Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Bakalářská práce

Randomizace algoritmů v systému ABC

Jakub Lerl

Vedoucí práce: doc.Ing. Petr Fišer, Ph.D.

12. května 2015

Poděkování

Chtěl bych upřímně poděkovat svému vedoucímu práce za trpělivost, ochotu spolupracovat a hlavně pevné nervy, které jsem často pokoušel. Dále bych chtěl poděkovat svému kamarádovi Tomášovi, kterého považuji za nejlepšího programátora co zná.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 12. května 2015 .....

Odkaz na tuto práci

Lerl, Jakub. *Randomizace algoritmů v systému ABC*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Logická syntéza je důležitým prvkem v oboru Číslicových obvodů. Nástroj ABC pro logickou syntézu využívá plně deterministických algoritmů. Během procesu není prováděn žádný náhodný výběr. Algoritmy při stejném vstupu tedy vrátí pokaždé stejný výstup. Algoritmy jsou ovšem velmi citlivé (mimo jiné) na změnu např. pořadí vstupů. Randomizování ekvivalentních voleb v těchto algoritmech tak vede k nalezení různých řešení, která mohou být kvalitnější než u deterministického algoritmu.

V této práci jsem úspěšně randomizoval algoritmy *balance* a *rewrite*. Testování algoritmu *rewrite* dopadlo podle očekávání a jeho používání v syntézním procesu může přinést pozitivní výsledky. Randomizovaný *balance* ovšem generoval horší výsledky než deterministický, byl proto několikrát upravován do formy, která je konkurenceschopná deterministickému. Podařilo se mi dosáhnout průměrného zhoršení pouze 0.09% oproti referenčnímu *balance*, což už už se dá považovat za statistickou chybu.

Klíčová slova Obvody, Randomizace algoritmu, Minimalizace, Logická syntéza, Balance, Rewriting, ABC.

Abstract

Logic synthesis is an important topic in circuit domain. The ABC tool uses fully deterministic algorithms for logic synthesis. No random decisions are made in the process and for any input it returns the same output. Nonetheless algorithms are very sensitive to changes of variable ordering. Randomization of equal choices in the process may lead to find better results.

In this thesis I successfully implemented randomization of *balance* and *rewrite*. Testing randomized *rewrite* ended as expected and its use in synthesis process may bring positive results. Nonetheless testing randomized *balance* generated worse results than deterministic one. So I reworked it many times to achieve algorithm which is able to compete that. I implemented semi-randomized *balance*. It generates better results on average than fully-randomized algorithm and it is worse than deterministic *balance* entirely by 0.09%. This difference can be looked at just as statistical margin for error.

Keywords Circuits, Randomized algorithm, Minimalization, Logic synthesis, Balance, Rewriting, ABC.

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Logická syntéza je proces, který požadované chování obvodu v abstraktní formě převede do designové implementace v rámci logických hradel. Logická optimalizace je pak část logické syntézy, která se snaží, aby výsledný obvod podléhal specifickým omezením. V praxi se nejčastěji klade důraz na minimální počet hradel a hloubku obvodu.

Základy logické syntézy se objevují v roce 1939, kdy Claude Shannon poukázal na fakt, že lze dvojhodnotovou logikou popsat operace obvodů [3]. Zprvu se k minimalizaci přistupovalo například s pomocí Karnaughových map [4]. Ovšem s rostoucím počtem vstupů tento způsob přístupu ztrácí na efektivitě. Později tento způsob vystřídaly sofistikovanější algoritmy (Quine - McCluskey algorithm [5]), které mohl provádět počítač. Nástup nových technologií (PLA, FPGA) [6] zvýšil potřebu po efektivnějších algoritmech. Logická optimalizace je NP-těžkým [7] problémem [8]. V takovém případě je vhodné s rostoucím počtem vstupů zvolit jinou metodiku k nalezení řešení.

V současných nástrojích [9] se v syntézním procesu používá kombinace více deterministických algoritmů a heuristik [10]. V průběhu těchto algoritmů dochází k výskytu více ekvivalentních voleb, jejichž výběr na funkcionalitu daného algoritmu nemá vliv. Zato na výsledné řešení vliv má. To znamená, že se během syntézního procesu neprovádí žádné náhodné volby, a při stejných vstupech budeme dostávat pokaždé stejný výstup. Na jednu stranu je velmi snadné se k danému řešení znovu dostat například opakovaným spouštěním. Na druhou stranu potenciálně lepšího řešení z důvodu determinizmu nemusí být nikdy dosaženo.

Kapitola 1

Cíl Práce

Cílem práce je do nástroje ABC (open-source nástroj pro logickou syntézu) zavést prvky náhodnosti. V ABC je implementována řada syntézních algoritmů, které jsou plně deterministické, tj. při opakovaném spouštění produkují stejné výsledky. Na druhou stranu jsou však citlivé např. na pořadí vstupů syntetizovaného obvodu, což svědčí o přítomnosti algoritmů, do kterých by mohlo být možné vložit prvek náhodného výběru, bez narušení funkčnosti. Mým úkolem tedy bude náhodnost do algoritmu vložit. Dále mám za úkol provést měření a porovnat randomizované řešení s deterministickým.

Kapitola 2

Seznam pojmů

Boolovská síť (Boolean network) je orientovaný acyklický graf, jehož uzly reprezentují logická hradla a hrany reprezentují propojení mezi nimi. Boolovskou síť můžeme nazývat také třeba síť nebo obvod.

Primární Vstup (PO - Primary Input) je uzel, ke kterému nevede žádná hrana.

Primární Výstup (PO - Primary Output) pak nazýváme uzel, ze kterého žádná hrana nevede.

And Inverted Graph (AIG) je boolovská síť, jež se skládá pouze z dvouvstupých AND uzlů, a invertorů.

Hloubka (depth/delay) AIG je pak největší počet uzlů od PI k PO.

Velikost (area/size) sítě je pak počet všech uzlů v síti.

Řez (cut) uzlu n je množina uzlů sítě, pro kterou platí, že všechny cesty od primárního vstupu k n prochází uzly z této množiny. Tento řez je fanout-free pakliže uzly, patřící do řezu nemají žádné další výstupy mimo daný řez.

Lokální funkce (local function) AIG uzlu n je boolovská funkce AIG podgrafu s kořenem v uzlu n vyjádřena listy řezu uzlu n.

Strukturní Hashování AIG je způsob ukládání uzlů. Strukturní hashování přináší několik výhod jako např. sloučení uzlů se stejnými vstupy. Při vytváření uzlů se jeho vstupy „zahashují“ a přidají do AIG manageru [11] [12] viz. dále.

Hloubka uzlu Hloubka uzlu n je počet uzlů mezi Primárním vstupem a n včetně uzlu samotného. Primární vstup má hloubku 0.

Topologické pořadí Topologické uspořádání je taková posloupnost uzlů grafu, že pro každou jeho hranu (u, v) platí, že uzel u má nižší hloubku než uzel v. Topologicky lze proto uspořádat pouze acyklické grafy.

Binární rozhodovací diagram (BDD - Binary Decision Diagram) je způsob reprezentace logické funkce [13].

SuperAND je vektor, který reprezentuje vícevstupý AND v AIG. Používá se při tree balancing kroku v algoritmu *balance*. Více 4.1.

Faktorizace neboli sestrojení Faktorované formy z logické funkce ve formátu SOP. Podrobnosti 4.3.

Kapitola 3

Úvod do ABC

3.1 ABC

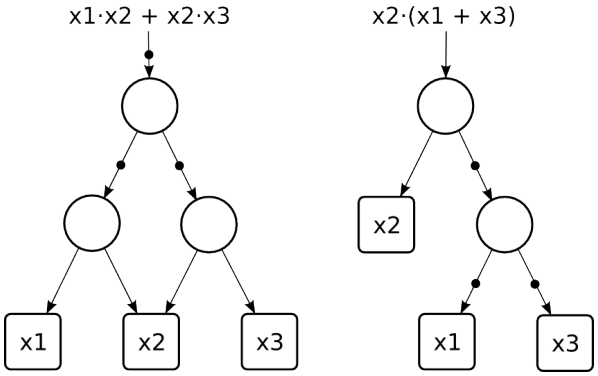
ABC je konzolový nástroj pro syntézu a minimalizaci číslicových obvodů [9] [12] [11]. Je v něm implementováno několik algoritmů pro optimalizaci sítě využívajících vlastností AIG. Umí načítat a ukládat soubory ve formátu PLA, BLIF, SLIF, Verilog a dalších. Dále dokáže ověřovat ekvivalenci dvou obvodů. ABC je napsaný v jazyce C. Autor zde používá „Maďarskou notaci“ [14]. ABC je open-source a zdrojový kód je tedy volně přístupný např. na online repozitáři [15]. V ABC je připravené inovativní programovatelné prostředí, přizpůsobené pro práci s obvody [11] [12]. Je zde implementována řada elementárních metod jako například:

- Vytvoření nové sítě (*Abc_NtkAlloc*)
- Označení primárních vstupů a výstupů(*Abc_NtkPi*)
- Vytvoření uzlu (*Abc_AigAnd*)
- Nastavení konstant (*Abc_AigReplace*)
- Provádět elementární logické funkce (*Abc_AigOr, Abc_AigXor atd...*)

3.2 AIG

AIG je primární reprezentace sítí v ABC. Je to orientovaný acyklický graf, jehož uzly reprezentují dvouvstupové AND hradla a hrany propojení. Každá hrana v AIG může mít doplňkový atribut, který reprezentuje invertor viz. 3.2. Tento způsob přístupu má mnoho výhod:

- Každá síť může být velmi snadno převedena na AIG
- Používá se zde jen jeden typ uzlu



Obrázek 3.1: Ukázka AIG. Uzly jsou ANDy, hrany spojení a černé puntíky na hranách invertory.

- Nedochází k prudkému vzrůstu velikosti s velikostí sítě a počtem PI (na rozdíl od BDD [13])
- Snadno se mapuje na technologie
- Části AIG se snadno upravují
- Snadné operace

AIG je vždy ukládán ve formě strukturního hashe. Většina algoritmů (*balance*, *collapse*, *renode*, *rewrite*, *refactor*, *retime*) v ABC pracuje s AIG a využívá výhod strukturního hashování:

- Každé 2 ANDy se stejnými vstupy jsou spojeny do jednoho.
- Nikde v grafu se nevyskytuje jednovstupý uzel.
- Propagování konstant

Převod obvodu na AIG se z konzole provede příkazem *strash*. Vizualizace malého AIG (do 100 uzlů) může být provedena příkazem *show*. Hashování probíhá za běhu při každém volání nějaké elementární AIG operace. Mimo jiné díky strukturnímu hashování je dosaženo následujících vlastností, které urychlují práci s AIG:

- Žádné volné uzly (tj. bez výstupního uzlu)
- AND uzly jsou uloženy v topologickém pořadí v poli objektů *pBins* viz. 3.1
- Konstantní uzel má vždy Identifikační číslo 0 v poli objektů (uzly, vstupy, výstupy, propojení)

- Hloubka každého uzlu je reflektována hloubkou jeho vstupů

Manažer pro strukturní hashování si uchovává informace o dané síti. V ukázce 3.2 můžeme vidět jak k samotnému hashování dochází při vytváření ANDu. Nejprve se otestuje, zda je hashovací tabulka dostatečně velká. Dále uspořádá uzly tak, aby levý předek měl menší Id než pravý předek a přidají se na vstup novému ANDu. Díky tomu máme zachovanou komutativitu vstupů. Mějme uzel F se vstupy a a b, pak funkce $F = ab$ $F = ba$ budou mít stejnou pozici v hashovací tabulce, protože se vždy uspořádají podle Id. Dále nám Hashovací funkce viz. 3.3 vrátí index, na který uložíme uzel. V případě, že již existuje uzel se stejnými vstupy, se na daný index nic neuloží, protože daná pozice už je obsazená, tudíž takový uzel už v grafu je.

```
struct Abc_Aig_t_
{
    // Aig síť
    Abc_Ntk_t *        pNtkAig;
    // Konstantní uzel
    Abc_Obj_t *        pConst1;
    // pole zahashovaných uzlu
    Abc_Obj_t **       pBins;
    // velikost hashovací tabulky
    int                nBins;
    // počet uzlů
    int                nEntries;
    // pole nezahashovaných uzlů
    Vec_Ptr_t *        vNodes;
    Vec_Ptr_t *        vStackReplaceOld;
    Vec_Ptr_t *        vStackReplaceNew;
    Vec_Vec_t *        vLevels;
    Vec_Vec_t *        vLevelsR;
    Vec_Ptr_t *        vAddedCells;
    Vec_Ptr_t *        vUpdatedNets;
};
```

Listing 3.1: Definice AIG manageru.

```
Abc_Obj_t * Abc_AigAndCreate( Abc_Aig_t * pMan, Abc_Obj_t * p0,
    Abc_Obj_t * p1 )
{
    Abc_Obj_t * pAnd;
    unsigned Key;
    // check if it is a good time for table resizing
    if ( pMan->nEntries > 2 * pMan->nBins )
        Abc_AigResize( pMan );
    // order the arguments
    if ( Abc_ObjRegular(p0)->Id > Abc_ObjRegular(p1)->Id )
        pAnd = p0, p0 = p1, p1 = pAnd;
    // create the new node
    pAnd = Abc_NtkCreateNode( pMan->pNtkAig );
    Abc_ObjAddFanin( pAnd, p0 );
    Abc_ObjAddFanin( pAnd, p1 );
    // set the level of the new node
    pAnd->Level  = 1 + Abc_MaxInt( Abc_ObjRegular(p0)->Level,
    Abc_ObjRegular(p1)->Level );
    pAnd->fExor  = Abc_NodeIsExorType(pAnd);
```

```
18      pAnd->fPhase = (Abc_ObjIsComplement(p0) ^ Abc_ObjRegular(p0)->
        fPhase) & (Abc_ObjIsComplement(p1) ^ Abc_ObjRegular(p1)->
        fPhase);
19      // add the node to the corresponding linked list in the table
20      Key = Abc_HashKey2( p0, p1, pMan->nBins );
21      pAnd->pNext        = pMan->pBins[Key];
22      pMan->pBins[Key] = pAnd;
23      pMan->nEntries++;
24      pAnd->pCopy = NULL;
25      // add the node to the list of updated nodes
26      if ( pMan->vAddedCells )
27          Vec_PtrPush( pMan->vAddedCells, pAnd );
28      return pAnd;
29  }
```

Listing 3.2: Ukázka hashování při vytváření AND uzlu.

```
1   static unsigned Abc_HashKey2( Abc_Obj_t * p0, Abc_Obj_t * p1, int
        TableSize )
2   {
3       unsigned Key = 0;
4       Key ^= Abc_ObjRegular(p0)->Id * 7937;
5       Key ^= Abc_ObjRegular(p1)->Id * 2971;
6       Key ^= Abc_ObjIsComplement(p0) * 911;
7       Key ^= Abc_ObjIsComplement(p1) * 353;
8       return Key % TableSize;
9   }
```

Listing 3.3: Ukázka hashovací funkce.

3.3 Uzly

Všechny komponenty v ABC jsou mimo jiné typu objekt (definice v souboru „src/base/abc/abc.h“), který si o sobě uchovává informace například jakého typu je (uzel, hrana, síť), má svoje unikátní ID - *int Id;* 32 bitové číslo, které je přidělování při vytváření obvodu.

Hodnota ID není v ABC skoro nikde využívána (kromě elementárních operací). Jediné Id, které nesmí být uzlům přiděleno je 0. Ta je rezervována pro konstantní uzel.

Objekt má dále mnoho dalších atributů viz. 3.4. Uzly mohou mít žádný nebo více výstupů, které mohou být dalšími uzly nebo propojení. Konstantní uzly nemají žádný vstupní uzel a Invertor má vždy jeden výstup. Každý uzel má svojí logickou funkci uloženou v proměnné *pNode->pData*, která je složená z vstupních uzlů. Funkce může nabývat více různých formátů jako jsou SOP [12] [1], BDD [16] [17] nebo hradla (AND). V AIG se ovšem logická funkce uchovávat nemusí díky vlastnostem strukturního hashování. Namísto toho je logická funkce uzlu definována oběma jeho syny, kteří jsou také ANDy.

3.4 Hrany

Hrana v AIG je objekt, který obsahuje nepovinný doplňkový atribut, který reprezentuje invertor. Hrany se vytvářejí za běhu při připojování uzlů funkcí *Abc_NtkCreateLatch(pAig);*.

```
1  struct Abc_Obj_t_       // 48/72 bytes (32-bits/64-bits)
2  {
3      Abc_Ntk_t *         pNtk;
4      Abc_Obj_t *         pNext;
5      int                 Id;
6      unsigned            Type     :  4;
7      unsigned            fMarkA   :  1;
8      unsigned            fMarkB   :  1;
9      unsigned            fMarkC   :  1;
10     unsigned            fPhase   :  1;
11     unsigned            fExor    :  1;
12     unsigned            fPersist:  1;
13     unsigned            fCompl0  :  1
14     unsigned            fCompl1  :  1;
15     unsigned            Level    : 20;
16     Vec_Int_t           vFanins;
17     Vec_Int_t           vFanouts;
18     union { void *      pData;
19       int               iData; };
20     union { void *      pTemp;
21       Abc_Obj_t *       pCopy;
22       int               iTemp;
23       float             dTemp; };
24 };
```

Listing 3.4: Definice třídy *Abc_Obj_t_*

Kapitola 4

Analýza Algoritmů

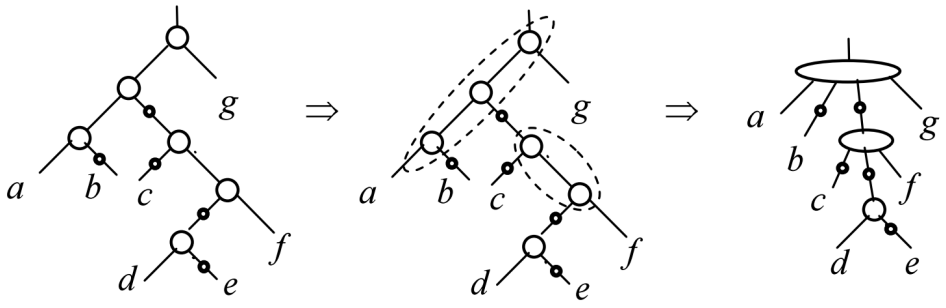
Syntézní proces napomáhá převést obvod z návrhu v abstraktní formě (např. RTL [18]) na technologický design. Na výsledný obvod jsou kladena různá specifika, která by měl splňovat. V současnosti se nejčastěji klade důraz na minimální počet hradel a minimální hloubku. Tato specifika ovšem z nalezení takového obvodu činí NP-těžký problém.

Pro syntézu se tak používají různě algoritmy, které napomáhají těchto specifik dosáhnout různými metodami. Algoritmus *balance* se snaží snížit hloubku obvodu, zatímco *rewriting* nebo *refactor* zase snižují počet AND hradel. Kombinace více algoritmů pak může najít uspokojivé řešení. V praxi se používají syntézní skripty, které obsahují sekvenci těchto algoritmů. V souboru „abc.rc“ jsou tyto skripty definovány jako sekvence příkazů ve formě aliasů (*st - strash, b - balance, rw - rwrite, rf - refactor, rs -resub...*). Jako ukázku zde uvedu několik syntézních skriptů, na které bude mít moje práce vliv:

- **resyn** - „b; rw; rwz; b; rwz; b“
- **resyn2** - „b; rw; rf; b; rw; rwz; b; rfz; rwz; b“
- **resyn2a** - „b; rw; b; rw; rwz; b; rwz; b“
- **resyn3** - „b; rs; rs -K 6; b; rsz; rsz -K 6; b; rsz -K 5; b“
- **compress** - „b -l; rw -l; rwz -l; b -l; rwz -l; b -l“
- **rwsat** - „st; rw -l; b -l; rw -l; rf -l“
- **choice** - „fraig_store; resyn; fraig_store; resyn2; fraig_store; fraig_restore“

4.1 Balance

Algoritmus *balance* dokáže snížit v mnoha případech hloubku AIG [1]. Vedlejším produktem může být i snížení počtu uzlů. Dosahuje toho pomocí dvou kroků:



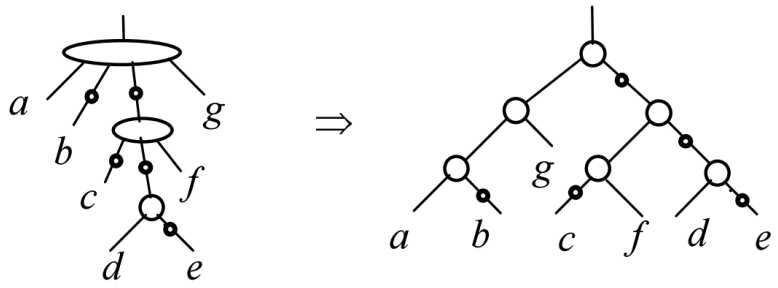
Obrázek 4.1: Ilustrace „Covering“ kroku. [1]

1. **Covering** - Algoritmus v prvním kroku nejprve identifikuje množiny sousedících (tj. takových, mezi kterými vede hrana) uzlů, které mezi sebou nemají žádný doplňkový atribut (tj. invertor) a spojí je do sebe, čímž vytvoří jeden mnohavstupý AND (dále superAND) 4.1. Vstupy superANDu seřadí podle hloubky sestupně. Pro třídění se v ABC používá funkce qsort [19].

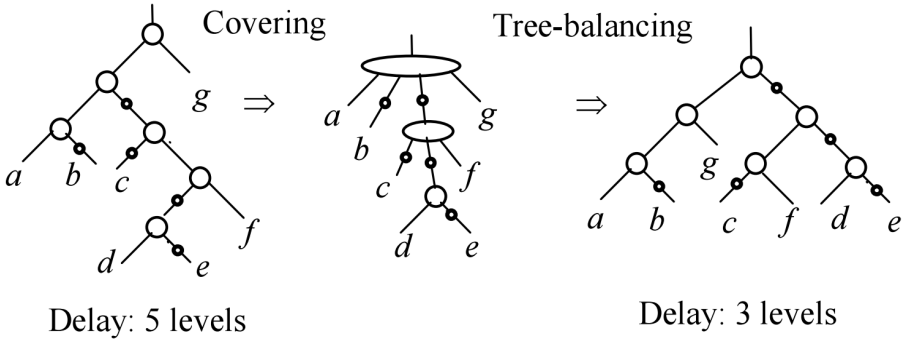
2. **Tree Balancing** - V následujícím kroku, pro každý takto vytvořený AND odebírá vstupy tak, že odebere dva s nejnižší hloubkou a spojí je do nového ANDu, který pak zpět připojí do mnohavstupého ANDu. Spojování opakujeme dokud v rebalancovaném mnohavstupého ANDu nezůstanou jen dva vstupy viz. 4.1. Při tomto procesu nemůže být navýšen počet hradel, ovšem může dojít k vytvoření uzlu, který sdílí lokální funkci s jiným již existujícím uzlem. Díky vlastnostem AIG (konkrétně „logic sharing“ [1]) můžeme tedy jeden takovýto uzel vynechat, čímž snížíme počet uzlů v AIG. K tomuto jevu dochází díky strukturnímu hashování, které zabraňuje jakékoliv duplikaci uzlů se stejnými vstupy.

Při „Tree balancing“ kroku se vybírají dva libovolné uzly s nejnižší hloubkou, které se připojí na vstup ANDu. Kvůli faktu, že algoritmus je plně deterministický, k tomuto chování ovšem nedochází. Pokud algoritmus nenajde uzly, které sdílí logickou funkci s jiným uzlem, vybere dva poslední uzly v pořadí. Toto chování ignoruje potenciální lepší výsledky při zvolení skutečně dvou náhodných takových uzlů.

Funkce *Abc_NodeBalance_rec* 4.1, *Abc_NodeBalanceCone_rec* 4.2 a *Abc_NodeBalancePermute* 4.3 jsou hlavním pilířem implementace algoritmu *balance*. Všechny funkce s předponou „Abc_NodeBalance“ jsou implementovány v souboru „src/base/abci/abcBalance.c“. Algoritmus začíná zadáním příkazu do příkazové řádky, kde se příkaz přeloží a zavolá se funkce *Abc_NtkBalance()*. Tato funkce pro každý primární vstup zavolá rekurzivní funkci
Abc_NodeBalance_rec s následujícími parametry:



Obrázek 4.2: Ilustrace „Tree-Balancing“ kroku. [1]



Obrázek 4.3: Ilustrace kompletního *balance*. [1]

- **pNtkNew** - ukazatel na síť, ve které se vyskytuje
- **pNodeOld** - ukazatel na kořenový uzel, který je zrovna balancován.
- **vStorage** - ukazatel na vektor obsahující množinu superANDů. Přistupuje se k němu jako k zásobníku.
- **Level** - úroveň zanoření rekurze. Při prvním volání je nastaveno na 0

Zbylé parametry jsou případné parametry zadané z příkazové řádky. Ihned po zavolání se identifikují uzly, které jsou na vstupu superANDu. Funkce *Abc_NodeBalanceCone*() 4.2 prohledává graf do hloubky a ukládá takové uzly do vektoru *vSuper*, které nemají invertor nebo nejsou PI. Po prohledání grafu máme ve vektoru vSuper uložené uzly, které jsou vstupy superANDu s kořenem *pNodeOld*. Na všechny uzly z vektoru vSuper se pak zavolá opětovně *Abc_NodeBalance_rec*() s hloubkou zanoření o jedna větší a daným uzlem jako kořenem. Funkce se přestane rekurzivně volat v případě že jsme prohledali celý graf.

Jako první se vyhodnotí ten uzel, který byl v rekurzi volán jako poslední. Díky tomu se bude AIG *balancovat* odspodu a každý uzel, který je právě

balancován má všechny listy již *vybalancované*. Uzly v superANDu *vSuper* se setřídí podle úrovně zanoření. Autor zde používá nestabilní třídící algoritmus qsort [19].

Uzly superANDu jsou setříděné podle hloubky sestupně. Nyní se zavolá funkce *NodeBalancePermute*, která v cyklu ověří, zda v superANDu není uzel, který s posledním uzlem po přidání do ANDu nevytvořil nový uzel, který už v grafu je. Pokud ano, prohodí nalezený uzel s uzlem na indexu *PočetuzlůV-superANDu - 2*.

Pro lepší pochopení, předpokládejme následující superAND:

Index uzlu	0	1	2	3	4	5	6	7
Hloubka uzlu	5	4	2	0	0	0	0	0

Funkce *NodeBalancePermute* 4.3 pak otestuje zda uzel na indexu 7 a jiný uzel na indexu v rozmezí 3 - 6 po spojení nevytvoří uzel, který už v grafu je. Pokud takový uzel najde, prohodí ho s uzlem na indexu 6. Pokud se žádný takový uzel nenajde, „Popnou“(Pop je obecný termín pro odebrání posledního prvku ze zásobníku) se dva uzly (tj. 6 a 7) a ty se dají na vstup nově vytvořenému uzlu a ten je zařazen zpět do superANDu na příslušnou pozici, abychom zachovali sestupnou hloubku uzlů. V případě, že se takový uzel najde, dojde k prohození. „Popnutím“ posledních dvou uzlů vytvoříme nový uzel, který v grafu už je. Díky strukturnímu hashování ušetříme minimálně jeden AND uzel, protože strukturní hashování zabrání duplikaci uzlů. Tento proces se opakuje tak dlouho odkud v superANDu nezbude poslední uzel, který se stane novým kořenem. Poté co jsou spojeny všechny uzly v superANDu se vrátí kořenový uzel a ten je zařazen na vstup rodičovského superANDu, kde se celý proces opakuje pro rodičovský superAND.

Dá se předpokládat, že náhodnou permutací uzlů stejné úrovně budeme dostávat více různých řešení, z nichž mohou být některá lepší, než deterministická. Balance je v ABC implementován jako příkaz *balance*.

```
1  Abc_Obj_t * Abc_NodeBalance_rec( Abc_Ntk_t * pNtkNew, Abc_Obj_t *
       pNodeOld, Vec_Vec_t * vStorage, int Level, int fDuplicate, int
        fSelective, int fUpdateLevel )
2  {
3      Abc_Aig_t * pMan = (Abc_Aig_t *)pNtkNew->pManFunc;
4      Abc_Obj_t * pNodeNew, * pNode1, * pNode2;
5      Vec_Ptr_t * vSuper;
6      int i, LeftBound;
7      // return if the result if known
8      if ( pNodeOld->pCopy ) return pNodeOld->pCopy;
9      // get the implication supergate
10     vSuper = Abc_NodeBalanceCone( pNodeOld, vStorage, Level,
       fDuplicate, fSelective );
11     if ( vSuper->nSize == 0 ){
12     // it means that the supergate contains two nodes in the
       opposite polarity
13         pNodeOld->pCopy = Abc_ObjNot(Abc_AigConst1(pNtkNew));
14         return pNodeOld->pCopy;
15     }
16     // for each old node, derive the new well-balanced node
```

```
17      for ( i = 0; i < vSuper->nSize; i++ ) {
18          pNodeNew = Abc_NodeBalance_rec( pNtkNew, Abc_ObjRegular((
        Abc_Obj_t *)vSuper->pArray[i]), vStorage, Level + 1,
        fDuplicate, fSelective, fUpdateLevel );
19          vSuper->pArray[i] = Abc_ObjNotCond( pNodeNew,
        Abc_ObjIsComplement((Abc_Obj_t *)vSuper->pArray[i]) );
20      }
21      if ( vSuper->nSize < 2 ) printf( "BUG!\n" );
22      // sort the new nodes by level in the decreasing order
23      Vec_PtrSort( vSuper, (int (*)(void))
        Abc_NodeCompareLevelsDecrease );
24      // balance the nodes
25      while ( vSuper->nSize > 1 ) {
26          // find the left bound on the node to be paired
27          LeftBound = (!fUpdateLevel)? 0 : Abc_NodeBalanceFindLeft(
        vSuper );
28          // find the node that can be shared (if no such node,
        randomize choice)
29          Abc_NodeBalancePermute( pNtkNew, vSuper, LeftBound );
30          // pull out the last two nodes
31          pNode1 = (Abc_Obj_t *)Vec_PtrPop(vSuper);
32          pNode2 = (Abc_Obj_t *)Vec_PtrPop(vSuper);
33          Abc_VecObjPushUniqueOrderByLevel( vSuper, Abc_AigAnd(pMan,
         pNode1, pNode2) );
34      }
35      // mark the old node with the new node
36      pNodeOld->pCopy = (Abc_Obj_t *)vSuper->pArray[0];
37      vSuper->nSize = 0;
38
39      return pNodeOld->pCopy;
40  }
```

Listing 4.1: Ukázka Balance v ABC.

```
1   int Abc_NodeBalanceCone_rec( Abc_Obj_t * pNode, Vec_Ptr_t * vSuper
        , int fFirst, int fDuplicate, int fSelective )
2   {
3       int RetValue1, RetValue2, i;
4       // check if the node is visited
5       if ( Abc_ObjRegular(pNode)->fMarkB )
6       {
7           // check if the node occurs in the same polarity
8           for ( i = 0; i < vSuper->nSize; i++ )
9               if ( vSuper->pArray[i] == pNode )
10                  return 1;
11          // check if the node is present in the opposite polarity
12          for ( i = 0; i < vSuper->nSize; i++ )
13              if ( vSuper->pArray[i] == Abc_ObjNot(pNode) )
14                  return -1;
15          assert( 0 );
16          return 0;
17      }
18      // if the new node is complemented or a PI, another gate
        begins
19      if ( !fFirst && (Abc_ObjIsComplement(pNode) || !Abc_ObjIsNode(
        pNode) || (!fDuplicate && !fSelective && (Abc_ObjFanoutNum(
        pNode) > 1)) || Vec_PtrSize(vSuper) > 10000) )
20      {
21          Vec_PtrPush( vSuper, pNode );
22          Abc_ObjRegular(pNode)->fMarkB = 1;
23          return 0;
24      }
```

```
25      assert( !Abc_ObjIsComplement(pNode) );
26      assert( Abc_ObjIsNode(pNode) );
27      // go through the branches
28      RetValue1 = Abc_NodeBalanceCone_rec( Abc_ObjChild0(pNode),
        vSuper, 0, fDuplicate, fSelective );
29      RetValue2 = Abc_NodeBalanceCone_rec( Abc_ObjChild1(pNode),
        vSuper, 0, fDuplicate, fSelective );
30      if ( RetValue1 == -1 || RetValue2 == -1 )
31          return -1;
32      // return 1 if at least one branch has a duplicate
33      return RetValue1 || RetValue2;
34  }
```

Listing 4.2: Ukázka funkce *Abc_NodeBalanceCone_rec*().

```
1   void Abc_NodeBalancePermute(Abc_Ntk_t * pNtkNew, Vec_Ptr_t *
        vSuper, int LeftBound)
2   {
3     Abc_Obj_t * pNode1, *pNode2, *pNode3;
4     int RightBound, i;
5     // get the right bound
6     RightBound = Vec_PtrSize(vSuper) - 2;
7     assert(LeftBound <= RightBound);
8     if (LeftBound == RightBound)
9       return;
10    // get the two last nodes
11    pNode1 = (Abc_Obj_t *)Vec_PtrEntry(vSuper, RightBound + 1);
12    pNode2 = (Abc_Obj_t *)Vec_PtrEntry(vSuper, RightBound);
13    // find the first node that can be shared
14    for (i = RightBound; i >= LeftBound; i--)
15    {
16      pNode3 = (Abc_Obj_t *)Vec_PtrEntry(vSuper, i);
17      if (Abc_AigAndLookup((Abc_Aig_t *)pNtkNew->pManFunc, pNode1,
        pNode3))
18      {
19        if (pNode3 == pNode2)
20          return;
21        Vec_PtrWriteEntry(vSuper, i, pNode2);
22        Vec_PtrWriteEntry(vSuper, RightBound, pNode3);
23        return;
24      }
25    }
26  }
```

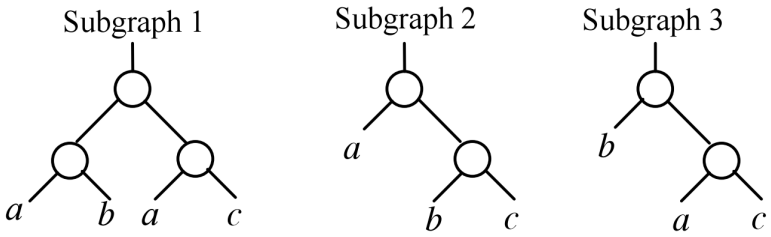
Listing 4.3: Ukázka funkce *Abc_NodeBalancePermute*().

4.2 Rewriting

Algoritmus *Rewriting* dokáže snížit počet AND uzlů v AIG [2]. Dosahuje toho v následujících krocích:

1. Najde 4-vstupý fanout-free řez.

2. z něj sestrojí BDD [13] (tj. kanonickou reprezentaci) a dále převede na 16-bitový integer, který reprezentuje logickou funkci v daném bodě.



Obrázek 4.4: Různé AIG struktury funkce F=abc. [2]

3. Vyhledá ekvivalentní podgraf ekvivalentní třídy NPN (1 z 222 ekvivalentních) [2] viz. 4.2. Tyto podgrafy se předpočítávají při alokaci manažera.

4. Nahradí starý podgraf novým, pokud bude výsledek lepší. Přepínač -z povolí nahrazování i v případě, že bude podgraf stejně velký.

Dvě logické funkce jsou NPN ekvivalentní pokud mají permutované vstupy, negované vstupy nebo negovaný výstup.

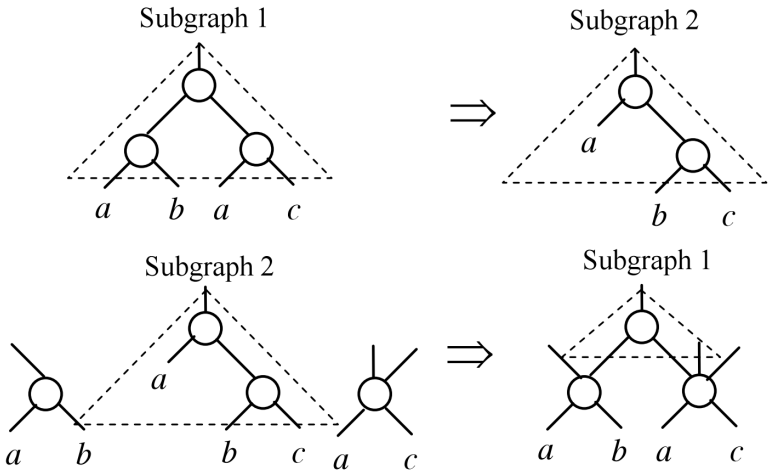
Příklad: funkce $F = ab + c$ a $G = ac + b$ jsou NPN ekvivalentní, protože prohozením vstupů b a c dostáváme identickou funkci. Funkce $F = ab + c$ a $G = ab$ nejsou NPN ekvivalentní, protože libovolným počtem permutací a negací nedokážeme z funkce o třech proměnných udělat funkci o dvou proměnných [2].

Pro čtyř vstupý graf dostáváme 222 NPN ekvivalentních možností [2]. Výběr jednoho z takových podgrafů je ovšem plně deterministický - vybere se první nalezený nejlepší podgraf.

Takové chování algoritmu přehlíží možnost výběru různých (potenciálně lepších) řešení. Je vhodné tedy množinu řešení v průběhu procesu náhodně permutovat, aby i při výběru prvního nalezeného grafu byl vybrán jeden ze všech „kvalitních“ kandidátů na řešení.

Algoritmus *rewriting* je v ABC implementovaný v souborech „src/base/abci/abcRewrite.c“, „src/opt/rwr/*“, „src/opt/cut/*“ (pro práci s řezem), „src/misc/vec/vecPtr.h“ (pro práci s vektorem). Rewriting můžeme evokovat příkazem „rewrite“.

Po zavolání příkazu *rewrite* nebo jeho aliasu definovaném v souboru „abc.rc“ *rw* se příkaz přeloží, nastaví přepínače (např. *-z* nastaví *fUseZeros* 4.5 na 1) a zavolá funkce *Ntk_Rewrite*(). Ta sama o sobě není příliš zajímavá. Důležitá je ovšem část, kdy iterátor *Abc_NtkForEachNode*() začne procházet uzly v grafu v topologickém pořadí a na každý takový uzel zavolá *Rwr_NodeRewrite*() 4.4. Ta vrátí nový počet uzlů, které jsme ušetřili při nahrazení. V tomto bodě už je starý graf nahrazen novým (nebo není, pokud nebyla nalezena menší struktura). Parametry:



Obrázek 4.5: Dva různé případy jak může dopadnout nahrazení novým podgrafem. [2]

- **pManRwr** a **pManCut** - manažerské funkce pro správu rewritingu a řezu
- **pNode** - kořenový uzel podgrafu, který bude nahrazený menším v případě že bude nalezen.
- další paramtery představují přepínače

```
1  Abc_NtkForEachNode( pNtk, pNode, i )
2      {
3          if ( i >= nNodes ) break;
4          if ( Abc_NodeIsPersistant(pNode) ) continue;
5          if ( Abc_ObjFanoutNum(pNode) > 1000 ) continue;
6
7          // for each cut, try to resynthesize it
8          nGain = Rwr_NodeRewrite( pManRwr, pManCut, pNode,
       fUpdateLevel, fUseZeros, fPlaceEnable );
9          if ( !(nGain > 0 || (nGain == 0 && fUseZeros)) )
10             continue;
11         // if we end up here, a rewriting step is accepted
12
13         pGraph = (Dec_Graph_t *)Rwr_ManReadDecs(pManRwr);
14         fCompl = Rwr_ManReadCompl(pManRwr);
15
16         if ( fPlaceEnable ) Abc_AigUpdateReset( (Abc_Aig_t *)pNtk
       ->pManFunc );
17
18         if ( fCompl ) Dec_GraphComplement( pGraph );
19         Dec_GraphUpdateNetwork( pNode, pGraph, fUpdateLevel, nGain
        );
20         if ( fCompl ) Dec_GraphComplement( pGraph );
21     }
```

Listing 4.4: Tělo funkce *Abc_NtkRewrite*

Funkce *Rwr_NodeRewrite* 4.5 na vstup dostane zadaný uzel a začne procházet všechny jeho řezy se čtyřmi listy. Pro každý takovýto řez si spočítá logickou funkci a uloží současný graf, aby se mohl, v případě že se nenajde žádný lepší podgraf, vrácen do původního stavu. *Rwr_CutEvaluate* 4.6 pak prohledá les podgrafů, které už má připravé v rewrite manažeru *p* a z něj vybere jeden z nejvhodnějších. Výběr probíhá tak, že se porovnává, zda je nový podgraf menší než stávající uložený. V případě zlepšení ho pak nahradí. Spočítá se počet ušetřených uzlů při nahrazení novým podgrafem a uloží do *nGain*. *p->vFanins* obsahuje informaci o nejlepším řezu, který bude použit při nahrazení. Funkce vrátí -1 pokud graf nemůže být přepsán.

```
int Rwr_NodeRewrite( Rwr_Man_t * p, Cut_Man_t * pManCut, Abc_Obj_t
    * pNode, int fUpdateLevel, int fUseZeros, int fPlaceEnable )
{
    int fVeryVerbose = 0;
    Dec_Graph_t * pGraph;
    Cut_Cut_t * pCut; //, * pTemp;
    Abc_Obj_t * pFanin;
    unsigned uPhase;
    unsigned uTruthBest = 0; // Suppress "might be used
    uninitialized"
    unsigned uTruth;
    char * pPerm;
    int Required, nNodesSaved;
    int nNodesSaveCur = -1; // Suppress "might be used
    uninitialized"
    int i, GainCur = -1, GainBest = -1;

    p->nNodesConsidered++;
    // get the required times
    Required = fUpdateLevel? Abc_ObjRequiredLevel(pNode) :
    ABC_INFINITY;

    // get the node's cuts
    pCut = (Cut_Cut_t *)Abc_NodeGetCutsRecursive( pManCut, pNode,
    0, 0 );
    assert( pCut != NULL );
    // go through the cuts
    for ( pCut = pCut->pNext; pCut; pCut = pCut->pNext )
    {
        // consider only 4-input cuts
        if ( pCut->nLeaves < 4 )
            continue;

        // get the fanin permutation
        uTruth = 0xFFFF & *Cut_CutReadTruth(pCut);
        pPerm = p->pPerms4[ (int)p->pPerms[uTruth] ];
        uPhase = p->pPhases[uTruth];
        // collect fanins with the corresponding permutation/phase
        Vec_PtrClear( p->vFaninsCur );
        Vec_PtrFill( p->vFaninsCur, (int)pCut->nLeaves, 0 );
        for ( i = 0; i < (int)pCut->nLeaves; i++ )
        {
            pFanin = Abc_NtkObj( pNode->pNtk, pCut->pLeaves[(int)
    pPerm[i]] );
            if ( pFanin == NULL )
                break;
            pFanin = Abc_ObjNotCond(pFanin, ((uPhase & (1<<i)) >
    0) );
```

```
42              Vec_PtrWriteEntry( p->vFaninsCur, i, pFanin );
43          }
44          if ( i != (int)pCut->nLeaves )
45          {
46              p->nCutsBad++;
47              continue;
48          }
49          p->nCutsGood++;
50          int Counter = 0;
51          Vec_PtrForEachEntry( Abc_Obj_t *, p->vFaninsCur, pFanin, i
    )
52          if ( Abc_ObjFanoutNum(Abc_ObjRegular(pFanin)) == 1 )
53              Counter++;
54          if ( Counter > 2 )
55               continue;
56
57          // mark the fanin boundary
58          Vec_PtrForEachEntry( Abc_Obj_t *, p->vFaninsCur, pFanin, i
    )
59              Abc_ObjRegular(pFanin)->vFanouts.nSize++;
60
61          // label MFFC with current ID
62          Abc_NtkIncrementTravId( pNode->pNtk );
63          nNodesSaved = Abc_NodeMffcLabelAig( pNode );
64          // unmark the fanin boundary
65          Vec_PtrForEachEntry( Abc_Obj_t *, p->vFaninsCur, pFanin, i
    )
66              Abc_ObjRegular(pFanin)->vFanouts.nSize--;
67
68          // evaluate the cut
69          pGraph = Rwr_CutEvaluate( p, pNode, pCut, p->vFaninsCur,
    nNodesSaved, Required, &GainCur, fPlaceEnable );
70
71          // check if the cut is better than the current best one
72          if ( pGraph != NULL && GainBest < GainCur )
73          {
74              // save this form
75              nNodesSaveCur = nNodesSaved;
76              GainBest  = GainCur;
77              p->pGraph  = pGraph;
78              p->fCompl = ((uPhase & (1<<4)) > 0);
79              uTruthBest = 0xFFFF & *Cut_CutReadTruth(pCut);
80              // collect fanins in the
81              Vec_PtrClear( p->vFanins );
82              Vec_PtrForEachEntry( Abc_Obj_t *, p->vFaninsCur,
    pFanin, i )
83                  Vec_PtrPush( p->vFanins, pFanin );
84          }
85      }
86
87      if ( GainBest == -1 )
88          return -1;
89      // copy the leaves
90      Vec_PtrForEachEntry( Abc_Obj_t *, p->vFanins, pFanin, i )
91          Dec_GraphNode((Dec_Graph_t *)p->pGraph, i)->pFunc = pFanin
    ;
92
93      p->nScores[p->pMap[uTruthBest]]++;
94      p->nNodesGained += GainBest;
95      if ( fUseZeros || GainBest > 0 )
96      {
97          p->nNodesRewritten++;
```

```
98      }
99
100     return GainBest;
101 }
```

Listing 4.5: Definice funkce *Rwr_NodeRewrite*

Poslední a nejdůležitější funkci, kterou zde zmíním je *Rwr_CutEvaluate* 4.6. Funkce se stará o vyhodnocování řezů. Na vstup dostane:

- *p* - rewrite manažer
- *pRoot* - kořenový uzel řezu jeden
- *pCut* - jeden z jeho řezů
- *vFaninsCur* - informace o momentálně nejlepším podgrafu
- *nNodesSaved* - počet uložených uzlů

Funkce jako první získá všechny grafy jejichž kořenový uzel má NPN ekvivalentní logickou funkci jako stávající a uloží je do *vSubgraphs*. Jedná se o vektor, který obsahuje ukazatele na kořenové uzly podgrafů. Vektor pak iterátor *Vec_PtrForEachEntry*() prochází a testuje zda podgraf z vektoru není lepší, než stávající. Pokud tomu tak je, uloží ho do manažeru *p* jako nový nejlepší podgraf.

Vložením randomizace do výběru z vektoru *vSubgraphs*, bychom mohli dosáhnout větší diverzity řešení. Nezáleží na tom, v jakém pořadí řešení z třídy NPN ekvivalence vybíráme, vybere se vždy to nejlepší. Pokud je nejlepších řešení více, díky náhodné permutace dosáhneme toho, že všechny budou mít stejnou pravděpodobnost, že budou mít stejnou pravděpodobnost na nahrazení.

```
1  Dec_Graph_t * Rwr_CutEvaluate( Rwr_Man_t * p, Abc_Obj_t * pRoot,
       Cut_Cut_t * pCut, Vec_Ptr_t * vFaninsCur, int nNodesSaved, int
        LevelMax, int * pGainBest, int fPlaceEnable ) {
2    extern int            Dec_GraphToNetworkCount( Abc_Obj_t * pRoot
       , Dec_Graph_t * pGraph, int NodeMax, int LevelMax );
3    Vec_Ptr_t * vSubgraphs;
4    Dec_Graph_t * pGraphBest = NULL; // Suppress "might be used
       uninitialized"
5    Dec_Graph_t * pGraphCur;
6    Rwr_Node_t * pNode, * pFanin;
7    int nNodesAdded, GainBest, i, k;
8    unsigned uTruth;
9    float CostBest;//, CostCur;
10   // find the matching class of subgraphs
11   uTruth = 0xFFFF & *Cut_CutReadTruth(pCut);
12   vSubgraphs = Vec_VecEntry( p->vClasses, p->pMap[uTruth] );
13   p->nSubgraphs += vSubgraphs->nSize;
14   // determine the best subgraph
15   GainBest = -1;
16   CostBest = ABC_INFINITY;
17   Vec_PtrForEachEntry( Rwr_Node_t *, vSubgraphs, pNode, i ){
18     // get the current graph
```

```
19      pGraphCur = (Dec_Graph_t *)pNode->pNext;
20      // copy the leaves
21      Vec_PtrForEachEntry( Rwr_Node_t *, vFaninsCur, pFanin, k~)
22        Dec_GraphNode(pGraphCur, k)->pFunc = pFanin;
23      // detect how many unlabeled nodes will be reused
24      nNodesAdded = Dec_GraphToNetworkCount( pRoot, pGraphCur,
        nNodesSaved, LevelMax );
25      if ( nNodesAdded == -1 )
26        continue;
27      assert( nNodesSaved >= nNodesAdded );
28      // count the gain at this node
29        if ( GainBest < nNodesSaved - nNodesAdded ){
30          GainBest    = nNodesSaved - nNodesAdded;
31          pGraphBest = pGraphCur;
32
33          // score the graph
34          if ( nNodesSaved - nNodesAdded > 0 ){
35            pNode->nScore++;
36            pNode->nGain += GainBest;
37            pNode->nAdded += nNodesAdded;
38          }
39        }
40      }
41    }
42    if ( GainBest == -1 )
43      return NULL;
44    *pGainBest = GainBest;
45    return pGraphBest;
46  }
```

Listing 4.6: Definice funkce *Rwr_CutEvaluate*

4.3 Refactor

Algoritmus *refactoring* [20] funguje jistým způsobem podobně jako rewriting. Jeho hlavním úkolem je snížení počtu uzlů v AIG. Dosahuje toho pomocí dvou kroků.

1. Pro každý uzel spočítám řez (na rozdíl od rewritingu, kde se braly v úvahu jen 4-vstupé fanout-free řezy)

2. Převedu ho na faktorovanou formu. Pokud je faktorovaná forma menší než stávající podgraf, nahradí se jím. Přepínačem *-z* můžeme nastavit že se faktorovaná forma má nahradit i v případě že je faktorovaná forma stejně velká jako stávající podgraf.

Faktorovaná forma je způsob zápisu logických funkcí v boolovské algebře, kde se „střídají“ operátory AND a OR. Je definovaná rekurentně:

- písmeno je faktorovaná forma
- součin faktorovaných forem je faktorovaná forma
- součet faktorovaných forem je faktorovaná forma

Funkce ve tvaru *(ab + ac)* může být převedena na faktorovanou formu *(a(b + c))*. Naopak *(ad + b'c)'* není ve faktorové formě, protože ze 4 písmen, které jsou ve faktorové formě, postupným sčítáním a násobením nemůžeme danou funkci vytvořit.

Algoritmus implementovaný v ABC se vyhodnotí spuštěním příkazu *refactor*. Po sytnaktickém rozboru přepínačů se zavolá funkce *Abc_NtkRefactor*, která samotná není příliš zajímavá. Důležitá je pasáž, kde iterátor začne procházet uzly v grafu v topologickém pořadí, pro každý takový najde řez a zavolá funkci *Abc_NodeRefactor* 4.7 4.8 s parametry:

1. *pManRef* - refactor manažer
2. *pNode* - ukazatel na uzel, který faktorizován
3. *vFanins* - uzly řezu uložené ve vektoru
4. *fUpdateLevel*, *fUseZeros*, *fUseDcs* a *fVerbose* jsou volitelné přepínače

Funkce *Abc_NodeRefactor* nejdříve spočítá logickou funkci daného řezu a uloží si jí ve formátu BDD [16], kterou vzápětí převede na SOP [12] a uloží ji do stringu *pSop* ve formátu „A B C... X\n“. Každý řádek reprezentuje jeden součin, sloupce *A B C..* mohou nabývat hodnot 1/0/-, podle toho zda jsou negované, nebo se v součinu vůbec nevyskytují. Hodnota X může nabývat hodnot 0/1 a označuje negaci funkce. **Příklad:** Logická funkce *F = abc + cde + a'b'd* by pak v pSop vypadala takto „1 1 1 - - 1\n- - 1 1 1 1\n0 0 - 0 1\n“.

Tu pak předá funkci *Dec_Factor*, která ji faktorizuje a otestuje zda dojde ke zmenšení celého grafu nebo ne. Jedna logická funkce může být ovšem vyjádřena více faktorovanými formami např.

ab + ac + bc = a(b+c) + bc

ab + ac + bc = c(a+b) + ab

Protože je Algoritmus plně deterministický, bude ignorováno mnoho kandidátů na řešení, které by mohly být potenciálně lepší. Při opakovaném spuštění randomizace algoritmu *refactor* by pak měla za důsledek větší pestrost řešení, z nichž některá by mohla být lepší než stávající deterministické.

```
1  Abc_NtkForEachNode( pNtk, pNode, i ){
2          // skip persistant nodes
3          if ( Abc_NodeIsPersistant(pNode) ) continue;
4          // skip the nodes with many fanouts
5          if ( Abc_ObjFanoutNum(pNode) > 1000 ) continue;
6          // stop if all nodes have been tried once
7          if ( i >= nNodes ) break;
8          // compute a reconvergence-driven cut
9          vFanins = Abc_NodeFindCut( pManCut, pNode, fUseDcs );
10         // evaluate this cut
11         pFForm = Abc_NodeRefactor( pManRef, pNode, vFanins,
       fUpdateLevel, fUseZeros, fUseDcs, fVerbose );
12         if ( pFForm == NULL ) continue;
13         // acceptable replacement found, update the graph
14         Dec_GraphUpdateNetwork( pNode, pFForm, fUpdateLevel,
       pManRef->nLastGain );
```

```
15            Dec_GraphFree( pFForm );
16        }
```

Listing 4.7: Iterátor v těle funkce *Abc_NtkRefactor*

```
1  Dec_Graph_t * Abc_NodeRefactor( Abc_ManRef_t * p, Abc_Obj_t *
       pNode, Vec_Ptr_t * vFanins, int fUpdateLevel, int fUseZeros,
       int fUseDcs, int fVerbose ){
2
3      Required = fUpdateLevel? Abc_ObjRequiredLevel(pNode) :
       ABC_INFINITY;
4      p->nNodesConsidered++;
5      // get the function of the cut
6      bNodeFunc = Abc_NodeConeBdd( p->dd, p->dd->vars, pNode,
       vFanins, p->vVisited );  Cudd_Ref( bNodeFunc );
7      // get the SOP of the cut
8      pSop = Abc_ConvertBddToSop( NULL, p->dd, bNodeFunc, bNodeFunc,
        vFanins->nSize, 0, p->vCube, -1 );
9      // get the factored form
10     pFForm = Dec_Factor( pSop );
11     ABC_FREE( pSop );
12     // mark the fanin boundary
13     Vec_PtrForEachEntry( Abc_Obj_t *, vFanins, pFanin, i )
14         pFanin->vFanouts.nSize++;
15     // label MFFC with current traversal ID
16     Abc_NtkIncrementTravId( pNode->pNtk );
17     nNodesSaved = Abc_NodeMffcLabelAig( pNode );
18     // unmark the fanin boundary and set the fanins as leaves in
       the form
19     Vec_PtrForEachEntry( Abc_Obj_t *, vFanins, pFanin, i ){
20         pFanin->vFanouts.nSize--;
21         Dec_GraphNode(pFForm, i)->pFunc = pFanin;
22     }
23     // detect how many new nodes will be added (while taking into
       account reused nodes)
24     nNodesAdded = Dec_GraphToNetworkCount( pNode, pFForm,
       nNodesSaved, Required );
25     // quit if there is no improvement
26     if ( nNodesAdded == -1 || (nNodesAdded == nNodesSaved && !
       fUseZeros) )
27     {
28         Cudd_RecursiveDeref( p->dd, bNodeFunc );
29         Dec_GraphFree( pFForm );
30         return NULL;
31     }
32     // compute the total gain in the number of nodes
33     p->nLastGain = nNodesSaved - nNodesAdded;
34     p->nNodesGained += p->nLastGain;
35     p->nNodesRefactored++;
36
37     Cudd_RecursiveDeref( p->dd, bNodeFunc );
38     return pFForm;
39 }
```

Listing 4.8: Ukázka funkce *Abc_NodeRefactor*

Kapitola 5

Realizace

Cílem randomizace je snaha zvětšit prohledávaný prostor řešení. Pokud randomizovaný algoritmus po opakovaném spuštění vygeneruje různé struktury, z nichž budou některé lepší než referenční, můžeme to považovat za úspěch. V průměru by ovšem nemělo dojít ke zlepšení ani ke zhoršení. Nicméně funkcionalita algoritmu by měla být zachována. Vložení náhodného výběru do místa, kde má dojít k výběru jednoho z ekvivalentních variant, by mohlo odpovídat požadovaným podmínkám. Dalším přínosem randomizace je možnost uniknutí z lokálního minima při dlouhodobé iteraci.

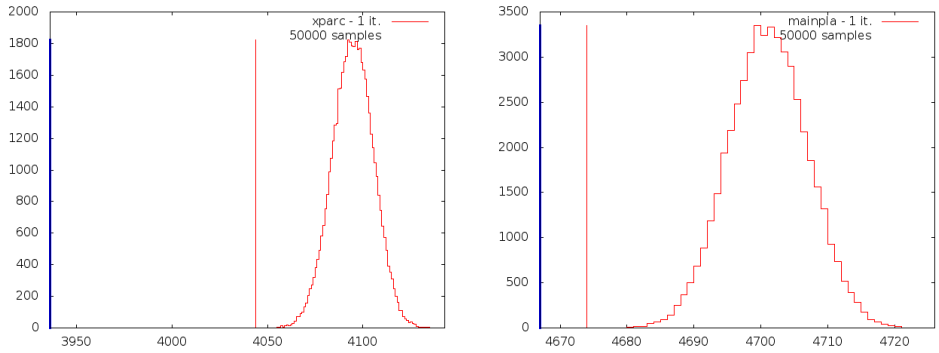
5.1 Balance

V tree-balancing kroku dochází k odebírání uzlů stejné úrovně ze superANDu. Uzly v superANDu (*vSuper*) jsou setříděné sestupně podle hloubky uzlů.

První návrh byl „Plná randomizace“ tj. permutace vstupů superANDu tak, aby uzly zůstaly setříděné podle úrovně. Bylo by pak náhodné nejen pořadí zpracovávaných vstupů, ale i nacházení vhodných vhodných kandidátů ke sdílení. Implementoval jsem proto funkci *Abc_NodeBalanceShuffle* 5.1. Ta na vstup dostane množinu vstupů superANDu a každou množinu uzlů každé úrovně zpermutuje. Při psaní funkce *Vec_PtrRandomShuffle* 5.2 jsem se inspiroval [21]. Ta se bude volat pokaždé před voláním funkce *Abc_NodeBalancePermute*.

```
1 void Abc_NodeBalanceShuffle( Vec_Ptr_t * vSuper ){
2     int RightBound, LeftBound;
3     RightBound = Vec_PtrSize( vSuper ) - 1;
4     LeftBound = Abc_NodeBalanceFindLeftFrom( vSuper, RightBound );
5     while ( 1 ){
6         LeftBound = Abc_NodeBalanceFindLeftFrom( vSuper,
      RightBound );
7         Vec_PtrRandomShuffle( vSuper, LeftBound, RightBound );
8     return;
9 }
```

Listing 5.1: Definice funkce *Abc_NodeBalanceShuffle*



Obrázek 5.1: Histogram xparc.blif a mainpla.blif při úplné randomizaci.

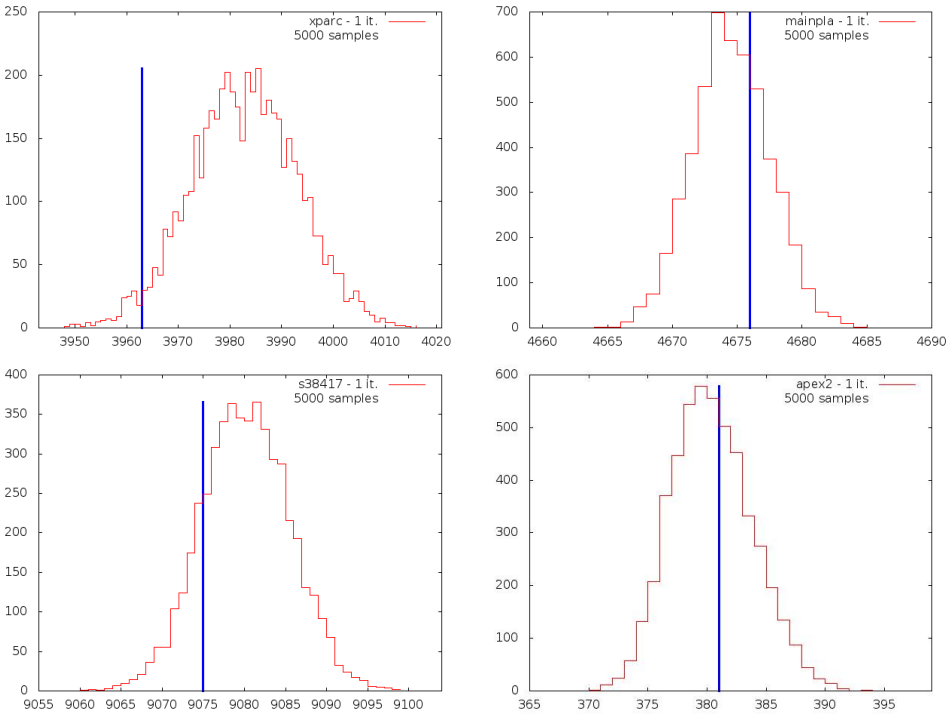
```
1  static inline void Vec_PtrRandomShuffle(Vec_Ptr_t * vSuper, int
       begin, int end ){
2    int i, rnd;
3    void * tmp;
4
5    for ( i = end ; i > begin; --i) {
6      rnd = rand() % ( i - begin + 1 );
7      tmp = Vec_PtrEntry(vSuper, i);
8      Vec_PtrWriteEntry(vSuper, i, Vec_PtrEntry(vSuper, begin + rnd)
       );
9      Vec_PtrWriteEntry(vSuper, begin + rnd, tmp);
10   }
11 }
```

Listing 5.2: Definice funkce *Vec_PtrRandomShuffle*

Testoval jsem randomizovaný *balance* 2000 krát pro 222 obvodů různé velikosti a počtu PI/PO. Testy ovšem vykázaly průměrné zhoršení oproti referenčnímu řešení více než 1% v počtu AND hradel. Navíc pro některé obvody (xparc.blif, mainpla.blif, spla.blif...viz. 5.1 - modrá hodnota označuje deterministický běh) ani nejlepší nalezené řešení nebylo lepší než referenční při 50 000 opakování.

Dále jsem testoval, jak se bude měnit zhoršení v závislosti na míře randomizace (k náhodné permutaci docházelo jen v 50%, 20%, 10%, 1% případů). Zhoršení klesalo s klesající randomizací. Proč se tak děje jsem ovšem neodhalil. Podle mně je tomu tak kvůli tomu, že specifický tvar AIG zvýhodňuje deterministický balance oproti plně randomizovanému. Jde ovšem pouze o mou domněnku, své tvrzení nemám čím podložit. Jedná se totiž o uzly s mnoha uzly a vstupy, je tedy poměrně obtížné z grafů získat nějaké informace.

Po dohodě s vedoucím práce jsem testoval případ, jak se zlepší průměrné řešení randomizovaného algoritmu v případě, že se bude hledat vhodný kandidát ke sdílení v celém superANDu, nikoli pouze v rámci stejné úrovně. Tato metoda má průměrné zhoršení o 0.4% méně než plná randomizace. Tato metoda ovšem testuje více prvků, tudíž se časová složitost algoritmu zvýší o



Obrázek 5.2: Histogramy při částečné randomizaci.

konstantu. I přes to jsem se rozhodl tuto změnu v algoritmu ponechat.

Další návrh byl „Částečná randomizace“. V případě, že se nenajde vhodný kandidát ke sdílení, vybere se jiný uzel s nejnižší dostupnou úrovní. Pokud je takových uzlů víc, vybere se náhodně 5.3. Na rozdíl od stávajícího modelu, kde se vybíraly uzly na indexech *početUzlů - 1* a *početUzlů - 2*.

Tento model byl v ABC již navržený v komentáři ve funkci *Abc_NodeBalancePermute*. Měření ukázalo viditelné zlepšení oproti plné randomizaci navíc bylo konkurenceschopné deterministické variantě. Nejen, že došlo k průměrnému zhoršení od průměru pouze o 0,09%, ale zároveň nacházel řešení, které nalezl deterministický algoritmus. V ABC jsem randomizaci balance nastavil na volitelný přepínač *-r* ve funkci *Abc_CommandBalance* v souboru „abc.c“.

Porovnával jsem chování příkazu *permute* (náhodná permutace vstupů a výstupů) oproti deterministickému řešení. Zhoršení oproti deterministickému řešení činilo 0,05%. Z toho by se dalo usuzovat, že permutace vstupů je robustnější než randomizace ekvivalentních voleb.

```
1 if( fRandomize ){
2   int Choice = rand() % (RightBound - LeftBound + 1);
3   pNode3 = Vec_PtrEntry( vSuper, LeftBound + Choice );
4   if ( pNode3 == pNode2 )
5     return;
```

```
6     Vec_PtrWriteEntry( vSuper, LeftBound + Choice, pNode2 );
7     Vec_PtrWriteEntry( vSuper, RightBound,         pNode3 );
8   }
```

Listing 5.3: Rozšíření funkce *Abc_NodeBalancePermute*

5.2 Rewriting

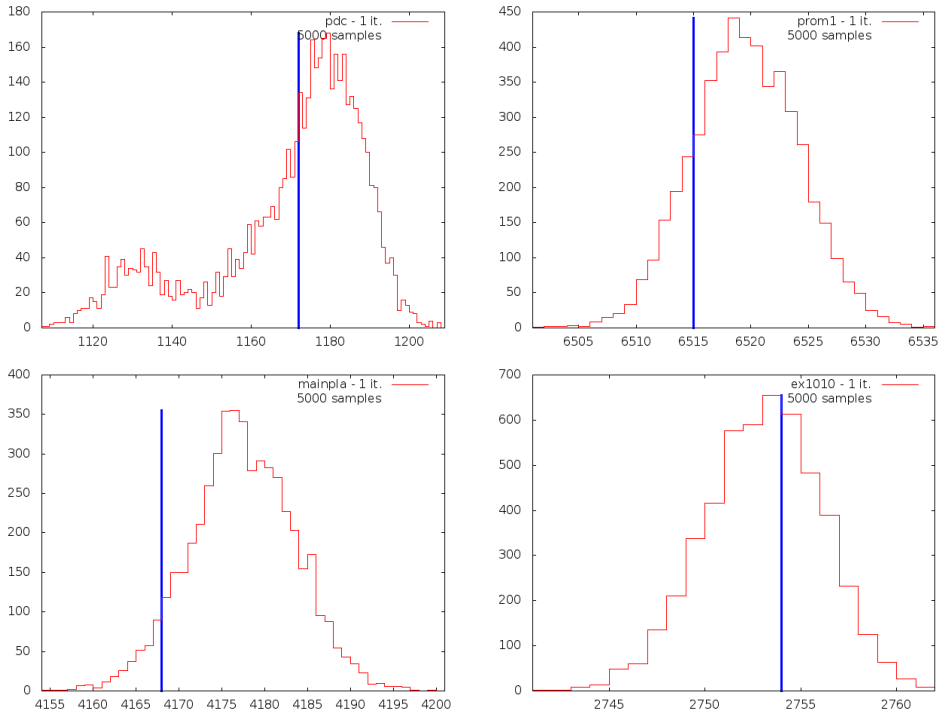
Při výběru z množiny NPN ekvivalentních podgrafů se vybírá ten, který je nejvhodnější. Pokud je nejvhodnějších více, zachová se první nalezený. V případě nastavení přepínače *-z* popř. aliasu *rwz* bude nahrazování probíhat i v případě, že počet uzlů zůstane stejný.

Množina podgrafů daného řezu je uložená ve vektoru *vSubgraphs*. Ten je lineárně procházen a testována každá možnost přepisu. Implementoval jsem tedy vektorovou funkci *Vec_PtrShuffle*, která náhodně permutuje prvky ve vektoru 5.4. Tuto funkci zavolám před tím než se začne vektor procházet.

```
1  static inline void Vec_PtrShuffle(Vec_Ptr_t * pVec){
2    int i, rnd;
3    void * tmp;
4    for (i = pVec->nSize - 1; i > 0; --i) {
5      rnd = rand() % (i + 1);
6      tmp = Vec_PtrEntry(pVec, i);
7      Vec_PtrWriteEntry(pVec, i, Vec_PtrEntry(pVec, rnd));
8      Vec_PtrWriteEntry(pVec, rnd, tmp);
9    }
10 }
```

Listing 5.4: Rozšíření funkce *Vec_PtrShuffle*

Testoval jsem randomizovaný *rewriting* 5000 krát pro 222 obvodů různé velikosti a počtu PI/PO. Testy ukázaly zlepšení v počtu uzlů o 0,18% a zlepšení v počtu úrovní o 1,09%. Dá se tedy předpokládat, že se randomizovaný *rewriting* chová podle očekávání.



Obrázek 5.3: Histogramy randomizovaného rewritingu.

5.3 Seznam upravených souborů

Vector

src/misc/vec/vecPtr.h	řádek: 1114

Rewrite

src/base/abc/abc.h	řádek: 860
src/base/abci/abc.c	řádek: 3801, 3812, 3836, 3869, 388
src/base/abci/abcRewrite.c	řádek: 129, 61
src/base/abci/abcIvy.c	řádek: 551, 554
src/base/abci/abcProve.c	řádek: 146, 335, 337
src/base/abci/abcQuant.c	řádek: 53, 60
src/opt/rwr/rwrEva.c	řádek: 32, 59, 152, 253, 268

Balance

src/base/abc/abc.h	řádek: 586
src/base/abci/abc.c	řádek: 3154, 3165, 3186, 3207, 3220, 3242
src/base/abci/abcBalance.c	řádek: 30, 31, 53, 74,107, 131, 139, 201, 230, 322, 345, 366
src/base/abci/abcIvy.c	řádek: 549, 552
src/base/abci/abcProve.c	řádek: 156, 336
src/base/abci/abcQuant.c	řádek: 56, 62

Process	AndMax	AndAvg	LevAvg
Částečná randomizace *balance*	-8,17%	0,09%	0,00%
Úplná randomizace *balance*	-7,40%	1,12%	0,00%
permute; balance	-10,52%	0,05%	0,00%
rewrite	-5,58%	-0,18%	-14,28%
permute; rewrite	-8,33%	-0,06%	-12,50%
choice	-40,00%	-1,15%	-5,52%

Tabulka 5.1: shrnutí výsledků

5.4 Výsledky měření

Algoritmy jsem testoval spouštěním sekvence příkazů „*read file.blif; st; command; ps*“, který jsem opakoval minimálně 1000krát pro každý obvod. Z výpisu jsem pak zachoval vždy nejlepší výsledek (AndMin), nejhorší výsledek (AndMax) a průměrný výsledek (AndAvg) jak pro velikost, tak pro hloubku (Lev). Testoval jsem 222 obvodů, které jsou stejně jako všechna naměřená data na přiloženém CD. Zhoršení/zlepšení (*AndAvg/AndRef * 100 - 100*) se pak odvíjí podle referenčního řešení, které v tabulkách 5.4 a 5.4 uvádím jako „AndRef“. Záporné hodnoty značí pokles počtu Uzlů, kladné nárůst. V tabulce 5.1 Je viditelné, že Úplná randomizace z nějakého důvodu stabilně vykazuje hroší výsledky než částečná. Ostatní algoritmy se chovají podle očekávání. Syntézní skript jsem po dohodě s vedoucím práce zvolil *choice*.

Choice - *fraig_store; resyn; fraig_store; resyn2; fraig_store; fraig_restore*
alias resyn - *b; rw; rf; b; rw; rwz; b; rfz; rwz; b*
alias resyn2 - *b; rw; rwz; b; rwz; b*

Z důvodu velké zátěže byl na testování využit stroj vedoucího práce. Skript byl iterován 20x a vzorek 1000 spuštění. Hodnoty byly porovnávány nativně jako u *balance* a *rewrite*. Počet AND uzlů se snížil v průměru o 1,15% a úroveň o 5,5%. Z toho se dá usuzovat, že randomizace má pozitivní vliv na iterativní optimalizaci obvodu.

	AndMin	AndMax	AndAvg	AndRef	Odchylka	LevMin	LevMax	LevAvg	LevRef
alu2	370	372	371	372	-0,26%	39	39	39	39
amd	496	509	502	506	-0,79%	15	16	15	16
apex1	2156	2178	2167	2159	0,37%	23	25	24	24
apex2	325	347	334	345	-3,18%	26	27	26	27
apex3	1905	1924	1916	1912	0,20%	19	19	19	19
apex4	2879	2897	2887	2886	0,03%	20	20	20	20
apex5	1001	999	982	1010	-2,77%	13	14	13	14
apex6	639	640	639	640	-0,15%	15	15	15	15
apla	215	225	219	221	-0,90%	13	13	13	13
bca	2656	2695	2675	2672	0,11%	26	26	26	26
bcb	2408	2454	2431	2428	0,12%	25	26	25	25
bcc	2347	2386	2366	2361	0,21%	23	25	24	25
bcd	1706	1738	1722	1721	0,05%	25	25	25	25
bc0	1210	1240	1227	1211	1,32%	24	28	25	25
br1	172	174	173	174	-0,57%	17	17	17	17
br2	117	131	124	125	-0,80%	13	14	13	13
bw	154	160	157	160	-1,87%5	7	8	7	8
jbp	414	433	421	424	-0,70%	16	16	16	16
k2	1421	1478	1444	1474	-2,03%	22	22	22	22
lal	85	94	90	86	4,65%	7	8	7	8
lin	1148	1166	1156	1157	-0,08%	17	17	17	17
luc	234	237	235	236	-0,42%	10	10	10	10
mainpla	4156	4198	4176	4168	0,19%	31	34	32	31
mark1	279	308	296	301	-1,66%	26	27	26	27
max1024	913	927	919	917	0,21%	20	20	20	20
max128	430	442	436	433	0,69%	12	12	12	12
max46	167	168	167	168	-0,59%	17	17	17	17
max512	612	625	619	613	0,97%	19	19	19	19
misex3	1277	1291	1284	1278	0,46%	21	21	21	21
misex3c	613	618	615	616	-0,16%	20	21	20	21
pdc	1269	1293	1279	1278	0,07%	23	23	23	23
pm1	35	44	40	40	0,00%	5	5	5	5
pope	675	675	675	675	0,00%	11	11	11	11
prom1	7073	7078	7075	7075	0,00%	17	17	17	17
prom2	3194	3201	3197	3198	-0,03%	16	16	16	16
p82	114	116	114	114	0,00%	8	8	8	8
rd53	54	54	54	54	0,00%	7	7	7	7
rd73	151	151	151	151	0,00%	12	12	12	12
rd84	218	220	219	217	0,92%	12	12	12	12
risc	100	99	99	99	0,00%	7	7	7	7
root	246	252	248	249	-0,40%	13	13	13	13
rot	526	535	530	531	-0,18%	21	21	21	21
ryy6	16	16	16	16	0,00%	6	6	6	6
sao2	147	157	151	152	-0,65%	10	10	10	10
sct	72	75	73	71	2,80%	6	6	6	6
seq	2022	2044	2032	2032	0,00%	22	22	22	22
sex	44	46	44	44	0,00%	6	6	6	6
shift	146	146	146	146	0,00%	6	6	6	6

Tabulka 5.2: Ukázka naměřených dat částečně randomizovaného *balance*.

	AndMin	AndMax	AndAvg	AndRef	Odchylka:	LevMin	LevMax	LevAvg	LevRef
alcom	78	78	78	78	0,00%	4	5	4	4
alu1	30	30	30	30	0,00%	3	3	3	3
alu2	370	372	371	372	-0,26%	39	39	39	39
alu3	71	71	71	71	0,00%	10	10	10	10
alu4	680	681	680	680	0,00%	41	41	41	41
al2	106	107	106	106	0,00%	6	6	6	6
amd	496	509	502	506	-0,79%	15	16	15	16
apex1	2156	2178	2167	2159	0,37%	23	25	24	24
apex2	325	347	334	345	-3,18%	26	27	26	27
apex3	1905	1924	1916	1912	0,20%	19	19	19	19
apex4	2879	2897	2887	2886	0,03%	20	20	20	20
apex5	1001	999	982	1010	-2,77%	13	14	13	14
apex6	639	640	639	640	-0,15%	15	15	15	15
apex7	201	202	201	201	0,00%	14	14	14	14
apla	215	225	219	221	-0,90%	13	13	13	13
bca	2656	2695	2675	2672	0,11%	26	26	26	26
bcb	2408	2454	2431	2428	0,12%	25	26	25	25
bcc	2347	2386	2366	2361	0,21%	23	25	24	25
bcd	1706	1738	1722	1721	0,05%	25	25	25	25
bc0	1210	1240	1227	1211	1,32%	24	28	25	25
br1	172	174	173	174	-0,57%	17	17	17	17
br2	117	131	124	125	-0,8 %	13	14	13	13
bw	154	160	157	160	-1,87%	7	8	7	8
b1	10	10	10	10	0,00%	4	4	4	4
b10	516	523	519	520	-0,19%	22	24	23	23
b11	82	83	82	82	0,00%	6	6	6	6
b12	68	68	68	68	0,00%	7	7	7	7
b2	1449	1479	1465	1466	-0,06%	20	20	20	20
b3	395	411	404	410	-1,46%	25	27	26	27
b4	290	298	294	292	0,68%	15	17	16	16
b7	82	83	82	82	0,00%	6	6	6	6
b9	87	87	87	87	0,00%	9	9	9	9
cc	54	54	54	54	0,00%	4	4	4	4
clip	141	141	141	141	0,00%	10	11	10	10
clpl	10	10	10	10	0,00%	10	10	10	10
cmb	43	43	43	43	0,00%	8	8	8	8
cm138a	16	16	16	16	0,00%	4	4	4	4
cm150a	50	51	50	50	0,00%	10	11	10	10
cm151a	24	24	24	24	0,00%	8	8	8	8
cm152a	21	21	21	21	0,00%	6	6	6	6
cm162a	34	34	34	34	0,00%	9	9	9	9
cm163a	32	32	32	32	0,00%	8	8	8	8
cm42a	18	18	18	18	0,00%	3	3	3	3
cm82a	20	20	20	20	0,00%	5	5	5	5
cm85a	36	36	36	36	0,00%	7	7	7	7
comp	98	98	98	98	0,00%	16	18	17	17
con1	17	17	17	17	0,00%	5	5	5	5

Tabulka 5.3: Ukázka naměřených dat randomizovaného *rewrite*.

	AndMin	AndMax	AndAvg	AndRef	Odchylka	LevMin	LevMax	LevAvg	LevRef
5xp1	37	69	55	54	1,85%	5	8	6	6
9sym	136	157	146	138	5,79%	8	9	8	9
9symml	119	142	130	127	2,36%	7	9	7	8
C1355	167	294	196	189	3,70%	10	9	10	11
C17	6	6	6	6	0,00%	3	3	3	3
C1908	220	312	263	284	-7,39%	14	19	16	16
C2670	362	469	414	431	-3,94%	10	13	11	12
C3540	589	681	629	624	0,80%	20	25	21	21
C432	100	99	97	99	-2,02%	13	18	15	15
C499	170	299	193	202	-4,45%	10	9	10	12
C5315	824	946	888	855	3,85%	17	21	18	19
C6288	1946	2275	2093	2080	0,62%	49	60	53	53
C7552	1000	999	1026	1066	-3,75%	14	17	15	14
C880	184	216	199	194	2,57%	11	17	13	13
Z5xp1	57	89	71	76	-6,57%	4	9	5	5
Z9sym	100	99	92	103	-10,67%	7	9	7	8
al2	71	88	78	80	-2,50%	3	5	3	3
alcom	54	64	57	57	0,00%	3	3	3	3
alu1	22	22	22	22	0,00%	2	2	2	2
alu2	191	236	212	208	1,92%	13	19	15	15
alu3	38	50	42	44	-4,54%	6	8	6	6
alu4	400	478	436	436	0,00%	16	21	18	20
amd	162	219	192	165	16,36%	10	9	8	9
apex1	1149	1227	1188	1176	1,02%	10	13	10	11
apex2	133	174	158	143	10,48%	10	13	11	12
apex3	1000	999	1008	998	1,00%	10	12	10	11
apex4	1590	1710	1648	1687	-2,31%	10	12	11	11
apex5	439	525	483	507	-4,73%	10	9	8	8
apex6	312	355	330	323	2,16%	10	9	8	9
apex7	101	133	116	119	-2,52%	10	9	8	10
apla	100	99	108	97	11,34%	5	9	6	7
b1	6	6	6	6	0,00%	2	2	2	2
b10	209	248	228	230	-0,86%	10	9	9	10
b11	52	62	56	59	-5,08%	3	4	3	4
b12	33	40	35	36	-2,77%	4	5	4	4
b2	638	730	684	695	-1,58%	10	13	11	12
b3	171	215	193	192	0,52%	10	9	8	9
b4	141	171	154	154	0,00%	6	8	6	8
b7	53	63	56	58	-3,44%	3	4	3	4
b9	54	60	56	57	-1,75%	5	6	5	4
bc0	442	531	492	517	-4,83%	10	12	11	11
bca	1292	1403	1342	1369	-1,97%	11	14	12	13
bcb	1146	1241	1198	1224	-2,12%	11	15	12	12
bcc	1080	1191	1135	1129	0,53%	10	14	11	13
bcd	793	881	835	833	0,24%	10	14	12	12
br1	62	91	75	65	15,38%	10	9	7	8
br2	48	69	56	50	12,00%	6	8	6	6
bw	100	99	95	95	0,00%	4	7	5	6
c8	57	72	64	66	-3,03%	4	6	5	6

Tabulka 5.4: Ukázka naměřených dat syntézního skriptu *choice*.

Závěr

Cílem mé práce byla randomizace algoritmů v systému ABC, aniž by byla funkcionalita algoritmů narušena. Randomizované algoritmy by měly v průměru dosahovat stejných, výsledků jako deterministické. Randomizoval jsem příkazy *balance* a *rewrite*. Randomizace rewrite dopadla podle očekávání a její využití může být prospěšné v syntézním procesu. Testy ukázaly průměrné zlepšení ve velikosti 0,18% a hloubky o 1,09%. Plně randomizovaný *balance* se nechoval podle očekávání a přinášel stabilně horší výsledky. Velikost obvodu je v průměru o 1,12% větší. Byl proto upraven na částečně randomizovanou formu, která přináší stabilně lepší výsledky, než úplná. Zhoršení v takovém případě činí pouze 0,09%, což se dá považovat za statistickou chybu.

Téma randomizace algoritmů v ABC je stále otevřené, protože se nejedná pouze o tyto dva, které jsem v této práci implementoval. V budoucnu je možná randomizace i dalších algoritmů jako jsou např. *resub*, *refactor* a další.

Literatura

[1] A. Mishchenko, S. Chatterjee, and R. Brayton: Delay optimization using SOP balancing. *International Workshop on Logic and Synthesis*, 2011. Dostupné z: `http://www.eecs.berkeley.edu/~alanmi/publications/2011/iccad11_sop.pdf`

[2] A. Mishchenko, R. Brayton, S. Jang, and V. Kravets: DAG-aware AIG rewriting: A fresh look at combinational logic synthesis. *International Conference on Computer-Aided Design*, 2006. Dostupné z: `http://www.eecs.berkeley.edu/~alanmi/publications/2006/dac06_rwr.pdf`

[3] Shannon, C. E.: *A Symbolic Analysis of Relay and Switching Circuits.* Diplomová práce, Massachusetts Institute of Technology, 1940.

[4] The University of Texas at Dallas: *A Symbolic Analysis of Relay and Switching Circuits.* 2012. Dostupné z: `http://www.utdallas.edu/~dodge/EE2310/lec5.pdf`

[5] McCluskey, E.J., Jr.: Minimization of Boolean Functions. *Bell System Technical Journal*, 1956.

[6] D, G.: Electrically programmable logic circuits. Červen 18 1974, uS Patent 3,818,452. Dostupné z: `http://www.google.com/patents/US3818452`

[7] G. Ausiello et al.: Complexity and Approximation. *Springer*, Nov. 1999.

[8] Kurt Keutzer: Technology Binding and Local Optimization by DAG Matching. *Design Automation, 1987. 24th Conference on*, 1988: str. 344. Dostupné z: `http://janders.eecg.toronto.edu/1387/readings/dagon.pdf`

[9] Synthesis, B. L.; Group, V.: ABC: A System for Sequential Synthesis and Verification. Dostupné z: `http://www.eecs.berkeley.edu/~alanmi/abc/`

[10] Petr Fišer, Jan Schmidt, Jiří Balcárek: Sources of Bias in EDA Tools and Its Influence. *Design and Diagnostics of Electronic Circuits and Systems, 17th International Symposium on*, 1988: str. 344. Dostupné z: `http://users.fit.cvut.cz/~fiserp/papers/ddecs14.pdf`

[11] Berkeley Logic Synthesis and Verification Group: *Quick Look under the Hood of ABC*. 2006. Dostupné z: `http://www.eecs.berkeley.edu/~alanmi/abc/programming.pdf`

[12] Berkeley Logic Synthesis and Verification Group: *Constructing AIGs in ABC*. 2007. Dostupné z: `http://www.eecs.berkeley.edu/~alanmi/abc/aig.pdf`

[13] Randal E. Bryant: Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys, 24, 3, 293- 318*, 1992: s. 2–3.

[14] Microsoft Corporation: *Hungarian Notation*. 1999. Dostupné z: `https://msdn.microsoft.com/en-us/library/aa260976(v=vs.60).aspx`

[15] Synthesis, B. L.; Group, V.: ABC: System for Sequential Logic Synthesis and Formal Verification. April 2015. Dostupné z: `https://bitbucket.org/alanmi/abc`

[16] S. B. Akers: Binary Decision Diagrams. *IEEE Transactions on Computers, vol. C-27, No. 6*, June 1978: s. 509–516.

[17] R. E. Bryant: Graph Based Algorithms for Boolean Function Manipulation. *ACM Computing Surveys, 24, 3, 293- 318, Vol. 35*, August 1986: s. 677–691.

[18] Vahid, F.: *Digital Design with RTL Design, Verilog and VHDL (2nd ed.)*. John Wiley and Sons., 2010.

[19] cplusplus.com: *qsort*. Dostupné z: `http://www.cplusplus.com/reference/cstdlib/qsort/`

[20] S. Chatterjee, A. Mishchenko, and R. Brayton: Factor cuts. *Proc. IC-CAD*, Nov. 2009. Dostupné z: `http://www.eecs.berkeley.edu/~alanmi/publications/2006/iccad06_cut.pdf`

[21] cplusplus.com: *random_shuffle*. Dostupné z: `http://www.cplusplus.com/reference/algorithm/random_shuffle/`

Příloha A

Seznam použitých zkratek

AIG And Inverted Graph

BDD Binary Decision Diagram

CI Primary Input and Latch Output

CO Primary Output and Latch Input

DAG Directed Acyclic Graph

FPGA Field Programmable Gate Array

GUI Graphical user interface

LUT Look up Table

PI Primary Input

PO Primary Output

PLA Programmable logic array

SOP Sum Of Products

XML Extensible markup language

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `abc_Randomized` ..... adresář se spustitelnou formou implementace
  - `readme.md` ..... manuál ke kompilaci a používání
  - `src` ..... adresář se zdrojovými soubory
  - `lib` ..... adresář externích knihoven
  - `circs` ..... adresář s testovanými obvody
- `tests` ..... adresář s histogramy a naměřenými daty
- `text` ..... adresář s potřebnými soubory k vysázení BP
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `thesis.tex` ..... text práce ve formátu LaTeX