DDG Reference Manual

2.0

Sid-Ahmed-Ali TOUATI

April 2009

Contents

1 DDG: A C++ High Level Data Dependence Graph Library

Author:

: Sid-Ahmed-Ali Touati

with contribution from Frederic Brault (from INRIA) and Sebastien Briais (from UVSQ).

1.1 Introduction

This is a generic C++ library that handles data dependence graphs (DDG) for optimising compilation. It is built on top of the LEDA graph library (see http://www.algorithmic-solutions.com/enleda.htm). Our graph library is specially though for research purposes where people are willing to make quick, robust and modular implementations of code optimisation techniques for basic blocks and simple innermost loops (modeled by regular mono-dimensional data dependences). We manage directed acyclic graphs (DAGs) for basic blocks and cyclic graphs for innermost loops. The user is able to take advantage of many standard algorithms for graphs. Also, numerous algorithms on data dependence graphs are implemented. It is also possible to configure the library for different instruction set architectures, and multiple register types.

LEDA is a famous C++ graphs and general data structures library. We have used it since many years, and we can safely say that it is *better* than other existing C++ graph and data structures libraries that we experimented (BOOST, STL, etc.). LEDA is a high level library greatly helping to implement complex algorithms in a quick, robust and modular way. According to our deep experience, a C++ code using LEDA looks like a high level algorithm, allowing to easily debug it without suffering from programming details. Furthermore, LEDA offers the largest set of implementation of well known algorithms in graph theory and data structures: these are very helpful in optimising

compilation, especially for instruction level parallelism problems such as instruction scheduling and register allocation. LEDA is well suited for academic research.

Remarks:

- Our C++ library does not perform or implement data dependence analysis. Currently, it's up to the user to make such analysis. Building a precise data dependence analysis module is another on progress software of us (see http://www.prism.uvsq.fr/~bem/fadalib/home.html).
- This version of DDG has been tested with g++ version 4.2. Since g++ implementations continuously change the accepted C++ codes, we do not know of further version of g++ would compile LEDA and DDG. Maybe a minor porting would be required.

1.2 Requested Libraries for Using DDG

- Getting a LEDA licence is required in order to be able to build new binaries using our DDG library. For our colleagues in universities, algorithmic-solutions offers free academic binary LEDA licences allowing you to use our DDG library. It is also recommended to learn the basic ways to use LEDA.
- For our XML parsing, we use libxerces-c. It is provided in almost all linux distribution.

1.3 What is new in this version 2.0

- Now, DDG may handle complex ISA, with multiple register types. Each instruction may write multiple results of distinct types.
- Add new classes : ARCHITECTURE, REGISTER_TYPES
- Enhancements of precedent classes : adding check methods, adding multiple register types
- Register saturation computation can be done of any register type.
- Bug fix in Register Saturation Computation
- Selection between two heuristics for Register Saturation Computation: fast or precise
- New enhanced XML formats for input/output DDG and architecture description.
- Deprecated DDG file formats: gl.
- Now, DDG is distributed under the LGPL software licence instead of GPL.

1.4 What was new in version 1.5

- It compiles with g++ 4.2
- Added : minimal chain decomposition (Dilworth decomposition)
- Added : loop unroll example
- Added : delta_r and delta_w functions with id as parameter
- Added functions in DAG : functions lt ge parralel comparable
- Improvement of function and member declarations : we put a "const" C++ qualifier to each method that does not modify the C++ object.

1.5 What was new in version 1.1

- Enhanced ISA architectural description : for instance, reading and writing delay from/to registers are now implemented inside DDG (δ_r and δ_w latencies). Also, the hardware latencies of operations are implemented too, which may distinct from the latencies of the edges.
- The DDGs now may have edges with nonpositive integer latencies and distances (δ and λ functions become general integers). This is an important aspect if we would like to model optimal scheduling problems for VLIW/EPIC architectures. Consequently, some DDG algorithms have been slightly modified to include this aspect (such as critical cycle computation, longest and shortest paths, etc.)
- The DDG library allows now to manage multiple types of edges : flow, antidep, output dep, input dep, serial
- Implementing loop retiming/shifting methods
- Enhanced gl DDG format
- Consistent DDG copying : copy methods preserve the internal consistency of DDGs objects.
- Consistent hidding nodes and edges
- Import and export methods from gml and leda graph formats
- Enhanced export method to vcg DDG format for interactive DDG visualization : colored edges classes, etc.
- Associate a generic textual attribute to each node. This would allow to associate, for instance, the textual code representation of each instruction.
- Several other minor functions
- Bug fixes

1.6 DAG Model

A DAG $G = (V, E, \delta)$ in our library is a directed acyclic graph (from top to bottom) that represents the data dependences between a set of statements and any other serial constraints. The DAG is defined by its set of nodes (statements) V, its set of edges (data dependences and serial constraints) $E = \{(u, v)|\ u, v \in V\}$, and δ such that $\delta(e)$ is the latency of the edge e (in terms of processor clock cycles for instance). While the latencies are non-negative in general (this assumption is important for many standard graph algorithms), we allow in the DDG library to have non-positive latencies in order to model some VLIW and EPIC characteristics. We consider a target RISC-style architecture. We differentiate between statements and precedence constraints, based on whether they refer to values to be stored in registers or not.

- $V_{R,t} \subseteq V$ is the set of statements (operations) which define values of type t to be stored in registers of type t. We simply call them *values*.
- $E_{R,t} \subseteq E$ is the set of data flow dependence edges through registers of type t.

1.6.1 Notation and Definitions on DAGs

In this manual, we use the following notations for a given DAG $G = (V, E, \delta)$ (as those usually used in lattices and orders algebra):

- $\Gamma^+_G(u) = \{v \in V|(u, v) \in E\}$ successors of u in the graph G. They are also called the `children` of u.
- $\Gamma^-_G(u) = \{v \in V|(v, u) \in E\}$ predecessors of u in the graph G;.They are also called the `parents` of u.
- $\forall e = (u, v) \in E \quad source(e) = u \wedge target(e) = v$. u and v are called *end-points*;
- $\forall u, v \in V, u \leq v \iff \exists$ a path $(u, \ldots, v)$ in G. This relationship is reflexive: $\forall u, u \leq u$.
-
- $\forall u, v \in V, u < v \iff u \neq vwedge\exists$ a path $(u, \ldots, v)$ in G;
- $\forall u, v \in V :\ u||v \iff \neg(u \leq v) \wedge \neg(v \leq u)$. u and v are said to be *parallel* ;
- $\forall u, v \in V :\ u \sim v \iff u \leq v \vee v \leq u$. u and v are said to be *comparable* ;
- $\forall u \in V \ \uparrow u = \{v \in V|v = u \vee v \leq u\}$ u's ascendants including u. In other terms, a node u is an ascendant of a node v iff there exists a path from u to v.
- $\forall u \in V \ \downarrow u = \{v \in V|v = u \vee u \leq v\}$ u 's descendants including u. In other terms, a node u is a descendant of a node v iff there exists a path from v to u.
- $A \subseteq V$ is an antichain iff all nodes belonging to A are parallel. Formally, $A \subseteq V$ is an antichain in G iff $\forall u, v \in A,\ u||v$;

- AM is a *maximal* antichain iff its size in terms of number of nodes is maximal. Formally, AM is a *maximal* antichain $if f \forall A$ antichain in G, $\quad |A| \leq |AM|$;

- The size of a maximal antichain is called the *width* of the DAG and is noted $w(G)$.

- $C \subseteq V$ is a chain iff all nodes belonging to C are compatable. Simply, all nodes of a chain belongs to the same path in the DAG. Formally, $C \subseteq V$ is a chain in G iff $\forall u, v \in C,\ u \sim v$;

- $CD = \{C_1, \cdots, C_p\}$ is a chain partition of G if any $C_i \in CD$ is a chain and: $\forall u \in V, \exists! i \in [0, p-1] : u \in C_i$.

- A chain decomposition CD is minimal if its indice p is minimal. Such minimal indice is noted $p(G)$.

- In 1950, Dilworth proved that $p(G) = w(G)$, and each maximal antichain is equivalent to a minimal chain decomposition (and vice-versa).

1.7 Loop Model

We consider a simple innermost loop (without branches, with possible recurrences). It is represented by a data dependence graph (DDG) $G = (V, E, \delta, \lambda)$, such that:

- V is the set of the statements in the loop.
- E is the set of precedence constraints (flow dependences, or other serial constraints). Any edge e has the form $e = (u, v)$, where $\delta(e)$ is the latency of the edge e (in terms of processor clock cycles for instance) and $\lambda(e)$ is the distance of the edge e in terms of number of iterations. Both δ and λ may be non-positive. However, valid loops DDGs have always cycles with summed non-negative distances, that is: $\forall C$ a cycle in $G, \sum_{e \in C} \lambda(e) \geq 0$.

We consider a target RISC-style architecture and we distinguish between statements and precedence constraints, depending upon whether they refer to values to be stored in registers or not.

- $V_{R,t} \subseteq V$ is the set of statements that produce values to be stored in registers of type t. We simply call them *values*.
- $E_{R,t} \subseteq E$ is the set of flow dependence edges through registers of type t. The set of consumers (readers through registers) of a value $u^t \in V_{R,t}$ is therefore the set $Cons(u^t) = \{v \in V \mid (u, v) \in E_{R,t}\}$

1.8 Processor Model

In order to consider static issue VLIW processors in which the hardware pipeline steps are visible to compilers (we consider dynamically scheduled superscalar processors

too), we assume that reading from and writing into a register may be delayed from the beginning of the schedule/issue time, and these delays are visible to the compiler (architecturally visible). We define two delay (offset) functions δ_r and δ_w in which

$\delta_{w,t} : V_{R,t} \rightarrow \mathbb{N}$

$u \mapsto \delta_{w,t}(u) | \ 0 \leq \delta_{w,t}(u) < lat(u)$

The writing cycle of u^t into a register is delayed by $\delta_{w,t}(u)$ clock cycles after the issue (schedule) date of u. $lat(u)$ is the hardware latency of the instruction u (it may be distinct from the latencies of the data dependence edges). Also, we define:

$\delta_r : V \rightarrow \mathbb{N}$

$u \mapsto \delta_r(u) | \ 0 \leq \delta_r(u) \leq < lat(u)$

The reading cycle of u from a register is delayed by $\delta_r(u)$ clock cycles after the issue (schedule) date of u.

According to the semantics of superscalar processors (sequential semantics) and EPIC/IA64, δ_r and $\delta_{w,t}$ are equal to zero. Also, there are many VLIW processors with zero reading/writing delays. But some VLIW processors such that Trimedia have non-zero reading/writing delays.

1.9 Some File Formats Defined and Used by the DDG Library

1.9.1 User Defined Instruction Set Architectures (XML format)

ISA objects can be constructed via C++ methods or via import from an XML file. Here is an example of such an XML file:

```
<arch version='1'>
  <regtype type='BR' number='2'>
    <register name='br0'>
    <register name='br1'>
  </regtype>
  <regtype type='GR' number='61'>
  </regtype>

  <instruction opcode='nop' latency='0' delta_r='0'>
  </instruction>
  <instruction opcode='inst_2' latency='1' delta_r='2'>
     <write regtype='GR' delta_w='1'/>
     <write regtype='GR' delta_w='0'/>
  </instruction>
  <semantic type='UAL'/>
</arch>
```

For now, only version 1 is supported, so the version attribute must be set to 1. For each register type, the type name and the number of registers must be provided. The type name has to be unique. Optionally, register names can be specified. For each instruction, a unique opcode must be provided, as well as the latency and read delay of the instruction. If the instruction writes into one or several registers, the register types and write delay must be provided for each one. The semantic can also be specified

(once in the file). If the type attribute is 'UAL', then UAL semantic is assumed. Else, or if the semantic is not specified, NUAL is assumed. The current DDG release contains examples of XML files describing some architectures.

1.9.2 Our enhanced XML format for Importing and Exporting Data Dependence Graphs

DDG objects can be constructed via C++ methods or imported/exported via XML files. Here is a simple example of such a file :

```
<ddg version='1'>
  <operation type='inst_1' id='1'/>
  <operation type='inst_2' id='2'/>
  <operation type='nop' id='3' description='Idle'/>
  <edge src='1' dst='2' latency='3' distance='0' typedep='flowdep_reg' regtype='GR'/>
  <edge src='1' dst='3' latency='0' distance='1' typedep='serial'/>
  <edge src='2' dst='3' latency='0' distance='1' typedep='antidep_reg' regtype='BR'/>
</ddg>
```

Again, version must be set to 1, since only version 1 is supported at the moment. For each operation, a unique integer id must be supplied. The instruction type must match the architecture definition. An optional description can also be provided. For each edge, the id of the source and of the destination operations must be provided, as well as the latency and the distance. The type of the dependence can be any of : flowdep_reg serial antidep_reg outputdep_reg inputdep_reg flowdep_mem antidep_-mem outputdep_mem; inputdep_mem spilldep_mem other_mem killerdep reusedep For some of those dependencies, the register type has to be specified. Again, it must match one of the types defined in the architecture file.

The current DDG release contains examples of XML files describing DDG provided from STmicroelectronics.

1.9.3 Our gl Format for Importing and Exporting Data Dependence Graphs (Deprecated Format)

The gl format is our own simple design for DDGs. This format is used in the case of simple ISA, with a unique register type. For ISA with multiple register types, the enhanced XML format should be used (see Our enhanced XML format for Importing and Exporting Data Dependence Graphs). Other DDGs format are also supported (see LEDA format, gml format). Let assume the name "loop" for the DDG, Hence the file loop.gl contains nodes, edges with their latencies and distances (in terms of number of iterations) in the form

```
DDG.GRAPH 1.1
#number_of_nodes
opcode_id_node_1
opcode_id_node_2
...      .
opcode_id_node_n
#number_of_edges
source_id dest_id latency distance edge_type
```

. . . .

Note that the latency of an edge may be distinct from the latency of the source instruction. edge_type is a string describing the type of the data dependence ("flowdep_reg", "serial", "antidep_reg", "outputdep_reg", etc.). "" do not belong to the strings.

1.9.3.1 Default Opcodes (Deprecated) The default opcodes are as follows. 0 add_op 1 sub_op 2 not_op 3 mul_op 4 div_op 5 ld_op 6 st_op 7 copy_op 8 nop_op Be aware that the current gl format does not contain any information on the ISA (except the opcode ids of the nodes). To completely define an ISA for a DDG, see User Defined Instruction Set Architectures (XML format).

1.9.3.2 Nodes identifiers (Deprecated) Each node in a DDG is uniquely defined by an integer identifier (id). For instance, the id of each node can be simply equal to the position of the node in the loop.gl file (starting from 1). Suppose that the edge section of the loop.gl file contains a line describing an edge in the form :

```
2  4  1  0
```

This means that the edge is from node number 2 to node number 4 with a latency equal to 1 and a distance equal to zero. Do not confuse between the node id and its opcode.

1.10 Some Algorithms Implemented by LEDA

Here is a simple list of algorithms implemented inside LEDA that we can use inside our DDG library

- Basic Graph Algorithms
  1. Sorts algorithms (DFS, BFS, etc.)
  2. Computing strongly connected components, connected components, bipartite components, transitive closure, transitive reduction
- Shortest Path Algorithms
- Maximum Flow
- Min Cost Flow Algorithms
- Minimum Cut
- Maximum Cardinality Matchings in Bipartite Graphs
- Bipartite Weighted Matchings and Assignments
- Maximum Cardinality Matchings in General Graphs
- General Weighted Matchings

- Stable Matching
- Minimum Spanning Trees
- Euler Tours
- Algorithms for Planar Graphs
- Graph Drawing Algorithms
- Graph Morphism Algorithms
- Graph Morphism Algorithm Functionality

1.11 Some Algorithms Implemented by the DDG Library

Our DDG library inherits from all the LEDA powerfull algorithmic implementations, such as high level graph operators, access and update DDG structures and attributes, etc. It also includes some specific algorithms used in optimising compilation.

- DAG algorithms
  1. Shortest and longest paths between any pair of nodes
  2. Dilworth decomposition, maximal antichain, minimal chain decomposition.
  3. Register Saturation of a DAG (maximal register pressure independently of instruction schedules). The current implementation considers multiple register types.
  4. Import and export methods from/to XML formats, see Some File Formats Defined and Used by the DDG Library
  5. Exporting to graph visualization tool (xvcg).
- Loop algorithms
  1. Critical cycle : the critical cycle in a loop DDG is defined as the one producting the maximal ratio $\max_{C \text{ a circuit}} \frac{\sum_{e \in C} \delta(e)}{\sum_{e \in C} \lambda(e)}$. It is an old problem in graph theory studied initially by Dantzing in 1966. Such circuit is assumed in general to have $\sum_{e \in C} \delta(e) 0 >$ and $\sum_{e \in C} \lambda(e) > 0$. However, our DDG library adds the special case of null circuits, *i.e.*,, those defined with $\sum_{e \in C} \delta(e) = 0$ and $\sum_{e \in C} \lambda(e) \geq 0$.
  2. DDG unrolling (loop unrolling) with recurrences
  3. DDG merging (loop merging)
  4. Loop retiming/shifting
  5. Import and export methods from/to XML formats, see Some File Formats Defined and Used by the DDG Library
  6. Exporting to graph visualization tool (xvcg).

1.12 Interactive Data Dependence Graphs Visualization using XVCG tool

The DDG library implements enhanced export methods to xvcg format that allow helpful and interactive DDG visualization. The xvcg tool is a free software that can be downloaded from http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html
The DDG library has export methods that produce vcg graphs with the following visual properties.

- The nodes writing into registers (values) are in green, while the other nodes are in grey. See screenshot 1 .
- The edges are valued by couples representing the edge latency and the iteration distance. That is, each edge e is valued by the couple $(\delta(e), \lambda(e))$.
- There are ma,y types of dependence edges : flow edges throught registers, flow edges throught memory, serial edges, antidependance edges, outputdep edges, etc. See screenshot 2 .
- The user may visualize the nodes attributes by clicking on Node Information :
  1. ISA opcodes : see screenshot 3 and screenshot 4 .
  2. different latencies : hardware operation latency, reading and writing delays from/into registers, see screenshot 5 and screenshot 6
  3. The distinct register types produced by the instruction.
  4. textual attributes attached to ndes (codes, etc.)
- The user may visualize andd/or hide any class of these edges using the xvcg tool (click on Expose/Hide Edges) : see screenshot 7.

Some other screenshots can be found here .

1.13 Getting Started

1.13.1 Download Sources

The source files of our DDG library (with the included reference manual) can be downloaded from http://www.prism.uvsq.fr/~touati/sw/DDG/DDG-2.0.tgz Our codes are under LGPL licence. If you are from academia, you can get a free academic licence for LEDA. Otherwise, you should buy a licence to be able to use DDG. In this Release you find:

- INSTALL instructions
- Full user guide documentation
- DDG and ISA examples (in XML format)
- C++ Code Examples
- All C++ sources (except LEDA, that should be instaleld separately by the user)

1.13.2 Download some Binaries Examples (linux X64)

Some binaries under linux/x86 have been built using g++ 4.2. Our binaries do not require LEDA, but they require other dynamic open source libraries such as libstd++ and libxerces-c. Here is the list of binaries we distribute as examples:

- Library : `binary archive ar file`.
- Examples (all sources included in the distribution) :
  1. `DAG example` .
  2. `LOOP example` .
  3. `Register Saturation Computation of a DAG`.
- Command line DDG file format filter: `xml2vcg`.

you can execute these programs on the DDG and ISA samples we provide inside the DDG release.

1.13.3 Building the DDG Library

See installation instructions in the source file distribution. The current version of DDG has been implemented under linux/x86 with g++ version 4.2.2 and LEDA version 5.

1.13.4 Some Coding Instructions for Users

The user should include "DDG.h" in his C++ program. This header file defines the API of the DDG library and includes some LEDA header files. Thus, the user should set the `LEDAROOT` environment variable to the path of LEDA installation directory and use the include directory `-I$LEDAROOT/incl` in the command line argument of `g++`.

Furthermore, we highly recommend to use the `-fno-inline` option of your C++ compiler.

1.13.5 Linking Instructions

After successfull compilation (installation) of the DDG library, a file library called `libDDG.a` is created. This library has to be linked with the user applications using the DDG features. Furthermore, the LEDA library should be present (make sure that the `LEDAROOT` environment variable is correctly set), especially `libG.a` and `libL.a`. The user should then link its application with all these libraries by using the following link option for `g++` line command `libDDG.a -L$LEDAROOT -lG -lL`. Note that the order of these linker options is important.

1.13.6 A Hello World Program

There are two main DDG C++ classes, DAG and LOOP, corresponding to the DAG and loop models defined in DAG Model and Loop Model respectively. So, each DDG is an

object of these two C++ classes. Data dependence graphs can be filled either by reading a DDG XML file (see Some File Formats Defined and Used by the DDG Library), or buy using C++ methods (`new_node`, `new_edges`, etc.). If the user wishes, he can implement his own import method from other graph formats. Here is a simple hello world program to show that it is very simple to handle data dependence graphs using our library. Detailed information could be found in the definitions of DAG and LOOP classes, and in the other code examples.

```
#include "DDG.h" // the header file of the DDG library. To be included by the user.

#include <iostream>
#include <cstdlib>


using namespace std;
using namespace DDG; // the namespace of the DDG library.



int main(int argc, char *argv[])
{
    int         i;
    int c; // for getopt
    // some strings for file names
    ARCHITECTURE isa_arch;

    char        *ddg_filename=NULL, *arch_filename=NULL;
    //declare a DDG of a loop (cyclic data dependence graph)
    DDG::LOOP loop;

    LEDA::node n,u;
    LEDA::edge e;


    while ((c = getopt (argc, argv, "i:a:")) != -1){
        switch(c){
            case 'a': arch_filename=optarg;
                break;
            case 'i': ddg_filename =optarg;
                break;
    case '?':
                cout<<"usage:"<< argv[0] <<" -i ddg_filename.xml [-a isa_desc.xml] "<<endl;
                return EXIT_FAILURE;
        }

    }

    if (arch_filename!=NULL)  {
        isa_arch.read_from_xml(arch_filename);
        if(isa_arch.check()){
            loop=LOOP(isa_example);  //build an empty DDG loop with a user defined architcural (ISA
        } else return EXIT_FAILURE;

    }
     //---- The user can read a DDG from an xml format defined by the DDG library
    if (ddg_filename != NULL){
        i=loop.read_from_xml(ddg_filename); // reads the DDG loop from an xml file
        if(i==-1) return EXIT_FAILURE;
        }
    else{
        cout<<"usage:"<< argv[0] <<" -i ddg_filename.xml [-a isa_desc.xml] "<<endl;
```

```
        return EXIT_FAILURE;
    }




    cout<< "DDG example : " <<ddg_filename<<endl;
    // Null circuits are accepted inside the DDG and detected
    cout<<"Critical Cycle  of "<< ddg_filename<<" = "<<loop.CRITICAL_CYCLE(el)<<endl;
    cout<<"Edges belonging the the critical cycle"<<endl;
    forall(e, el){
        ie=loop[e];
        cout<< loop.source_id(e)<< " -> "<< loop.target_id(e);
        cout<<" : "<< ie<<endl;
    }


    return EXIT_SUCCESS;
}
```

1.14 Future plans

Our C++ DDG library is generic and modular enough to be extended to various directions. The first possible extension is to be able to model multi-dimensional data dependences (inside a non perfect loop nest) instead of a simple innermost loop. We are willing to work on this issue, depending on our future fundings. In addition, we are are working on a release of a module computing data flow information using the FADA method (Fuzzy Data Flow Analysis). See http://www.prism.uvsq.fr/~bem/fadalib/home.html

2 DDG Namespace Documentation

2.1 DDG Namespace Reference

2.1.1 Detailed Description

All the API symbols used in the DDG library.

Data Structures

- class ARCHITECTURE
- class DAG

  This class represents a directed acyclic graph (DAG).

- class IllegalAccessNode
- class InvalidISADesc
- class BadIODDG
- class BadRegType
- class BadInstructionType

- class NonUniqueRegisterType
- class IllegalDependance
- class NonUniqueNodeID
- class INFO_EDGE

  This class represents the attributes (information) attached to an edge in a DDG.

- class INFO_NODE

  This class represents the attributes (information) attached to each node in a DDG.

- class INSTRUCTIONS_TYPES

  This class describes a generic instruction, used by the class ARCHITECTURE to describe an ISA. A generic instruction object contains opcode, unique id per opcode, string opcode, list of writtent registers, list of written register types, etc.

- class LOOP

  This class represent cyclic data dependence graphs of simple loops (without branches).

- class REGISTER_TYPES
- class CBC
- class EN

Enumerations

- enum edge_type {

  flowdep_reg = 1, antidep_reg, outputdep_reg, inputdep_reg,

  flowdep_mem, antidep_mem, outputdep_mem, inputdep_mem,

  spilldep_mem, other_mem, serial, killerdep,

  reusedep }

  Edges corresponds to data dependences. Data dependences are classified into multiple types : flowdep_reg, antidep_reg, outputdep_reg, inputdep_reg, flowdep_mem, etc.

- enum NodeFlag {

  NodeFlag_Nothing = 0x0, NodeFlag_PartialDef = 0x1, NodeFlag_SpillCode = 0x2, NodeFlag_Volatile = 0x4,

  NodeFlag_SPUpdate = 0x8, NodeFlag_Prefetch = 0x10, NodeFlag_Preload = 0x20, NodeFlag_Barrier = 0x40,

  NodeFlag_Clobber = 0x80, NodeFlag_Hoisted = 0x100, NodeFlag_DeadCode = 0x200, NodeFlag_SafeAccess = 0x800,

  NodeFlag_SafePerfs = 0x1000, NodeFlag_EntryCode = 0x4000, NodeFlag_-ExitCode = 0x8000, NodeFlag_KillOp = 0x5000 }

  Nodes may have some flag representing, special nodes. If a node corresponds to a regular instruction, then its flag is NodeFlag_Nothing.

Functions

- template<class T>
  LEDA::list< T > set_to_list (const set< T > l)
- template<class T>
  set< T > list_to_set (const list< T > l)
- bool lt (node u, node v)
- bool le (node u, node v)
- bool gt (node u, node v)
- bool ge (node u, node v)
- bool parallel (node u, node v)
- bool comparable (node u, node v)
- int MAXIMAL_ANTI_CHAIN (const graph &G, set< node > &MA)
- int MINIMAL_CHAIN (const graph &G, node_array< int > &chain)
- int MINIMAL_CHAIN (const DAG &G, array< list< node > > &C)
- int REGISTER_SATURATION (const DAG &G, leda::list< node > &RS_values, bool fast=true)
- int REGISTER_SATURATION (const DAG &G, leda::list< node > &RS_values, node_array< node > &k, bool fast=true)
- int REGISTER_SATURATION (const DAG &G, int t, leda::list< node > &RS_values, bool fast=true)
- int REGISTER_SATURATION (const DAG &G, int t, leda::list< node > &RS_values, node_array< node > &k, bool fast=true)
- std::string edge_type_to_string (const edge_type)

  Convert an edge_type to string ("flow", "serial", "antidep", ...).

- edge_type string_to_edge_type (const std::string)

  Convert a string to an edge_type.

- void unroll (const LOOP &L, DAG &unrolled, int unroll_degree)

  Returns the body of the unrolled loop.

- void unroll (const LOOP &L, LOOP &unrolled, int unroll_degree)

  Returns an unrolled loop.

- void loop_merge (const LOOP &L1, const LOOP &L2, LOOP &L3)

  Merges two independent loop DDGs in order to produce a new DDG.

- bool retime_ddg (LOOP &G, LEDA::node_array< int > &r)

  Computes and applies a valid loop retiming (called also loop shifing).

- bool retime_ddg (LOOP &G)

  Computes and applies a valid loop retiming (called also loop shifing).

- bool apply_retiming (LOOP &G, const LEDA::node_array< int > r)

  Applies the loop retiming given as input.

- bool is_lexicographic_positive (LOOP &G, leda::list< edge > &le)

  Checks if all circuits of G are lexicographic positive. That is, $\forall C$ a circuit in G, $\lambda(C) = \sum_{e \in C} \lambda(e) > 0$.

2.1.2 Enumeration Type Documentation

2.1.2.1 enum typedef enum edge_type

Edges corresponds to data dependences. Data dependences are classified into multiple types : flowdep_reg, antidep_reg, outputdep_reg, inputdep_reg, flowdep_mem, etc.

Remarks:

The user can modify this set of edge types in order to take into account user-defined data dependences.

Enumerator:

flowdep_reg

antidep_reg

outputdep_reg

inputdep_reg

flowdep_mem

antidep_mem

outputdep_mem

inputdep_mem

spilldep_mem

other_mem

serial

killerdep

reusedep

Definition at line 50 of file info_edge.h.

2.1.2.2 enum typedef enum NodeFlag

Nodes may have some flag representing, special nodes. If a node corresponds to a regular instruction, then its flag is NodeFlag_Nothing.

Enumerator:

NodeFlag_Nothing

NodeFlag_PartialDef

NodeFlag_SpillCode

NodeFlag_Volatile

NodeFlag_SPUpdate

NodeFlag_Prefetch

NodeFlag_Preload

NodeFlag_Barrier

NodeFlag_Clobber

NodeFlag_Hoisted

NodeFlag_DeadCode

NodeFlag_SafeAccess

NodeFlag_SafePerfs

NodeFlag_EntryCode

NodeFlag_ExitCode

NodeFlag_KillOp

Definition at line 51 of file info_node.h.

2.1.3 Function Documentation

2.1.3.1 bool apply_retiming (LOOP & *G*, const LEDA::node_array< int > *r*)

Applies the loop retiming given as input.

Parameters:

↔ ***G*** The loop to retime (modified by the loop retiming)

← ***r*** the input retiming given by the user.

Returns:

true if the input retiming is valid (all retimed edge distances are nonnegative).
false if the input retiming is not valid (in case of the presence of a nonpositive retimed edge distance).

Warning:

If the retiming is not valid, the loop would been modified anywhere.

Loop retiming consists of the following graph transformation: for each loop statement u, we associate an integer shift $r(u)$ which means that we delay the operation u by $r(u)$ iterations. Then, each statement u that was representing the operation u of loop iteration i represents now the operation u of loop iteration $i - r(u)$. The new distance of each arc $e = (u, v)$ becomes equal to $\lambda_r(e) = \lambda(e) + r(v) - r(u)$. Such new distance is always nonnegative $\lambda_r(u) \geq 0, \forall u \in V$.

For more details about loop retiming, please study the excellent research article of Leiserson and Saxe published in Algorithmica 1991 (Retiming asynchronous circuits).

2.1.3.2 bool comparable (node *u*, node *v*)

Parameters:

← ***u*** A first node

← ***v*** A second node

Returns:

True if $u \sim v$, i.e., $u \leq v \vee v \leq u$.

Precondition:

u and v belongs to the same DAG. This is a notation from the lattices and order theory

Remarks:

$\forall u \in V, u \sim u$

Examples:

dag_example.cpp.

2.1.3.3 std::string DDG::edge_type_to_string (const *edge_type*)

Convert an edge_type to string ("flow", "serial", "antidep", ...).

Warning:

Returns empty string in case of unknownn edge_type

2.1.3.4 bool ge (node *u*, node *v*)

Parameters:

← ***u*** A first node

← ***v*** A second node

Returns:

Returns True if $u \geq v$, i.e., $u = v \vee \exists$ a path $(v, \ldots, u)$.

Precondition:

u and v belongs to the same DAG. This is a notation from the lattices and order theory

Remarks:

this relationship is reflexive: $\forall u \in V, u \geq u$

Examples:

dag_example.cpp.

2.1.3.5 bool gt (node *u*, node *v*)

Parameters:

← ***u*** A first node

← ***v*** A second node

Returns:

Returns True if $u > v$, i.e., $u \neq \wedge \exists$ a path $(v, \ldots, u)$.

Precondition:

u and v belongs to the same DAG. This is a notation from the lattices and order theory

Examples:

dag_example.cpp.

2.1.3.6 is_lexicographic_positive (LOOP & *G*, leda::list< edge > & *le*)

Checks if all circuits of G are lexicographic positive. That is, $\forall C$ a circuit in G, $\lambda(C) = \sum_{e \in C} \lambda(e) > 0$.

Parameters:

← ***G*** The loop check

→ ***le*** A list of a delinquent circuit C, that is $\lambda(C) \leq 0$.

Returns:

true if the input DDG is lexicographic positive. False otherwise.

Parameters:

← ***G*** The loop check $*$

Returns:

true if the input DDG is lexicographic positive. False otherwise.

2.1.3.7 bool le (node *u*, node *v*)

Parameters:

← ***u*** A first node

← ***v*** A second node

Returns:

Returns True if $u \leq v$, i.e., $u = v \vee \exists$ a path $(u, \ldots, v)$.

Precondition:

u and v belongs to the same DAG. This is a notation from the lattices and order theory

Remarks:

this relationship is reflexive: $\forall u \in V, u \leq u$

Examples:

dag_example.cpp, and RS_example.cpp.

2.1.3.8 void loop_merge (const LOOP & *L1*, const LOOP & *L2*, LOOP & *L3*)

Merges two independent loop DDGs in order to produce a new DDG.

Parameters:

← ***L1*** The first DDG to consider.

← ***L2*** The second DDG to consider.

→ ***L3*** A new loop consisting of merging distinct copies of L1 and L2.

Precondition:

Unrolling degree and all dependence distances should be nonnegative. $\forall u \in V, \lambda(u) \geq 0$.

2.1.3.9 bool lt (node *u*, node *v*)

Parameters:

← ***u*** A first node

← ***v*** A second node

Returns:

Returns True if $u < v$, i.e., $u \neq \wedge \exists$ a path $(u, \ldots, v)$.

Precondition:

u and v belongs to the same DAG. This is a notation from the lattices and order theory

Examples:

dag_example.cpp.

2.1.3.10 int MAXIMAL_ANTI_CHAIN (const graph & *G*, set< node > & *MA*)

Parameters:

← ***G*** An acyclic graph (DAG, order)

→ ***MA*** The set of nodes belonging to a maximal antichain

Returns:

The size of a maximal antichain in the graph (-1 if the graph is not acyclic). Returns 0 if the input acyclic graph is empty. Returns -1 if error.

This function implements Dilworth decomposition to compute maximal antichain inside an order (DAG).

Remarks:

A maximal antichain may not be unique. This function returns an arbitrary one.

Precondition:

The input graph G should be acyclic.

Examples:

dag_example.cpp.

2.1.3.11 int DDG::MINIMAL_CHAIN (const DAG & *G*, array< list< node > > & *C*)

int MINIMAL_CHAIN(const DAG &G, array< list<node> > &C) This function computes a minimal chain decomposition of a DAG (an order).

Parameters:

← ***G*** An input DAG

→ ***C*** $C[i]$ is a node list reprensenting the chain number i, with $0 \leq i < p(G)$. The lists are sorted in increasing order: $\forall i, \forall u, v \in C[i], u \leq v \iff \exists$ a path $(u, \ldots, v)$ in G.

Returns:

$p(G)$ the minimal number of chains of the DAG. Dilworth proved that $p(G) = w(G)$, that is, it is equal to the size of a maximal antichain. Returns 0 if the input acyclic graph is empty. Returns -1 if error.

Remarks:

A minimal chain decomposition may not be unique. This function returns an arbitrary one.

Precondition:

G is acyclic.

Examples:

dag_example.cpp.

2.1.3.12 int MINIMAL_CHAIN (const graph & *G*, node_array< int > & *chain*)

Parameters:

← ***G*** An acyclic graph (DAG, order)

→ ***chain*** Contains the chain number for each node of the DAG. $\forall u \in V, chain[u]$ is the number of the chain to which u belongs.

Returns:

The size of a minimal chain decomposition of the graph. Returns 0 if the input acyclic graph is empty. Returns -1 if error.

This function implements Dilworth decomposition to compute minimal chain decomposition of an order (DAG).

Precondition:

The input graph G should be acyclic.

Remarks:

A minimal chain decomposition may not be unique. This function returns an arbitrary one.

2.1.3.13 bool parallel (node *u*, node *v*)

Parameters:

← ***u*** A first node

← ***v*** A second node

Returns:

True if $u||v$, i.e., $\neg(u \leq v) \wedge \neg(v \leq u)$.

Precondition:

u and v belongs to the same DAG. This is a notation from the lattices and order theory

Examples:

dag_example.cpp.

2.1.3.14 int REGISTER_SATURATION (const DAG & *G*, int *t*, leda::list< node > & *RS_values*, node_array< node > & *k*, bool *fast* = `true`)

Parameters:

← ***G*** An input DAG

← ***t*** A register type identifier.

→ ***RS_values*** The list of saturating values. It is proved that there exists an instruction schedule for any underlying processor constraints that make all the saturating values simultaneously alive.

→ ***k*** A valid saturating killing function. If the computed register saturation is equal to RS, then it is proved that there exists a schedule for the input DAG (for any hardware property) that requires RS registers if the killer of each value $u \in V_{R,t}$ is set to $k(u^t)$.

← ***fast*** boolean to indicate to use either fast heuristic or accurate one

Returns:

An approximate register saturation for the DAG.

Precondition:

The set $V_{R,t} \subseteq V$ of values and the set $E_{R,t} \subseteq E$ of flow edges should be set. This function implements a heuristics for computing the register saturation of the DAG (computing the optimal register saturation is an NP-complete problem). It consists of computing the maximal register need of a DAG independently of instruction schedules. Our experiments on a large set of DAGs show that our heuristics is optimal in 95% of the cases, and far from the optimal by only one register in 5% of the cases. This concept has been mathematically and experimentally validated in the follwing research article :

Sid-Ahmed-Ali Touati. Register Saturation in Instruction Level Parallelism. International Journal of Parallel Programming, Springer-Verlag, Volume 33, Issue 4, August 2005.

Examples:

dag_example.cpp, and RS_example.cpp.

2.1.3.15 int REGISTER_SATURATION (const DAG & *G*, int *t*, leda::list< node > & *RS_values*, bool *fast* = `true`)

Parameters:

← ***G*** A DAG

← ***t*** A register type identifier.

→ ***RS_values*** The list of saturating values of type t. It is proved that there exists an instruction schedule for any underlying processor constraints that make all the saturating values simultaneously alive.

← ***fast*** boolean to indicate to use either fast heuristic or accurate one

Returns:

An approximate register saturation for the DAG.

Precondition:

The set $V_{R,t} \subseteq V$ of values and the set $E_{R,t} \subseteq E$ of flow edges should be set. This function implements a heuristics for computing the register saturation of the DAG (computing the optimal register saturation is an NP-complete problem). It consists of computing the maximal register need of a DAG independently of instruction schedules. Our experiments on a large set of DAGs show that our heuristics is optimal in 95% of the cases, and far from the optimal by only one register in 5% of the cases. This concept has been mathematically and experimentally validated in the follwing research article :

Sid-Ahmed-Ali Touati. Register Saturation in Instruction Level Parallelism. International Journal of Parallel Programming, Springer-Verlag, Volume 33, Issue 4, August 2005.

2.1.3.16 int REGISTER_SATURATION (const DAG & *G*, leda::list< node > & *RS_values*, node_array< node > & *k*, bool *fast* = `true`)

Parameters:

← ***G*** An input DAG

→ ***RS_values*** The list of saturating values. It is proved that there exists an instruction schedule for any underlying processor constraints that make all the saturating values simultaneously alive.

→ ***k*** A valid saturating killing function. If the computed register saturation is equal to RS, then it is proved that there exists a schedule for the input DAG (for any hardware property) that requires RS registers if the killer of each value $u \in V_R$ is set to $k(u)$.

← ***fast*** boolean to indicate to use either fast heuristic or accurate one

Returns:

An approximate register saturation for the DAG.

Precondition:

The set $V_R \subseteq V$ of values and the set $E_R \subseteq E$ of flow edges should be set.
a unique register type exists in the ISA.

Exceptions:

NonUniqueRegisterType This function implements a heuristics for computing the register saturation of the DAG (computing the optimal register saturation is an NP-complete problem). It consists of computing the maximal register

need of a DAG independently of instruction schedules. Our experiments on a large set of DAGs show that our heuristics is optimal in 95% of the cases, and far from the optimal by only one register in 5% of the cases. This concept has been mathematically and experimentally validated in the follwing research article :

Sid-Ahmed-Ali Touati. Register Saturation in Instruction Level Parallelism. International Journal of Parallel Programming, Springer-Verlag, Volume 33, Issue 4, August 2005.

2.1.3.17 int REGISTER_SATURATION (const DAG & *G*, leda::list< node > & *RS_values*, bool *fast* = `true`)

Parameters:

← ***G*** A DAG

→ ***RS_values*** The list of saturating values. It is proved that there exists an instruction schedule for any underlying processor constraints that make all the saturating values simultaneously alive.

← ***fast*** boolean to indicate to use either fast heuristic or accurate one

Returns:

An approximate register saturation for the DAG.

Precondition:

The set $V_R \subseteq V$ of values and the set $E_R \subseteq E$ of flow edges should be set.
a unique register type exists in the ISA.

Exceptions:

NonUniqueRegisterType This function implements a heuristics for computing the register saturation of the DAG (computing the optimal register saturation is an NP-complete problem). It consists of computing the maximal register need of a DAG independently of instruction schedules. Our experiments on a large set of DAGs show that our heuristics is optimal in 95% of the cases, and far from the optimal by only one register in 5% of the cases. This concept has been mathematically and experimentally validated in the follwing research article :

Sid-Ahmed-Ali Touati. Register Saturation in Instruction Level Parallelism. International Journal of Parallel Programming, Springer-Verlag, Volume 33, Issue 4, August 2005.

2.1.3.18 bool retime_ddg (LOOP & *G*)

Computes and applies a valid loop retiming (called also loop shifing).

Parameters:

↔ ***G*** The loop to retime (modified by the loop retiming)

Returns:

true if a valid retiming has been found.
false if no valid retiming has been found (in case of the presence of nonpositive distance cycle).

Loop retiming consists of the following graph transformation: for each loop statement u, we associate an integer shift $r(u)$ which means that we delay the operation u by $r(u)$ iterations. Then, each statement u that was representing the operation u of loop iteration i represents now the operation u of loop iteration $i - r(u)$. The new distance of each arc $e = (u, v)$ becomes equal to $\lambda_r(e) = \lambda(e) + r(v) - r(u)$. Such new distance is always nonnegative $\lambda_r(u) \geq 0, \forall u \in V$.

For more details about loop retiming, please study the excellent research article of Leiserson and Saxe published in Algorithmica 1991 (Retiming asynchronous circuits).

Warning:

If the retiming is not valid, the loop would been modified anywhere.

2.1.3.19 bool retime_ddg (LOOP & *G*, LEDA::node_array< int > & *r*)

Computes and applies a valid loop retiming (called also loop shifing).

Parameters:

↔ ***G*** The loop to retime (modified by the loop retiming)

→ ***r*** the computed retiming.

Returns:

true if a valid retiming has been found.
false if no valid retiming has been found (in case of the presence of nonpositive distance cycle).

Loop retiming consists of the following graph transformation: for each loop statement u, we associate an integer shift $r(u)$ which means that we delay the operation u by $r(u)$ iterations. Then, each statement u that was representing the operation u of loop iteration i represents now the operation u of loop iteration $i - r(u)$. The new distance of each arc $e = (u, v)$ becomes equal to $\lambda_r(e) = \lambda(e) + r(v) - r(u)$. Such new distance is always nonnegative $\lambda_r(u) \geq 0, \forall u \in V$.

For more details about loop retiming, please study the excellent research article of Leiserson and Saxe published in Algorithmica 1991 (Retiming asynchronous circuits).

Warning:

If the retiming is not valid, the loop would been modified anywhere.

2.1.3.20 edge_type DDG::string_to_edge_type (const std::string)

Convert a string to an edge_type.

Precondition:

The input string should be one of ("flow", "serial", "antidep", "outputdep, "input-dep")

2.1.3.21 void unroll (const LOOP & *L*, LOOP & *unrolled*, int *unroll_degree*)

Returns an unrolled loop.

Parameters:

← ***L*** The loop to be considered. Not altered by this function.

→ ***unrolled*** The resulting unrolled loop.

← ***unroll_degree*** The degree of loop unrolling. If equal to zero, no loop unrolling is performed and the resulting loop is empty.

Remarks:

- The degree unrolling tells how much loops bodies are duplicated. If the degree of unrolling is equal to 1, the resulting loop contains only one copy of the initial loop.
- This unrolling function preserves and handles recurrent edges.

Precondition:

Unrolling degree and all dependence distances should be nonnegative. $\forall u \in V, \lambda(u) \geq 0$.

Examples:

dag_example.cpp, and loop_example.cpp.

2.1.3.22 void unroll (const LOOP & *L*, DAG & *unrolled*, int *unroll_degree*)

Returns the body of the unrolled loop.

Parameters:

← ***L*** The loop to be considered. Not altered by this function.

→ ***unrolled*** The resulting DAG (the loop body of the unrolled loop).

← ***unroll_degree*** The degree of loop unrolling. If equal to zero, no loop unrolling is performed and the resulting DAG is empty.

Remarks:

- The degree unrolling tells how much loop bodies are duplicated. If the degree of unrolling is equal to 1, the resulting DAG contains only one loop body.
- This unrolling function preserves and handles recurrent edges.

2.2 leda Namespace Reference

3 DDG Data Structure Documentation

3.1 ARCHITECTURE Class Reference

3.1.1 Detailed Description

Author:

Touati Sid <Sid-nospam.Touati-nospam@inria.fr>

Examples:

dag_example.cpp, gl2vcg.cpp, loop_example.cpp, and RS_example.cpp.

Definition at line 61 of file architecture.h.

Public Member Functions

- LEDA::d_array< int, REGISTER_TYPES > get_dictionary_regtypes () const
- LEDA::d_array< int, INSTRUCTIONS_TYPES > get_dictionary_instructions_types () const
- bool is_default_arch ()
- bool get_ual_semantic () const
- void set_ual_semantic (bool ual)
- REGISTER_TYPES get_register_type_by_id (int id) const
- INSTRUCTIONS_TYPES get_instruction_type_by_id (int id) const
- LEDA::list< REGISTER_TYPES > T () const
- int get_instruction_type_id (std::string stropcode) const

Protected Attributes

- int cpt_regtypes
- int cpt_instructions_types
- bool default_arch
- bool is_ual_semantic

3.1.2 Member Function Documentation

3.1.2.1 is_default_arch () [inline]

Returns true if the architecture is the default one

Definition at line 101 of file architecture.h.

References ARCHITECTURE::default_arch.

3.1.2.2 set_ual_semantic (bool *ual*) [inline]

Set to true if semantic is UAL

Definition at line 118 of file architecture.h.

References ARCHITECTURE::is_ual_semantic.

3.2 BadInstructionType Class Reference

3.2.1 Detailed Description

Definition at line 71 of file ddg_exceptions.h.

Public Member Functions

- BadInstructionType ()
- BadInstructionType (int id)

3.3 BadIODDG Class Reference

3.3.1 Detailed Description

Definition at line 58 of file ddg_exceptions.h.

Public Member Functions

- BadIODDG ()

3.4 BadRegType Class Reference

3.4.1 Detailed Description

Definition at line 65 of file ddg_exceptions.h.

Public Member Functions

- BadRegType ()

3.5 CBC Class Reference

3.5.1 Detailed Description

Definition at line 61 of file RS.h.

Public Member Functions

- CBC ()
- void clear ()

Data Fields

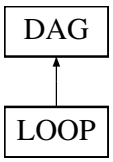
- set< node > S
- set< node > T
- set< edge > E_CB

Friends

- ostream & operator<< (ostream &os, const CBC &x)
- istream & operator>> (istream &is, CBC &x)

3.6 DAG Class Reference

Inheritance diagram for DAG::



3.6.1 Detailed Description

This class represents a directed acyclic graph (DAG).

Author:

Sid-Ahmed-Ali Touati

It inherits from the LEDA class GRAPH<INFO_NODE, INFO_EDGE>. The attributes of the nodes are of type INFO_NODE, and the attributes of edges are of type INFO_EDGE. The implemented DAG model is described in DAG Model.

For more information about LEDA, please consult http://www.algorithmic-solutions.info/leda_manual

Examples:

dag_example.cpp, and RS_example.cpp.

Definition at line 227 of file DAG.h.

Public Member Functions

Creation

- DAG ()
  Declares an empry DAG.
- DAG (const ARCHITECTURE)
- ~**DAG** ()

Access Operations

- LEDA::set< edge > edges_from_to (node u, node v) const
  Returns the set of all edges from the source node u to the destination node v.
- LEDA::set< edge > edges_from_to (int id_u, int id_v) const
  Returns the set of all edges from the source node to the destination node.
- int LongestPath () const
  Returns the value of the longest path of the DAG.
- int LongestPath (LEDA::list< node > &nl) const
  Returns the value of the longest path of the DAG.
- int LongestPath (list< edge > &el) const
  Returns the value of the longest path of the DAG.
- int CriticalPath ()
  This is an alias of the method DAG::LongestPath(). It returns the value of the longest (critical) path.
- int CriticalPath (LEDA::list< node > &nl)
  Returns the value of the longest (critical) path.
- int CriticalPath (LEDA::list< edge > &el)
  Returns the value of the longest (critical) path.
- int LongestPathTo (node) const
  Returns the value of the longest path from the top of the DAG (sources) up to a node given as argument.
- int LongestPathFrom (node) const
  Returns the value of the longest path from a node given as argument to the bottom the DAG (sinks).
- int ShortestPathTo (node) const
  Returns the value of the shortest path from the top of the DAG (sources) up to a node given as argument.
- int ShortestPathFrom (node) const

Returns the value of the shortest path from a node given as argument to the bottom the DAG (sinks).

- int lp (node u, node v) const

  Returns the value of the longest path from a node u to a node v given as arguments∗.

- LEDA::set< node > get_down (node u) const

  Returns the set $\downarrow u$ (the set of all descending nodes)., see Notation and Definitions on DAGs.

- LEDA::set< node > get_up (node u) const

  Returns the set $\uparrow u$.(the set of all ascenfing nodes), see Notation and Definitions on DAGs.

- LEDA::set< node > get_parents (node u) const

  Returns the set $\Gamma_G^+(u)$ (set of children), see Notation and Definitions on DAGs.

- LEDA::set< node > get_children (const node u) const

  Returns the set $\Gamma_G^-(u)$ (set of parents), see Notation and Definitions on DAGs.

- node node_with_id (int i) const

  Returns the node with the integer identifier given as argument.

- node node_with_name (std::string name) const

  Returns the node with the name given as argument.

- bool is_value (node n) const

  Returns true if the node is an instruction writing into a register (i.e. if it belongs to the V_R set).

- bool is_value (int i, int t) const

  Returns true if the node defined by its indentifier is an instruction writing into a register of type t (i.e. if it belongs to the V_R, t set).

- bool is_value (node n, int t) const

  Returns true if the node is an instruction writing into a register of type t (i.e. if it belongs to the $V_{R,t}$ set).

- bool is_value (int i) const

  Returns true if the node defined by its indentifier is an instruction writing into a register (i.e. if it belongs to the V_R set).

- bool is_flow (edge e) const

  Returns true if the edges is a flow dependence (not necessarily through registers).

- bool is_flow_reg (edge e) const

  Returns true if the edges is a flow dependence through a register. This is the case of a flow edge having a source writing into a register.

- int get_register_type_id (edge e) const

Returns the register type of a flow edge.

- bool is_flow_reg (edge e, int t) const

  Returns true if the edges is a flow dependence through a register of type t. This is the case of a flow edge having a source writing into a register.

- DDG::edge_type etype (edge e) const

  Returns the type of the edge e. (flow, serial, antidep, ...).

- set< node > get_V_R () const

  Returns the set $V_R \subseteq V$ of instructions (nodes) writing into registers.

- set< node > get_V_R (int t) const

  Returns the set $V_{R,t} \subseteq V$ of instructions (nodes) writing into registers of type t (given by its id).

- LEDA::list< REGISTER_TYPES > **T** () const
- LEDA::set< edge > get_E_R () const

  Returns the set $E_R \subseteq E$ of flow dependence edges through registers.

- LEDA::set< edge > get_E_R (int t) const

  Returns the set $E_{R,t} \subseteq E$ of flow dependence edges through registers of type t (given by its id).

- LEDA::set< node > get_Sources () const

  Returns the set of nodes which are the sources of the DAG.

- LEDA::set< node > get_Targets () const

  Returns the set of nodes which are the targetsof (sinks) of the DAG.

- LEDA::set< node > get_Cons (node u) const

  Returns the set of consumers of a node.

- LEDA::set< node > get_Cons (node u, int id_regtype) const

  Returns the set of consumers of a node writing inside a register of a given type.

- LEDA::set< int > get_Sources_id () const

  Returns the set of nodes identifiers which are the sources of the DAG.

- LEDA::set< int > get_Targets_id () const

  Returns the set of nodes identifiers which are the targetsof (sinks) the DAG.

- LEDA::set< int > all_nodes_id ()

  Returns the set of nodes identifiers.

- int source_id (edge e) const

  Returns the integer identifier of the edge source.

- int target_id (edge e) const

  Returns the integer identifier of the edge target.

- ARCHITECTURE get_ISA () const
  Returns the ARCHITECTURE of the DAG.
- int delta (edge e) const
  Returns the latency $\delta(e)$ of the edge e.
- int latency (node n) const
  Returns the latency $latency(n)$ of the generic instruction of the node n.
- int delta_r (node n) const
  Returns the reading latency $\delta_r(n)$ of the node n.
- int delta_r (int idn) const
  Returns the reading latency $\delta_r(idn)$ of the node given by its id.
- int delta_w (node n) const
  Returns the writing latency $\delta_w(n)$ of the node n.
- int delta_w (int idn) const
  Returns the writing latency $\delta_w(idn)$ of the node given by its id.
- int delta_w (node n, int regtype_id) const
  Returns the writing latency $\delta_{w,t}(n)$ of the node n.
- int delta_w (int idn, int regtype_id) const
  Returns the writing latency $\delta_{w,t}(idn)$ of the node given by its id.
- int id (node n) const
  Returns the integer identifier of the node n.
- INSTRUCTIONS_TYPES instruction_type (node n) const
  Returns an INSTRUCTIONS_TYPES object attached to the node n.
- INSTRUCTIONS_TYPES instruction_type (int id) const
  Returns an INSTRUCTIONS_TYPES object attached to the node given by its id.
- std::string get_string_attribute (node n) const
  Returns the textual attribute of the node n.
- const INFO_NODE & inf (node v) const
  Returns the information of node v.
- const INFO_NODE & operator[ ] (node v) const
  Returns a reference to the information of v.
- INFO_NODE & operator[ ] (node v)
- const INFO_EDGE & inf (edge e) const
  Returns the information of node e.

- const INFO_EDGE & operator[ ] (edge e) const
  Returns a reference to the information of e.
- INFO_EDGE & operator[ ] (edge e)
- int outdeg (node v) const
  Returns the number of edges leaving to node v.
- int indeg (node v) const
  Returns the number of edges entering v.
- int degree (node v) const
  Returns $outdeg(v) + indeg(v)$.
- node source (edge e) const
  Returns the source node of edge e.
- node target (edge e) const
  Returns the target node of edge e.
- node opposite (node v, edge e) const
  Returns $target(e)$ if $v = source(e)$ and $source(e)$ otherwise.
- node opposite (edge e, node v) const
  Returns $target(e)$ if $v = source(e)$ and $source(e)$ otherwise.
- int number_of_nodes () const
  Returns the number of nodes in the DAG.
- int number_of_edges () const
  Returns the number of edges in the DAG.
- const LEDA::list< node > & all_nodes () const
  Returns the list of all nodes in the DAG.
- const LEDA::list< edge > & all_edges () const
  Returns the list of all edges in the DAG.
- LEDA::list< edge > out_edges (node v) const
  Returns the list of edges leaving the node v.
- LEDA::list< edge > in_edges (node v) const
  Returns the list of edges entering the node v.
- node first_node () const
  Returns the first node in the DAG.
- node last_node () const
  Returns the last node in the DAG.

- node choose_node () const
  Returns a random node of the DAG (nil if empty).
- node succ_node (node v) const
  Returns the successor of node v (nil if it does not exist).
- node pred_node (node v) const
  Returns the predecessor of node v (nil if it does not exist).
- edge first_edge () const
  Returns the first edge in the DAG.
- edge last_edge () const
  Returns the last edge in the DAG.
- edge choose_edge () const
  Returns a random edge of the DAG (nil if empty).
- edge succ_edge (edge e) const
  Returns the successor of edge e (nil if it does not exist).
- edge pred_edge (edge e) const
  Returns the predecessor of edge e (nil if it does not exist).
- bool empty () const
  Returns true if the DAG is empty.
- bool member (node v) const
  Returns true if the node v belongs to the DAG.
- bool member (edge e) const
  Returns true if the edge e belongs to the DAG.
- bool is_hidden (edge e) const
  Returns true if e is hidden, false otherwise. See DAG::hide_edge(edge e).
- bool is_hidden (node v) const
  Returns true if v is hidden, false otherwise. See DAG::hide_node(node v).
- LEDA::list< edge > hidden_edges () const
  Returns the list of all hidden edges.
- LEDA::list< node > hidden_nodes () const
  Returns the list of all hidden nodes.
- int max_delta (node u, node v) const
  Returns the maximal edge latency between to nodes, i.e., $\max_{e=(u,v)} \delta(e)$.

- int max_delta (int u, int v) const

  Returns the maximal edge latency between to nodes (given by their ids), i.e., $\max_{e=(u,v)} \delta(e)$.

- int min_delta (node u, node v) const

  Returns the minimal edge latency between to nodes, i.e., $\min_{e=(u,v)} \delta(e)$.

- int min_delta (int u, int v) const

  Returns the minimal edge latency between to nodes (given by their ids), i.e., $\min_{e=(u,v)} \delta(e)$.

- INSTRUCTIONS_TYPES search_instruction_type (const std::string opcode) const

  Looks for the instruction type (inside the DAG ISA) that has the opcode given as argument. If not found, it returns an INSTRUCTIONS_TYPES object with opcode_id equal to -1.

- INSTRUCTIONS_TYPES search_instruction_type (int opcode_id) const

  Looks for the instruction type (inside the DAG ISA) that has the opcode id given as argument. Returns NULL if not found. If not found, it returns an INSTRUCTIONS_TYPES object with opcode_id equal to -1.

- bool check ()

  A simple check method that returns true if the DAG object seems ok. False otherwise.

- bool check (int regtype_id)

  A simple check method that returns true if the DAG object seems ok for the specified register type. False otherwise.

- NodeFlag node_flag (node n) const

  Returns the node flag of n.

Update Operations

Returns the list of register types of the ISA of the DAG. The ISA object should be attached to the DAG before calling this function. See the DAG construtor method, or the DDG:DAG:set_ISA method. LEDA::list<REGISTER_TYPES> T() const ;

/**

- void clear ()

  Empty the DAG.

- int new_node (int it_id)

  Creates a new node.

- int new_node (int it_id, std::string text_inst)

  Creates a new node.

- int new_node (int it_id, int n_id)

Creates a new node.

- int new_node (int it_id, int n_id, std::string text_inst)
  Creates a new node.
- void del_node (node)
  Deletes the node given as argument.
- void del_node (int node_id)
  Deletes the node whose id is given as argument.
- edge new_edge (int s, int t, INFO_EDGE ie)
  Creates a new edge.
- edge new_edge (node s, node t, INFO_EDGE ie)
  Creates a new edge.
- void del_edge (edge e)
  Deletes the edge given as argument.
- void copy (const DAG &G2)
  Duplicates a DAG.
- void copy (const DAG &G2, node_map< node > &mn, edge_map< edge > &me)
  Duplicates a DAG.
- void get_loop_body (const LOOP &G2, node_map< node > &mn, edge_map< edge > &me)
  Retrieves a loop body.
- void get_loop_body (const LOOP &G2)
  Retrieves a loop body.
- void build_internal_structures ()
  Build internal data structures for the DAG.
- void set_ISA (ARCHITECTURE isa)
  Sets the set of generic instruction types (ISA) encapsulated inside the DAG.
- void set_delta (edge e, int l)
  Sets a latency $\delta(e)$ to the edge e.
- void set_instruction_type (node n, int it_id)
  Sets an INSTRUCTIONS_TYPES object (given by its opcode id) attached to the node n.
- void set_string_attribute (node u, std::string inst_text)
  Sets a textual attribute to the node n. For instance, this textual attribute can be used to associate the textual code of the instruction.

- void set_etype (edge e, DDG::edge_type et)
  Sets an edge type (flow, serial, antidep, etc.).
- void hide_edge (edge e)
  Removes temporarily the edge e. The edge can be restored by DAG::restore_edge(edge e).
- void hide_edges (const LEDA::list< edge > &el)
  Hides all edges in the list el.
- void restore_edge (edge e)
  Restores the hidden edge e .
- void restore_edges (const LEDA::list< edge > &el)
  Restores all edges in the list el.
- void restore_all_edges ()
  Restores all hidden edges.
- void hide_node (node v)
  Removes temporarily the node v. Leaving and entering edges are also hidden. The node can be restored by DAG::restore_node(node v).
- void hide_nodes (const LEDA::list< node > &nl)
  Hides all nodes in the list nl.
- void hide_node (node v, LEDA::list< edge > &h_edges)
  Removes temporarily the node v. Leaving and entering edges are also hidden. The node can be restored by DAG::restore_node(node n).
- void restore_node (node v)
  Restores the hidden node v. Note that no edge adjacent to v that was hidden by G.hide_node(v) is restored by this operation.
- void restore_all_nodes ()
  Restores all hidden nodes. Note that no adjacent edges to the hidden nodes are restored by this operation.
- void del_all_edges ()
  Deletes all edges from the DAG.
- void move_edge (edge e, node v, node w)
  Moves the edge e to source v and target w.
- void set_node_flag (node n, const NodeFlag theValue)
  Set the flag of the node n.

Import and Export Methods

- void write_to_gl (const char *filename) const

Exports the DAG to a gl file.

- void write_to_gl (const char *filename, int regtype_id) const
  Exports the DAG to a gl file.

- bool read_from_gl (const char *filename)
  Imports the DAG from a gl file when a unique register type exists in the architecture.

- bool read_from_gl (const char *filename, int regtype_id)
  Imports the DAG from a gl file when multiple registers types exist in the architecture. This function asks to fix which register type to consider.

- void write_to_vcg (const char *filename) const
  Exports the DAG to a vcg file to in order to be visualized with the xvcg tool. The user can navigate using vcg on all DDG attributes (types of edges, opcodes, etc.).

- void write_to_vcg (const char *filename, int t) const
  Exports the DAG to a vcg file to in order to be visualized with the xvcg tool while considering one register type. The user can navigate using vcg on all DDG attributes (types of edges, opcodes, etc.) For more information about xvcg, see `http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html`.

- void write_to_xml (const char *filename)
  Exports the DAG to an XML description.

- bool read_from_xml (const char *filename)
  Imports the DAG from its xml description.

Protected Attributes

- unsigned int cpt_nodes
- LEDA::d_array< int, LEDA::set< node > > V_R
- LEDA::d_array< int, LEDA::set< edge > > E_R
- ARCHITECTURE ISA
- node_array< set< node > > down
- node_array< set< node > > up
- node_array< int > longuest_path_from
- node_array< int > longuest_path_to
- node_matrix< int > lp_array
- node_array< int > shortest_path_from
- node_array< int > shortest_path_to
- set< node > Sources
- set< node > Targets
- map2< node, int, set< node > > Cons
- LEDA::map< int, node > node_index_table

3.6.2 Constructor & Destructor Documentation

3.6.2.1 DAG ()

Declares an empry DAG.

Default constructor.

It creates an empty DAG. A default architecture (ISA) is assumed, but the user may attach another ISA to the DAG. See DDG::DAG::set_ISA

3.6.2.2 DAG (const *ARCHITECTURE*)

Constructs a DAG after reading the generic description of the architecture (ISA).

Parameters:

← ***ARCHITECTURE*** The ARCHITECTURE (ISA) object containing the ISA description.

3.6.3 Member Function Documentation

3.6.3.1 LEDA::set<edge> edges_from_to (int *id_u*, int *id_v*) const

Returns the set of all edges from the source node to the destination node.

Parameters:

← ***id_u*** The integer identifier of the source node.

← ***id_v*** The integer identifier of the destination node.

3.6.3.2 int LongestPath (LEDA::list< node > & *nl*) const

Returns the value of the longest path of the DAG.

Parameters:

→ ***nl*** Contains the list of nodes belonging to the computed longest path.

Warning:

Despite the unicity of its value, the longest path in a DAG may not be unique.

3.6.3.3 int LongestPath (list< edge > & *el*) const

Returns the value of the longest path of the DAG.

Parameters:

→ ***el*** Contains the list of edges belonging to the computed longest path.

Warning:

Despite the unicity of its value, the longest path in a DAG may not be unique.

3.6.3.4 int CriticalPath (LEDA::list< node > & *nl*)

Returns the value of the longest (critical) path.

Parameters:

→ ***nl*** Contains the list of nodes belonging to the computed longest path.

Warning:

The longest path in a DAG may not be unique

3.6.3.5 int CriticalPath (LEDA::list< edge > & *el*)

Returns the value of the longest (critical) path.

Parameters:

→ ***el*** Contains the list of edges belonging to the computed longest path.

Warning:

The longest path in a DAG may not be unique

3.6.3.6 int lp (node *u*, node *v*) const

Returns the value of the longest path from a node u to a node v given as arguments$*$.

Precondition:

Such path exists

3.6.3.7 node node_with_id (int *i*) const

Returns the node with the integer identifier given as argument.

Exceptions:

IllegalAccessNode If the node id does not exist, or if the node is hidden, then an exception of type DDG:IllegalAccessNode is thrown.

Referenced by DAG::max_delta(), and DAG::min_delta().

3.6.3.8 node node_with_name (std::string *name*) const

Returns the node with the name given as argument.

Exceptions:

IllegalAccessNode If the node id does not exist, or if the node is hidden.

3.6.3.9 bool is_value (node *n*) const

Returns true if the node is an instruction writing into a register (i.e. if it belongs to the V_R set).

Precondition:

There is a unique register type in the architecture

Exceptions:

NonUniqueRegisterType

3.6.3.10 bool is_value (int *i*) const

Returns true if the node defined by its indentifier is an instruction writing into a register (i.e. if it belongs to the V_R set).

Precondition:

There is a unique register type in the architecture

Exceptions:

NonUniqueRegisterType

3.6.3.11 bool is_flow_reg (edge *e*) const `[inline]`

Returns true if the edges is a flow dependence through a register. This is the case of a flow edge having a source writing into a register.

Precondition:

There is a unique register type in the architecture

Exceptions:

NonUniqueRegisterType

Definition at line 383 of file DAG.h.

References ARCHITECTURE::get_the_unique_register_type_id(), DAG::inf(), INFO_EDGE::is_flow_reg(), and DAG::ISA.

3.6.3.12 int get_register_type_id (edge *e*) const

Returns the register type of a flow edge.

Precondition:

The edge should be a flow through register (flowdep_reg)

3.6.3.13 set<node> get_V_R () const [inline]

Returns the set $V_R \subseteq V$ of instructions (nodes) writing into registers.

Precondition:

a unique register type should exist in the architecture

Exceptions:

NonUniqueRegisterType

Examples:

dag_example.cpp, loop_example.cpp, and RS_example.cpp.

Definition at line 408 of file DAG.h.

References ARCHITECTURE::get_the_unique_register_type_id(), DAG::ISA, and DAG::V_R.

3.6.3.14 set<node> get_V_R (int *t*) const [inline]

Returns the set $V_{R,t} \subseteq V$ of instructions (nodes) writing into registers of type t (given by its id).

Exceptions:

BadRegType

Definition at line 414 of file DAG.h.

References ARCHITECTURE::exist_register_type_id(), DAG::ISA, and DAG::V_R.

3.6.3.15 LEDA::set<edge> get_E_R () const [inline]

Returns the set $E_R \subseteq E$ of flow dependence edges through registers.

Precondition:

A unique register type must exist.

Exceptions:

NonUniqueRegisterType

Examples:

loop_example.cpp.

Definition at line 426 of file DAG.h.

References DAG::E_R, ARCHITECTURE::get_the_unique_register_type_id(), and DAG::ISA.

3.6.3.16 LEDA::set<edge> get_E_R (int *t*) const [inline]

Returns the set $E_{R,t} \subseteq E$ of flow dependence edges through registers of type t (given by its id).

Exceptions:

BadRegType

Definition at line 432 of file DAG.h.

References DAG::E_R, ARCHITECTURE::exist_register_type_id(), and DAG::ISA.

3.6.3.17 LEDA::set<node> get_Cons (node *u*) const [inline]

Returns the set of consumers of a node.

Precondition:

A unique register type should exist in the architecture.

Exceptions:

NonUniqueRegisterType

Definition at line 450 of file DAG.h.

References DAG::Cons, ARCHITECTURE::get_the_unique_register_type_id(), and DAG::ISA.

3.6.3.18 LEDA::set<node> get_Cons (node *u*, int *id_regtype*) const [inline]

Returns the set of consumers of a node writing inside a register of a given type.

Precondition:

id_regtype is an id of register type in the architetcure

Exceptions:

BadRegType

Definition at line 458 of file DAG.h.

References DAG::Cons, ARCHITECTURE::exist_register_type_id(), and DAG::ISA.

3.6.3.19 int delta_w (node *n*) const

Returns the writing latency $\delta_w(n)$ of the node n.

Precondition:

a unique register type should exist in the architecture

Examples:

dag_example.cpp, and loop_example.cpp.

3.6.3.20 int delta_w (int *idn*) const

Returns the writing latency $\delta_w(idn)$ of the node given by its id.

Precondition:

a unique register type should exist in the architecture

3.6.3.21 int new_node (int *it_id*)

Creates a new node.

Parameters:

← ***it_id*** The id of the INSTRUCTIONS_TYPES object describing the generic instruction of the node

Returns:

The integer node identifier.

Remarks:

This method assigns a unique inter identifier for the created node.

See also DAG::build_internal_structures().

Examples:

dag_example.cpp.

3.6.3.22 int new_node (int *it_id*, std::string *text_inst*)

Creates a new node.

Parameters:

← ***it_id*** The id of the INSTRUCTIONS_TYPES object describing the generic instruction of the node

← ***text_inst*** A textual attribute of the instruction

Returns:

The integer node identifier.

Remarks:

This method assigns a unique inter identifier for the created node.

See also DAG::build_internal_structures().

3.6.3.23 int new_node (int *it_id*, int *n_id*)

Creates a new node.

Parameters:

← ***it_id*** The id of the INSTRUCTIONS_TYPES object describing the generic instruction of the node

← ***n_id*** A unique id for the node to create

Returns:

The integer node identifier. Returns -1 if the there exists an node with the input id.

Exceptions:

NonUniqueNodeID if the input node id already exists

Remarks:

This method creates a node with an identifier given as input.

Precondition:

n_id is positive

See also DAG::build_internal_structures().

3.6.3.24 int new_node (int *it_id*, int *n_id*, std::string *text_inst*)

Creates a new node.

Parameters:

← ***it_id*** The id of the INSTRUCTIONS_TYPES object describing the generic instruction of the node

← ***n_id*** A unique id for the node to create

← ***text_inst*** A textual attribute of the instruction

Returns:

The integer node identifier.

Exceptions:

NonUniqueNodeID if the input node id already exists

Remarks:

This method creates a node with an identifier given as input.

Precondition:

n_id is positive

See also DAG::build_internal_structures().

3.6.3.25 void del_node (node)

Deletes the node given as argument.

All the edges incident to the deleted node are removed from the DAG too.

3.6.3.26 void del_node (int *node_id*)

Deletes the node whose id is given as argument.

All the edges incident to the deleted node are removed from the DAG too.

See also DAG::build_internal_structures().

3.6.3.27 edge new_edge (int *s*, int *t*, INFO_EDGE *ie*)

Creates a new edge.

It takes three arguments. If flow dependence, the inserted edge is added to the set E_R.

Parameters:

← ***s*** The integer identifier of the source node.

← ***t*** The integer identifier of the destination node.

← ***ie*** The INFO_EDGE object describing the edge attribute. See also DAG::build_internal_structures().

Examples:

dag_example.cpp.

3.6.3.28 edge new_edge (node *s*, node *t*, INFO_EDGE *ie*)

Creates a new edge.

It takes three arguments. If flow dependence, the inserted edge is added to the set E_R.

Parameters:

← ***s*** The source node.

← ***t*** The destination node.

← ***ie*** The INFO_EDGE object describing the edge attribute. See also DAG::build_internal_structures().

3.6.3.29 void copy (const DAG & *G2*)

Duplicates a DAG.

Parameters:

← ***G2*** The current DAG object will contain a duplicated copy of G2. This method will keep the same node ids, same edge and node information, etc. The internal consistency of the duplicated DAG is guaranteed.

Remarks:

DAG::build_internal_structures() is called inside this method.

Examples:

dag_example.cpp.

3.6.3.30 void copy (const DAG & *G2*, node_map< node > & *mn*, edge_map< edge > & *me*)

Duplicates a DAG.

Parameters:

← ***G2*** The current DAG object will contain a duplicated copy of G2. This method will keep the same node ids, same edge and node information, etc. The internal consistency of the duplicated DAG is guaranteed.

→ ***mn*** mn is a mapping from the set of nodes of G2 to the set of nodes of the current DAG. $\forall u \in G_2, mn[u]$ contains the node belonging to the current DAG corresponding to its original copy $u \in G_2$.

→ ***me*** me is a mapping from the set of edges of G2 to the set of edges of the current DAG. $\forall e \in G_2, me[e]$ contains the edge belonging to the current DAG corresponding to its original copy $e \in G_2$.

Remarks:

DAG::build_internal_structures() is called inside this method.

3.6.3.31 void get_loop_body (const LOOP & *G2*, node_map< node > & *mn*, edge_map< edge > & *me*)

Retrieves a loop body.

Parameters:

← ***G2*** The current DAG object will contain the loop body of G2. This method will keep the same node ids, same edge and node information, etc. The internal consistency of the DAG is guaranteed.

→ ***mn*** mn is a mapping from the set of nodes of G2 to the set of nodes of the current DAG. $\forall u \in G_2, mn[u]$ contains the node belonging to the current DAG corresponding to its original copy $u \in G_2$.

→ ***me*** me is a mapping from the set of edges of G2 to the set of edges of the current DAG. $\forall e \in G_2, me[e]$ contains the edge belonging to the current DAG corresponding to its original copy $e \in G_2$.

Remarks:

DAG::build_internal_structures() is called inside this method. The loop body contains the set of all nodes, but the set of edges contains only the edges with null distance: $\forall e$ in the loop body, $\lambda(e) = 0$.

3.6.3.32 void get_loop_body (const LOOP & *G2*)

Retrieves a loop body.

Parameters:

← ***G2*** The current DAG object will contain the loop body of G2. This method will keep the same node ids, same edge and node information, etc. The internal consistency of the duplicated DAG is guaranteed.

Remarks:

DAG::build_internal_structures() is called inside this method. The loop body contains the set of all nodes, but the set of edges contains only the edges with null distance: $\forall e$ in the loop body, $\lambda(e) = 0$.

3.6.3.33 void build_internal_structures ()

Build internal data structures for the DAG.

For internal information consistency, this function member should be called after update operations such as DAG::new_node(INSTRUCTIONS_TYPES it), DAG::new_edge(INFO_EDGE), etc. This method compute some internal information returned by other methods (the set V_R, the set of consumers per value, all pair shortest paths, transitive closure, set of descendant and ascendant nodes, set of values, etc.). If the DAG is not modified after calling build_internal_structures(), all its internal information remain consistent.

Remarks:

For efficiency, the user can group series of update operations before ultimately calling build_internal_structures()

Precondition:

The graph should not contain a circuit.

Reimplemented in LOOP.

Examples:

dag_example.cpp.

3.6.3.34 void set_ISA (ARCHITECTURE *isa*) [inline]

Sets the set of generic instruction types (ISA) encapsulated inside the DAG.

Parameters:

← ***isa*** An ARCHITECTURE object describing a user-defined ISA.

Warning:

It is better to clear the DAG before changing its ISA. If the ISA of a non empty DAG is modified on-the-fly, the successive behavior of the DAG object would be indefined.

Definition at line 886 of file DAG.h.

3.6.3.35 void set_string_attribute (node *u*, std::string *inst_text*)

Sets a textual attribute to the node n. For instance, this textual attribute can be used to associate the textual code of the instruction.

Remarks:

- Currently, a textual attribute is a single textual line. Hence, it should not contain the newline character \n. If they exist, the current implementation replaces them by white spaces.
- Double quotes characters " should be specialized. They should be written as \" inside the string attribute.

3.6.3.36 void hide_edge (edge *e*)

Removes temporarily the edge e. The edge can be restored by DAG::restore_edge(edge e).

Remarks:

A hidden edge does no longer belong to the list of graph edges

3.6.3.37 void hide_edges (const LEDA::list< edge > & *el*)

Hides all edges in the list el.

Remarks:

A hidden edge does no longer belong to the list of graph edges

3.6.3.38 void restore_edge (edge *e*)

Restores the hidden edge e .

Warning:

If an endpoint of e is hidden, an exception occurs.

3.6.3.39 void restore_edges (const LEDA::list< edge > & *el*)

Restores all edges in the list el.

Warning:

If an endpoint of an edge is hidden, an exception occurs.

3.6.3.40 void restore_all_edges ()

Restores all hidden edges.

Warning:

If an endpoint of an edge is hidden, an exception occurs.

Examples:

dag_example.cpp, and loop_example.cpp.

3.6.3.41 void hide_node (node *v*)

Removes temporarily the node v. Leaving and entering edges are also hidden. The node can be restored by DAG::restore_node(node v).

Remarks:

A hidden node does no longer belong to the list of graph nodes

Examples:

dag_example.cpp, and loop_example.cpp.

3.6.3.42 void hide_nodes (const LEDA::list< node > & *nl*)

Hides all nodes in the list nl.

Remarks:

A hidden node does no longer belong to the list of graph nodes

3.6.3.43 void hide_node (node *v*, LEDA::list< edge > & *h_edges*)

Removes temporarily the node v. Leaving and entering edges are also hidden. The node can be restored by DAG::restore_node(node n).

Parameters:

← ***v*** the node to hide.

→ ***h_edges*** the list of leaving and entering edges hidden in consequence of hiding the node v.

Remarks:

A hidden node does no longer belong to the list of graph nodes

3.6.3.44 void del_all_edges ()

Deletes all edges from the DAG.

See also DAG::empty();

3.6.3.45 void write_to_gl (const char * *filename*) const

Exports the DAG to a gl file.

Deprecated

Parameters:

← ***filename*** the file name of the DDG in gl format

Remarks:

if multiple registers types exist in the architecture, then the output gl file will consider them as the same register type, because the current gl format does not support multiple register files.

Reimplemented in LOOP.

3.6.3.46 void write_to_gl (const char * *filename*, int *regtype_id*) const

Exports the DAG to a gl file.

Deprecated

Parameters:

← ***filename*** the file name of the DDG in gl format

← ***regtype_id*** An id of a register type to consider for the flow_reg dependances in the gl file. Only flow edges through registers of type regtype_id will be output as flow_reg. Other flow of other register types are considered as serial.

Reimplemented in LOOP.

3.6.3.47 bool read_from_gl (const char * *filename*)

Imports the DAG from a gl file when a unique register type exists in the architecture.

Deprecated

Parameters:

← ***filename*** the file name of the DDG in gl format

Returns:

true if OK.

Remarks:

- To get the DAG from a cyclic data dependence graph, all edges with non null distances are ignored. That is, only the loop body is considered when importing a cyclic data dependence graph from a gl file.

- The set E_R of flow edges through registers is constructed for the unique register type in the architecture. -DAGbuild_internal_structures() is called inside this method.

Exceptions:

BadIODDG If an IO error occurs (corrupted or non present input files for instance), an exception object of type DDG::BadIODDG is thrown

Precondition:

Before reading a DAG from a gl file, it is assumed that the ISA of the DAG defines all the opcodes ids of the nodes as present in the gl file. Also, it is assumed that the ISA has a unique register type (this is a restriction of the gl format). The constructor DDG::DAG(ARCHITECTURE) defines a DAG with a user defined ISA, otherwise a default ISA is assumed, see ISA_xml.

Reimplemented in LOOP.

3.6.3.48 bool read_from_gl (const char * *filename*, int *regtype_id*)

Imports the DAG from a gl file when multiple registers types exist in the architecture. This function asks to fix which register type to consider.

Deprecated

Parameters:

← ***filename*** the file name of the DDG in gl format

← ***regtype_id*** The id of the register type to consider for flow_reg dependances.

Returns:

true if OK.

Remarks:

- To get the DAG from a cyclic data dependence graph, all edges with non null distances are ignored. That is, only the loop body is considered when importing a cyclic data dependence graph from a gl file.

- The set E_R of flow edges through registers is constructed for the register type given as parameter. -DAGbuild_internal_structures() is called inside this method.

Exceptions:

BadIODDG If an IO error occurs (corrupted or non present input files for instance), an exception object of type DDG::BadIODDG is thrown

Precondition:

Before reading a DAG from a gl file, it is assumed that the ISA of the DAG defines all the opcodes ids of the nodes as present in the gl file. The constructor DDG::DAG(ARCHITECTURE) defines a DAG with a user defined ISA, otherwise a default ISA is assumed, see User Defined Instruction Set Architectures (XML format).

Reimplemented in LOOP.

3.6.3.49 void write_to_vcg (const char * *filename*) const

Exports the DAG to a vcg file to in order to be visualized with the xvcg tool. The user can navigate using vcg on all DDG attributes (types of edges, opcodes, etc.).

Parameters:

← ***filename*** the file name of the DDG in vcg format For more information about xvcg, see http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html

Reimplemented in LOOP.

Examples:

dag_example.cpp.

3.6.3.50 void write_to_vcg (const char * *filename*, int *t*) const

Exports the DAG to a vcg file to in order to be visualized with the xvcg tool while considering one register type. The user can navigate using vcg on all DDG attributes (types of edges, opcodes, etc.) For more information about xvcg, see http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html.

Parameters:

← ***filename*** the file name of the DDG in vcg format

← ***t*** the id of the considered register type

Reimplemented in LOOP.

3.6.3.51 void write_to_xml (const char * *filename*)

Exports the DAG to an XML description.

All DDG attributes are included into this XML graph format.

3.6.3.52 bool read_from_xml (const char * *filename*)

Imports the DAG from its xml description.

Returns:

True if OK. False if problem.

Reimplemented in LOOP.

Examples:

dag_example.cpp, and RS_example.cpp.

3.7 EN Class Reference

3.7.1 Detailed Description

Definition at line 75 of file RS.h.

Data Fields

- node t
- rational rho

Friends

- istream & operator>> (istream &is, EN &x)
- ostream & operator<< (ostream &os, const EN &x)

3.8 IllegalAccessNode Class Reference

3.8.1 Detailed Description

Definition at line 44 of file ddg_exceptions.h.

Public Member Functions

- IllegalAccessNode (int i)

3.9 IllegalDependance Class Reference

3.9.1 Detailed Description

Definition at line 89 of file ddg_exceptions.h.

Public Member Functions

- IllegalDependance ()

3.10 INFO_EDGE Class Reference

3.10.1 Detailed Description

This class represents the attributes (information) attached to an edge in a DDG.

Author:

Sid-Ahmed-Ali Touati

Each edge e (data dependence or serial edge) has basically three attributes :

- Its type (flow, serial, antidep, ...)
- Its integer latency : $\delta(e)$

- Its distance in terms of loop iterations (equal to 0 in case of a DAG) : : $\lambda(e)$

Examples:

loop_example.cpp.

Definition at line 84 of file info_edge.h.

Public Member Functions

Creation

- INFO_EDGE ()
  Constructor by default.
- INFO_EDGE (edge_type et)
  Constructor with edge type.
- INFO_EDGE (int dur, int dis, edge_type et)
  Constructor with latency, distance and edge type.
- INFO_EDGE (int dur, edge_type et)
  Constructor with latency and edge type.
- **INFO_EDGE** (const INFO_EDGE &ie)

Access Operations

- bool is_flow () const
  Returns true if the edge is a flow dependence (throug registers or memory).
- edge_type etype () const
  Returns the type of the edge (flow, serial, antidep, ...).
- int delta () const
  Returns the latency of the edge.
- int lambda () const
  Returns the distance of the edge in terms of iterations.
- bool check ()
  A simple check method that returns true if the object seems ok.
- int get_register_type_id () const
  Returns the register type id of the flowdep_reg edge.
- bool is_flow_reg (int regtypeid) const
  Returns true if the edge is a flowdep_reg on the given register type.

- bool is_flow_reg () const

  Returns true if the edge is a flowdep_reg.

- bool is_dep_reg () const

  Returns true if the edge is a dependence on a register (all type of dependences on registers).

Update Operations

- void set_etype (const edge_type &et)

  Sets the edge type.

- bool set_register_type_id (int regtypeid)

  set a register type for the edge if it is a dependence through registers.

- void set_delta (int val)

  Sets the edge latency.

- void set_lambda (int lambda)

  Sets the edge distance.

Friends

Input/Output stream operations

- ostream & operator<< (ostream &os, INFO_EDGE x)
- istream & operator>> (istream &is, INFO_EDGE &x)

Related Functions

(Note that these are not member functions.)

- bool operator== (const INFO_EDGE, const INFO_EDGE)

3.10.2 Constructor & Destructor Documentation

3.10.2.1 INFO_EDGE ()

Constructor by default.

It sets the latency and the distance to 0 and the edge type to serial

3.10.2.2 INFO_EDGE (edge_type *et*)

Constructor with edge type.

The latency and the distance are set to 0.

3.10.2.3 INFO_EDGE (int *dur*, edge_type *et*)

Constructor with latency and edge type.

The distance is set to 0.

3.10.3 Friends And Related Function Documentation

3.10.3.1 ostream& operator<< (ostream & *os*, INFO_EDGE *x*) [friend]

Output stream of the edge attribute. Prints $\delta(e)\lambda(e)$ and its edge_type as a string ("flow", "serial, ...).

3.10.3.2 istream& operator>> (istream & *is*, INFO_EDGE & *x*) [friend]

Input stream of the edge attribute. Reads $\delta(e)\lambda(e)$ and its edge_type as a string ("flow", "serial, ...).

3.10.3.3 bool operator== (const *INFO_EDGE*, const *INFO_EDGE*) [related]

Returns True if two edge attributes are assumed equal : they should have the same latency (δ)) and the same distance (λ).

3.11 INFO_NODE Class Reference

3.11.1 Detailed Description

This class represents the attributes (information) attached to each node in a DDG.

Author:

Sid-Ahmed-Ali Touati

Each node (instruction) in a DDG has bsically two attributes :

- an C++ INSTRUCTIONS_TYPES object describing its generic instruction.
- an integer identifier that uniquely defines the node.

In order to avoid possible bugs, the user should not try to set the internal integer node identifier. Such unique identifier is automatically created internally by the DDG library when creating a node.

Definition at line 83 of file info_node.h.

Public Member Functions

Creation

- INFO_NODE ()
  Default constructor. It constructs a node attribute with an instruction type "add_op" and and integer identifier equal to 0.

- INFO_NODE (int id, INSTRUCTIONS_TYPES opt)
- **INFO_NODE** (const INFO_NODE &in)
- INFO_NODE (int id, INSTRUCTIONS_TYPES opt, std::string inst_text)

Access Operations

- bool check ()
  Returns OK if the object seems ok.

- const int id () const
  Returns the integer identifier of the node attribute.

- const INSTRUCTIONS_TYPES instruction_type () const
  Returns the instruction type of the node attribute.

- std::string get_string_attribute () const
  Returns the textual attribute of the node.

- int instruction_type_id () const
  Returns the id of tge instruction type of the node.

- NodeFlag get_node_flag () const
  Returns the flag of the node.

Update Operations

- void set_instruction_type (INSTRUCTIONS_TYPES opt)
  Sets an INSTRUCTIONS_TYPES object in the node attribute.

- void set_string_attribute (std::string inst_text)
  Sets a textual attribue to the node. For instance, this textual attribute can be used to associate the textual code of the instruction.

- void set_node_flag (const NodeFlag theValue)
  Set the flag of the node.

Friends

Input/Output stream operations

- ostream & operator<< (ostream &os, INFO_NODE x)
  Output stream of the node attributes in the form : node_id INSTRUCTIONS_TYPES_attributes.

- istream & operator>> (istream &is, INFO_NODE &x)
  Input stream of the node attribute. Reads the node id and its INSTRUCTIONS_TYPES attribute.

3.11.2 Constructor & Destructor Documentation

3.11.2.1 INFO_NODE (int *id*, INSTRUCTIONS_TYPES *opt*)

Constructor with two arguments

Parameters:

← ***id*** An integer identifier.

← ***opt*** The INSTRUCTIONS_TYPES object describing the generic instruction of the node.

3.11.2.2 INFO_NODE (int *id*, INSTRUCTIONS_TYPES *opt*, std::string *inst_text*)

Constructor with three arguments

Parameters:

← ***id*** An integer identifier.

← ***opt*** The INSTRUCTIONS_TYPES object describing the generic instruction of the node.

← ***inst_text*** The textual representation of the instruction

3.11.3 Member Function Documentation

3.11.3.1 void set_string_attribute (std::string *inst_text*) `[inline]`

Sets a textual attribue to the node. For instance, this textual attribute can be used to associate the textual code of the instruction.

Remarks:

- Currently, a textual attribute is a single textual line. Hence, it should not contain the newline character \n. If such character exists in the string, the current implementation of the method replaces it by a white space.
- Double quotes characters " should be specialized. They should be written as \" inside the string attribute.

Definition at line 162 of file info_node.h.

3.12 INSTRUCTIONS_TYPES Class Reference

3.12.1 Detailed Description

This class describes a generic instruction, used by the class ARCHITECTURE to describe an ISA. A generic instruction object contains opcode, unique id per opcode, string opcode, list of writtent registers, list of written register types, etc.

Author:

Sid-Ahmed-Ali Touati

Each node in a data dependence graph corresponds to an instruction instance or to an operation. The generic instruction defined by the ISA is defined by the class INSTRUCTIONS_TYPES. A generic instruction is defined by a unique id, an opcode (of type string) and a flag saying if the instances of such generic instruction write into a register.

To each node in a DAG or in a LOOP, we associate an attribute of type INSTRUCTIONS_TYPES describing its generic instruction description. A generic instruction has the following attributes:

- an integer latency
- an integer unique opcode id
- the string of the opcode (such as "add_op", "sub_op", etc.)
- a boolean flag saying wether the instruction writes into a register (as destination operand)
- an integer latency for reading delay from registers (δ_r)
- an integer latency for writing delay into registers (δ_w)

The user can construct its own ISA as explained in User Defined Instruction Set Architectures (XML format).

Examples:

dag_example.cpp, loop_example.cpp, and RS_example.cpp.

Definition at line 62 of file instructions_types.h.

Public Member Functions

Creation

- INSTRUCTIONS_TYPES ()

  Constructor by default. Build a generic instruction of type "add_op" by default, and gives it a null opcode id, and sets a flag saying that such instruction writes into a register. Reading and writing delays are set to zero. Latency set to 1.

- ~INSTRUCTIONS_TYPES ()
- INSTRUCTIONS_TYPES (std::string sopcode, int id)
- INSTRUCTIONS_TYPES (const char *sopcode, int id)
- INSTRUCTIONS_TYPES (const char *sopcode, int id, int latency)
- INSTRUCTIONS_TYPES (std::string sopcode, int id, int latency)
- **INSTRUCTIONS_TYPES** (const INSTRUCTIONS_TYPES &it)

Access Operations

- int opcode_id () const

  Returns the integer id that defines the instruction type.

- int delta_r () const

  Returns δ_r the architecturally visible latency of reading from a register (in processor clock cycles).

- int delta_w (int regtype_id) const

  Returns δ_w the architecturally visible latency of writing into a register (in processor clock cycles).

- int latency () const

  Returns the latency of the instruction type (in processor clock cycles).

- std::string opcode () const

  Returns the string opcode of the instruction type.

- const char * opcode_str () const

  Returns the string opcode of the instruction type.

- bool is_value () const

  Returns true if instruction type writes into a register of any type.

- leda::list< int > get_list_reg_id ()

  Returns the list of the id of the register types written by this generic instruction.

- leda::list< int > get_list_regtypes_id () const

  Same as get_list_reg_id().

- int nb_write_reg () const

  Returns the number of written register types.

- bool is_value (int t) const

  Returns true if instruction type writes into a register of type t.

- bool does_write_into_reg () const

  An alias of the method is_value(). Returns true if the instruction type writes into a register.

- bool does_write_into_reg (int regtype_id) const

  Returns true if the instruction type writes into a register of a given type.

- bool check () const

  This method makes a simple check of the consistancy of the object. Returns true if OK, false otherwise.

Update Operations

- void set_opcode_id (const int &theValue)

Sets the opcode id.

- void set_opcode (const std::string &theValue)
  Sets the string opcode of the instruction type.
- void set_delta_r (int dr_value)
  Sets δ_r the architecturally visible latency of reading from a register (in processor clock cycles).
- bool set_delta_w (int regtype_id, int dw_value)
  Sets δ_w the architecturally visible latency of writing into a register (in processor clock cycles).
- void set_latency (int latency_value)
  Sets the latency of the instruction type (in processor clock cycles).
- void add_write_reg_id (int t)
  Add a written register type given by its id.
- void add_write_reg_id (int regtype_id, int deltaw)
  Add a written register type given by its id and sets δ_w.

Friends

Input/Output stream operations

- ostream & operator<< (ostream &os, INSTRUCTIONS_TYPES x)
- istream & operator>> (istream &is, INSTRUCTIONS_TYPES &x)

Related Functions

(Note that these are not member functions.)

- bool operator== (INSTRUCTIONS_TYPES a, INSTRUCTIONS_TYPES b)

3.12.2 Constructor & Destructor Documentation

3.12.2.1 INSTRUCTIONS_TYPES (std::string *sopcode*, int *id*)

Constructor with two arguments.

Parameters:

← ***sopcode*** The string of the opcode, such as "load_op", "store_op", etc.

← ***id*** A (unique) opcode id defining the instruction. Reading and writing delays are set to zero.

3.12.2.2 INSTRUCTIONS_TYPES (const char * *sopcode*, int *id*)

Constructor with two arguments.

Parameters:

← ***sopcode*** The string (of type char *) of the opcode, such as "load_op", "store_op", etc.

← ***id*** A (unique) opcode id defining the instruction. Reading and writing delays are set to zero.

3.12.2.3 INSTRUCTIONS_TYPES (const char * *sopcode*, int *id*, int *latency*)

Constructor with three arguments.

Parameters:

← ***sopcode*** The string (of type char *) of the opcode, such as "load_op", "store_op", etc.

← ***id*** A (unique) opcode id defining the instruction.

← ***latency*** The latency of the instruction type (in processor clock cycles).

3.12.2.4 INSTRUCTIONS_TYPES (std::string *sopcode*, int *id*, int *latency*)

Constructor with three arguments.

Parameters:

← ***sopcode*** The string (of type std::string) of the opcode, such as "ld_op", "st_op", etc.

← ***id*** A (unique) opcode id defining the instruction.

← ***latency*** The latency of the instruction type (in processor clock cycles).

3.12.3 Friends And Related Function Documentation

3.12.3.1 ostream& operator<< (ostream & *os*, INSTRUCTIONS_TYPES *x*) [friend]

Output stream of the INSTRUCTIONS_TYPES attribute. Print all INSTRUCTIONS_TYPES attributes in the form : string_opcode, opcode_id, latency, δ_r, δ_w, does_write_into_reg?

3.12.3.2 istream& operator>> (istream & *is*, INSTRUCTIONS_TYPES & *x*) [friend]

Input stream of the INSTRUCTIONS_TYPES. Read all INSTRUCTIONS_TYPES attributes in the following order : string_opcode, opcode_id, latency, δ_r, δ_w, does_write_into_reg?

3.12.3.3 bool operator== (INSTRUCTIONS_TYPES *a*, INSTRUCTIONS_TYPES *b*) [related]

Returns True if two INSTRUCTIONS_TYPES objects are assumed equal : they are "equal" if they have the same opcode identifier.

3.13 InvalidISADesc Class Reference

3.13.1 Detailed Description

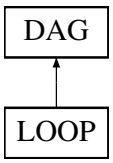
Definition at line 52 of file ddg_exceptions.h.

Public Member Functions

- InvalidISADesc ()

3.14 LOOP Class Reference

Inheritance diagram for LOOP::



3.14.1 Detailed Description

This class represent cyclic data dependence graphs of simple loops (without branches).

Author:

Sid-Ahmed-Ali Touati

It inherits from the LEDA class GRAPH<INFO_NODE, INFO_EDGE>. The attributes of the nodes are of type INFO_NODE, and the attributes of edges are of type INFO_EDGE. It models cyclic data dependence graphs. The implemented LOOP model is described in Loop Model.

All LEDA access, creation and update methods can be used on a LOOP object.

Examples:

dag_example.cpp, gl2vcg.cpp, and loop_example.cpp.

Definition at line 71 of file loop.h.

Public Member Functions

Creation

- LOOP ()
  Default constructor.
- void clear ()
  Empty the DAG.
- LOOP (ARCHITECTURE isa)
  Constructs an empty LOOP object encapsulating a user-defined generic description of the architecture (ISA).
- ~**LOOP** ()

Access Operations

- int MaxLambda () const
  Returns the maximal iteration edge distance in the loop. Formally equal to $\max_{e \in E} \lambda(e)$.
- int MinLambda () const
  Returns the minimal iteration edge distance in the loop. Formally equal to $\min_{e \in E} \lambda(e)$.
- LEDA::set< edge > flow_from_to (node u, node v) const
  Returns the set of all `flow` *edges (not necessarily through registers) from the source node* u *to the destination node* v. *Formally equal to* $\{e = (u, v) \in E | e \text{is a flow edge}\}$.
- LEDA::set< edge > flow_from_to (int id_u, int id_v) const
  Returns the set of all `flow` *edges (not necessarily through registers) from the source node to the destination node. Formally equal to* $\{e = (u, v) \in E | e \text{is a flow edge}\}$.
- LEDA::set< edge > flow_reg_from_to (node u, node v) const
  Returns the set of all `flow` *edges through registers from the source node* u *to the destination node* v. *Formally equal to* $\{e = (u, v) \in E_R\}$.
- LEDA::set< edge > flow_reg_from_to (int id_u, int id_v) const
  Returns the set of all `flow` *edges through registers from the source node to the destination node. Formally equal to* $\{e = (u, v) \in E_R\}$.
- LEDA::set< edge > flow_reg_from_to (node u, node v, int t) const
  Returns the set of all `flow` *edges of type* t *from the source node* u *to the destination node* v. *Formally equal to* $\{e = (u, v) \in E_{R,t}\}$.
- LEDA::set< edge > flow_reg_from_to (int id_u, int id_v, int t) const
  Returns the set of all `flow` *edges through registers from the source node to the destination node. Formally equal to* $\{e = (u, v) \in E_R\}$.

- int max_dist (node u, node v) const

  Returns the maximal distance between two nodes (in terms of number of iterations). Formally equal to $\max_{e=(u,v)\in E} \lambda(e)$.

- int max_dist (int id_u, int id_v) const

  Returns the maximal iteration distance between two nodes. Formally equal to $\max_{e=(u,v)\in E} \lambda(e)$.

- int max_flow_dist (node u, node v) const

  Returns the maximal iteration flow distance (not necessarily through registers) between two nodes. Formally equal to $\max_{a\ flow\ edge\ e=(u,v)\in E} \lambda(e)$.

- int max_flow_dist (int id_u, int id_v) const

  Returns the maximal iteration flow distance (not necessarily through registers) between two nodes. Formally equal to $\max_{a\ flow\ edge\ e=(u,v)\in E} \lambda(e)$.

- int max_flow_reg_dist (node u, node v) const

  Returns the maximal iteration flow distance through registers between two nodes. Formally equal to $\max_{e=(u,v)\in E_R} \lambda(e)$.

- int max_flow_reg_dist (int id_u, int id_v) const

  Returns the maximal iteration flow distance through registers between two nodes. Formally equal to $\max_{e=(u,v)\in E_R} \lambda(e)$.

- int max_flow_reg_dist (node u, node v, int t) const

  Returns the maximal iteration flow distance through registers of type t between two nodes. Formally equal to $\max_{e=(u,v)\in E_{R,t}} \lambda(e)$.

- int max_flow_reg_dist (int id_u, int id_v, int t) const

  Returns the maximal iteration flow distance through registers between two nodes. Formally equal to $\max_{e=(u,v)\in E_{R,t}} \lambda(e)$.

- int min_dist (node u, node v) const

  Returns the minimal iteration distance between two nodes. Formally equal to $\min_{e=(u,v)\in E} \lambda(e)$.

- int min_dist (int id_u, int id_v) const

  Returns the minimal iteration distance between two nodes. Formally equal to $\max_{e=(u,v)\in E} \lambda(e)$.

- int min_flow_dist (node u, node v) const

  Returns the minimal iteration flow distance (not necessarily through registers) between two nodes. Formally equal to $\min_{aflowedgee=(u,v)\in E} \lambda(e)$.

- int min_flow_dist (int id_u, int id_v) const

  Returns the minimal iteration flow distance (not necessarily through registers) between two nodes. Formally equal to $\max_{a\ flow\ edge\ e=(u,v)\in E} \lambda(e)$.

- int min_flow_reg_dist (node u, node v) const

  Returns the minimial iteration flow distance through registers between two nodes. Formally equal to $\min_{e=(u,v)\in E_R} \lambda(e)$.

- int min_flow_reg_dist (int id_u, int id_v) const

  Returns the minimal iteration flow distance through registers between two nodes. Formally equal to $\min_{e=(u,v)\in E_R} \lambda(e)$.

- int min_flow_reg_dist (node u, node v, int t) const

  Returns the minimial iteration flow distance through registers of type t between two nodes. Formally equal to $\min_{e=(u,v)\in E_{R,t}} \lambda(e)$.

- int min_flow_reg_dist (int id_u, int id_v, int t) const

  Returns the minimal iteration flow distance through registers of type t between two nodes. Formally equal to $\min_{e=(u,v)\in E_{R,t}} \lambda(e)$.

- LEDA::rational CRITICAL_CYCLE () const

  Returns the ratio of the critical cycle of the loop.

- LEDA::rational CRITICAL_CYCLE (LEDA::list< edge > &el) const
- int lambda (edge e) const

  Returns the distance $\lambda(e)$ *of the edge* e *(in terms of number of iterations).*

Update Operations

- void copy (const LOOP &G2)

  Duplicates a LOOP.

- void copy (const LOOP &G2, node_map< node > &mn, edge_map< edge > &me)

  Duplicates a LOOP.

- void build_internal_structures ()

  Build internal data structures for the LOOP object.

- void set_lambda (edge e, int l)

  Sets a distance $\lambda(e) \geq 0$ *(in terms of number of iterations) to the edge* e .

- void write_to_vcg (const char *filename) const

  Exports the LOOP DDG to a vcg file in order to be visualized with the xvcg tool. For more information about xvcg, see http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html
  The user can navigate using vcg on all DDG attrbutes (types of edges, opcodes, etc.).

- void write_to_vcg (const char *filename, int t) const

  Exports the LOOP DDG to a vcg file in order to be visualized with the xvcg tool. This function considers one register type.

- void write_to_gl (const char *filename) const

  Exports the LOOP to a gl file. If multiple registers types exist, they are all considered as the same type.

- void write_to_gl (const char *filename, int regtype_id) const
  Exports the LOOP to a gl file considering one register type.
- bool read_from_gl (const char *filename, int regtype_id)
  Imports a DDG loop from a gl file. In case of multiple registers types, this function allows to considere one of them.
- bool read_from_gl (const char *filename)
  Imports a DDG loop from a gl file.
- void write_to_xml (const char *filename) const
  Exports the LOOP to a leda XML format. .
- bool read_from_xml (const char *filename)
  Imports the LOOP from an XML file.
- bool check ()
  A simple check method that returns true if the DAG object seems ok. False otherwise.
- bool check (int regtype_id)
  A simple check method that returns true if the DAG object seems ok for the specified register type. False otherwise.

Access Operations

- LEDA::set< edge > edges_from_to (node u, node v) const
  Returns the set of all edges from the source node u to the destination node v.
- LEDA::set< edge > edges_from_to (int id_u, int id_v) const
  Returns the set of all edges from the source node to the destination node.
- int LongestPath () const
  Returns the value of the longest path of the DAG.
- int LongestPath (LEDA::list< node > &nl) const
  Returns the value of the longest path of the DAG.
- int LongestPath (list< edge > &el) const
  Returns the value of the longest path of the DAG.
- int CriticalPath ()
  This is an alias of the method DAG::LongestPath(). It returns the value of the longest (critical) path.
- int CriticalPath (LEDA::list< node > &nl)
  Returns the value of the longest (critical) path.
- int CriticalPath (LEDA::list< edge > &el)

Returns the value of the longest (critical) path.

- int LongestPathTo (node) const

  Returns the value of the longest path from the top of the DAG (sources) up to a node given as argument.

- int LongestPathFrom (node) const

  Returns the value of the longest path from a node given as argument to the bottom the DAG (sinks).

- int ShortestPathTo (node) const

  Returns the value of the shortest path from the top of the DAG (sources) up to a node given as argument.

- int ShortestPathFrom (node) const

  Returns the value of the shortest path from a node given as argument to the bottom the DAG (sinks).

- int lp (node u, node v) const

  Returns the value of the longest path from a node u to a node v given as arguments∗.

- LEDA::set< node > get_down (node u) const

  Returns the set $\downarrow u$ (the set of all descending nodes)., see Notation and Definitions on DAGs.

- LEDA::set< node > get_up (node u) const

  Returns the set $\uparrow u$.(the set of all ascenfing nodes), see Notation and Definitions on DAGs.

- LEDA::set< node > get_parents (node u) const

  Returns the set $\Gamma_G^+(u)$ (set of children), see Notation and Definitions on DAGs.

- LEDA::set< node > get_children (const node u) const

  Returns the set $\Gamma_G^-(u)$ (set of parents), see Notation and Definitions on DAGs.

- node node_with_id (int i) const

  Returns the node with the integer identifier given as argument.

- node node_with_name (std::string name) const

  Returns the node with the name given as argument.

- bool is_value (node n) const

  Returns true if the node is an instruction writing into a register (i.e. if it belongs to the V_R set).

- bool is_value (int i, int t) const

  Returns true if the node defined by its indentifier is an instruction writing into a register of type t (i.e. if it belongs to the V_R, t set).

- bool is_value (node n, int t) const

Returns true if the node is an instruction writing into a register of type t (i.e. if it belongs to the $V_{R,t}$ set).

- bool is_value (int i) const

  Returns true if the node defined by its indentifier is an instruction writing into a register (i.e. if it belongs to the V_R set).

- bool is_flow (edge e) const

  Returns true if the edges is a flow dependence (not necessarily through registers).

- bool is_flow_reg (edge e) const

  Returns true if the edges is a flow dependence through a register. This is the case of a flow edge having a source writing into a register.

- bool is_flow_reg (edge e, int t) const

  Returns true if the edges is a flow dependence through a register of type t. This is the case of a flow edge having a source writing into a register.

- int get_register_type_id (edge e) const

  Returns the register type of a flow edge.

- DDG::edge_type etype (edge e) const

  Returns the type of the edge e. (flow, serial, antidep, ...).

- set< node > get_V_R () const

  Returns the set $V_R \subseteq V$ of instructions (nodes) writing into registers.

- set< node > get_V_R (int t) const

  Returns the set $V_{R,t} \subseteq V$ of instructions (nodes) writing into registers of type t (given by its id).

- LEDA::list< REGISTER_TYPES > **T** () const
- LEDA::set< edge > get_E_R () const

  Returns the set $E_R \subseteq E$ of flow dependence edges through registers.

- LEDA::set< edge > get_E_R (int t) const

  Returns the set $E_{R,t} \subseteq E$ of flow dependence edges through registers of type t (given by its id).

- LEDA::set< node > get_Sources () const

  Returns the set of nodes which are the sources of the DAG.

- LEDA::set< node > get_Targets () const

  Returns the set of nodes which are the targetsof (sinks) of the DAG.

- LEDA::set< node > get_Cons (node u) const

  Returns the set of consumers of a node.

- LEDA::set< node > get_Cons (node u, int id_regtype) const

  Returns the set of consumers of a node writing inside a register of a given type.

- LEDA::set< int > get_Sources_id () const
  Returns the set of nodes identifiers which are the sources of the DAG.
- LEDA::set< int > get_Targets_id () const
  Returns the set of nodes identifiers which are the targetsof (sinks) the DAG.
- LEDA::set< int > all_nodes_id ()
  Returns the set of nodes identifiers.
- int source_id (edge e) const
  Returns the integer identifier of the edge source.
- int target_id (edge e) const
  Returns the integer identifier of the edge target.
- ARCHITECTURE get_ISA () const
  Returns the ARCHITECTURE of the DAG.
- int delta (edge e) const
  Returns the latency $\delta(e)$ of the edge e.
- int latency (node n) const
  Returns the latency $latency(n)$ of the generic instruction of the node n.
- int delta_r (node n) const
  Returns the reading latency $\delta_r(n)$ of the node n.
- int delta_r (int idn) const
  Returns the reading latency $\delta_r(idn)$ of the node given by its id.
- int delta_w (node n) const
  Returns the writing latency $\delta_w(n)$ of the node n.
- int delta_w (int idn) const
  Returns the writing latency $\delta_w(idn)$ of the node given by its id.
- int delta_w (node n, int regtype_id) const
  Returns the writing latency $\delta_{w,t}(n)$ of the node n.
- int delta_w (int idn, int regtype_id) const
  Returns the writing latency $\delta_{w,t}(idn)$ of the node given by its id.
- int id (node n) const
  Returns the integer identifier of the node n.
- INSTRUCTIONS_TYPES instruction_type (node n) const
  Returns an INSTRUCTIONS_TYPES object attached to the node n.
- INSTRUCTIONS_TYPES instruction_type (int id) const

Returns an INSTRUCTIONS_TYPES object attached to the node given by its id.

- std::string get_string_attribute (node n) const
  Returns the textual attribute of the node n.
- const INFO_NODE & inf (node v) const
  Returns the information of node v.
- const INFO_EDGE & inf (edge e) const
  Returns the information of node e.
- const INFO_NODE & operator[ ] (node v) const
  Returns a reference to the information of v.
- INFO_NODE & operator[ ] (node v)
- const INFO_EDGE & operator[ ] (edge e) const
  Returns a reference to the information of e.
- INFO_EDGE & operator[ ] (edge e)
- int outdeg (node v) const
  Returns the number of edges leaving to node v.
- int indeg (node v) const
  Returns the number of edges entering v.
- int degree (node v) const
  Returns $outdeg(v) + indeg(v)$.
- node source (edge e) const
  Returns the source node of edge e.
- node target (edge e) const
  Returns the target node of edge e.
- node opposite (node v, edge e) const
  Returns $target(e)$ if $v = source(e)$ and $source(e)$ otherwise.
- node opposite (edge e, node v) const
  Returns $target(e)$ if $v = source(e)$ and $source(e)$ otherwise.
- int number_of_nodes () const
  Returns the number of nodes in the DAG.
- int number_of_edges () const
  Returns the number of edges in the DAG.
- const LEDA::list< node > & all_nodes () const
  Returns the list of all nodes in the DAG.

- const LEDA::list< edge > & all_edges () const
  Returns the list of all edges in the DAG.
- LEDA::list< edge > out_edges (node v) const
  Returns the list of edges leaving the node v.
- LEDA::list< edge > in_edges (node v) const
  Returns the list of edges entering the node v.
- node first_node () const
  Returns the first node in the DAG.
- node last_node () const
  Returns the last node in the DAG.
- node choose_node () const
  Returns a random node of the DAG (nil if empty).
- node succ_node (node v) const
  Returns the successor of node v (nil if it does not exist).
- node pred_node (node v) const
  Returns the predecessor of node v (nil if it does not exist).
- edge first_edge () const
  Returns the first edge in the DAG.
- edge last_edge () const
  Returns the last edge in the DAG.
- edge choose_edge () const
  Returns a random edge of the DAG (nil if empty).
- edge succ_edge (edge e) const
  Returns the successor of edge e (nil if it does not exist).
- edge pred_edge (edge e) const
  Returns the predecessor of edge e (nil if it does not exist).
- bool empty () const
  Returns true if the DAG is empty.
- bool member (node v) const
  Returns true if the node v belongs to the DAG.
- bool member (edge e) const
  Returns true if the edge e belongs to the DAG.
- bool is_hidden (edge e) const

Returns true if e is hidden, false otherwise. See DAG::hide_edge(edge e).

- bool is_hidden (node v) const
  Returns true if v is hidden, false otherwise. See DAG::hide_node(node v).
- LEDA::list< edge > hidden_edges () const
  Returns the list of all hidden edges.
- LEDA::list< node > hidden_nodes () const
  Returns the list of all hidden nodes.
- int max_delta (node u, node v) const
  Returns the maximal edge latency between to nodes, i.e., $\max_{e=(u,v)} \delta(e)$.
- int max_delta (int u, int v) const
  Returns the maximal edge latency between to nodes (given by their ids), i.e., $\max_{e=(u,v)} \delta(e)$.
- int min_delta (node u, node v) const
  Returns the minimal edge latency between to nodes, i.e., $\min_{e=(u,v)} \delta(e)$.
- int min_delta (int u, int v) const
  Returns the minimal edge latency between to nodes (given by their ids), i.e., $\min_{e=(u,v)} \delta(e)$.
- INSTRUCTIONS_TYPES search_instruction_type (const std::string opcode) const
  Looks for the instruction type (inside the DAG ISA) that has the opcode given as argument. If not found, it returns an INSTRUCTIONS_TYPES object with opcode_id equal to -1.
- INSTRUCTIONS_TYPES search_instruction_type (int opcode_id) const
  Looks for the instruction type (inside the DAG ISA) that has the opcode id given as argument. Returns NULL if not found. If not found, it returns an INSTRUCTIONS_TYPES object with opcode_id equal to -1.
- NodeFlag node_flag (node n) const
  Returns the node flag of n.

Update Operations

Returns the list of register types of the ISA of the DAG. The ISA object should be attached to the DAG before calling this function. See the DAG construtor method, or the DDG:DAG:set_ISA method. LEDA::list<REGISTER_TYPES> T() const ;

*/***

- int new_node (int it_id)
  Creates a new node.
- int new_node (int it_id, std::string text_inst)

Creates a new node.

- int new_node (int it_id, int n_id)
  Creates a new node.
- int new_node (int it_id, int n_id, std::string text_inst)
  Creates a new node.
- void del_node (node)
  Deletes the node given as argument.
- void del_node (int node_id)
  Deletes the node whose id is given as argument.
- edge new_edge (int s, int t, INFO_EDGE ie)
  Creates a new edge.
- edge new_edge (node s, node t, INFO_EDGE ie)
  Creates a new edge.
- void del_edge (edge e)
  Deletes the edge given as argument.
- void copy (const DAG &G2)
  Duplicates a DAG.
- void copy (const DAG &G2, node_map< node > &mn, edge_map< edge > &me)
  Duplicates a DAG.
- void get_loop_body (const LOOP &G2, node_map< node > &mn, edge_map< edge > &me)
  Retrieves a loop body.
- void get_loop_body (const LOOP &G2)
  Retrieves a loop body.
- void set_ISA (ARCHITECTURE isa)
  Sets the set of generic instruction types (ISA) encapsulated inside the DAG.
- void set_delta (edge e, int l)
  Sets a latency $\delta(e)$ to the edge e.
- void set_instruction_type (node n, int it_id)
  Sets an INSTRUCTIONS_TYPES object (given by its opcode id) attached to the node n.
- void set_string_attribute (node u, std::string inst_text)
  Sets a textual attribute to the node n. For instance, this textual attribute can be used to associate the textual code of the instruction.

- void set_etype (edge e, DDG::edge_type et)
  Sets an edge type (flow, serial, antidep, etc.).

- void hide_edge (edge e)
  Removes temporarily the edge e. The edge can be restored by DAG::restore_edge(edge e).

- void hide_edges (const LEDA::list< edge > &el)
  Hides all edges in the list el.

- void restore_edge (edge e)
  Restores the hidden edge e .

- void restore_edges (const LEDA::list< edge > &el)
  Restores all edges in the list el.

- void restore_all_edges ()
  Restores all hidden edges.

- void hide_node (node v)
  Removes temporarily the node v. Leaving and entering edges are also hidden. The node can be restored by DAG::restore_node(node v).

- void hide_node (node v, LEDA::list< edge > &h_edges)
  Removes temporarily the node v. Leaving and entering edges are also hidden. The node can be restored by DAG::restore_node(node n).

- void hide_nodes (const LEDA::list< node > &nl)
  Hides all nodes in the list nl.

- void restore_node (node v)
  Restores the hidden node v. Note that no edge adjacent to v that was hidden by G.hide_node(v) is restored by this operation.

- void restore_all_nodes ()
  Restores all hidden nodes. Note that no adjacent edges to the hidden nodes are restored by this operation.

- void del_all_edges ()
  Deletes all edges from the DAG.

- void move_edge (edge e, node v, node w)
  Moves the edge e to source v and target w.

- void set_node_flag (node n, const NodeFlag theValue)
  Set the flag of the node n.

Import and Export Methods

- void write_to_xml (const char *filename)
  Exports the DAG to an XML description.

Protected Attributes

- unsigned int cpt_nodes
- LEDA::d_array< int, LEDA::set< node > > V_R
- LEDA::d_array< int, LEDA::set< edge > > E_R
- ARCHITECTURE ISA
- node_array< set< node > > down
- node_array< set< node > > up
- node_array< int > longuest_path_from
- node_array< int > longuest_path_to
- node_matrix< int > lp_array
- node_array< int > shortest_path_from
- node_array< int > shortest_path_to
- set< node > Sources
- set< node > Targets
- map2< node, int, set< node > > Cons
- LEDA::map< int, node > node_index_table

3.14.2 Constructor & Destructor Documentation

3.14.2.1 LOOP ()

Default constructor.

Creates an empty LOOP object. A default ISA is assumed.

3.14.2.2 LOOP (ARCHITECTURE *isa*)

Constructs an empty LOOP object encapsulating a user-defined generic description of the architecture (ISA).

Parameters:

← ***isa*** ISA object, see ISA_xml.

Exceptions:

InvalidISADesc When the ISA description file is corrupted, and object of class DDG::InvalidISADesc is thrown.

3.14.3 Member Function Documentation

3.14.3.1 LEDA::set<edge> flow_from_to (int *id_u*, int *id_v*) const

Returns the set of all `flow` edges (not necessarily through registers) from the source node to the destination node. Formally equal to $\{e = (u, v) \in E | e \text{is a flow edge}\}$.

Parameters:

← ***id_u*** The integer identifier of the source node.

← ***id_v*** The integer identifier of the destination node.

3.14.3.2 LEDA::set<edge> flow_reg_from_to (node *u*, node *v*) const

Returns the set of all `flow` edges through registers from the source node u to the destination node v. Formally equal to $\{e = (u, v) \in E_R\}$.

Precondition:

A unique register type exists.

Exceptions:

NonUniqueRegisterType

3.14.3.3 LEDA::set<edge> flow_reg_from_to (int *id_u*, int *id_v*) const

Returns the set of all `flow` edges through registers from the source node to the destination node. Formally equal to $\{e = (u, v) \in E_R\}$.

Parameters:

← ***id_u*** The integer identifier of the source node.

← ***id_v*** The integer identifier of the destination node.

Precondition:

A unique register type exists.

Exceptions:

NonUniqueRegisterType

3.14.3.4 LEDA::set<edge> flow_reg_from_to (int *id_u*, int *id_v*, int *t*) const

Returns the set of all `flow` edges through registers from the source node to the destination node. Formally equal to $\{e = (u, v) \in E_R\}$.

Parameters:

← ***id_u*** The integer identifier of the source node.

← ***id_v*** The integer identifier of the destination node.

← ***t*** considered register type

3.14.3.5 int max_dist (node *u*, node *v*) const

Returns the maximal distance between two nodes (in terms of number of iterations). Formally equal to $\max_{e=(u,v)\in E} \lambda(e)$.

Warning:

Returns -MAXINT if no edge exists between u and v.

3.14.3.6 int max_dist (int *id_u*, int *id_v*) const

Returns the maximal iteration distance between two nodes. Formally equal to $\max_{e=(u,v)\in E} \lambda(e)$.

Parameters:

← ***id_u*** The integer identifier of the source node.

← ***id_v*** The integer identifier of the destination node.

Warning:

Returns 0 if no edge exists between u and v.

3.14.3.7 int max_flow_dist (node *u*, node *v*) const

Returns the maximal iteration flow distance (not necessarily through registers) between two nodes. Formally equal to $\max_{\text{a flow edge } e=(u,v)\in E} \lambda(e)$.

Warning:

Returns -MAXINT if no edge exists between u and v.

3.14.3.8 int max_flow_dist (int *id_u*, int *id_v*) const

Returns the maximal iteration flow distance (not necessarily through registers) between two nodes. Formally equal to $\max_{\text{a flow edge } e=(u,v)\in E} \lambda(e)$.

Parameters:

← ***id_u*** The integer identifier of the source node.

← ***id_v*** The integer identifier of the destination node.

Warning:

Returns -MAXINT if no edge exists between u and v.

3.14.3.9 int max_flow_reg_dist (node *u*, node *v*) const

Returns the maximal iteration flow distance through registers between two nodes. Formally equal to $\max_{e=(u,v)\in E_R} \lambda(e)$.

Warning:

Returns -MAXINT if no edge exists between u and v.

Precondition:

A unique register type exists in the ISA

Exceptions:

NonUniqueRegisterType

3.14.3.10 int max_flow_reg_dist (int *id_u*, int *id_v*) const

Returns the maximal iteration flow distance through registers between two nodes. Formally equal to $\max_{e=(u,v)\in E_R} \lambda(e)$.

Parameters:

← ***id_u*** The integer identifier of the source node.

← ***id_v*** The integer identifier of the destination node.

Warning:

Returns -MAXINT if no edge exists between u and v.

Precondition:

A unique register type exists in the ISA

Exceptions:

NonUniqueRegisterType

3.14.3.11 int max_flow_reg_dist (node *u*, node *v*, int *t*) const

Returns the maximal iteration flow distance through registers of type t between two nodes. Formally equal to $\max_{e=(u,v)\in E_{R,t}} \lambda(e)$.

Warning:

Returns -MAXINT if no edge exists between u and v.

3.14.3.12 int max_flow_reg_dist (int *id_u*, int *id_v*, int *t*) const

Returns the maximal iteration flow distance through registers between two nodes. Formally equal to $\max_{e=(u,v)\in E_{R,t}} \lambda(e)$.

Parameters:

← ***id_u*** The integer identifier of the source node.

← ***id_v*** The integer identifier of the destination node.

← ***t*** considered register type

Warning:

Returns -MAXINT if no edge exists between u and v.

3.14.3.13 int min_dist (node *u*, node *v*) const

Returns the minimal iteration distance between two nodes. Formally equal to $\min_{e=(u,v)\in E} \lambda(e)$.

Warning:

Returns +MAXINT if no edge exists between u and v.

3.14.3.14 int min_dist (int *id_u*, int *id_v*) const

Returns the minimal iteration distance between two nodes. Formally equal to $\max_{e=(u,v)\in E} \lambda(e)$.

Parameters:

← ***id_u*** The integer identifier of the source node.

← ***id_v*** The integer identifier of the destination node.

Warning:

Returns +MAXINT if no edge exists between u and v.

3.14.3.15 int min_flow_dist (node *u*, node *v*) const

Returns the minimal iteration flow distance (not necessarily through registers) between two nodes. Formally equal to $\min_{\text{a}flowedgee=(u,v)\in E} \lambda(e)$.

Warning:

Returns +MAXINT if no edge exists between u and v.

3.14.3.16 int min_flow_dist (int *id_u*, int *id_v*) const

Returns the minimal iteration flow distance (not necessarily through registers) between two nodes. Formally equal to $\max_{\text{a flow edge } e=(u,v)\in E} \lambda(e)$.

Parameters:

← ***id_u*** The integer identifier of the source node.

← ***id_v*** The integer identifier of the destination node.

Warning:

Returns +MAXINT if no edge exists between u and v.

3.14.3.17 int min_flow_reg_dist (node *u*, node *v*) const

Returns the minimial iteration flow distance through registers between two nodes. Formally equal to $\min_{e=(u,v)\in E_R} \lambda(e)$.

Warning:

Returns +MAXINT if no edge exists between u and v.

3.14.3.18 int min_flow_reg_dist (int *id_u*, int *id_v*) const

Returns the minimal iteration flow distance through registers between two nodes. Formally equal to $\min_{e=(u,v)\in E_R} \lambda(e)$.

Parameters:

← ***id_u*** The integer identifier of the source node.

← ***id_v*** The integer identifier of the destination node.

Warning:

Returns +MAXINT if no edge exists between u and v.

3.14.3.19 int min_flow_reg_dist (node *u*, node *v*, int *t*) const

Returns the minimial iteration flow distance through registers of type t between two nodes. Formally equal to $\min_{e=(u,v)\in E_{R,t}} \lambda(e)$.

Warning:

Returns +MAXINT if no edge exists between u and v.

3.14.3.20 int min_flow_reg_dist (int *id_u*, int *id_v*, int *t*) const

Returns the minimal iteration flow distance through registers of type t between two nodes. Formally equal to $\min_{e=(u,v)\in E_{R,t}} \lambda(e)$.

Parameters:

← ***id_u*** The integer identifier of the source node.

← ***id_v*** The integer identifier of the destination node.

← ***t*** considered register type

Warning:

Returns +MAXINT if no edge exists between u and v.

3.14.3.21 LEDA::rational CRITICAL_CYCLE () const

Returns the ratio of the critical cycle of the loop.

The critical cycle of a loop is defined by the one maximizing the ratio $\max_{C \text{ a cycle}} \frac{\sum_{e \in C} \delta(e)}{\sum_{e \in C} \lambda(e)}$. Contrary to what it is usually implemented, this method can detect null cycles, i.e., cycles C such as $\sum_{e \in C} \delta(e) = 0$ and $\sum_{e \in C} \lambda(e) \geq 0$. This method returns -1 if no cycle exists in the graph.

Precondition:

All cycles should have nonnegative distance, that is $\forall C$ a cycle in $G, \sum_{e \in C} \lambda(e) \geq 0$. Note that the edges may have nonpositive latencies ($\delta(e)$ may be positive or negative).

Remarks:

- The implemented algorithm is the one published by E.L Lawler in Periodic Optimization (1972, Springer Verlag). Its complexity is $O(n \times m \times \log(n))$, where n is the number of nodes and m is the number of edges.
- Since the original algorithm does not try to detect null cycles, we added our own implementation mechanism to do it.

Examples:

loop_example.cpp.

3.14.3.22 LEDA::rational CRITICAL_CYCLE (LEDA::list< edge > & *el*) const

Returns the ratio of the critical cycle of the loop and gives its list of edges.

A critical cycle in a loop data dependence graph is defined as the one which maximizes the ratio $\max_{C \text{ a cycle}} \frac{\sum_{e \in C} \delta(e)}{\sum_{e \in C} \lambda(e)}$. Contrary to what it is usually implemented, this method can detect null cycles, i.e., cycles C such as $\sum_{e \in C} \delta(e) = 0$ and $\sum_{e \in C} \lambda(e) \geq 0$. This method returns -1 if no cycle exists in the graph.

Parameters:

→ ***el*** Contains the list of edges that belongs to the critical cycle.

Warning:

If the critical cycle has a null latency ($\sum_{e \in C} \delta(e) = 0$) and a nonnegative distance $\sum_{e \in C} \lambda(e) > 0$, then the output list of edges nl is empty and the method returns zero. However, our method is able to build the list of edges belonging to a critical cycle if $\sum_{e \in C} \delta(e) = \sum_{e \in C} \lambda(e) = 0$.

Precondition:

All cycles should have nonnegative distance, that is $\forall C$ a cycle in $G, \sum_{e \in C} \lambda(e) \geq 0$. Note that the edges may have nonpositive latencies ($\delta(e)$ may be positive or negative).

Remarks:

- The implemented algorithm is the one published by E.L Lawler in Periodic Optimization (1972, Springer Verlag). Its complexity is $O(n \times m \times \log(n))$, where n is the number of nodes and m is the number of edges.
- Since the original algorithm does not try to detect null cycles, we added our own implementation mechanism to do it.

3.14.3.23 void copy (const LOOP & *G2*)

Duplicates a LOOP.

Parameters:

← ***G2*** The current LOOP object will contain a duplicated copy of G2.

Examples:

loop_example.cpp.

3.14.3.24 void copy (const LOOP & *G2*, node_map< node > & *mn*, edge_map< edge > & *me*)

Duplicates a LOOP.

Parameters:

← ***G2*** The current LOOP object will contain a duplicated copy of G2.

→ ***mn*** mn is a mapping from the set of nodes of G2 to the set of nodes of the current LOOP. $\forall u \in G_2, mn[u]$ contains the node belonging to the current LOOP corresponding to its original copy $u \in G_2$.

→ ***me*** me is a mapping from the set of edges of G2 to the set of edges of the current LOOP. $\forall e \in G_2, me[e]$ contains the edge belonging to the current LOOP corresponding to its original copy $e \in G_2$.

Remarks:

LOOP::build_internal_structures() is called inside this method.

3.14.3.25 void build_internal_structures ()

Build internal data structures for the LOOP object.

For internal information consistency, this function member should be called after update operations such as LOOP::new_node(INSTRUCTIONS_TYPES it), LOOP::new_edge(INFO_EDGE), etc. This method compute some internal information returned by other methods (the set V_R, the set of consumers per value, etc.). If the LOOP object is not modified after calling build_internal_structures(), all its internal information remain consistent. remarks For efficiency, the user can group series of update operations before ultimately calling build_internal_structures()

Reimplemented from DAG.

3.14.3.26 void write_to_vcg (const char * *filename*) const

Exports the LOOP DDG to a vcg file in order to be visualized with the xvcg tool. For more information about xvcg, see http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html
The user can navigate using vcg on all DDG attrbutes (types of edges, opcodes, etc.).

Import and Export Methods

Parameters:

← ***filename*** the file name of the DDG in vcg format

Reimplemented from DAG.

Examples:

gl2vcg.cpp, and loop_example.cpp.

3.14.3.27 void write_to_vcg (const char * *filename*, int *t*) const

Exports the LOOP DDG to a vcg file in order to be visualized with the xvcg tool. This function considers one register type.

Parameters:

← ***filename*** the file name of the DDG in vcg format

← ***t*** the considered register type For more information about xvcg, see http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html
The user can navigate using vcg on all DDG attrbutes (types of edges, opcodes, etc.)

Reimplemented from DAG.

3.14.3.28 void write_to_gl (const char * *filename*) const

Exports the LOOP to a gl file. If multiple registers types exist, they are all considered as the same type.

Parameters:

← ***filename*** the file name of the DDG in gl format

Reimplemented from DAG.

3.14.3.29 void write_to_gl (const char * *filename*, int *regtype_id*) const

Exports the LOOP to a gl file considering one register type.

Parameters:

← ***filename*** the file name of the DDG in gl format

← ***regtype_id*** All flow_reg edges through this register type are output to gl as flow_reg. Other flow_reg edges are output as "serial" edges.

Reimplemented from DAG.

3.14.3.30 bool read_from_gl (const char * *filename*, int *regtype_id*)

Imports a DDG loop from a gl file. In case of multiple registers types, this function allows to considere one of them.

Parameters:

← ***filename*** the file name of the DDG in gl format

← ***regtype_id*** The flow_reg edges in the gl file are considered as flow_reg through this register type.

Returns:

true if OK

Remarks:

- LOOP::build_internal_structures() is called inside this method.

Exceptions:

BadIODDG If an IO error occurs (corrupted or non present input files for instance), an exception object of type DDG::BadIODDG is thrown *

Precondition:

Before reading a LOOP from a gl file, it is assumed that the ISA of the LOOP defines all the opcodes ids of the nodes as present in the gl file. The constructor DDG::LOOP(ARCHITECTURE) defines a LOOP with a user defined ISA, otherwise a default ISA is assumed, see User Defined Instruction Set Architectures (XML format).

Reimplemented from DAG.

Examples:

gl2vcg.cpp.

3.14.3.31 bool read_from_gl (const char * *filename*)

Imports a DDG loop from a gl file.

Parameters:

← ***filename*** the file name of the DDG in gl format

Returns:

true if OK.

Precondition:

A unique register type should exist in the architecture.
Before reading a LOOP from a gl file, it is assumed that the ISA of the LOOP defines all the opcodes ids of the nodes as present in the gl file. The constructor DDG::LOOP(ARCHITECTURE) defines a LOOP with a user defined ISA, otherwise a default ISA is assumed, see ISA_xml.

Remarks:

- LOOP::build_internal_structures() is called inside this method.

Exceptions:

BadIODDG If an IO error occurs (corrupted or non present input files for instance), an exception object of type DDG::BadIODDG is thrown

Reimplemented from DAG.

3.14.3.32 bool read_from_xml (const char * *filename*)

Imports the LOOP from an XML file.

Returns:

true if ok.

Reimplemented from DAG.

Examples:

dag_example.cpp, and loop_example.cpp.

3.14.3.33 LEDA::set<edge> edges_from_to (int *id_u*, int *id_v*) const [inherited]

Returns the set of all edges from the source node to the destination node.

Parameters:

← ***id_u*** The integer identifier of the source node.

← ***id_v*** The integer identifier of the destination node.

3.14.3.34 int LongestPath (LEDA::list< node > & *nl*) const [inherited]

Returns the value of the longest path of the DAG.

Parameters:

→ ***nl*** Contains the list of nodes belonging to the computed longest path.

Warning:

Despite the unicity of its value, the longest path in a DAG may not be unique.

3.14.3.35 int LongestPath (list< edge > & *el*) const [inherited]

Returns the value of the longest path of the DAG.

Parameters:

→ ***el*** Contains the list of edges belonging to the computed longest path.

Warning:

Despite the unicity of its value, the longest path in a DAG may not be unique.

3.14.3.36 int CriticalPath (LEDA::list< node > & *nl*) [inherited]

Returns the value of the longest (critical) path.

Parameters:

→ ***nl*** Contains the list of nodes belonging to the computed longest path.

Warning:

The longest path in a DAG may not be unique

3.14.3.37 int CriticalPath (LEDA::list< edge > & *el*) [inherited]

Returns the value of the longest (critical) path.

Parameters:

→ ***el*** Contains the list of edges belonging to the computed longest path.

Warning:

The longest path in a DAG may not be unique

3.14.3.38 int lp (node *u*, node *v*) const [inherited]

Returns the value of the longest path from a node u to a node v given as arguments$*$.

Precondition:

Such path exists

3.14.3.39 node node_with_id (int *i*) const [inherited]

Returns the node with the integer identifier given as argument.

Exceptions:

IllegalAccessNode If the node id does not exist, or if the node is hidden, then an exception of type DDG:IllegalAccessNode is thrown.

Referenced by DAG::max_delta(), and DAG::min_delta().

3.14.3.40 node node_with_name (std::string *name*) const [inherited]

Returns the node with the name given as argument.

Exceptions:

IllegalAccessNode If the node id does not exist, or if the node is hidden.

3.14.3.41 bool is_value (node *n*) const [inherited]

Returns true if the node is an instruction writing into a register (i.e. if it belongs to the V_R set).

Precondition:

There is a unique register type in the architecture

Exceptions:

NonUniqueRegisterType

3.14.3.42 bool is_value (int *i*) const [inherited]

Returns true if the node defined by its indentifier is an instruction writing into a register (i.e. if it belongs to the V_R set).

Precondition:

There is a unique register type in the architecture

Exceptions:

NonUniqueRegisterType

3.14.3.43 bool is_flow_reg (edge *e*) const [inline, inherited]

Returns true if the edges is a flow dependence through a register. This is the case of a flow edge having a source writing into a register.

Precondition:

There is a unique register type in the architecture

Exceptions:

NonUniqueRegisterType

Definition at line 383 of file DAG.h.

References ARCHITECTURE::get_the_unique_register_type_id(), DAG::inf(), INFO_EDGE::is_flow_reg(), and DAG::ISA.

3.14.3.44 int get_register_type_id (edge *e*) const [inherited]

Returns the register type of a flow edge.

Precondition:

The edge should be a flow through register (flowdep_reg)

3.14.3.45 set<node> get_V_R () const [inline, inherited]

Returns the set $V_R \subseteq V$ of instructions (nodes) writing into registers.

Precondition:

a unique register type should exist in the architecture

Exceptions:

NonUniqueRegisterType

Examples:

dag_example.cpp, loop_example.cpp, and RS_example.cpp.

Definition at line 408 of file DAG.h.

References ARCHITECTURE::get_the_unique_register_type_id(), DAG::ISA, and DAG::V_R.

3.14.3.46 set<node> get_V_R (int *t*) const [inline, inherited]

Returns the set $V_{R,t} \subseteq V$ of instructions (nodes) writing into registers of type t (given by its id).

Exceptions:

BadRegType

Definition at line 414 of file DAG.h.

References ARCHITECTURE::exist_register_type_id(), DAG::ISA, and DAG::V_R.

3.14.3.47 LEDA::set<edge> get_E_R () const [inline, inherited]

Returns the set $E_R \subseteq E$ of flow dependence edges through registers.

Precondition:

A unique register type must exist.

Exceptions:

NonUniqueRegisterType

Examples:

loop_example.cpp.

Definition at line 426 of file DAG.h.

References DAG::E_R, ARCHITECTURE::get_the_unique_register_type_id(), and DAG::ISA.

3.14.3.48 LEDA::set<edge> get_E_R (int *t*) const [inline, inherited]

Returns the set $E_{R,t} \subseteq E$ of flow dependence edges through registers of type t (given by its id).

Exceptions:

BadRegType

Definition at line 432 of file DAG.h.

References DAG::E_R, ARCHITECTURE::exist_register_type_id(), and DAG::ISA.

3.14.3.49 LEDA::set<node> get_Cons (node *u*) const [inline, inherited]

Returns the set of consumers of a node.

Precondition:

A unique register type should exist in the architecture.

Exceptions:

NonUniqueRegisterType

Definition at line 450 of file DAG.h.

References DAG::Cons, ARCHITECTURE::get_the_unique_register_type_id(), and DAG::ISA.

3.14.3.50 LEDA::set<node> get_Cons (node *u*, int *id_regtype*) const [inline, inherited]

Returns the set of consumers of a node writing inside a register of a given type.

Precondition:

id_regtype is an id of register type in the architetcure

Exceptions:

BadRegType

Definition at line 458 of file DAG.h.

References DAG::Cons, ARCHITECTURE::exist_register_type_id(), and DAG::ISA.

3.14.3.51 int delta_w (node *n*) const [inherited]

Returns the writing latency $\delta_w(n)$ of the node n.

Precondition:

a unique register type should exist in the architecture

Examples:

dag_example.cpp, and loop_example.cpp.

3.14.3.52 int delta_w (int *idn*) const [inherited]

Returns the writing latency $\delta_w(idn)$ of the node given by its id.

Precondition:

a unique register type should exist in the architecture

3.14.3.53 int new_node (int *it_id*) [inherited]

Creates a new node.

Parameters:

← ***it_id*** The id of the INSTRUCTIONS_TYPES object describing the generic instruction of the node

Returns:

The integer node identifier.

Remarks:

This method assigns a unique inter identifier for the created node.

See also DAG::build_internal_structures().

Examples:

dag_example.cpp.

3.14.3.54 int new_node (int *it_id*, std::string *text_inst*) [inherited]

Creates a new node.

Parameters:

← ***it_id*** The id of the INSTRUCTIONS_TYPES object describing the generic instruction of the node

← ***text_inst*** A textual attribute of the instruction

Returns:

The integer node identifier.

Remarks:

This method assigns a unique inter identifier for the created node.

See also DAG::build_internal_structures().

3.14.3.55 int new_node (int *it_id*, int *n_id*) [inherited]

Creates a new node.

Parameters:

← ***it_id*** The id of the INSTRUCTIONS_TYPES object describing the generic instruction of the node

← ***n_id*** A unique id for the node to create

Returns:

The integer node identifier. Returns -1 if the there exists an node with the input id.

Exceptions:

NonUniqueNodeID if the input node id already exists

Remarks:

This method creates a node with an identifier given as input.

Precondition:

n_id is positive

See also DAG::build_internal_structures().

3.14.3.56 int new_node (int *it_id*, int *n_id*, std::string *text_inst*) [inherited]

Creates a new node.

Parameters:

← ***it_id*** The id of the INSTRUCTIONS_TYPES object describing the generic instruction of the node

← ***n_id*** A unique id for the node to create

← ***text_inst*** A textual attribute of the instruction

Returns:

The integer node identifier.

Exceptions:

NonUniqueNodeID if the input node id already exists

Remarks:

This method creates a node with an identifier given as input.

Precondition:

n_id is positive

See also DAG::build_internal_structures().

3.14.3.57 void del_node (node) [inherited]

Deletes the node given as argument.

All the edges incident to the deleted node are removed from the DAG too.

3.14.3.58 void del_node (int *node_id*) [inherited]

Deletes the node whose id is given as argument.

All the edges incident to the deleted node are removed from the DAG too.

See also DAG::build_internal_structures().

3.14.3.59 edge new_edge (int *s*, int *t*, INFO_EDGE *ie*) [inherited]

Creates a new edge.

It takes three arguments. If flow dependence, the inserted edge is added to the set E_R.

Parameters:

← ***s*** The integer identifier of the source node.

← ***t*** The integer identifier of the destination node.

← ***ie*** The INFO_EDGE object describing the edge attribute. See also DAG::build_internal_structures().

Examples:

dag_example.cpp.

3.14.3.60 edge new_edge (node *s*, node *t*, INFO_EDGE *ie*) [inherited]

Creates a new edge.

It takes three arguments. If flow dependence, the inserted edge is added to the set E_R.

Parameters:

← ***s*** The source node.

← ***t*** The destination node.

← ***ie*** The INFO_EDGE object describing the edge attribute. See also DAG::build_internal_structures().

3.14.3.61 void copy (const DAG & *G2*) [inherited]

Duplicates a DAG.

Parameters:

← ***G2*** The current DAG object will contain a duplicated copy of G2. This method will keep the same node ids, same edge and node information, etc. The internal consistency of the duplicated DAG is guaranteed.

Remarks:

DAG::build_internal_structures() is called inside this method.

Examples:

dag_example.cpp.

3.14.3.62 void copy (const DAG & *G2*, node_map< node > & *mn*, edge_map< edge > & *me*) [inherited]

Duplicates a DAG.

Parameters:

← ***G2*** The current DAG object will contain a duplicated copy of G2. This method will keep the same node ids, same edge and node information, etc. The internal consistency of the duplicated DAG is guaranteed.

→ ***mn*** mn is a mapping from the set of nodes of G2 to the set of nodes of the current DAG. $\forall u \in G_2, mn[u]$ contains the node belonging to the current DAG corresponding to its original copy $u \in G_2$.

→ ***me*** me is a mapping from the set of edges of G2 to the set of edges of the current DAG. $\forall e \in G_2, me[e]$ contains the edge belonging to the current DAG corresponding to its original copy $e \in G_2$.

Remarks:

DAG::build_internal_structures() is called inside this method.

3.14.3.63 void get_loop_body (const LOOP & *G2*, node_map< node > & *mn*, edge_map< edge > & *me*) [inherited]

Retrieves a loop body.

Parameters:

← ***G2*** The current DAG object will contain the loop body of G2. This method will keep the same node ids, same edge and node information, etc. The internal consistency of the DAG is guaranteed.

→ ***mn*** mn is a mapping from the set of nodes of G2 to the set of nodes of the current DAG. $\forall u \in G_2, mn[u]$ contains the node belonging to the current DAG corresponding to its original copy $u \in G_2$.

→ ***me*** me is a mapping from the set of edges of G2 to the set of edges of the current DAG. $\forall e \in G_2, me[e]$ contains the edge belonging to the current DAG corresponding to its original copy $e \in G_2$.

Remarks:

DAG::build_internal_structures() is called inside this method. The loop body contains the set of all nodes, but the set of edges contains only the edges with null distance: $\forall e$ in the loop body, $\lambda(e) = 0$.

3.14.3.64 void get_loop_body (const LOOP & *G2*) [inherited]

Retrieves a loop body.

Parameters:

← ***G2*** The current DAG object will contain the loop body of G2. This method will keep the same node ids, same edge and node information, etc. The internal consistency of the duplicated DAG is guaranteed.

Remarks:

DAG::build_internal_structures() is called inside this method. The loop body contains the set of all nodes, but the set of edges contains only the edges with null distance: $\forall e$ in the loop body, $\lambda(e) = 0$.

3.14.3.65 void set_ISA (ARCHITECTURE *isa*) [inline, inherited]

Sets the set of generic instruction types (ISA) encapsulated inside the DAG.

Parameters:

← ***isa*** An ARCHITECTURE object describing a user-defined ISA.

Warning:

It is better to clear the DAG before changing its ISA. If the ISA of a non empty DAG is modified on-the-fly, the successive behavior of the DAG object would be indefined.

Definition at line 886 of file DAG.h.

3.14.3.66 void set_string_attribute (node *u*, std::string *inst_text*) [inherited]

Sets a textual attribute to the node n. For instance, this textual attribute can be used to associate the textual code of the instruction.

Remarks:

- Currently, a textual attribute is a single textual line. Hence, it should not contain the newline character \n. If they exist, the current implementation replaces them by white spaces.
- Double quotes characters " should be specialized. They should be written as \" inside the string attribute.

3.14.3.67 void hide_edge (edge *e*) [inherited]

Removes temporarily the edge e. The edge can be restored by DAG::restore_edge(edge e).

Remarks:

A hidden edge does no longer belong to the list of graph edges

3.14.3.68 void hide_edges (const LEDA::list< edge > & *el*) [inherited]

Hides all edges in the list el.

Remarks:

A hidden edge does no longer belong to the list of graph edges

3.14.3.69 void restore_edge (edge *e*) [inherited]

Restores the hidden edge e .

Warning:

If an endpoint of e is hidden, an exception occurs.

3.14.3.70 void restore_edges (const LEDA::list< edge > & *el*) [inherited]

Restores all edges in the list el.

Warning:

If an endpoint of an edge is hidden, an exception occurs.

3.14.3.71 void restore_all_edges () [inherited]

Restores all hidden edges.

Warning:

If an endpoint of an edge is hidden, an exception occurs.

Examples:

dag_example.cpp, and loop_example.cpp.

3.14.3.72 void hide_node (node *v*) [inherited]

Removes temporarily the node v. Leaving and entering edges are also hidden. The node can be restored by DAG::restore_node(node v).

Remarks:

A hidden node does no longer belong to the list of graph nodes

Examples:

dag_example.cpp, and loop_example.cpp.

3.14.3.73 void hide_node (node *v*, LEDA::list< edge > & *h_edges*) [inherited]

Removes temporarily the node v. Leaving and entering edges are also hidden. The node can be restored by DAG::restore_node(node n).

Parameters:

← ***v*** the node to hide.

→ ***h_edges*** the list of leaving and entering edges hidden in consequence of hiding the node v.

Remarks:

A hidden node does no longer belong to the list of graph nodes

3.14.3.74 void hide_nodes (const LEDA::list< node > & *nl*) [inherited]

Hides all nodes in the list nl.

Remarks:

A hidden node does no longer belong to the list of graph nodes

3.14.3.75 void del_all_edges () [inherited]

Deletes all edges from the DAG.

See also DAG::empty();

3.14.3.76 void write_to_xml (const char * *filename*) [inherited]

Exports the DAG to an XML description.

All DDG attributes are included into this XML graph format.

3.15 NonUniqueNodeID Class Reference

3.15.1 Detailed Description

Definition at line 96 of file ddg_exceptions.h.

Public Member Functions

- NonUniqueNodeID (int u)
- NonUniqueNodeID ()

3.16 NonUniqueRegisterType Class Reference

3.16.1 Detailed Description

Definition at line 82 of file ddg_exceptions.h.

Public Member Functions

- NonUniqueRegisterType ()

3.17 REGISTER_TYPES Class Reference

3.17.1 Detailed Description

Author:

Touati Sid <Sid-nospam.Touati-nospam@inria.fr>

Examples:

dag_example.cpp, loop_example.cpp, and RS_example.cpp.

Definition at line 51 of file register_types.h.

Public Member Functions

- void set_nb_registers (int theValue)
- int get_nb_registers () const
- LEDA::list< std::string > get_list_registers ()
- void set_register_typename (std::string theValue)
- std::string get_register_typename () const
- const char * get_str_register_typename () const
- void set_register_type_id (int theValue)
- int get_register_type_id () const

4 DDG File Documentation

4.1 architecture.h File Reference

4.1.1 Detailed Description

The header file of the C++ ARCHITECTURE class. This file containts all function prototypes for building an ISA (architecture) object: register types, instructions types, latencies, read and written registers for each instruction type, code semantics (UAL/NUAL). An ISA object can be constructed by XML import, or by C++ interface using the different class methods.

Definition in file architecture.h.

Namespaces

- namespace DDG

Data Structures

- class ARCHITECTURE

Defines

- #define MAX_STRING 1000

4.2 DAG.h File Reference

4.2.1 Detailed Description

This file specifies the API of the DAG class and INFO_NODE and INFO_EDGE classes.

Directed Acyclic Graphs (DAG) are well known in graph theory and it optimizing compilation. This file contains the API of our implementation of Directed Acyclic Graphs as used in optimizing compilation. The DAG model we implement here is described in DAG Model.

Definition in file DAG.h.

Namespaces

- namespace DDG
- namespace **std**
- namespace **LEDA**

Data Structures

- class DAG

  This class represents a directed acyclic graph (DAG).

Defines

- #define MAX_NODE 10001

Functions

- template<class T>
  LEDA::list< T > set_to_list (const set< T > l)
- template<class T>
  set< T > list_to_set (const list< T > l)
- bool lt (node u, node v)
- bool le (node u, node v)
- bool gt (node u, node v)
- bool ge (node u, node v)
- bool parallel (node u, node v)
- bool comparable (node u, node v)
- int MAXIMAL_ANTI_CHAIN (const graph &G, set< node > &MA)
- int MINIMAL_CHAIN (const graph &G, node_array< int > &chain)
- int MINIMAL_CHAIN (const DAG &G, array< list< node > > &C)
- int REGISTER_SATURATION (const DAG &G, leda::list< node > &RS_values, bool fast=true)
- int REGISTER_SATURATION (const DAG &G, leda::list< node > &RS_values, node_array< node > &k, bool fast=true)
- int REGISTER_SATURATION (const DAG &G, int t, leda::list< node > &RS_values, bool fast=true)
- int REGISTER_SATURATION (const DAG &G, int t, leda::list< node > &RS_values, node_array< node > &k, bool fast=true)

4.3 DDG.h File Reference

4.3.1 Detailed Description

Main API of the DDG library. This is the unique file that must be included by user.

Definition in file DDG.h.

4.4 ddg_exceptions.h File Reference

4.4.1 Detailed Description

All exception objects thrown by DDG library. DDG objects and methods may throw exceptions. In the documentation of each method, we precise the thrown exception if any.

Definition in file ddg_exceptions.h.

Namespaces

- namespace DDG

Data Structures

- class IllegalAccessNode
- class InvalidISADesc
- class BadIODDG
- class BadRegType
- class BadInstructionType
- class NonUniqueRegisterType
- class IllegalDependance
- class NonUniqueNodeID

4.5 info_edge.h File Reference

4.5.1 Detailed Description

Header file of INFO_EDGE class. Edges may contain many useful information, such as: latencies, distances, type of dependence, etc.

Definition in file info_edge.h.

Namespaces

- namespace DDG

Data Structures

- class INFO_EDGE

  This class represents the attributes (information) attached to an edge in a DDG.

Enumerations

- enum edge_type {

  flowdep_reg = 1, antidep_reg, outputdep_reg, inputdep_reg,

  flowdep_mem, antidep_mem, outputdep_mem, inputdep_mem,

  spilldep_mem, other_mem, serial, killerdep,

  reusedep }

  Edges corresponds to data dependences. Data dependences are classified into multiple types : flowdep_reg, antidep_reg, outputdep_reg, inputdep_reg, flowdep_mem, etc.

Functions

- std::string edge_type_to_string (const edge_type)

  Convert an edge_type to string ("flow", "serial", "antidep", ...).

- edge_type string_to_edge_type (const std::string)

  Convert a string to an edge_type.

4.6 info_node.h File Reference

4.6.1 Detailed Description

Header file of INFO_NODE class. Nodes may contain many useful information, such as: unique identifiers, instruction type, latency, etc.

Definition in file info_node.h.

Namespaces

- namespace DDG

Data Structures

- class INFO_NODE

  This class represents the attributes (information) attached to each node in a DDG.

Enumerations

- enum NodeFlag {

  NodeFlag_Nothing = 0x0, NodeFlag_PartialDef = 0x1, NodeFlag_SpillCode = 0x2, NodeFlag_Volatile = 0x4,

  NodeFlag_SPUpdate = 0x8, NodeFlag_Prefetch = 0x10, NodeFlag_Preload = 0x20, NodeFlag_Barrier = 0x40,

  NodeFlag_Clobber = 0x80, NodeFlag_Hoisted = 0x100, NodeFlag_DeadCode = 0x200, NodeFlag_SafeAccess = 0x800,

  NodeFlag_SafePerfs = 0x1000, NodeFlag_EntryCode = 0x4000, NodeFlag_-ExitCode = 0x8000, NodeFlag_KillOp = 0x5000 }

  Nodes may have some flag representing, special nodes. If a node corresponds to a regular instruction, then its flag is NodeFlag_Nothing.

4.7 instructions_types.h File Reference

4.7.1 Detailed Description

A Documented file. Details.

Definition in file instructions_types.h.

Namespaces

- namespace DDG

Data Structures

- class INSTRUCTIONS_TYPES

  This class describes a generic instruction, used by the class ARCHITECTURE to describe an ISA. A generic instruction object contains opcode, unique id per opcode, string opcode, list of writtent registers, list of written register types, etc.

4.8 loop.h File Reference

4.8.1 Detailed Description

This file specifies the API of the LOOP class that describes cyclic data dependence graphs.

In optimizing compilation, simple internal loops are modeled by cyclic data dependence graphs. This library implements a C++ class called LOOP. The LOOP model we implement here is described in Loop Model.

Definition in file loop.h.

Namespaces

- namespace DDG

Data Structures

- class LOOP

  This class represent cyclic data dependence graphs of simple loops (without branches).

Defines

- #define Min(X, Y) X<Y ? X:Y

Functions

- void unroll (const LOOP &L, DAG &unrolled, int unroll_degree)
  Returns the body of the unrolled loop.
- void unroll (const LOOP &L, LOOP &unrolled, int unroll_degree)
  Returns an unrolled loop.
- void loop_merge (const LOOP &L1, const LOOP &L2, LOOP &L3)
  Merges two independent loop DDGs in order to produce a new DDG.
- bool retime_ddg (LOOP &G, LEDA::node_array< int > &r)
  Computes and applies a valid loop retiming (called also loop shifing).
- bool retime_ddg (LOOP &G)
  Computes and applies a valid loop retiming (called also loop shifing).
- bool apply_retiming (LOOP &G, const LEDA::node_array< int > r)
  Applies the loop retiming given as input.
- bool is_lexicographic_positive (LOOP &G, leda::list< edge > &le)
  Checks if all circuits of G are lexicographic positive. That is, $\forall C$ a circuit in G, $\lambda(C) = \sum_{e \in C} \lambda(e) > 0$.

4.9 register_types.h File Reference

4.9.1 Detailed Description

The header file of register types class. This file contains the method declaration to construct an object of register type.

Definition in file register_types.h.

Namespaces

- namespace DDG

Data Structures

- class REGISTER_TYPES

4.10 RS.h File Reference

4.10.1 Detailed Description

This is the header file for the implementation of register saturation for a DAG. Register Saturation is the fruit of deep fundamental research results. The full theory is described in the following article: Sid-Ahmed-Ali Touati. Register Saturation in Instruction Level Parallelism. International Journal of Parallel Programming, Springer-Verlag, Volume 33, Issue 4, August 2005. 57 pages.

Definition in file RS.h.

Namespaces

- namespace DDG
- namespace leda

Data Structures

- class CBC
- class EN

5 DDG Example Documentation

5.1 dag_example.cpp

This is an example of how to use the DAG class.

```
1 /*****************************************************************************
2  *   Copyright (C) 2009 by Touati Sid                                        *
3  *   Sid-nospam.Touati-nospam@uvsq.fr                                        *
4  *   Copyright  INRIA and the University of Versailles                       *
5  *   This program is free software; you can redistribute it and/or modify    *
6  *   it under the terms of the GNU General Public License as published by    *
7  *   the Free Software Foundation; either version 2 of the License, or       *
8  *   (at your option) any later version.                                     *
9  *                                                                           *
10  *   This program is distributed in the hope that it will be useful,         *
11  *   but WITHOUT ANY WARRANTY; without even the implied warranty of          *
12  *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the           *
13  *   GNU General Public License for more details.                            *
14  *                                                                           *
15  *   You should have received a copy of the GNU Library General Public       *
16  *   License along with this program (LGPL); if not, write to the            *
17  *   Free Software Foundation, Inc.,                                         *
18  *   59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.               *
19  *   Exception : this software requires a licence  of the LEDA libary.       *
20  *   If you are from academia, you can get a free leda binary licence
21  *   allowing you to use freely our DDG library.                             *
22  *   Otherwise, you can buy a LEDA licence from algorithmic-solutions        *
23  *   http://www.algorithmic-solutions.com                                    *
24  *   LEDA binaries and source codes are excluded from this LGPL.             *
```

```
25  *   Obtaining a LEDA licence is not mandatory to use our GPL software     *
26  *   If the user wants, he can  create a free source-compatible replacement*
27  *   for LEDA, and he is allowed to use our software under LGPL.           *
28  ****************************************************************************/
29
34 #include "DDG.h" // the header file of the DDG library.
35         // To be included by the user.
36 #include "LEDA/graph/graph_misc.h" // Is_Acyclic
37 #include <iostream>
38 #include <cstdlib>
39
40
41 using namespace std;
42 using namespace DDG; // the namespace of the DDG library.
43
44
45
46 int main(int argc, char *argv[])
47 {
48     //int are always usefull
49     int         i,j,cpt ;
50     int c; // for getopt
51     // some strings for file names
52     char        *ddg_filename=NULL, *arch_filename=NULL;
53     char        str[MAX_STRING];
54     REGISTER_TYPES rt;
55     LEDA::list<REGISTER_TYPES> T;
56
57     // some objects for graphs
58     //declare a DDG of a loop (cyclic data dependence graph)
59     DDG::LOOP loop;
60     ARCHITECTURE ISA_example;
61     DDG::DAG dag, dag_copy;  // declare a DDG of a basic block (directed acyclic graph : DAG)
62
63     LEDA::node n,u;
64     LEDA::edge e;
65     LEDA::set<node> VR;
66     LEDA::set<node> MA; // for a maximal antichain computation
67     LEDA::node_map<int> map; // when duplicating a graph,
68                 // we need to save node correspondance between original and new graph
69     LEDA::node_array<node> killer; // for register saturation examples
70     array< list<node> > ALN ; // array of sorted chains
71
72     DDG::INSTRUCTIONS_TYPES it; // information attached to an opcode
73
74     LEDA::list<edge> el;
75     LEDA::list<node> nl;
76     LEDA::set<node> parents, children; // set of parents and children of a node
77
78     while ((c = getopt (argc, argv, "i:a:")) != -1){
79         switch(c){
80             case 'a': arch_filename=optarg;
81                 break;
82             case 'i': ddg_filename =optarg;
83                 break;
84             case '?':
85                 cout<<"usage:"<< argv[0] <<" -i ddg_filename.xml [-a isa_desc.xml] "<<endl;
86                 return EXIT_FAILURE;
87         }
88
89     }
90
```

```
91     //------- the user can define its own ISA ----------------
92     if (arch_filename!=NULL)  {
93         if (!ISA_example.read_from_xml(arch_filename)) return EXIT_FAILURE;
94         if(ISA_example.check()){
95             loop=LOOP(ISA_example);  //build an empty DDG loop with a user defined architcural
96             dag=DAG(ISA_example);   //build an empty DAG with a user defined architcural (ISA) c
97         } else return EXIT_FAILURE;
98     }
99
100      //---------------- The user can read a DDG from an xml format defined by the DDG library
101
102     if (ddg_filename != NULL){
103
104         i=loop.read_from_xml(ddg_filename); // reads the DDG loop from a n xml file
105
106         if(i==-1){
107             return EXIT_FAILURE;
108         }
109
110         i=dag.read_from_xml(ddg_filename); // reads the DDG loop from a n xml file
111
112         if(i==-1){
113             return EXIT_FAILURE;
114         }
115
116     }
117     else{
118         cout<<"usage:"<< argv[0] <<" -i ddg_filename.xml [-a isa_desc.xml] "<<endl;
119         return EXIT_FAILURE;
120     }
121
122     if(! Is_Acyclic(dag, el)) {
123         cerr<<"Input Error: Non acyclic graph.\n";
124         cerr<<"There are "<< el.size() <<" illegal arcs:\n";
125         forall(e, el){
126             cerr<<dag.id(dag.source(e))<<" -> "<< dag.id(dag.target(e))<<endl;
127         }
128         return EXIT_FAILURE;
129     }
130
131
132      // The user can modify the DDG by many other methods :
133     // new_node, new_edge, clear, del_node, del_edge, etc.
134      // The user can access any information about nodes, edges,
135      // apply well known algorithms using the LEDA library
136      // See the LEDA manual : http://www.algorithmic-solutions.info/leda_manual
137      // See the DDG user manual inside this distribution
138
139
140     cout<< "DAG example program: " <<ddg_filename<<endl;
141     cout<< "-----------------------------" <<endl;
142     T=dag.T();
143
144     forall(rt,T){
145
146         cout<<"Register Saturation of the DAG (loop body) for register type "<< rt.get_registe
147         cout<<"Saturating values of type "<<rt.get_register_typename()<<endl;
148         forall(n, nl){
149         it= dag.instruction_type(n);
150         cout<< dag.id(n)<< " with opcode " << it.opcode()<< " ("<<dag.latency(n)<<","<<dag.delt
151         }
152     }
```

```
153     // -------------------- The critical path is the longest path of the DAG
154     cout<<endl<<"Critical Path ="<<dag.CriticalPath(el)<<endl;
155
156     cout<<"Edges belonging to the critical path"<<endl;
157     forall(e, el){
158         n=dag.source(e);
159         u=dag.target(e);
160         it= dag.instruction_type(n);
161         cout<< dag.id(n)<< " -> " << dag.id(u) <<endl;
162     }
163
164     //------------------------ We can access the DAG in many ways
165     forall_nodes(u, dag){
166         parents=dag.get_parents(u);
167         cout<<"The parents of the node number "<<dag.id(u)<<" are : ";
168         forall (n, parents){
169             cout<<dag.id(n)<<" ";
170         }
171         cout<<endl;
172     }
173     forall_nodes(u, dag){
174         children=dag.get_children(u);
175         cout<<"The children of the node number "<<dag.id(u)<<" are : ";
176         forall (n, children){
177             cout<<dag.id(n)<<" ";
178         }
179         cout<<endl;
180     }
181     forall_nodes(u, dag){
182         parents=dag.get_up(u);
183         cout<<"The ascendants of the node number "<<dag.id(u)<<" are : ";
184         forall (n, parents){
185             cout<<dag.id(n)<<" ";
186         }
187         cout<<endl;
188     }
189     forall_nodes(u, dag){
190         children=dag.get_down(u);
191         cout<<"The descendants of the node number "<<dag.id(u)<<" are : ";
192         forall (n, children){
193             cout<<dag.id(n)<<" ";
194         }
195         cout<<endl;
196     }
197
198     // ------------------- The user can visualize the DAG ------------------------
199     snprintf(str, MAX_STRING, "%s_dag.vcg",ddg_filename); // sets the .vcg filename for the DAG
200     dag.write_to_vcg(str); //outputs the DAG to a vcg file to be visualized by the xvcg tool
201     dag_copy.copy(dag); // create an exact copy.
202     snprintf(str, MAX_STRING, "%s_dag_copy.vcg",ddg_filename);
203     dag_copy.write_to_vcg(str); //outputs the copy DAG to a vcg file to be visualized by the xv
204
205     cout<<endl<<str<< " can be visualized using xvcg."<<endl;
206
207     /*---------------- The user can extract a copy of a loop body -------------- */
208
209     map.init(loop);/* initializes a mapping function from loop nodes to dag nodes.
210     Forall u a node in the loop, map(u) stores its corresponding node in the exctracted DAG (lo
211     dag.clear();
212     forall_nodes(u,loop){
213         map[u]=dag.new_node((loop.instruction_type(u)).opcode_id());
214     }
```

```
215     forall_edges(e,loop){
216         if(loop.lambda(e)==0) { // It is an edge belonging to the loop body ==> insert it into
217             dag.new_edge(map[loop.source(e)], map[loop.target(e)],loop[e]);
218         }
219     }
220
221     // --- Dilworth decomposition -----------
222     i=MAXIMAL_ANTI_CHAIN(dag, MA);
223     cout<<"Maximal antichain in the DAG : "<<i<<endl;
224     cout<<"Nodes belonging to a maximal antichain "<<endl;
225     forall(n, MA){
226         cout<< dag.id(n)<< " with opcode " << it.opcode()<<endl;
227     }
228
229     //---The user can extract the body of a loop unrolled many times
230     // (recurrent data dependence edges are preserved) ----------
231
232     cout<<"Unrolling the loop "<<ddg_filename<<" four times and extract its body"<<endl;
233     unroll (loop, dag, 4);
234     snprintf(str, MAX_STRING,"%s_dag_unroll_4.vcg",ddg_filename); // sets the .vcg filename for
235     dag.write_to_vcg(str);
236     // The user can access any kind of information
237     forall(rt,T){
238         cout <<"The set of values of type "<< rt.get_register_typename() <<" of the body  is :"
239         VR=dag.get_V_R(rt.get_register_type_id());
240         forall(u, VR){
241             it=dag.instruction_type(u);
242             cout<< dag.id(u)<< " with opcode " << it.opcode()<<endl;
243
244         }
245     }
246     cout<<endl<< "The set of sources of the body are :"<<endl;
247     VR=dag.get_Sources();
248     forall(u, VR){
249         it=dag.instruction_type(u);
250         cout<< dag.id(u)<< " with opcode " << it.opcode()<<endl;
251     }
252     cout<<endl<< "The set of sinks of the body are :"<<endl;
253     VR=dag.get_Targets();
254     forall(u, VR){
255         it=dag.instruction_type(u);
256         cout<< dag.id(u)<< " with opcode " << it.opcode()<<endl;
257     }
258
259     // Computes the critical path of the DAG and puts the list of its nodes in nl
260     cout<<endl<<"Critical Path of the unrolled body ="<<dag.CriticalPath(nl)<<endl;
261     cout<<"Nodes belonging the critical path of the unrolled body "<<endl;
262     forall(n, nl){
263         it= dag.instruction_type(n);
264         cout<< dag.id(n)<< " with opcode " << it.opcode()<<endl;
265     }
266
267     // ------------- Hiding and restoring nodes -------------
268     // look for two arbitrary values of any register type
269     forall(rt, T){
270         if (dag.get_V_R(rt.get_register_type_id()).size()>=2){
271             dag.hide_node(dag.get_V_R(rt.get_register_type_id()).choose()); // just to hide one
272                         // note that all adjacent edges are hidden too
273             dag.hide_node(dag.get_V_R(rt.get_register_type_id()).choose()); // hide another one
274             break;
275         }
276     }
```

```
277     rt=T.front();
278     cout<<"Register Saturation of the DAG (after hiding two values of type "<< rt.get_register_
279     cout<<"Saturating values of type "<< rt.get_register_typename() <<endl;
280     forall(n, nl){
281         it= dag.instruction_type(n);
282         cout<< dag.id(n)<< " with opcode " << it.opcode()<<endl;
283     }
284
285     dag.restore_all_nodes();
286     dag.restore_all_edges();
287     dag.build_internal_structures(); // restore the internal state of the DAG
288
289     cout<<"Register Saturation of the DAG (after restoring the two values of type"<< rt.get_reg
290     cout<<"Saturating values of type "<< rt.get_register_typename()<<endl;
291     forall(n, nl){
292         it= dag.instruction_type(n);
293         cout<< dag.id(n)<< " with opcode " << it.opcode()<<endl;
294     }
295
296     //---------------------- Lattices and Orders --------------------------/
297     cout<<"Lattices and Order Notations:\n";
298     forall_nodes(u, dag){
299         forall_nodes(n, dag){
300             if(lt(u,n)){
301                 cout<<dag.id(u)<< " < " << dag.id(n) << endl;
302             }
303             if(le(u,n)){
304                 cout<<dag.id(u)<< " <= " << dag.id(n)<< endl;
305             }
306             if(gt(u,n)){
307                 cout<<dag.id(u)<< " > " << dag.id(n)<< endl;
308             }
309             if(ge(u,n)){
310                 cout<<dag.id(u)<< " >= " << dag.id(n)<< endl;
311             }
312             if(parallel(u,n)){
313                 cout<<dag.id(u)<< " || " << dag.id(n)<< endl;
314             }
315             if(comparable(u,n)){
316                 cout<<dag.id(u)<< " ~ " << dag.id(n)<< endl;
317             }
318         }
319     }
320
321
322     cout<<"Minimal Chain Decomposition (Dilworth)"<<endl;
323     cout<<"--------------------------------------"<<endl;
324     j=MINIMAL_CHAIN(dag, ALN);
325
326     cout<<"There are "<< j <<" chains"<<endl;
327     cpt=0;
328     for(i=0;i< j ;i++){
329         cout<<"Chain["<<i<<"]: ";
330         forall(u,ALN[i]){
331             cout<<dag.id(u);
332             cout<<" ";
333             cpt++;
334         }
335         cout<<endl;
336     }
337     cout<<"Total "<< cpt <<" nodes"<<" in this distribution."<<endl;
338     //---------------------- Many other software opportunities exist. See the reference manual.
```

```
339
340     return EXIT_SUCCESS;
341 }
```

5.2 gl2vcg.cpp

A simple code example to show how to export a DDG file from gl format to vcg format to be visualized with xvcg

```
1 /*************************************************************************
2  *   Copyright (C) 2009 by Touati Sid                                    *
3  *   Sid-nospam.Touati-nospam@uvsq.fr                                    *
4  *   Copyright  INRIA and the University of Versailles                   *
5  *   This program is free software; you can redistribute it and/or modify  *
6  *   it under the terms of the GNU General Public License as published by  *
7  *   the Free Software Foundation; either version 2 of the License, or     *
8  *   (at your option) any later version.                                 *
9  *                                                                       *
10  *   This program is distributed in the hope that it will be useful,       *
11  *   but WITHOUT ANY WARRANTY; without even the implied warranty of        *
12  *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the         *
13  *   GNU General Public License for more details.                          *
14  *                                                                         *
15  *   You should have received a copy of the GNU Library General Public     *
16  *   License along with this program (LGPL); if not, write to the          *
17  *   Free Software Foundation, Inc.,                                       *
18  *   59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.             *
19  *   Exception : this software requires a licence  of the LEDA libary.     *
20  *   If you are from academia, you can get a free leda binary licence
21  *   allowing you to use freely our DDG library.                           *
22  *   Otherwise, you can buy a LEDA licence from algorithmic-solutions      *
23  *   http://www.algorithmic-solutions.com                                  *
24  *   LEDA binaries and source codes are excluded from this LGPL.           *
25  *   Obtaining a LEDA licence is not mandatory to use our GPL software     *
26  *   If the user wants, he can  create a free source-compatible replacement*
27  *   for LEDA, and he is allowed to use our software under LGPL.           *
28  *************************************************************************/
29
33 #include "DDG.h" // the header file of the DDG library. To be included by the user.
34
35 #include <iostream>
36 #include <cstdlib>
37
38
39 using namespace std;
40 using namespace DDG; // the namespace of the DDG library.
41
42 char *getStringForArgument(char * s, int argc, char *argv[])
43  {
44     for (int i = 1 ; i<argc; i++)
45     {
46         if (strcmp(argv[i],s) == 0)
47         {
48             if (i+1<argc){
49                 return argv[i+1];}
50                 else
51                     fprintf(stderr,"Argument value missing for %s\n",s);
52                 return(NULL);
53         }
54     }
```

```
55     return(NULL);
56 }
57
58
59 int main(int argc, char *argv[])
60 {
61     // some strings for file names
62     char        *ifi;
63     char        cc[400], str[400];
64
65     // some objects for graphs
66     DDG::LOOP loop; //declare a DDG of a loop (cyclic data dependence graph)
67     ARCHITECTURE isa_example;
68
69     ifi = getStringForArgument("-i",argc,argv);
70     if (ifi==NULL)  { // no line commande arguments
71         cout<<"gl2vcg exports a DDG from gl format to xvcg format for visualization"<<endl;
72         cout<<"usage: gl2vcg -i inputDDG.gl -o outputDDG.vcg [-a arch_desc]"<<endl;
73         return EXIT_FAILURE;
74     }
75     strcpy(cc,ifi);
76
77     ifi = getStringForArgument("-a",argc,argv);
78 //check wether the user specifies an ISA description or not
79     if (ifi!=NULL)  {
80         //build an empty DDG loop with a user defined ISA description
81         isa_example.read_from_arch(ifi);
82         if (isa_example.check())
83             loop=LOOP(isa_example);
84         else exit(1);
85     }
86
87     loop.read_from_gl(cc); // reads the DDG loop from a gl file
88
89     ifi = getStringForArgument("-o",argc,argv);
90     if (ifi==NULL)  { // no line commande arguments
91         cout<<"gl2vcg exports a DDG from gl format to xvcg format for visualization"<<endl;
92         cout<<"usage: gl2vcg -i inputDDG.gl -o outputDDG.vcg [-a arch_desc]"<<endl;
93         return EXIT_FAILURE;
94     }
95     //outputs the DDG loop to a vcg file to be visualized by the xvcg tool
96     loop.write_to_vcg(ifi);
97     return EXIT_SUCCESS;
98 }
99
```

5.3 loop_example.cpp

This is an example of how to use the LOOP class.

```
1 /*************************************************************************
2  *   Copyright (C) 2009 by Touati Sid                                    *
3  *   Sid-nospam.Touati-nospam@uvsq.fr                                    *
4  *   Copyright  INRIA and the University of Versailles                   *
5  *   This program is free software; you can redistribute it and/or modify  *
6  *   it under the terms of the GNU General Public License as published by  *
7  *   the Free Software Foundation; either version 2 of the License, or     *
8  *   (at your option) any later version.                                 *
9  *                                                                       *
10  *   This program is distributed in the hope that it will be useful,       *
```

```
11  *   but WITHOUT ANY WARRANTY; without even the implied warranty of       *
12  *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the        *
13  *   GNU General Public License for more details.                         *
14  *                                                                        *
15  *   You should have received a copy of the GNU Library General Public    *
16  *   License along with this program (LGPL); if not, write to the         *
17  *   Free Software Foundation, Inc.,                                      *
18  *   59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.            *
19  *   Exception : this software requires a licence  of the LEDA libary.    *
20  *   If you are from academia, you can get a free leda binary licence
21  *   allowing you to use freely our DDG library.                          *
22  *   Otherwise, you can buy a LEDA licence from algorithmic-solutions     *
23  *   http://www.algorithmic-solutions.com                                 *
24  *   LEDA binaries and source codes are excluded from this LGPL.          *
25  *   Obtaining a LEDA licence is not mandatory to use our GPL software    *
26  *   If the user wants, he can  create a free source-compatible replacement*
27  *   for LEDA, and he is allowed to use our software under LGPL.          *
28  *************************************************************************/
32 #include "DDG.h" // the header file of the DDG library. To be included by the user.
33
34 #include <iostream>
35 #include <cstdlib>
36
37
38 using namespace std;
39 using namespace DDG; // the namespace of the DDG library.
40
41
42
43 int main(int argc, char *argv[])
44 {
45     //int are always usefull
46     int         i,j ;
47     int c; // for getopt
48     // some strings for file names
49     ARCHITECTURE isa_arch;
50
51     char        str[MAX_STRING];
52     char        *ddg_filename=NULL, *arch_filename=NULL;
53     LEDA::list<REGISTER_TYPES> T;
54     REGISTER_TYPES rt;
55     // some objects for graphs
56     //declare a DDG of a loop (cyclic data dependence graph)
57     DDG::LOOP loop, unrolled, loop_copy;
58     ARCHITECTURE isa_example;
59
60     LEDA::node n,u;
61     LEDA::edge e;
62     set<node> VR;
63     set<edge> ER;
64     DDG::INFO_EDGE ie; //information attached to a given edge
65
66     INSTRUCTIONS_TYPES it; // information attached to an opcode
67     LEDA::list<edge> el; // a list of edges
68
69     while ((c = getopt (argc, argv, "i:a:")) != -1){
70         switch(c){
71             case 'a': arch_filename=optarg;
72                 break;
73             case 'i': ddg_filename =optarg;
74                 break;
75             case '?':
```

```
76                 cout<<"usage:"<< argv[0] <<" -i ddg_filename.xml [-a isa_desc.xml] "<<endl;
77                 return EXIT_FAILURE;
78         }
79
80     }
81
82     if (arch_filename!=NULL)  {
83         if (!isa_example.read_from_xml(arch_filename)) return EXIT_FAILURE;
84         if(isa_example.check()){
85             loop=LOOP(isa_example);  //build an empty DDG loop with a user defined architcural
86         } else return EXIT_FAILURE;
87
88     }
89      //---- The user can read a DDG from an xml format defined by the DDG library
90     if (ddg_filename != NULL){
91         i=loop.read_from_xml(ddg_filename); // reads the DDG loop from an xml file
92         if(i==-1) return EXIT_FAILURE;
93
94     }
95     else{
96         cout<<"usage:"<< argv[0] <<" -i ddg_filename.xml [-a isa_desc.xml] "<<endl;
97         return EXIT_FAILURE;
98     }
99
100
101
102
103      // The user can modify the DDG by many other methods
104     // : new_node, new_edge, clear, del_node, del_edge, etc.
105      // The user can access any information about nodes, edges,
106     // apply well known algorithms using the LEDA library
107      // See the LEDA manual : http://www.algorithmic-solutions.info/leda_manual
108      // See the DDG user manual inside this distribution
109
110     cout<< "DDG example program: " <<ddg_filename<<endl;
111     cout<< "------------------------------" <<endl;
112     // ------------------- The user can compute the critical path ---------------
113     // Null circuits are accepted inside the DDG and detected
114     cout<<"Critical Cycle  of "<< ddg_filename<<" = "<<loop.CRITICAL_CYCLE(el)<<endl;
115     cout<<"Edges belonging the the critical cycle"<<endl;
116     forall(e, el){
117         ie=loop[e];
118         cout<< loop.source_id(e)<< " -> "<< loop.target_id(e);
119         cout<<" : "<< ie<<endl;
120     }
121
122
123     //------- The user can outputs the DDGs to a .xml format --------
124     snprintf(str, MAX_STRING, "%s_temp.xml", ddg_filename);;
125     loop.write_to_xml(str); // Outputs the DDG loop to a .xml file format
126
127     i=loop.read_from_xml(str); // RE-reading the DDG loop from a .xml file erases the previous
128     if(i==-1) return EXIT_FAILURE;
129
130     // ------ The user can visualize the DDGs ----------------------
131     snprintf(str, MAX_STRING, "%s.vcg",ddg_filename); // sets the .vcg filename for the ddg loo
132     loop.write_to_vcg(str); //outputs the DDG loop to a vcg file to be visualized by the xvcg t
133     cout<<endl<<str<< " can be visualized using xvcg."<<endl;
134
135     loop_copy.copy(loop); // create an exact copy of the loop
136     snprintf(str, MAX_STRING, "%s_copy.vcg",ddg_filename);
137     //exports the copy loop to a vcg file to be visualized by the xvcg tool
```

```
138     loop_copy.write_to_vcg(str);
139
140     // The user can access any kind of information
141     T=loop.T();
142     forall(rt, T){
143         cout <<"The set of values of type "<< rt.get_register_typename() <<" is :"<<endl;
144         VR=loop.get_V_R(rt.get_register_type_id());
145         forall(u, VR){
146             it=loop.instruction_type(u);
147             cout<< loop.id(u)<< " with opcode " << it.opcode()<< " ("<<loop.latency(u)<<","<<lo
148         }
149     }
150     //----- We can unroll the loop while preserving recurrent edges
151     unroll(loop, unrolled, 3);
152     snprintf(str, MAX_STRING, "%s_unrolled_3.vcg",ddg_filename); // sets the .vcg filename for
153     unrolled.write_to_vcg(str); //outputs the DDG loop to a vcg file to be visualized by the xv
154     cout<<endl<<str<< " can be visualized using xvcg."<<endl;
155     forall (rt, T){
156         cout <<"The set of flow edges of the loop of type "<< rt.get_register_typename()<<" is
157         ER=unrolled.get_E_R(rt.get_register_type_id());
158         forall(e, ER){
159         ie=unrolled[e];
160         cout<< unrolled.source_id(e)<<" -> "<< unrolled.target_id(e)<<" : "<< ie<<endl;
161         }
162     }
163
164     // ------------- Hiding and restoring nodes -------------
165     // look for two arbitrary nodes to hide
166     forall (rt, T){
167
168         if (loop.get_V_R(rt.get_register_type_id()).size()>=2) {
169         loop.hide_node(loop.get_V_R(rt.get_register_type_id()).choose()); // just to hide one a
170         loop.hide_node(loop.get_V_R(rt.get_register_type_id()).choose()); // hide another arbit
171         break;
172         }
173     }
174     cout<<"Critical Cycle  of "<< ddg_filename<<" (after hiding two arbitrary values of type "<
175     cout<<"Edges belonging the the critical cycle"<<endl;
176     forall(e, el){
177         ie=loop[e];
178         cout<< loop.source_id(e)<< " -> "<< loop.target_id(e);
179         cout<<" : "<< ie<<endl;
180     }
181
182
183     loop.restore_all_nodes();
184     loop.restore_all_edges();
185
186     cout<<"Critical Cycle  of "<< ddg_filename<<" (after restoring all values and edges) = "<<l
187     cout<<"Edges belonging the the critical cycle"<<endl;
188     forall(e, el){
189         ie=loop[e];
190         cout<< loop.source_id(e)<< " -> "<< loop.target_id(e);
191         cout<<" : "<< ie<<endl;
192     }
193
194
195     //--- Many other software opportunities exist. See the reference manual.
196
197     return EXIT_SUCCESS;
198 }
199
```

5.4 RS_example.cpp

This is an example of how to compute the register saturation of a DAG.

```
1 /*****************************************************************************
2  *   Copyright (C) 2009 by Touati Sid                                        *
3  *   Sid-nospam.Touati-nospam@uvsq.fr                                        *
4  *   Copyright  INRIA and the University of Versailles                       *
5  *   This program is free software; you can redistribute it and/or modify    *
6  *   it under the terms of the GNU General Public License as published by    *
7  *   the Free Software Foundation; either version 2 of the License, or       *
8  *   (at your option) any later version.                                     *
9  *                                                                           *
10  *   This program is distributed in the hope that it will be useful,        *
11  *   but WITHOUT ANY WARRANTY; without even the implied warranty of          *
12  *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the           *
13  *   GNU General Public License for more details.                            *
14  *                                                                           *
15  *   You should have received a copy of the GNU Library General Public       *
16  *   License along with this program (LGPL); if not, write to the            *
17  *   Free Software Foundation, Inc.,                                         *
18  *   59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.               *
19  *   Exception : this software requires a licence  of the LEDA libary.       *
20  *   If you are from academia, you can get a free leda binary licence
21  *   allowing you to use freely our DDG library.                             *
22  *   Otherwise, you can buy a LEDA licence from algorithmic-solutions        *
23  *   http://www.algorithmic-solutions.com                                    *
24  *   LEDA binaries and source codes are excluded from this LGPL.             *
25  *   Obtaining a LEDA licence is not mandatory to use our GPL software       *
26  *   If the user wants, he can  create a free source-compatible replacement*
27  *   for LEDA, and he is allowed to use our software under LGPL.             *
28  *****************************************************************************/
29
34 #include "DDG.h"
35 #include <iostream>
36 #include <cstdlib>
37
38
39 using namespace std;
40 using namespace DDG; // the namespace of the DDG library.
41
42
43
44 int main(int argc, char *argv[])
45 {
46
47     // some strings for file names
48
49     char        str[MAX_STRING];
50
51     // some objects for graphs
52     DDG::DAG dag;  // declare a DAG of a basic block (directed acyclic graph)
53     DDG::ARCHITECTURE isa_arch;
54     char        *ddg_filename=NULL, *arch_filename=NULL, *regtypename=NULL;
55     LEDA::node n,u;
56     LEDA::edge e;
57     int c,i,t;
58     LEDA::list<node> nl;
59     INSTRUCTIONS_TYPES it;
60     LEDA::set<node> MA; // for a maximal antichain computation
61     LEDA::set<node> VR; // Set of values
62     LEDA::node_array<node> killer;
```

```
63      REGISTER_TYPES rt;
64      LEDA::list<REGISTER_TYPES> T;
65
66      while ((c = getopt (argc, argv, "i:a:t:")) != -1){
67          switch(c){
68              case 'a': arch_filename=optarg;
69                  break;
70              case 'i': ddg_filename =optarg;
71                  break;
72              case 't': regtypename=optarg;
73                  break;
74              case '?':
75                  cout<<"usage:"<< argv[0] <<" -i ddg_filename.xml [-a isa_desc.xml] [-t register_
76                  return EXIT_FAILURE;
77          }
78
79      }
80      if (arch_filename!=NULL)  {
81          if (!isa_arch.read_from_xml(arch_filename)) return EXIT_FAILURE;
82          if(isa_arch.check()){
83              dag=DAG(isa_arch);   //build an empty DDG loop with a user defined architcural (ISA)
84          } else return EXIT_FAILURE;
85
86      }
87       //---- The user can read a DDG from an xml format defined by the DDG library
88      if (ddg_filename != NULL){
89          list<edge> le;
90          edge e;
91          i=dag.read_from_xml(ddg_filename); // reads the DDG loop from an xml file
92          if(i==-1) return EXIT_FAILURE;
93          if(! Is_Acyclic(dag,le)==true) {
94              cerr<<"Input Error: Non Acyclic graph. Illegal arcs:\n";
95              forall(e, le){
96                  cerr<<dag.id(dag.source(e))<<" -> "<< dag.id(dag.target(e))<<endl;
97              }
98              return EXIT_FAILURE;
99          }
100
101      }
102      else{
103          cout<<"usage:"<< argv[0] <<" -i ddg_filename.xml [-a isa_desc.xml][-t register_type_nam
104          return EXIT_FAILURE;
105      }
106
107      if (regtypename!=NULL){
108
109          if(!isa_arch.exist_register_typename(regtypename)){
110              cerr<<"Register type of name "<<regtypename<<" does not exist in the ISA.\n";
111              return EXIT_FAILURE;
112          }
113          t=isa_arch.get_register_type_id( regtypename );
114          cout<<"Register Saturation of type "<< regtypename<<" of the DAG (loop body) of "<<ddg_
115          <<REGISTER_SATURATION (dag, t, nl, killer)<<endl;
116          cout<<"Saturating values of type "<< regtypename <<endl;
117          forall(n, nl){
118              it= dag.instruction_type(n);
119              cout<< dag.id(n)<< " with opcode " << it.opcode()<<endl;
120          }
121          cout<<"Saturating killing function of type "<< regtypename <<endl;
122          VR=dag.get_V_R(t);
123          forall(n, VR){
124          if(killer[n]!=NULL)     cout<<"k("<<dag.id(n)<<")="<<dag.id(killer[n])<<endl;
```

```
125         else cout<<"k("<<dag.id(n)<<")= bottom"<<endl;
126         }
127         return EXIT_SUCCESS;
128     }
129
130     T=dag.T();
131
132
133     forall(rt, T) {
134         t=rt.get_register_type_id();
135         cout<<"Register Saturation of type "<< rt.get_register_typename()<<" of the DAG (loop k
136         <<REGISTER_SATURATION (dag, t, nl, killer)<<endl;
137         cout<<"Saturating values of type "<< rt.get_register_typename() <<endl;
138         forall(n, nl){
139             it= dag.instruction_type(n);
140             cout<< dag.id(n)<< " with opcode " << it.opcode()<<endl;
141         }
142         cout<<"Saturating killing function of type "<< rt.get_register_typename() <<endl;
143         VR=dag.get_V_R(t);
144         forall(n, VR){
145         if(killer[n]!=NULL)     cout<<"k("<<dag.id(n)<<")="<<dag.id(killer[n])<<endl;
146         else cout<<"k("<<dag.id(n)<<")= bottom"<<endl;
147         }
148     }
149     return EXIT_SUCCESS;
150 }
```

6 DDG Page Documentation

6.1 Deprecated List

Global DDG::DAG::write_to_gl(const char *filename) const

Global DDG::DAG::write_to_gl(const char *filename, int regtype_id) const

Global DDG::DAG::read_from_gl(const char *filename)

Global DDG::DAG::read_from_gl(const char *filename, int regtype_id)

Index