# Documentation

Student: Shangkun LI
Student ID: 20307130215
Department of Physics

# 目录



# 1 项目说明

## 1.1 功能说明

本项目利用代码框架中提供的`parser.h`和`parser.cpp`文件读取并解析IR文件，生成对应的DFG（Data-Flow Graph）和CFG（Control-Flow Graph）。
随后对得到的数据控制流图在资源限制下进行调度，最后输出调度后的结果（包括CDFG的结构，，CDFG内每一个算子的开始结束周期及其种类）。

## 1.2 文件结构

该项目以如下的文件结构组织：

```
HLS_Project # 项目文件夹
```

```
├── build   # 编译文件夹
│   ├── CMakeCache.txt
│   ├── CMakeFiles
│   ├── cmake_install.cmake
│   ├── Makefile
│   └── Schedule
├── CMakeLists.txt    # CMake文件，可针对不同系统生成对应的Makefile
├── include # 源文件包含的头文件
│   ├── controlflow.h
│   ├── dataflow.h
│   ├── hls.h
│   ├── parser.h
│   └── schedule.h
├── README.md  # 说明文档
├── src  # 源文件
│   ├── controlflow.cpp
│   ├── dataflow.cpp
│   ├── hls.cpp
│   ├── main.cpp
│   ├── parser.cpp
│   └── schedule.cpp
└── test # 测试文件及测试结果
    ├── cdfg.txt
    ├── scheduled_CDFG.txt
    └── test.ll
```

# 2 算法说明

本项目可分为3部分实现：

1. 第一部分是设计整体架构，包括定义了`HLS`类，用于整合本项目所需要的数据结构和操作。
2. 第二部分是生成数据控制流图（CDFG），这一部分我们分别定义了`DataFlowGraph`类和`ControlFlowGraph`类用于整合生成CDFG所需的数据结构和操作。
3. 第三部分是对CDFG进行调度，这一部分定义了函数操作和数据结构。

## 2.1 顶层架构

包含文件：

```
HLS_Project
├── include
│   ├── hls.h
│   └── parser.h
└── src
    ├── hls.cpp
    ├── main.cpp
    └── parser.cpp
```

### 2.1.1 数据结构定义

定义了`HLS`类，其中包含：

```
# 变量
HLS::ir_parser # 实例化parser，用于解析IR文件
HLS::CFG # 存储的Control-flow Graph，后续操作均对其进行

# 函数
void HLS::generate_CFG() # 调用后后生成CFG并存储在HLS::CFG中，并输出未调度的CFG结果于HLS_Project/test目录下
void HLS::scheduling() # 调用后执行调度操作，并输出调度后的CFG结果于HLS_Project/test目录下

```

### 2.1.2 函数说明

1. `void HLS::generate_CFG()`
   调用该函数后，将对输入的`*.ll`文件进行解析，生成CFG变量，并输出CFG结果，其中包括每个算子的输入输出变量，与每个DFG相连的DFG及变量，以及DFG之间的跳转条件。每个DFG和他们之间的跳转条件构成了最终的CDFG。
2. `void HLS::scheduling()`
   调用该函数之后，将对已有的CFG进行调度，生成调度后的结果。其中调度后的结果除输出上述的CDFG结构外，还包括每个算子的开始与结束周期。本调度算法支持多周期算子的调度。

## 2.2 生成CDFG

包含文件：

```
HLS_Project
├── include
│   ├── controlflow.h
│   └── dataflow.h
└── src
    ├── controlflow.cpp
    └── dataflow.cpp
```

### 2.2.1 数据结构定义

1. 定义`DataFlowGraph`类，重要的变量及函数包括：

```
# 变量
DataFlowGraph::vector<int> Mark  # 用于标记该DFG中的node是否已被访问
DataFlowGraph::vector<int> Indegree # 存储DFG中每个node的入度
DataFlowGraph::vector<int> Outdegree   # 存储DFG中每个node的出度

# 函数
void DataFlowGraph::CreateEdge(int source, int des)   # 创建一个source node到des node的边
vector<Node> &DataFlowGraph::get_nodeList()  # 返回节点列表
vector<BranchEdge> &DataFlowGraph::get_branches()  # 返回边列表
vector<InputEdge> &DataFlowGraph::get_inputList()  # 返回DFG的输入变量
vector<OutputEdge> &DataFlowGraph::get_outputList()  # 返回DFG的输出变量
vector<int> &DataFlowGraph::get_desNode(int source)   # 返回与当前node相连的node
unordered_map<string, int> &DataFlowGraph::get_varTable()   # 返回一个node名称和对应index的映射的结构
```

2. 定义`ControlFlowGraph`类，重要的变量及函数包括：

```
# 变量
string func_name  # 每个CFG对应一个函数，这是CFG（函数）的名称
vector<graph_node> DFGs # CFG中的DFG节点
unordered_map<string, int> IndexDFG # 将每个DFG的名字与index映射起来

# 函数
vector<graph_node> &ControlFlowGraph::get_DFGNodes()  # 返回CFG中的DFG节点
int ControlFlowGraph::get_Index(string label)   # 根据DFG的名字返回对应的index
vector<int> ControlFlowGraph::NextNode(int index)  # 返回与目前DFG相连的下一个DFG的index
```

### 2.2.2 函数说明

1. `DataFlowGraph`类中函数说明
    - `vector<Node> &DataFlowGraph::get_nodeList()`：使用`vector`存储，方便快速索引并且访问速度快
    - `vector<BranchEdge> &DataFlowGraph::get_branches()`：可以得到在一个DFG内node与node之间的相连情况，其中`BranchEdge`类中存储了这一条边的跳转方向和跳转条件
    - `unordered_map<string, int> &DataFlowGraph::get_varTable()`：这是一个Hash Table，可以从一个变量名快速索引到对应的index，从而实现快速查找。
2. `ControlFlowGraph`类中函数说明
    - `vector<graph_node> &ControlFlowGraph::get_DFGNodes()`：返回一个由CFG中DFG构成的`vector`，并实现DFG节点的快速索引

注意，本部分在创建DFG的时候，在首尾各创建了一个空的虚拟节点，首节点意义是作为外部变量输入的索引位置，而末节点的意义是作为需要输出到外部的变量的索引位置。
利用哈希表在所有的数据流图中匹配需要的信息并进行更新和存储最终生成了这样的一个完整的数据流图。然后是在创建控制流图的时候为了输出三个输入变量生成了一个名为`fiction_head`的虚拟节点。

## 2.3 调度

包含文件：

```
HLS_Project
├── include
│   └── schedule.h
└── src
    └── schedule.cpp
```

### 2.3.1 数据结构定义

1. 在 `include/dataflow.h` 文件中定义了相关宏，用于描述每个算子运行所需的周期数，用于后续完成多周期算子的调度。
2. 定义 `HardwareRes` 类，其中包括：

```
# 变量
int adder   # 加法器个数
int mul  # 乘法器个数
int div  # 除法器个数
int sram # sram个数
int adder_aval # 空闲的加法器
int mul_aval   # 空闲的乘法器
int div_aval   # 空闲的除法器
int sram_aval  # 空闲的sram
```

3. 定义 `InitSchedule` 类，用于存储ASAP和ALAP的调度结果，其中包括：

```
# 变量
vector<pair<int, int>> ASAP_RESULT  # ASAP算法调度结果，第一个int为算子开始时间，第二个int为算子结束时间
vector<pair<int, int>> ALAP_RESULT  # ALAP算法调度结果
```

### 2.3.2 函数说明

除了上述变量之外，还在 `schedule.h` 中声明了其他函数，简介如下：

1. `bool meet_resources_constraint(HardwareRes &res, int i, DataFlowGraph &DFG)`：用于判断当前运算操作是否满足硬件约束，如果满足才可以被调度，并且消耗掉对应的一个硬件资源（`HardwareRes` 中对应的 `*_aval` 变量减1）。
2. `void reset(HardwareRes &hardware, int i, DataFlowGraph &DFG)`：根据运算操作释放对应的运算单元， `HardwareRes` 中对应的 `*_aval` 变量加1.
3. `void ASAP(DataFlowGraph &DFG, InitSchedule &InitResult)`：利用ASAP算法对DFG进行调度，主要基于图的拓扑排序操作，并将结果存储在 `InitSchedule` 类中。
4. `void ALAP(DataFlowGraph &DFG, InitSchedule &InitResult)`：利用ALAP算法对DFG进行调度，需要先进行一次ASAP调度获得最长调度周期后，根据出度反向进行拓扑排序操作，并将结果存储在 `InitSchedule` 类中。
5. `void ListSchedule_for_DFG(DataFlowGraph &DFG)`：对DFG图和开始进行的ASAP、ALAP调度结果进行列表调度法

6. `void Schedule_for_CFG(ControlFlowGraph &CFG)`：对每个CFG中的DFG进行列表调度，最后得到总的调度结果。

# 3 编译及运行示例

## 3.1 编译

1. 跳转到`HLS_Project`目录下，在命令行中执行`mkdir build && cd build`
2. 再执行`cmake ..`生成对应的Makefile，然后执行`make`生成最终的可执行文件
3. 可执行文件位于`HLS_Project/build`目录中，为`Schedule`

## 3.2 执行

1. 当前位于`HLS_Project/build`目录下，将需要解析的IR文件放在`HLS_Project/test`目录下
2. 在命令行中执行`./Schedule ../test/test.ll`，然后将在命令行中输出`parser`的解析结果以及如下提示信息

```
Finish CDFG Generation

Finish Scheduling

```

3. 在`HLS_Project/test`目录下会发现两个文件分别为`cdfg.txt`和`scheduled_CDFG.txt`。其中第一个文件存储了生成的CDFG结构；第二个文件存储了调度后的CDFG结果，调度结果以每个node的开始时间和结束时间表示。

# 附录

Support IR format:

1. Function define:

```
define int foo(int a, int b); return value could be int/void
```

2. Operators:

```
support arrays: int a[100]; define int foo(int a[], int b[], int N);
load: load a value from array. a[10] cannot be used directly. load(a, 10);
store: save a value to array. store(a, 10, c) -> a[10] = c;
=: assign value to a variable, follows the static single assignment rule
+:
   c = a + b;
-:
   c = a - b;
*:
   c = a * b;
/:
   c = a / b;
```

```
==:
   cond = a == b;
<:
   cond = a < b;
>:
   cond = a > b;
>=:
   cond = a >= b;
<=:
   cond = a <=b;
br:
   br label or br cond label1 label2
phi:
   phi function, select the right value from different blocks, according to SSA rule
   phi(value1, block_label1, value2, block_label2, ...); The default block label from the definition of the function is 0.
return:
   return a value
```

3. Examples

```
define int dotprod(int a[], int b[], int n)
    c = 0;

start:
    i = phi(0, 0, i_inc, calc);
    cond = i >= n;
    br cond ret calc

calc:
    ai = load(a, i);
    bi = load(b, i);
    cl = phi(c, 0, cr, start);
    ci = ai * bi;
    cr = cl + ci;
    i_inc = i + 1;
    br start;

ret:
    cf = phi(0, c, start, cr);
    return cf;
```