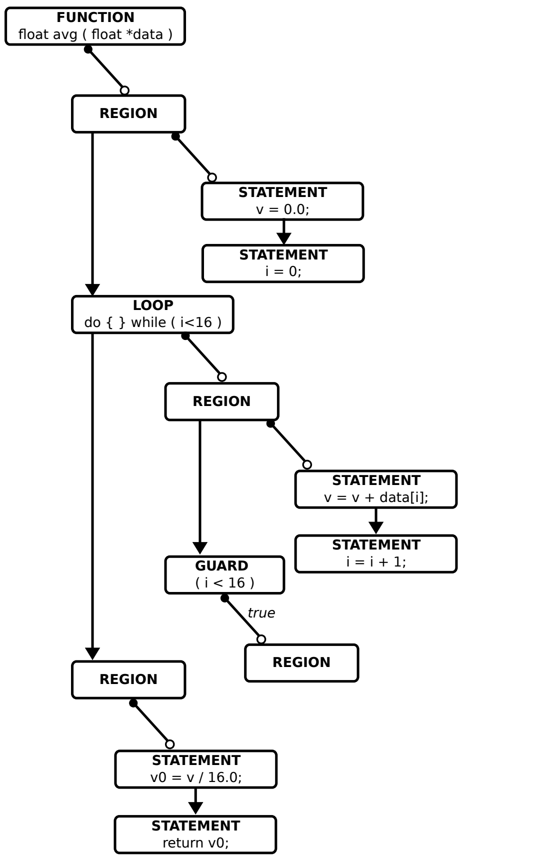


Figure 12. Kimble graph with serialized DAGs.

Besides control and data dependencies, a special kind of dependency named *indirect* is introduced. It is used to tell that if there is a dependency between N and M and between M and O, then N indirectly depends on O.

Given a dependencies matrix counting N nodes, algorithm 2 links nodes according to their dependencies. Symbolic values are used to represent the different types of dependencies between two nodes. If you consider nodes M and N and their respective indexes m and n in the matrix, then the value $matrix[n][m]$ has the following meaning:

- I means there is an indirect dependency between M and N
- $\emptyset$ means there is no dependency between M and N
- C means N is control-dependent of M
- D means N is data-dependent of M

An empty node (a region or a cluster according to the hierarchy level) is created to be the *source* node of the DAG. Algorithm 2 makes sure that besides *source* there is no node without a predecessor.

Figure 13 gives two example of dependency matrices at two different levels in the hierarchy of the *avg* function defined above.

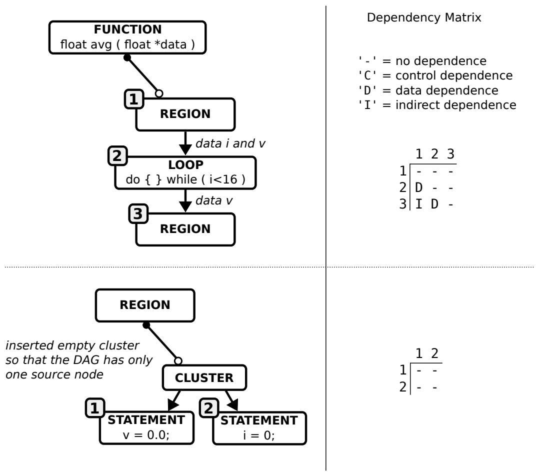
4.4 Step 4: Cleaning

After the update of the DAGs, a Kimble graph may contain empty nodes or structures that can be simplified:

- empty source nodes which have only one successor can be removed;
- empty regions can be removed;
- sequence of statements can be aggregated into clusters.

Algorithm 2 DAG update from a dependencies matrix of N nodes

```
Require: detachment of nodes
Require: creation of an empty source node
 1: for $n \leftarrow 1, N$ do
 2:     /* update indirect dependencies */
 3:     for $m \leftarrow 1, (n-1)$ do
 4:         if $matrix[n][m] = \emptyset$ or $matrix[n][m] = I$ then
 5:             continue
 6:         end if
 7:         for $o \leftarrow 1, (m-1)$ do
 8:             if $matrix[m][k] \neq \emptyset$ then
 9:                 $matrix[n][k] \leftarrow I$
10:             end if
11:         end for
12:     end for
13:     /* link nodes if they expose a (direct) dependency */
14:     for $m \leftarrow 1, (n-1)$ do
15:         if $matrix[n][m] = C$ or $matrix[n][m] = D$ then
16:             link_predecessor_and_successor ( $m$, $n$ )
17:         end if
18:     end for
19:     /* link DAG source to $n$ if $n$ has no predecessor */
20:     if $nb\_predecessors(n) = 0$ then
21:         link_predecessor_and_successor ( source, $n$ )
22:     end if
23: end for
```



	1	2	3
1	-	-	-
2	D	-	-
3	I	D	-

	1	2
1	-	-
2	-	-

Figure 13. Kimble dependencies matrix examples.

4.5 Step 5: Characterization

In order to hint parallelism mapping decisions, several information can be collected and used to decorate the nodes of a Kimble graph.

It includes for example the number and types operations, the volume of data written, etc. Such information are collected and aggregated from the finer-levels of granularity to the coarsest, so that the root of Kimble graph shall represent a characterization of the whole program. While aggregating the information bottom-up, two sources of precision loss may appear: the potential irregular behavior exposed by loops and guards, and the use of external functions. To address this issue, profiling data and source-code annotations seem to the most reliable approach.

Loop nodes can be further characterized by means of annotations:

- *mapping* when all iterations are independent of each other.
- *reduction* when iterations expose a write-after-read dependency on a single scalar variable.

Information regarding iteration range and reduction variable are attached to the nodes so they can be re-used at scheduling and code generation time. They can be obtained from source-code annotations such as OpenMP [13] or code analysis, e.g. the Graphite framework [15] within GCC. Our Kimble-enabled extension of GCC complements these with an inter-procedural data-dependence analysis based on a partial interpretation of the input program.

4.5.1 Hierarchical Aggregation of a Data-Parallelism Metric

In order to aggregate hierarchically the data-parallelism information delivered by the annotation of parallel loops, we propose to label each node n with a cost K_n and a ratio R_n.

As costs are derived from the characteristics gathered previously, the aggregation focuses on computing ratios. Those have the same meaning as the commonly used Amdahl parallelism ratio [1] and reflect the proportion of the cost which is divided by a parallel execution.

Nodes with no subgraph are assigned a parallelism ratio of zero. Loop nodes have their parallelism ratio estimated according to their characteristics, e.g. ideal parallel loops have a ratio of 1. Then, each node s with a subgraph S has its ratio R_s given by:

$$R_s = \frac{\sum_{n \in S} K_n \times R_n}{K_S} \quad (2)$$

where K_s is the aggregation of the costs within the subgraph S:

$$K_S = \sum_{n \in S} K_n \quad (3)$$

To compute the cost and ratio of a guard node exposing two subgraphs, we propose to combine each subgraph values using a weighted sum according to the branches probability.

4.5.2 Hierarchical Aggregation of a Task-Parallelism Metric

Within a DAG, we define the task-parallelism metric as the number of nodes that can be executed concurrently. It can be expressed as the cardinality of the maximum antichains within the graph. However, as Figure 14 shows, summing the concurrency metric of the nodes belonging to the maximum antichain raises the problem of selecting the appropriate maximum antichain. A possible approach is to select the maximum antichain which exposes the highest concurrency metric, at the expense of enumerating and comparing all maximum antichains.

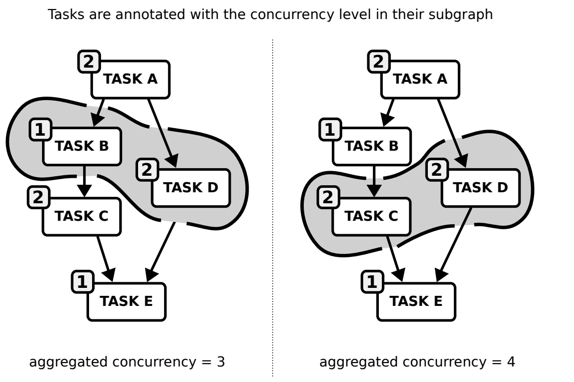

Tasks are annotated with the concurrency level in their subgraph
2 TASK A
1 TASK B
2 TASK D
2 TASK C
1 TASK E
aggregated concurrency = 3
2 TASK A
1 TASK B
2 TASK D
2 TASK C
1 TASK E
aggregated concurrency = 4

Figure 14. Concurrency metric aggregation using maximum antichains.

An alternative approach can consist in coloring the interference graph of the DAG being characterized, as shown in Figure 15. For each set of nodes of the same color, the maximum concurrency metric is kept and selected values are summed. However, as two valid coloring schemes may return different aggregated values, it may be required to determine the coloring scheme which maximizes the concurrency metric.

To arbitrate between those techniques which both deal with NP hard problems, one should take into consideration the following observation. In the proposed example, if task B takes much longer to execute than task D, then there is little chance of executing tasks

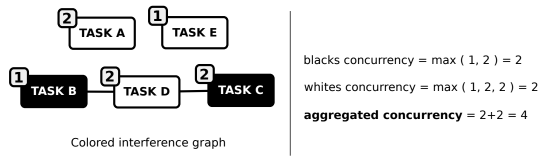

2 TASK A
1 TASK E
1 TASK B
2 TASK D
2 TASK C
Colored interference graph
blacks concurrency = max (1, 2) = 2
whites concurrency = max (1, 2, 2) = 2
aggregated concurrency = 2+2 = 4

Figure 15. Concurrency metric aggregation using interference graph coloring.

C and D concurrently. Consequently, a more precise aggregated concurrency metric in this context would be 3.

As execution costs can easily be taken into account with the coloring based technique by updating the interference graph, we advise to use it in favor of the maximum antichains based one.

5. Applications and Implementation of Kimble

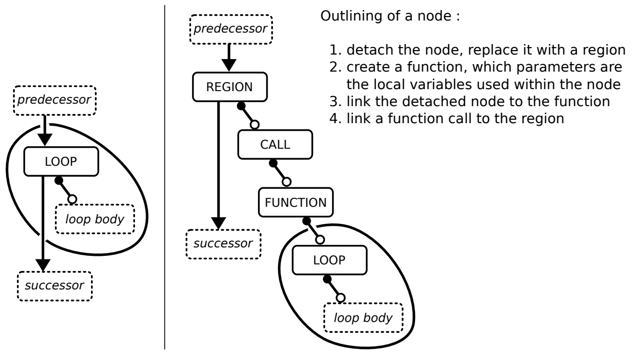
This section describes various applications of Kimble IR and its usage in a source-to-source transforming tool built on top of GCC.

5.1 Visual Representation

The hierarchical organization of Kimble graphs offers a convenient way of visualizing and exploring complex programs. By folding and unfolding hierarchy levels, the user can focus on a given granularity level. Besides, annotations can be used to enhance the visualization, for example by highlighting parallel loops. To illustrate this, a set of Kimble graphs is available upon email request to the authors.

5.2 Program Transformation

The hierarchical organization of the Kimble IR makes it a good candidate for reordering transformations, functions inlining and outlining, etc. For example, Figure 16 shows the result of the outlining of a loop node. Such operation requires to collect the



Outlining of a node :

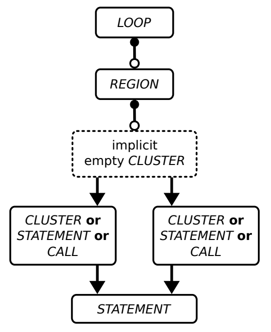
1. detach the node, replace it with a region
2. create a function, which parameters are the local variables used within the node
3. link the detached node to the function
4. link a function call to the region

Figure 16. Outlining in Kimble IR.

local variables used within the outlined node, those will be the parameters of the new function. Once this is done, detachment and insertion of new constructs in the graph is straightforward and immediately reflects the new structure of the code.

Similarly, Kimble organization benefits to code insertion at compile-time, e.g. for instrumentation purpose. Indeed, it is simplified as complex CFG constructs can be built with Kimble nodes in a way that is closer to high-level source code.

In addition, the typing of Kimble nodes facilitates the expression of pattern matching rules and their implementation. For example, the recognition of candidates for loop fission could be expressed by a single rule illustrated by Figure 17. Such rule, applied to each function, would return all the loops containing a single region, itself containing two independent cluster, statement or call nodes followed by a single statement (the expected loop induction which would be checked afterward).



Loop fission candidates by pattern matching

```
L : R : ( ((T+S+C) | (T+S+C)) ; S )
```

'L' corresponds to a Loop node
'R' corresponds to a Region node
'T' corresponds to a Cluster node
'S' corresponds to a Statement node
'C' corresponds to a Call node

':' expresses hierarchical nesting
'|' expresses independency
';' expresses dependency
'+' expresses logical OR

Figure 17. Example of pattern matching rule.

5.3 Scheduling

A Kimble graph being a hierarchy of DAGs, it can easily be used as the raw-material to perform program scheduling.

For instance, techniques for scheduling a hierarchy of DAGs are proposed in [3, 9]. Though the chosen representations differ from Kimble as program constructs are not described exhaustively, they rely on node decomposition heuristics that can be applied to Kimble graphs.

The *auto-scheduling* technique presented in [14] is also relevant for Kimble graphs. In this technique, characteristics derived from the hierarchical IR are used to hint dynamic scheduling directives inserted by the compiler at code generation.

Another example of hierarchical scheduling can be found in [11] where the emphasis is put on the memory footprint of the execution.

5.4 Source Code Generation

While the information retained in the Kimble IR are not detailed enough to provide clean and easily human readable sources, its program constructs can be expressed with a wide range of high-level programming languages. Therefore, source code generation from Kimble IR to the destination of compilation tools can be considered. Similarly to program transformation, pattern matching can be used to simplify and improve the output.

5.5 Implementation in a Source-to-Source Transforming Tool

The Kimble IR is currently implemented within an experimental extension of GCC - named Gomet [2] - performing code generation for heterogeneous architectures.

Intervening during the optimization passes sequence of GCC, Gomet benefits from high-level optimizations such as Static-Single-Assignment (SSA) form and Graphite loop transformations [15]. The first one simplifies scalar data dependencies by preventing multiple assignments of the same variable. The second one reduces the number of loop-carried dependencies. This allows Gomet to assume that parallelism at the statement and loop levels is exposed and requires no further transformations.

In order to extract parallelism at coarser levels, an inter-procedural data-dependence analysis based on a partial interpretation of the program is performed. The raw material of this analysis is the serial Kimble representation obtained after the step described in subsection 4.3.1. Pragmas inserted within the source code indicate to the interpretation process the entry point and the initial context of the code portions to analyze. As Kimble representation follows the three-address code format, the interpretation work is simplified and deals mostly with arithmetic expressions. Memory operations such as pointer dereferencing and array indexing are supported by using a virtual address space manager. Functions that are part of external libraries can be emulated as well by hardcoding their behavior within the interpreter. For example, our implementation emulates the behavior of dynamic memory management functions, e.g. *malloc()* and *free()*. The collected information about memory accesses are used to infer data-dependencies and build dependencies matrices.

After an update of the Kimble graph according to the dependencies matrices, the characterization process takes place. It is similar to the process described in subsection 4.5, cost metrics are computed according to the target architecture description. Once the graph is ready, it is sent to a dedicated scheduling tool that takes advantage of the task-parallelism and data-parallelism metrics to reduce the mapping optimization space. The result is a set of Kimble graphs which correspond to the code assigned to each to processing element. Synchronization and communication directives added by the scheduler are converted by Gomet to the available architectural mechanisms and inserted as Kimble constructs. Finally, each Kimble graph is unparsed to a C source file in function of the processing element it is bound to.

6. Conclusion

The unified representation of multiple levels of code granularities is critical to address the multiple levels of parallelism exposed by modern computer architectures. In this paper, we presented Kimble, a hierarchical Intermediate Representation designed for program analysis and mapping that is implemented in a source-to-source transforming tool built on top of GCC. Within this IR, a program is represented with a set of nested high-level constructs. At each level of granularity, constructs are linked to each other according to their dependencies. Program characteristics are aggregated in a bottom-up fashion, allowing a graph traversal to make decisions without having to inspect constructs at a finer level of granularity.

The exploitation of the structure of the Kimble IR is still experimental. Future work will focus on the addition of a pipeline parallelism loop annotation, and on the scheduling of Kimble graphs. We expect to improve the proposed characterizing metrics in order to further facilitate this process.

Acknowledgments

The authors would like to thank Professor William Jalby for its insightful comments about this work.

References

[1] G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Conference Proceedings*, 30(8):483–485, 1967.

[2] N. Benoit and S. Louise. Extending GCC with a multi-grain parallelism adaptation framework for MPSoCs. In *Proceedings of the 2nd International Workshop on GCC Research Opportunities*, 2010.

[3] K. S. Chatha and R. Vemuri. Magellan: multiway hardware-software partitioning and scheduling for latency minimization of hierarchical control-dataflow task graphs. In *Proceedings of the ninth international symposium on Hardware/software codesign*, CODES '01. ACM, 2001.

[4] A. Duller, G. Panesar, and D. Towner. Parallel Processing: the picoChip Way. *Communicating Process Architectures*, 2003.

[5] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3), 1987. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/24039.24041.

[6] M. Girkar and C. D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Trans. Parallel Distrib. Syst.*, 3, March 1992.

[7] A. Greiner. Tsar : a scalable, shared memory, many-cores architecture with global cache coherence. In *9th Int. Forum on Embedded MPSoC and Multicore*, 2009.

[8] M. Gschwind, H. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *IEEE MICRO*, 26(2), 2006.

[9] P. Hoang and J. Rabaey. Scheduling of DSP Programs onto Multiprocessors for Maximum Throughput. *IEEE Transactions on Signal Processing*, 41(6), 1993.

[10] K. Kimura, Y. Wada, H. Nakano, T. Kodaka, J. Shirako, K. Ishizaka, and H. Kasahara. Multigrain Parallel Processing on Compiler Cooperative Chip Multiprocessor. 2005.

[11] G. J. Narlikar and G. E. Blelloch. Space-efficient implementation of nested parallelism. In *PPOPP '97: Proceedings of the sixth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 1997. ISBN 0-89791-906-8. doi: http://doi.acm.org/10.1145/263764.263770.

[12] M. Okamoto, K. Yamashita, H. Kasahara, and S. Narita. Hierarchical macro-dataflow computation scheme. In *IEEE PACRIM '95*, 1995.

[13] OpenMP Architectural Review Board. OpenMP 3.0 specification, 2008.

[14] C. D. Polychronopoulos. The hierarchical task graph and its use in auto-scheduling. In *Proceedings of the 5th international conference on Supercomputing*, ICS '91. ACM, 1991.

[15] S. Pop, A. Cohen, C. Bastoul, S. Girbal, P. Jouvelot, G. Silber, and N. Vasilache. GRAPHITE: Loop optimizations based on the polyhedral model for GCC. In *Proc. of the 4th GCC Developper's Summit, Ottawa, Canada, June*, 2006.

[16] J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross. Exploiting task and data parallelism on a multicomputer. *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 13–22, 1993.

[17] J. Warren. A hierarchical basis for reordering transformations. In *POPL '84*. ACM, 1984. ISBN 0-89791-125-3. doi: http://doi.acm.org/10.1145/800017.800539.

[18] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C. Miao, J. Brown, and A. Agarwal. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro*, 27(5), 2007.