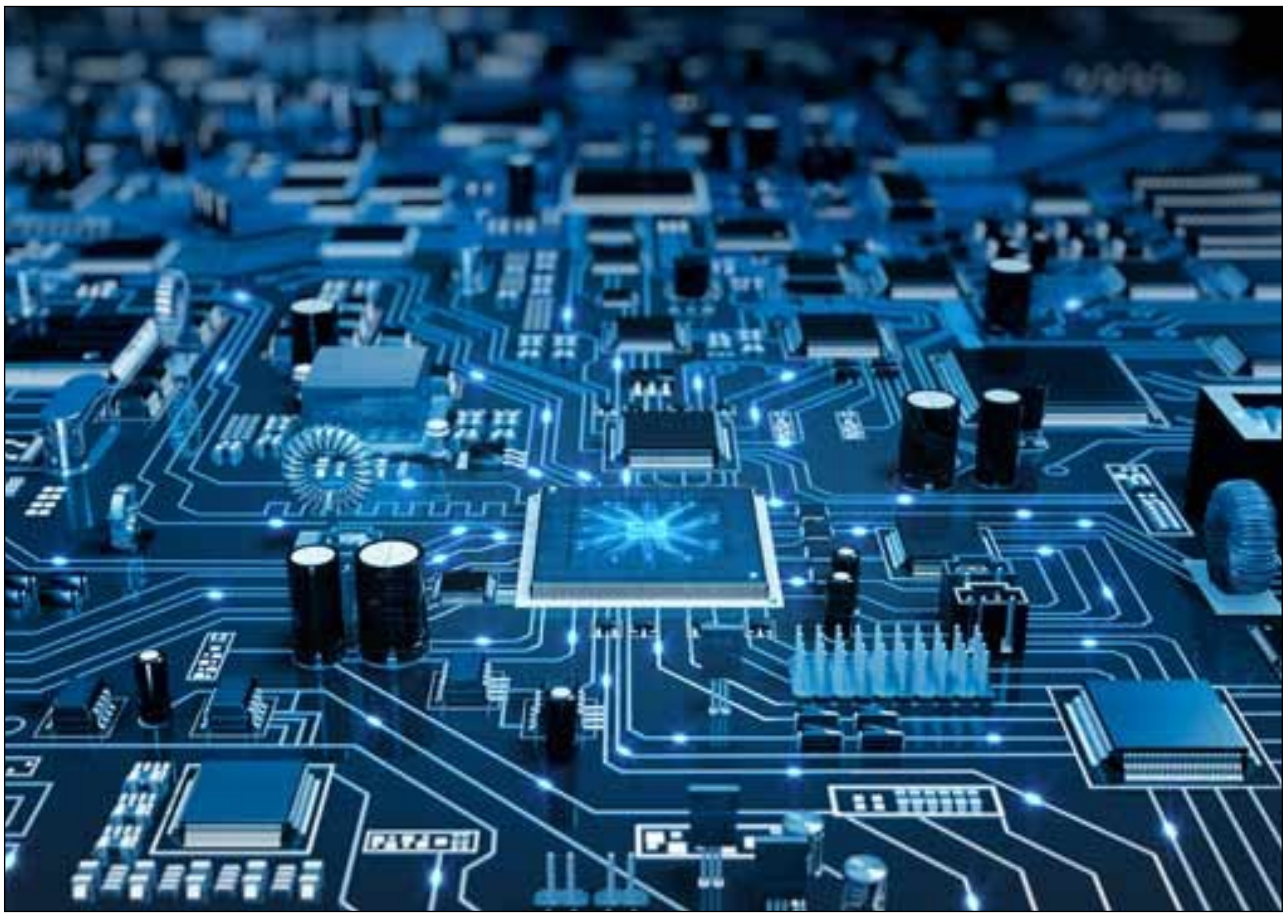
FRAIG專題報告

陳博彥 電機三 B04901067

Contact : 0933351384 / b04901067@ntu.edu.tw



資料結構/演算法設計

1.Sweep

Sweep的演算法很簡單，就是先把DFS List中所有的Gate都Set Global Reference，之後在Total Gate List中走過一遍，如果該Gate不是Global Reference，就呼叫CirGate::sweep()。值得注意的是，如果某個AIG Gate的Fanin也是即將要被Sweep掉的，那麼就沒有理由要處理這兩個Gate間的連接問題，反正等等都要清掉，所以CirGate::sweep()會先判斷它的那支Fanin是否為Global Reference，如果是（也就是在DFS List中）才要把這個Gate從它Fanin的Fanout List中移除。

2.Optimize

Optimize的演算法就是對Gate做基本的合併，如果某個Gate跟另一個Gate合併，那麼它Fanout以及Fanout的Fanout等等所有之後的Gate都可能會受影響，因此要從PI端開始簡化，另外，只有在DFS List中的Gate才需要做簡化。我原本是用function recursive call的方式寫，opt()某個gate前，先recursive呼叫他兩個Fanin的opt()，再執行自己，從PO端開始呼叫。但是後來發現，其實只要按照DFS List中的順序，一個接著一個呼叫，就可以了，根本不用recursive。改完之後程式碼變得精簡一些，只是效能幾乎一樣，因為opt被呼叫的次數都恰等於DFS List中的Gate數。

3.Strash

首先，我使用我自己HW7的HashSet來實作，並稍微改寫。

關於StrashNode，我的Node中有兩個data member：一個是該Gate的pointer，另一個是根據這個Gate的Fanin所作成的key。key的形式是一個size_t，產生方式是把兩個fanin的gateID個別乘以二，有invert再加上一，再把算出來的數字中，比較小的存在size_t的前32個bit中，較大的存在後32個bit當中，如此一來，每個不同的fanin組合，都可以產生一個獨一無二的key，並且不受兩fanin順序影響。

StrashNode需要overload兩個operator，一個是operator==，一個是operator()，其中==就是直接比較兩個Node的key是否一樣也就等價於比較fanin是否一樣，()就是直接return該key的數值。

最後，我把HashSet的insert()改寫，把return type改成CirGate*，原本的insert是：

if (有找到一樣的) { 存進去; return false; }

改成

if (有找到一樣的) return 原先存的gate的pointer;

else return 0;

如此一來，程式碼就變得很精簡，也很有效率，在CirMgr::strash中，只要寫：

for all AIG gate in DFS List

StrashNode N(……);

CirGate* G = _hash.insert(N);

if (G) merge two gate; //回傳的pointer不是零

其中merge可以重複利用optimize中所用的merge。

最後關於Hash Size，其實我原先不大知道怎麼設定Hash Size比較合適，但是回想HW7，老師的做法是，把預計會有的node數量，丟到一個function中，該function根據數值回傳應該設的Hash Size。在Strash中，node的數量，worst case就是AIG的數量，因此我就直接把AIG的數量，丟到老師原本設計好的function中，依照回傳值設Hash Size。

4.Simulation

Simulation的步驟有：(1)讀/產生pattern (2)根據pattern來simulate(3)根據各個gate的simulate pattern來分FEC Group。

(1)讀/產生pattern

把所有patternfile的資料存到_InputPattern之中，_InputPattern是一個二維的vector，第一個維度是PI的數目，第二個維度是roof(pattern數/64)，內部的資料是用size_t來存，也就是把第一個PI的前64個input存在_InputPattern[0][0]，第65~128個存在_InputPattern[0][1]中，第二個PI的前64個input存在_InputPattern[1][0]，依此類推。

(2)根據pattern來simulate

我採用Parallel-Pattern Simulation也就是一次做64筆資料，這樣比較有效率，程序如下：

for _InputPattern.size()/64

把前64筆pattern存到PI的_SimPat中;

for _DFS_List.size()

_DFS_List[i]->simulate();

ConstructFEC();

在simulation中，每個gate會用到兩個data member，_SimPat和_SimPatList，其中_SimPat是一個size_t，用來存最近64筆pattern，_SimPatList是一個vector<size_t>，用來存過去的所有pattern。

simulate()就是把該gate的兩/一個fanin gate的_SimPat，拿來做bitwise ADD(AIG)，或者直接引用(PO)，當數值存在_SimPat，之後再_SimPatList.push_back(_SimPat)。當全

部的_InputPattern都處理完之後，再來跟去所有gate的_SimPatList來分FEC group，也就是ConstructFEC()，會在下一部分中說明。

(3)根據各個gate的simulate pattern來分FEC Group

ConstructFEC()中，我使用了一個HashSet來比對各個gate的_SimPatList是否一樣（或者完全相反），再使用一個二維的vector（其中一個維度用class包起來）來儲存FEC group。定義如下：

HashSet<PatternNode> _PatternHash;

vector<FEC*> _FEC_List;

其中，PatternNode裡面有兩個data member：一個是某gate的pointer，另一個是該gate的_SimPatList（當然，是存這個List的pointer，而不是備份整個List）。

FEC則是建立FEC group的class（每個group一個FEC），裡面只有一個data member，為一個vector，用來存該FEC中有哪些gate，以及是否反向。

首先，按照ID順序，由小到大，把每個gate做成的PatternNode丟到_PatternHash中，一樣是使用我自己改寫的insert()：如果_PatternHash中，沒有任何一個node跟現在這個node（N）有相同的_SimPatList數值，就回傳0；如果_PatternHash中有某個舊的PatternNode（O）跟現在這個（N）有相同的_SimPatList數值（或者完全相反），就把N丟到O所屬的FEC中，如果O還沒有所屬的FEC（就是之前還沒有別的Node跟他有一樣的pattern），就直接創一個新的FEC，並把該FEC的pointer push到vector<FEC*> _FEC_List中，再把O、N依序放置到FEC中，並紀錄是否反向。

也就是：

```
for i = 0 ~ #(AIG+CONST){
    PatternNode N( _totalGateList[i] );  //用ID為i的那個gate的pointer及_SimPatList來建立PatternNode
    CirGate* O = _PatternHash.insert( N );  //如果已經有相同pattern的node在hash中，回傳舊的gate的pointer
    if ( O ) {
        if ( O is already in a FEC ) put N in the FEC of O;
        else{
            construct a FEC that contains O;
            put N in the FEC of O;      //有紀錄是否反向
            push this FEC into _FEC_List;
        }
    }
}
```

由於是按照gate ID小到大執行，所以做完之後，每個FEC中的gate都已經按照gate ID小到大排好了，但是題目要求，每個FEC還要再按照該FEC中，最小的gate ID來排序，而現

在_FEC_List中的順序還是亂的，所以呼叫std::sort來排序，排序方式就是比較第FEC中第一個gate的ID。

剛剛提到，FEC中的data member，是一個vector，記錄有哪些gate在這個FEC當中，並記錄是否跟第一個反向，而這個資訊（gate的pointer跟是否反向）剛好跟連接fanin、fanout所使用的CirGateV一樣！所以我直接把FEC裡的那個vector宣告成vector<CirGateV>。

注：對於每個gate，我需要記錄他的fanin、fanout有誰，以及是否反向，需要儲存對應gate的pointer（CirGate*）以及是否反向（bool），我按照教授建議的做法，自訂一個class CirGateV，member是一個size_t，前63個bit記錄了CirGate*最後一個bit當bool用，FEC剛好也可以套用同樣的class。

至於，PatNode要套用HashSet，需要overload operator==和operator()，對於相同或者完全相反的_PatternList， operator==要回傳true，operator()要回傳一樣的key。

operator==實現方式很普通，就是逐一比對_PatternList中的每一個pattern（size_t）有沒有相同，或者完全相反。

至於operator()，需要跟去pattern產生某個key，盡可能獨特，與別的pattern所產出的不同。起初，我「相信」把多一些pattern拿來產生這個key，比較能夠產生不一樣的key，所以演算法如下：

size_t key == 0;

將key bitwise XOR _PatternList的第1,11,21...項; //至多XOR十項

return key;

之所以要XOR第1,11,21...項，是「相信」多考慮幾項，比較能生出獨特的key，之所以要每次跳10項，是「相信」pattern file中可能有某種規律，不要集中取樣同一區域，比較容易產生獨特的key。

然後，為了要讓完全相反的pattern也能產生同樣的key，我會先記錄該pattern的第一項的最後一個bit，如果該bit是1，而且XOR的次數是奇數，最後就return ~key;

注：完全相反的pattern，XOR奇數次會完全相反，XOR偶數次會完全相同。

但是，最後實測發現，這樣好像沒什麼用（除非pattern file有某種規律），所以最後就把它改掉，直接把_PatternList的第一項（size_t）當作key，如果第一項的最後一個bit是1，就return ~key。

最後，怎麼取Hash Size？我的方法跟strash一樣，就是把AIG個數丟到老師寫的function中。

5.FRAIG

關於fraig，我的做法是先在SatSolver中建好電路，然後以每個FEC中ID最小的那一項為leader，去prove它跟其他gate的異同，如果相同或者完全相反就merge，不同就把counter example記錄在_InputPattern中（就是前面用來sim的那個vector），FRAIG完後根據counter example再simulate一次、重建DFS List、strash，如果在simulate過後，有新的FEC產生，就再Fraig一次（至多兩次）。

除此之外，我有用某個變數，記錄有多少個gate倍SatSolver刪減，為了增加效率，每當有n個gate倍SatSolver刪掉、重新strash一次，然後再根據新的電路建立新的SatSolver，根據一連串試驗後得知，n設為200是一個滿恰當的選擇，最後也採用此設計。

至於怎麼在SatSolver中放CONST 0？我的方法是在Solver中宣告兩個Var，第一個Var（A）沒有接fanin，在Solver中會被視為PI，第二個Var（B）被宣告為AIG，兩個fanin分別為(A, !A)，如此一來，就可以把B當做CONST 0來用了。

瓶頸

整體上，我的程式有幾個的問題：

1.Gate的fanin使用vector來儲存

當初在打HW6的時候，沒有考慮清楚，使用vector來儲存gate的fanin，而不是用fanin1、fanin2來存，因此記憶體用量比ref大。寫Final時，覺得修改難度過大，外加程式碼會變長許多，就先不做修改。

2.Simulation記憶體用量過大

這應該是最重要的問題所在。在sim時，我是把所有pattern Sim完，之後再來分FEC group（一開始沒考慮清楚），這樣就需要把所有pattern存在記憶體當中，導致記憶體用量很大。就sim13.aag而言，ref的記憶體用量約為19MB，我的約為630MB。

然而，這理論上不大會影響runtime（如果不存所有的pattern，每跑一段就要分一次FEC，時間差不多），我sim和fraig的時間用量，都大概是ref的1.5~2倍，sim並沒有特別多。

實驗/比較

1.用Parallel-Pattern Simulation與不用Parallel-Pattern Simulation

我寫了兩個不同版本的simulation架構，一個有用Parallel-Pattern，也就是每64筆資料再AND一次，另一個沒有用Parallel-Pattern，也就是每筆pattern傳入都sim過整個電路，並比較其效能。

實驗結果，以sim13.aag，採用pattern.13為例：

使用parallel：

Total time used ：5.02 seconds

Total memory used: 633.6 M Bytes

不使用parallel：

Total time used ：79.77 seconds

Total memory used: 626.2 M Bytes

Refrence：

Total time used ：2.85 seconds

Total memory used: 18.89 M Bytes

以上數據都是測試三次後的平均

由數據可知，採用Parallel-Pattern，所用時間大約為不使用的1/16，而記憶體用量差不多。由以上數據也顯示，在大型的電路simulation中，我的時間用量約為ref的1.5~2倍，記憶體則多出許多，應證前面提到的問題。

雖然同樣都是O(n)複雜度，但是由於電腦實作上的問題，想者速度相差甚遠。

寫兩個版本的simulation超級花時間的！這可是一個用肝臟換出來的實驗結果，十分珍貴！好啦不過其實我是一開始在寫sim的時候，忘記可以用parallel所以才先寫一份沒parallel的啦...不過難得有這樣的結果可以比較（我想應該很少人有做這樣的實驗吧？），也滿特別的。

2.使用counter example

在fraig的過程中，我們可以從solver取得許多counter example，可以拿這些counter example再去simulate，如果有新的FEC產生，就再fraig一次。但是為了控制runtime，會設置一個遞迴次數上限，在此做比較：

以樣以sim13.aag為例，首次sim使用pattern.13：

原本Gate數：88410

只fraig一次（不使用counter example）：

Total time used ：198.7 seconds

Total memory used: 660.9 M Bytes

#Gates remained: 85639（2771 reduced）

fraig兩次：

Total time used ：220.3 seconds

Total memory used: 669.2 M Bytes

#Gates remained: 85572（2838 reduced）

fraig三次：

Total time used ：426.2 seconds

Total memory used: 656.4 M Bytes

#Gates remained: 85572（2838 reduced）

Reference：

Total time used ：107.2 seconds

Total memory used: 43.53 M Bytes

#Gates remained: 85578（2832 reduced）

第一次fraig，花了約200秒，刪減了2271個gate，第二次fraig（用counter example）只多花20秒，又刪掉了67個gate，算是值得。

然而第三次fraig，竟然又花了200秒，而且竟然一個gate都沒有刪減！由此可見，就在這個數量級的電路中，fraig完一次後用counter example simulate完，再recursive呼叫一次fraig就差不多了，最後也採用此設計。

未來可能改良方法

我在寫project中想到一些可能可以讓程式效能更好的方式，但是實作的時候，一直Segmentation fault，外加Segmentation fault的地方都在SAT的函式裡面，很難debug，最後沒來得及在死線前完成，或許以後還有機會改良。

1.FRAIG過程中，動態更新DFS List，FRAIG在List中的FCC

在fraig時的過程中，每fraig掉n個gate，就optimize和strash一下，並且更新DFSList，原本在FEC中的gate，很多會變得不再新的DFS List中，之後只處理FEC中「還在DFS List裡」的gate，在處理大型電路時，理論上會變得更有效率。

2.由比較接近PI的Gate開始FRAIG，每刪掉n個gate，就重新new一個solver

我前面有提到，每刪掉n個gate，就重新new一個solver，可以讓solver中的電路越來越小，證明越來越快。我們之後還可以更進一步，從靠近PI的gate開始prove，因為靠近PI

的gate，fanin層數比較少，SAT在prove的時候會比較快，所以我們可以在短時間內先刪掉比較多的gate，讓solver更新的時候，複雜度降低更快。

課程建議

我覺得，對於任何一門課，重要的不是要教會學生什麼高深的學問，而是要讓學生認識這個領域、愛上這個領域、並且有能力自己深入研究，因為課堂上能教的一定十分有限。

資料結構與程式設計，我覺得重要的不是要讓學生學會什麼厲害的資料結構，而是要透過資料結構，讓學生練習程式設計，並且在設計的過程中，有著資料結構的意識。

個人淺見，這門課有些小問題：

1.學生對於程式碼運作的方式理解不深

大多數的同學（包含我），都寫完final了，但是還是不會寫makefile，對於#include也只是很模糊的理解，也就是說，我們還是沒有自己從頭寫出一個中型程式的能力。我也覺得，如果我們修了這門課，就可以獨立完成一個中型的程式，那我會更愛上這個門課、以及這個領域，那麼就算對於比較高深的資料結構理解不深，我也很樂意去查資料自己學習。

總之，建議教授可以多花點時間介紹makefile跟#include，並且「讓同學自己寫一次」（makefile跟include教授上課都有講，但是沒有真的寫一次，大多數的人都還是不會）。建議在課程規劃上，可以把部分的makefile跟include留給同學寫，讓我們思考程式運作的方式，可以讓同學對於程式碼有更佳的理解。

但是如果再增加以上作業內容，loading空怕會太重，因此以下兩點可以參考後面兩點。

2.部分比較深入的資料結構，可以花少一點時間介紹

像是Red-Black tree、Max-min heap、Fibonacci heap等等較深入的資料結構，我覺得老師可以不用花這麼多時間去介紹，因為HW沒有用到，沒有自己實作一次，也很難真正理解。我覺得老師只要讓我們資料有這些資料結構就夠了（不是說不重要，而是上面第一點更重要）。

假設未來我在做研究，需要一個很快的Heap，我也不可能因為當初修資結時老師有介紹Fibonacci heap，就直接打出一個Fibonacci heap來用。我還是會上網看看各種heap有什麼優缺點，選一個適合的，理解它、實作它。

總之我覺得，如果想要讓學生多學到我上面第一點提到的，這部分的授課內容可以稍做刪減。

3.部分作業內容可以刪減

我覺得部分作業中有些內容，只是讓我們花時間打、花時間debug，卻並沒有學到太多東西，可以稍做刪減，讓同學練習我上面提到的第一點。

總結：
我覺得「講課內容上」可以刪減的部分有：

- 一大堆進階的tree
- 一些Graph的Terminologies和Properties像是Clique、Cutset 、Bipartite graph、Plannar graph、Clique number、Chromatic number 、Clique cover number 、Stability number、Perfect Graph等等。
- 進階的heap像是Binomial Heap 、Fibonacci Heap。

總之我覺得，作業沒有實作到的，就稍微帶過就好。

「作業上」可以刪減的內容有：

- HW2的各種功能可以砍半，留比較難的像是Up arrow之類的就好。
- HW3的功能也可以砍半，留比較難的像是DOfile、HIStory等等。
- HW6的CIRGate -Fanout、CIRWrite。

我覺得有一些TODO只是在用勞力換分數，學到的東西沒有很多，可以考慮把他們刪掉（或是改成不是TODO）。

總之，我覺得教授可以刪掉一些太深的講課內容，多介紹程式的架構，並且把一些比較沒有「營養價值」的TODO改掉，讓同學自己寫幾次makefile，還有自己include。

心得

有人說，修完資結就知道自己適不適合走CS了。大一時由於貪玩，計程沒有好好學，coding能力一直很爛，因此抱持著挑戰與探索的心態來修資結。

一整個覺期下來，說崩潰也不是非常崩潰，但是真的花了不少時間。從HW2到HW6，每次作業剛公佈時，都像是一座難以征服的大山，但當我開始動工，認真地去瞭解問題時，往往會發現其實沒有想像中的困難。一次次的作業雖然很讓人頭大，但是我覺得我寫作業時的心情是雀躍的，甚至會期待下一次的作業，因為我知道，無論現在再怎麼崩

潰，無論花多少時間，兩個禮拜後，我一定還是會把它完成，而到時，我的實力將與現在完全不同。

這門課讓我覺到最多的，不是資料結構，不是程式設計，而是面對龐大程式碼的無畏。這門課讓我學到，沒有de不出來的bug，沒有打不出來的程式，只有不肯花時間的自己。以前大一時，打程式遇到瓶頸，就會認為自己不可能解決，想要放棄，但是修了資結之後，無論未來面對幾萬行的程式碼，或者在難找的bug，我相信我都能調整心態，慢慢將它完成。

資料結構與程式設計，這無疑是我大學以來學到最多東西的一門課，很感謝教授與助教，讓我從coding渣渣，變得有能力自己去找方法解決coding上的問題，也讓我對CS產生好感，未來打算更深入發展。