CUDD
Colorado University Decision Diagram Package

Software per Sistemi Embedded

Corso di Laurea in Informatica

Davide Bresolin       Tiziano Villa
1. Introduction

2. CUDD: Basic Architecture

3. Example: Half-Adder
CUDD

- CUDD is the Colorado University Decision Diagram Package.

- It is a C/C++ library for creating different types of decision diagrams:
  - binary decision diagrams (BDD);
  - zero-suppressed BDDs (ZDD);
  - algebraic decision diagrams (ADD)

- This lesson is only on the BDD functionality of CUDD
Getting CUDD

- You can download CUDD by FTP with anonymous login from vlsi.colorado.edu
- The latest version is 3.0.0
- How to install it on the Lab Computers (and any Linux distribution):

  ```
  export CUDD_INSTALL_DIRECTORY=$HOME/<install_dir>
  mkdir CUDD_INSTALL_DIRECTORY
  wget ftp://vlsi.colorado.edu/pub/cudd-3.0.0.tar.gz
  tar xzfv cudd-3.0.0.tar.gz
  cd cudd-3.0.0
  mkdir objdir && cd objdir
  ../configure --prefix=$CUDD_INSTALL_DIRECTORY
  make && make install
  ```
The CUDD library has two main header files:

- `#include <cudd.h>` for the C library
- `#include <cuddObj.h>` for the C++ library

We will use the C library

The package is split into many different libraries:

`libcudd.a, libutil.a, ...`

To compile and link a C program that uses CUDD:

```
gcc -o main main.c -lcudd -lutil -lepd -lmtr -lst -lm
```
Outline

1. Introduction
2. CUDD: Basic Architecture
3. Example: Half-Adder
CUDD has a built-in garbage collection system.

When a BDD is not used anymore, its memory can be reclaimed.

To facilitate the garbage collector, we need to “reference” and “dereference” each node in our BDD:

- \texttt{Cudd\_Ref(DdNode*)} to reference a node
- \texttt{Cudd\_RecursiveDeref(DdNode*)} to dereference a node and all its descendants.
Complemented arcs

- Each node of a BDD can be:
  - a variable with two children
  - a leaf with a constant value

- The two children of a node are referred to as the “then” child and the “else” child

- To assign a value to a BDD, we follow “then” and “else” children until we reach a leaf:
  - the value of our assignment is the value of the leaf we reach

- However: “else” children can be complemented:
  - when an “else” child is complemented, then we take the complement of the value of the leaf:
    - i.e., if the value of the leaf is 1 and we have traversed an odd number of complemented arcs, the value of our assignment is 0.
Complemented arcs: example

- $\text{out} = x_0 \overline{x}_1$
- “then” arcs are solid
- normal “else” arcs are dashed
- complemented “else” arcs are dotted
- the $\text{out}$ arc is complemented:

$$\overline{\text{out}} = \overline{x}_0 + x_1$$
$$= \overline{x}_0 + x_0 x_1$$
The half-adder circuit

This is a half adder circuit that we will compile into an OBDD.

It has the following truth table:

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>sum</th>
<th>carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
The DdManager is the central data structure of CUDD:

- It must be created before calling any other CUDD function.
- It needs to be passed to almost every CUDD function.

To initialize the DdManager, we use the following function:

```c
DdManager * Cudd_Init(
    unsigned int numVars, // initial number of BDD variables (i.e., subtables)
    unsigned int numVarsZ, // initial number of ZDD variables (i.e., subtables)
    unsigned int numSlots, // initial size of the unique tables
    unsigned int cacheSize, // initial size of the cache
    unsigned long maxMemory // target maximum memory occupation. (0 means unlimited)
);```

There is one caveat with CUDD. "else" children can be complemented. If the else child is complemented, then when we reach a leaf node, we would take the complement of the value of the leaf. i.e., if the value of the leaf is 1 and we have traversed through and odd number of complement arcs, the value of our assignment is 0. In 5.4.3, we will discuss how to deal with complement arcs.
#include<stdio.h>
#include<cudd.h>

int main() {
    DdManager* manager=Cudd_Init(0, 0,
    CUDD_UNIQUE_SLOTS, CUDD_CACHE_SLOTS, 0);
    if(manager == NULL) {
        printf("Error when initializing CUDD.\n");
        return 1;
    }

    ...

    return 0;
}
The **DdNode**

The **DdNode** is the core building block of BDDs:

```c
struct DdNode {
    DdHalfWord index; // Index of the variable represented by this node
    DdHalfWord ref; // Reference count
    DdNode *next; // Next pointer for unique table
    union {
        CUDD_VALUE_TYPE value; // For constant nodes
        DdChildren kids; // For internal nodes
    } type;
};
```

- **index** is a unique index for the variable represented by this node.
  - It is permanent: if we reorder variables, the index remains the same
- **ref** stores the reference count for this node.
  - It is incremented by `Cudd_Ref` and decremented by `Cudd_Recursive_Deref`
Create the BDD for \textbf{sum}

DdNode* x1 = Cudd_bddIthVar(manager, 0);
DdNode* x2 = Cudd_bddIthVar(manager, 1);

DdNode* and1;
and1 = Cudd_bddAnd(manager, x1, Cudd_Not(x2));
Cudd_Ref(and1);

DdNode* and2;
and2 = Cudd_bddAnd(manager, Cudd_Not(x1), x2);
Cudd_Ref(and2);

DdNode* sum;
sum = Cudd_bddOr(manager, and1, and2);
Cudd_Ref(sum);

Cudd_RecursiveDeref(manager, and1);
Cudd_RecursiveDeref(manager, and2);

\textbf{Exercise:} write the code for \textit{carry}
Restricting a BDD means assigning a truth value to some of the variables.

- **BDD** is the original BDD to restrict
- **restrictBy** is the truth assignment of the variables:
  - AND of variables and complemented variables
- the function returns the restricted BDD

```c
DdNode * Cudd_bddRestrict(
    DdManager * manager,  // DD manager
    DdNode * BDD,          // The BDD to restrict
    DdNode * restrictBy)   // The BDD to restrict by.
```

```c
for (int i = 0; i < SIZE; i++) {
    Cudd RecursiveDeref(manager, testSum[i]);
    Cudd RecursiveDeref(manager, testCarry[i]);
}
```
Print the truth table of the Half-adder

DdNode *restrictBy;
restrictBy = Cudd_bddAnd(manager, x1, Cudd_Not(x2));
Cudd_Ref(restrictBy);

DdNode *testSum;
testSum = Cudd_bddRestrict(manager, sum, restrictBy);
Cudd_Ref(testSum);
DdNode *testCarry;
testCarry = Cudd_bddRestrict(manager, carry, restrictBy);
Cudd_Ref(testCarry);

printf("x1 = 1, x2 = 0: sum = %d, carry = %d\n",
       1 - Cudd_IsComplement(testSum),
       1 - Cudd_IsComplement(testCarry));

Cudd_RecursiveDeref(manager, restrictBy);
Cudd_RecursiveDeref(manager, testSum);
Cudd_RecursiveDeref(manager, testCarry);

Exercise:
write the code for the complete truth table
Print the BDD: graphviz

- The function `Cudd_DumpDot` dumps the BDD to a file in GraphViz format.
- The `.dot` file can be converted to a PDF by the command `dot`:
  ```sh
dot -O -Tpdf half_adder.dot
  ```
char* inputNames[2];
inputNames[0] = "x1";
inputNames[1] = "x2";
char* outputNames[2];
outputNames[0] = "sum";
outputNames[1] = "carry";

DdNode* outputs[2];
outputs[0] = sum;
Cudd_Ref(outputs[0]);
outputs[1] = carry;
Cudd_Ref(outputs[1]);

FILE* f = fopen("half_adder.dot", "w");

Cudd_DumpDot(manager, 2, outputs, inputNames, outputNames, f);

Cudd_RecursiveDeref(manager, outputs[0]);
Cudd_RecursiveDeref(manager, outputs[1]);
fclose(f);
Variable reordering

- The order of variables can have a tremendous effects on the size of BDDs
- CUDD provides a rich set of tools for reordering BDDs:
  - Automatic reordering (using heuristics) when the number of nodes in the BDD passes a certain threshold
  - Manual reordering using different heuristics
  - Manual reordering with a user-specified variable order

The function `Cudd_ShuffleHeap` is used to define the variable order:

```c
int Cudd_ShuffleHeap(
    DdManager * manager,   // DD manager
    int * permutation       // required variable permutation
)
```

The final parameter is the minimum number of nodes that must be in the BDD in order to reorder. This prevents the cost of reordering small enough BDDs.
Exercise: play with the variable order!

- Create the BDD for the function $x_1 x_2 + x_3 x_4 + x_5 x_6$

- Try the following variable orders and compare the results:
  - $x_1 < x_2 < x_3 < x_4 < x_5 < x_6$
  - $x_1 < x_3 < x_5 < x_2 < x_4 < x_6$

**HINTS**

- `int Cudd_ReadPerm(manager, x2->index)` returns the position of variable $x_2$ in the order
- `int Cudd_ReadNodeCount(manager)` returns the number of nodes in the BDD