

DSnP Final Project Report

Functionally Reduced And-Inverter Graph(FRAIG)

School ID: B07901016

Name: 朱哲廣

Email: b07901016@ntu.edu.tw

Professor: Chung-Yang Ric Huang

0. Outline

1. Circuit Implementation
2. Function Implementation
3. Simulation Method and Implementation
4. Fraig Method
5. Review

1. Circuit Implementation

0. CirGate

在思索架構時，我認為所有的 Gate Type 都很類似，所以在定義 class 時幾乎都把 property 都定義在 CirGate 上，在讓其他而每個 gate 去繼承。我的 fanin 跟 fanout 都是用 vector 來實作，這樣的實作方式讓我在做很多操作時不用特判這個 Gate 是哪個形態再算要取哪個 fanin。

除此之外，每個 Gate 的 fanin 跟 fanout 裡都是存 class CirGateV，以此來記這條連接是否為反向，在做之後的 merge\replace 時比較好做。

1. CirMgr

以此類別來支援各種 function，與輸入輸出等。在此與 HW6 架構大致相同。

2. Function Implementation

0. Sweep

我的實作方法是：

Step 1：從 gatelist 掃所有的 gate，偵測其是否在 dfslist 內

Step 2：若其不在 dfslist 內，就將其刪除(_removeGate)

那這部分我想特別提到的是如何看一個 gate 是否在 dfslist 裡面。最一開始我的實作方始是在每一個 Gate 上掛一個變數，但在做完後面的 function 後，我發現如此方法在重新建一次 dfslist 時無法重新設值。因此，我改成在 CirMgr 中增加一個 unordered_set 來記有哪些 gate 在 dfslist 裡。而 unordered_set 的尋找複雜度是不錯的，所以以此結構來特別實作。

1. Optimize

在我的思索過後，我認為 Optimize 就是不斷地在替換 Gate，所以特別寫了一個 _replaceGate 來實作。

2. Strash

在這個 function 的實作中，我花最多時間在想其 fanin 的 hash function，也以此查了許多資料如 Cantor Pairing function、Szudik's function 等，或是將其轉成 string 後利用 HW4 中的方法，來將 fanin 跟其反值綁在一起。但最後我採取的方法是將每個 Gate 對應的 Literal 取出，以此就有涵蓋使否反向的資訊，另外將 literalA << 32 | literalB，以此快速生成我的 hash 值。

3. Simulation Method and Implementation

0. Parallel Simulation

我的實作方式是對每個 Gate 都有一個當前的`_simVal`，而此是一個`size_t`。因為對於程式，使用 `boolean operator` 的效率很高，所以可以一次將 64 個模擬值包成一個`size_t`下去做模擬，加速模擬時間。

1. File Simulation

在實作 `fileSim` 時，我一開始是將所有資料都存起來再 `parse` 成一個個`size_t`的模擬值。當全部資料都 `parse` 完後再一起做模擬。這樣實作完後我有兩個大問題：

- A. 大大增加了 `Simulation` 的前置時間
- B. 記憶體空間用量極大(曾經對 `sim13.aag` 測到 600M 以上)

因此在觀察老師的程式後，我想到對於一個檔案模擬，邊讀檔邊模擬才是個明智的選擇。在讀檔時候順便做完 `parsing`，每讀到 64 個 `simulation value` 就做一次的 `parallel simulation`。在這樣做之後，我的時間跟空間都有大大的下降(1s / 30M)。

2. Simulation Order

在這裡我選擇的是掃一次 `dfslist`，對其中每一個 `gate` 分別做一次 `simulation`。在決定這樣做之前，我有參考老師的講義實作 `Event driven` 的方法，但是在實測之下發現其效率遠不如直接掃 `dfslist` 的效率。這裡我認為的幾個原因有：

- A. 遞迴不斷 `call function` 的速度相較直接 `for loop` 還要慢
- B. 這次的模擬的只需要一個`& operator` 就可以完成，而其速度極快

在此我認為 `Event driven` 的做法要相較於直接跑還有效率的情況是當每一個 `Element` 的 `Simulation Time` 很長，但因為這個只有`&`要做，就採用直接搜。

3. Class `FecGrp`

這是我定義的 `FecGrp` class。其用途是儲存同一個 `group` 下有哪些 `gate`。而我選擇用來存資料結構是 `map<size_t, CirGateV>`，因為其查詢、插入跟移除的效率都很不錯(`logN`)，而在創每一個 `fecGrp` 時會用到很多的就是插入跟移除。

另外這裡會在次用到 `CirGateV` 的原因是每一個 `fecGrp` 下的每一個 `Gate` 可能是 `FEC` 或 `IFEC`，所以剛好也可以用此 class 來做。

而在 `CirMgr` 裡是一個 `vector (_fecGrps)`來存這個 `Circuit` 有哪些 `fecGrp`。

4. Generate FEC (Functionally Equivalent Candidate) Group

在此我的邏輯是：將每一個 `FecGrp` 下的第一個拿來當基準(`base`)，而將其它的與 `base` 做 `simVal` 的比較，若有不符合的就將其丟到一個 `temporary map`，這個 `map` 是記錄 `simVal` 與對應的新 `fecGrp*`。因此在同一個 `fecGrp` 下遇到其他已經出現過的 `simVal` 就可透過此 `map` 快速找到其對應的 `fecGrp`。若是沒找到，就開一個新的 `fecGrp`。

5. IFEC Group Detection

一開始我的做法是將生 `Fec Group` 的做法再做一次，只是將 `simVal` 改成`~simVal`。但是這樣會有的問題就是時間太長且空間更大，最後更要再花時間 `merge`。

在苦思之後，我想出一個可能正確的邏輯如下：

- A. 一個 `Gate` 如果 `simVal` 與`~simVal` 皆與 `base` 不相符，那就代表他一定跟 `base` 在不同的 `FecGroup`。

- B. 在 A 的前提下，搜尋 map 中找 simVal 或~simVal 相等的 fecGroup，若有找到就可以加進去。這時要考慮其找到的是 simVal 還是~simVal 的對應，而要去改其進入時的 CirGateV。
- C. 在 A 下，若是找不到對應，就取這個 gate 的 simVal 作為新的 fecGrp 與其 base。

在這樣的邏輯下，我推出一個可能會導致錯誤的 IFec grp 分配：若第一次 genFecGrp 時，gateA 與 gateB 是同向相等，但在第二次的模擬下，gateA 與 gateB 是反向相等，那這樣 gateA 與 gateB 可以肯定的是在不同 Fec Grp，應該要被分開，然而我的邏輯並不會在第一時間將其分開。

但在實測下並沒有明顯的此問題，我認為的其原因是只要 simulation pattern 夠多，要同時一直滿足這樣的條件少之又少。

6. Fec Group Reduction

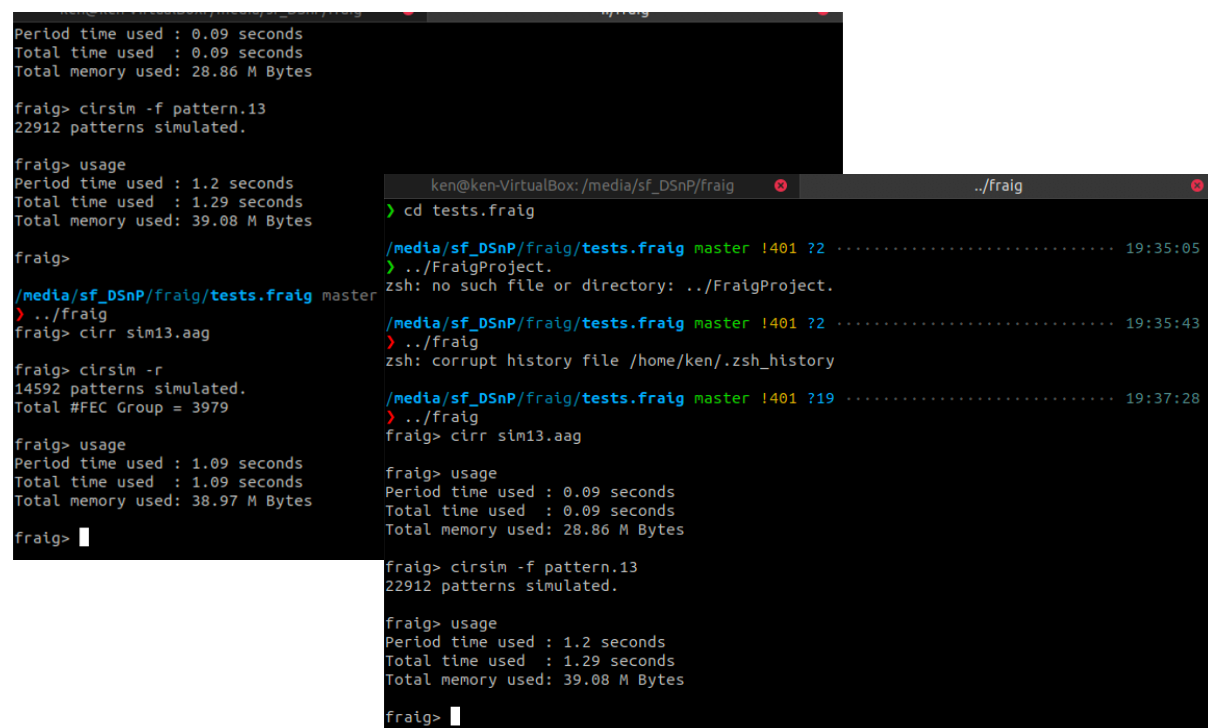
在我第一次做完 fec grp 實測的時，發現又多又冗，這時才想到若是一個 fecGroup 只含 1 個 child，那其就應該被刪除。我的實作方法是在每次遇到新的 fecGrp 時都先不急著加到 CirMgr::_fecGrps 裡，而是跑完之後，若其 child > 1 才 push 入，可以節省事後再 erase 的時間，而且 vector 的 erase 效率也不好。

7. RandomSim

我在此的實作方法是對每一個 PI 隨機生成兩個整數($< 2^{32}$)，將其中一個 left shift 32 並加另一個。以此下去實作，並記錄_fecGrps 的 size。若是連續 simulate 30 次左右都沒有新的 fecGrps 的情況下就停止。

8. Result

以 sim13.aag 實測 random sim 與 讀檔案(pattern.13) sim



```
Period time used : 0.09 seconds
Total time used : 0.09 seconds
Total memory used: 28.86 M Bytes

fraig> cirsim -f pattern.13
22912 patterns simulated.

fraig> usage
Period time used : 1.2 seconds
Total time used : 1.29 seconds
Total memory used: 39.08 M Bytes

fraig>
/media/sf_DSnP/fraig/tests.fraig master !401 ?2 ..... 19:35:05
> ../FraigProject.
zsh: no such file or directory: ../FraigProject.

/media/sf_DSnP/fraig/tests.fraig master !401 ?2 ..... 19:35:43
> ../fraig
zsh: corrupt history file /home/ken/.zsh_history

/media/sf_DSnP/fraig/tests.fraig master !401 ?19 ..... 19:37:28
> ../fraig
fraig> cirr sim13.aag

fraig> usage
Period time used : 1.09 seconds
Total time used : 1.09 seconds
Total memory used: 38.97 M Bytes

fraig>

fraig> usage
Period time used : 0.09 seconds
Total time used : 0.09 seconds
Total memory used: 28.86 M Bytes

fraig> cirsim -f pattern.13
22912 patterns simulated.

fraig> usage
Period time used : 1.2 seconds
Total time used : 1.29 seconds
Total memory used: 39.08 M Bytes

fraig>
```

4. Fraig Implementation and Method

0. Logic

我的 Fraig 大致邏輯是對於每個 fecGrp 下，對 base 與其他剩下的值都做 SAT Prove。若是有 SAT，變將此反例存起來留著之後做 simulation，若是全部都是 UNSAT，變可執行 `_mergeFecGrp()`，此 function 的用途是將 fecGrp 下的所有 gate 都與 base 做 merge，最後再刪除此 fecGrp。另外，對於 Const0 在 SAT Model 中的實作，我是以 push 一個 input 互相反向的 gate 來完成的，以此省很多特判的麻煩。

1. Optimization (I) Order

我做的第一個優化是其「順序」。最一開始我是照著 dfslist 的順序去跑，對每個 gate 找他的 fecGrp 做 SAT。那做完之後我遇到的問題就是他剩下的 fecGrp 都存在在 dfslist 後半段的 gate 上，導致每次都要先走很多多餘的路才能抵達，因此排除再用 dfslist。

那接著我嘗試直接跑 fecGrps，對每個 fecGrp 做。但是這裡有問題的就是在於 fecGrp 在 Circuit 中的 base 是很隨機的，所以效率不好，而且在 fecGrp 的操作中很容易不小心誤刪導致 segmentation fault。

於是我試著想一個獨立 dfslist 與 fecGrp 的做法，那這裡我使用的是 BFS + queue 來做 fecGrp 的紀錄，並用一個 `unordered_set<unsigned> inQue` 來存 gate 是否在 queue 中：

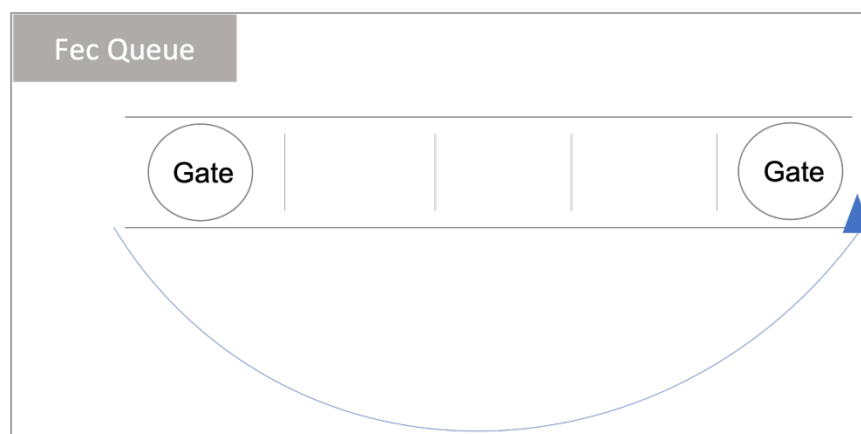
Step 1：將 dfslist 中的 PI Gate 推入 fecQue

(若 fecQue != empty)

Step 2：將其第一個 pop 出來，找其 fecGrp 作處理

Step 3：將此 Gate 的 fanout 中，沒在 Queue 裡的推入 Queue 裡(!inQue)

Step 4：處理 fecGrp 會回傳這個 Gate 的 fecGrp 是否處需要再處理，若不需要則不用放回 queue，若需要的話則在 push_back



會用 Queue 的原因是因為他的 pop 跟 push 的複雜度都還不錯，而且符合我的需求，並且在實作上因為與 dfslist/fecGrp 獨立，所以不用想太多指標或會不會 segmentation fault 的問題。

2. Optimization (II) Pattern & simulation

對於每一個 SAT 的結果都可以得到一個反例，也就是說這個反例可以幫助我們拆散現在正在被處理的 fecGrp。那只要再次 simulation 這個 pattern，就可以得到新的 fecGrp，而可以將舊的 merge 掉。

我在程式中紀錄每一個 SAT pattern，並將其合併，直到有 64 個 SAT pattern 出現過，就以新的 SAT pattern 做一次 simulation，就會有新的 fecGrp 並會再之後繼續被推入 queue 中處理。

3. Optimization (III) Pattern

對於每一個 SAT 結果，我都存入 unordered_map<string> _duPat，這個 set 的用途是來記住有哪些 pattern 已經有了，那如果遇到一模一樣的反例，就可以不用將其再放入 SAT pattern。這樣可以用更少次的 simulation 得到更多的 fecGrp。

4. Optimization (IV) set<unsigned> _fecToMerge

在爆搜所有的 Fec Pair 時，我發現如果知道這個 fecGrp 已經被搜過了，那就可以先把它存起來之後再一起 merge。所以會有這個 set 來記錄哪些 fecGrp 已經被搜過，那如果遇到下一個 gate 他的 fecGrp 已經被搜過，那就不會再對這個 fecGrp 作處理。

要這樣做的情況有以下 3 個：

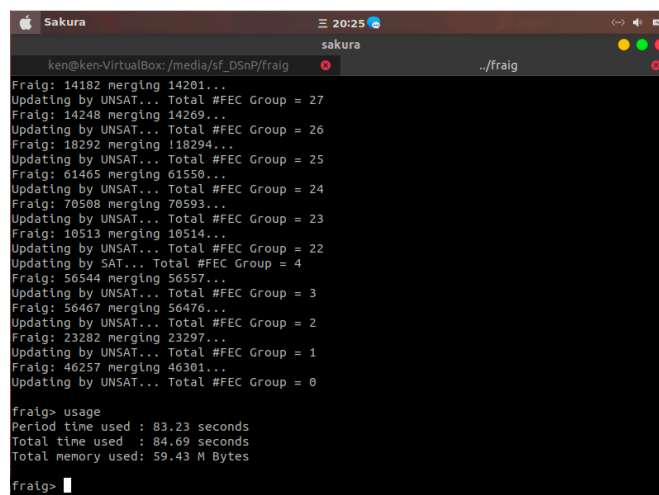
A. All UNSAT -> push to _fecToMerge

B. Partial SAT -> push to _fecToMerge

C. SAT cnt > 64 -> push to Queue, wait for next time process

其中 A, B 的情況都會將 pattern 存起來，所以只要 simulation 完之後，剩下在 A, B 的 fecGroup 就一定是全 UNSAT 就可以選擇把 A 或 B 類型的 fecGrp 直接 merge 掉。而 C 情況因為還無法保證其全部都是 UNSAT 的，所以將此重新放回 Queue，待下一次處理。

5. Result :



```
ken@ken-VirtualBox: /media/sf_DSnP/fraig
fraig: 14182 merging 14201...
Updating by UNSAT... Total #FEC Group = 27
Fraig: 14248 merging 14269...
Updating by UNSAT... Total #FEC Group = 26
Fraig: 18292 merging 118294...
Updating by UNSAT... Total #FEC Group = 25
Fraig: 61465 merging 61550...
Updating by UNSAT... Total #FEC Group = 24
Fraig: 70508 merging 70593...
Updating by UNSAT... Total #FEC Group = 23
Fraig: 10513 merging 10514...
Updating by UNSAT... Total #FEC Group = 22
Updating by SAT... Total #FEC Group = 4
Fraig: 56544 merging 56557...
Updating by UNSAT... Total #FEC Group = 3
Fraig: 56467 merging 56470...
Updating by UNSAT... Total #FEC Group = 2
Fraig: 23282 merging 23297...
Updating by UNSAT... Total #FEC Group = 1
Fraig: 46257 merging 46301...
Updating by UNSAT... Total #FEC Group = 0

fraig> usage
Period time used : 83.23 seconds
Total time used : 84.69 seconds
Total memory used: 59.43 M Bytes
fraig>
```

5. Review

這門課是我從大一就很想修的一門課，聽說是最後一年開了還好我有感到這班末班車。這門課真的收穫很多，改變我很多以前寫 `code` 的習慣，畢竟也不會有機會在有厲害的人可以帶著寫大型又算有趣的程式。每一次的作業都是我需要認真動腦，想怎麼樣寫最有效率，最方便寫而且可以達成目標的，每一次都很期待下一次作業會帶來什麼樣的挑戰。這次的 **Fraig** 是最後一天凌晨通宵努力的結果，但卻寫得很開心很興奮，想著可以怎麼樣讓我的程式更快更好。這門課真的非常多學習的地方，從 **parsing** 的寫法架構、記憶體管理(每個人心中都有記憶體)、**debug** 技巧，思考一大堆可能的 **segmentation fault** 的原因、查了非常非常多資料，快速看大型 `code` 的能力等。真的學到很多，也非常感謝老師這一學期的教導！