FRAIG Report

B04901060 黃文璁

Contact: vichuang1997@gmail.com

1. **實作和優化sweep、opt、strash 的效能和可讀性**

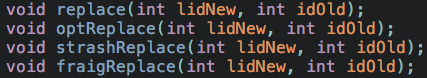
這幾個函數因為多數情況下都只是從PO開始遍歷整個電路，所以能夠優化的演算法部分比較少。

不過值得注意的是FRAIG中常常會需要在ID和Literal ID之間做轉換，也就是常常會有乘以2、除以2、模2等動作，這部分則可以藉由bitwise operation進行優化。關於利用bitwise operation進行優化請參考後文「其它優化細節」部分。

雖然在效率方面能夠優化的比較少，但是在可讀性方面可以進行優化。

此外由於這幾個函數常會有移除、取代Gate的動作，故將這些功能包裝成函數，可以有效增加程式碼的可讀性，關於這些函數我實作了下列輔助函數：

* 下列函數負責將gate[idOld] 取代成 lidNew。



* 由於replace動作會需要將原Gate（gate[idOld]）的FanIn和FanOut中和原Gate有關的部分進行修正，故還有以下輔助函數。

replaceIn的功能是修正原Gate的FanOut，其FanIn中和idOld有關的部分

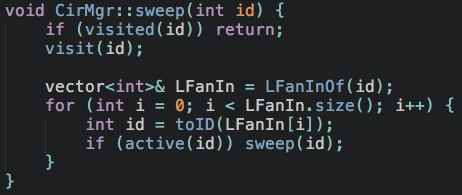
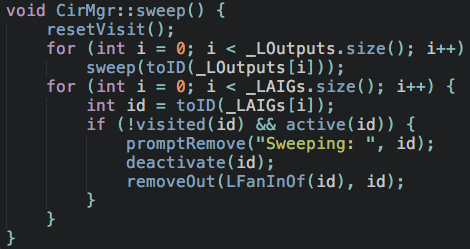
removeOut則是從原Gate的FanIn，其FanOut中移除和idOld有關的部分



* 此外針對被移除的Gate，我使用一個陣列 \_active來記錄這個Gate是不是還有功能，而不先將其 delete，並加上下列介面：

有了這些輔助函數，程式碼的可讀性也隨之增加。

舉CirMgr::sweep() 為例（左為sweep介面，右為進行DFS的工作函數）：



開頭有L的陣列存的是Literal ID。

有了這些輔助函數，應該不難看出左邊sweep() 函數程式碼中的意義。

至於還沒介紹過的resetVisit()、visited()、visit()、toID() 函數可以參考後文「其它優化細節」部分。

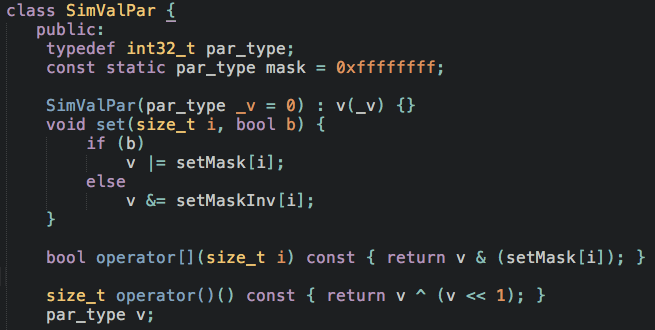
1. **優化simulation**
2. **改為使用Parallel-pattern Simulation**

原先我使用的方法不是Parallel-pattern，原因也很簡單，因為Parallel-pattern寫起來難度較高，為了求正確性就決定先不寫Parallel-pattern Simulation。不過目前的寫法已經改寫成Parallel-pattern Simulation，舊的程式碼剛好可以用來比較Non-parallel和parallel的效能差異。

所謂Parallel-pattern其實概念不算複雜，就是一次做整個數字（32bit）的位元運算，可能會比做32次bool的位元運算來得快，此外還能大幅減少重複執行相同的程式碼，在編譯出來的機器碼中應該可以看出差異。

實作Parallel-pattern時，首先自訂一個Parallel-pattern使用的class SimVal，目的是取代非Parallel-pattern模擬時使用的單一bool。這個class的功能像是一個32bit的vector<bool> 可以進行operator[] 操作，讀出該位元的值，運用在simulation上可以達到一次計算32次模擬。

實際上此class底層操作的是一個int\_32t，利用int\_32t在各個機器上都是32個bit的原理，可以將這個 class當做大小為32bit的vector<bool> 使用，利用一些簡單的bitwise operation。實作如下：



事實上STL中的vector<bool> 也不同於一般vector，是特別經過記憶體優化過的，但其功能比較複雜，底層實作的複雜性可能也較高，可能導致效能低於自訂的SimVal。不過實際上還是需要測試後才能看出差異。

至於實作中使用setMask和setMaskInv的原因可以參考後文「其它優化細節」的bitwise operation部分。

1. **優化尋找FEC Pair的時間**

這個部分也和上一節相同，一開始使用寫起來較簡單的方法，做法是：先模擬出所有gate所有輸入pattern的結果，存到一個vector<vector<bool>> val 中，其中val[id][i] 代表的是gate[id] 在第i筆輸入的模擬結果，最後再利用HashMap<vector<bool>, int> FECMap，其中FECMap[ val[id] ] 代表的是和gate[id] 模擬值相同的gate。

可以發現這個做法在時間和空間上都有很多缺點，首先這個做法需要先將所有gate的所有模擬出來的值存起來，佔用了很多空間，其次雖然C++ 的STL中內建有vector<bool> 的Hash function，但是以整個vector<bool> 下去做Hash相對來說又更慢了點。

簡化過的步驟如下：

1. 將檔案中所有輸入值存到一個vector<vector<bool>> 中。
2. 利用輸入模擬所有Gate的值
3. 模擬後可以得到針對「每個Gate、每個Input」的結果
4. 由每個Gate的結果（為一個vector<bool>）

利用HashMap<vector<bool>> 建立對應關係，找出是否有相同的值。

1. 將模擬值相同的Gate合併為一個FEC Group。

改寫成Parallel-pattern後，尋找FEC Pair的步驟也跟著改寫，這裡我們不再用HashMap直接建立「所有模擬值－> Gate」的對應關係，改為使用linked list來建立一個動態的FEC Group分群關係，將這個linked list稱為\_FECGroupList。

\_FECGroupList的實作上直接使用STL中的std::list，使用起來也比自己重新打造一個linked list來得簡單。\_FECGroupList宣告在class CirMgr中：



至於為什麼是list<list<int>> 的原因可參考下列步驟：

整個CirSim的步驟大致上可以寫成：

1. *讀入32行檔案，若讀到檔案尾端，則離開迴圈。*
2. *將讀入的字串轉換成PI的SimVal。*
3. *從每個PO進行DFS模擬。*
4. ***根據得到的模擬值更新FEC Group***
5. *若還有讀入的字串尚未轉換（也就是行數非32的倍數）*

*則再進行一次步驟2到4*

如此一來就不用將「所有gate的所有模擬值」存下來，同時只要存有32個pattern的模擬值即可，對pattern數量很多的檔案來說，記憶體方面的優化是非常明顯的。不過能這麼做的原因和更新FEC Group的方法有關，更新FEC Group這個動作比較複雜，所以提出來另外說明。

1. 更新FEC Group

前面提到 \_FECGroupList是一個list<list<int>>，利用list的原因是因為list有容易插入、移除元素的特性，對電路模擬這種順序較不重要的應用上十分有用。更新FEC Group的方法概念上也不算複雜，只不過實作上就比較麻煩些。詳細作法請參考下頁。

*// 將所有gate加入第0個Group（也就是 \_FECGroupList.front()）*

***for\_each*** *Gate* ***in*** *GateList*

*FECGroupList.front().push\_back(Gate)*

*// 根據SimVal將Gate分群、更新*

***for\_each*** *FECGroup* ***in*** *FECGroupList*

***if*** *sizeof(FECGroup)==1*

*remove(FECGroup)*

***else***

*pivot🡨0*

*firstNewGroup🡨FECGroupList.end()*

*isFirstNew🡨****true***

***for\_each*** *Gate* ***in*** *FECGroup*

***if*** *Gate is first element of FECGroup*

*pivot🡨SimVal[Gate]*

***elif*** *Gate.isNotInverted* ***and*** *SimVal[Gate]==~pivot*

*Invert(Gate)*

***elif*** *Gate.isInverted* ***and*** *SimVal[Gate]==pivot*

*Invert(Gate)*

***else*** *// 模擬值不同，必須分到別群*

*found🡨****false***

***for\_each*** *Group* ***in*** *FECGroupList …………………………………(1)*

***if*** *SimVal[Gate]==SimVal[Group.front()]*

*found🡨****true***

*Group.push\_back(Gate)*

***break***

***if******not*** *found // 找不到相同的模擬值，需要新增群*

*FECGroupList.push\_back(newGroup)*

*newGroup.push\_back(Gate)*

***if*** *isFirstNew*

*firstNewGroup🡨FECGroupList.back()*

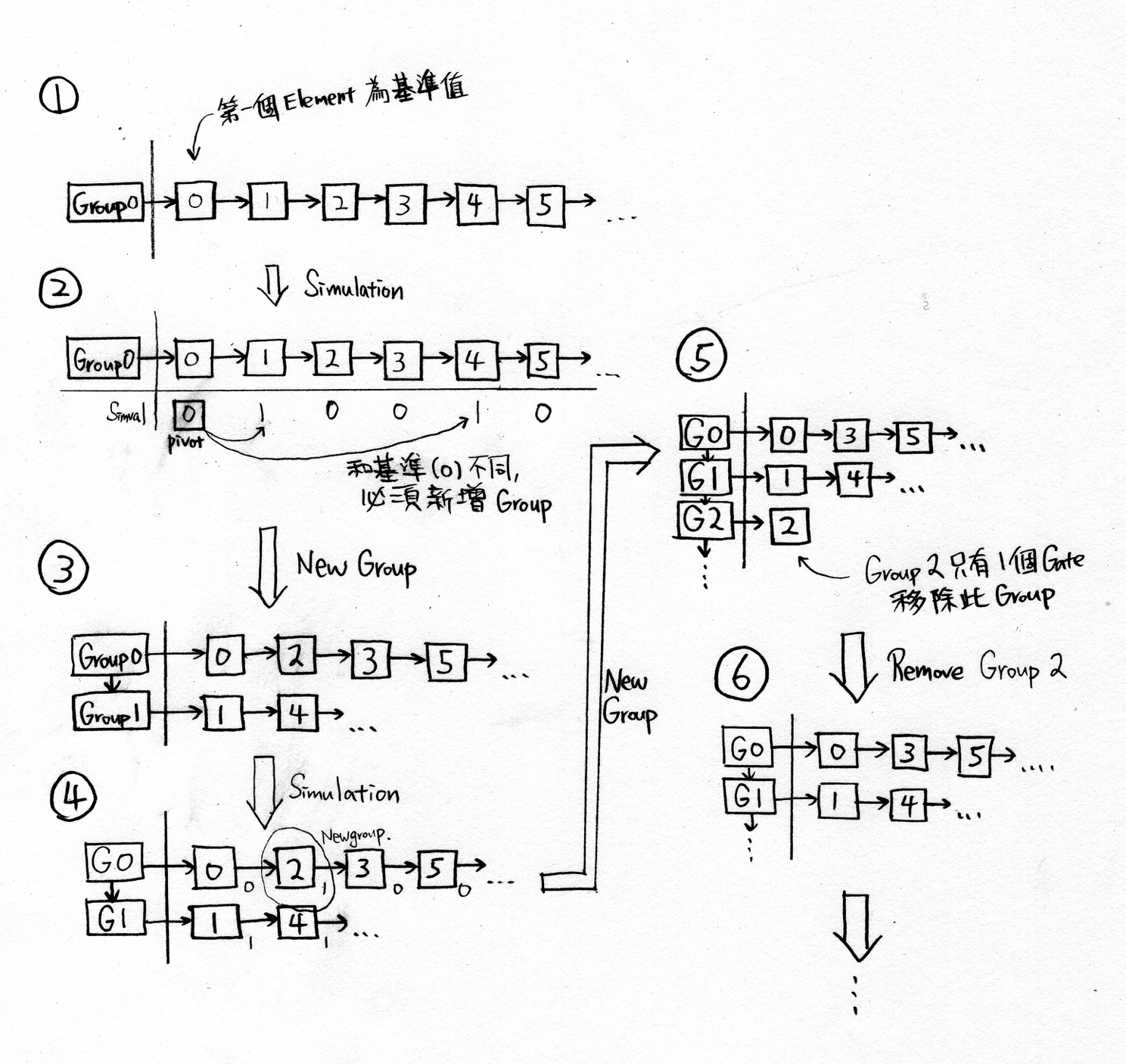
*isFirstNew🡨****false***

*// 從原群移除Gate*

*FECGroup.erase(Gate)*

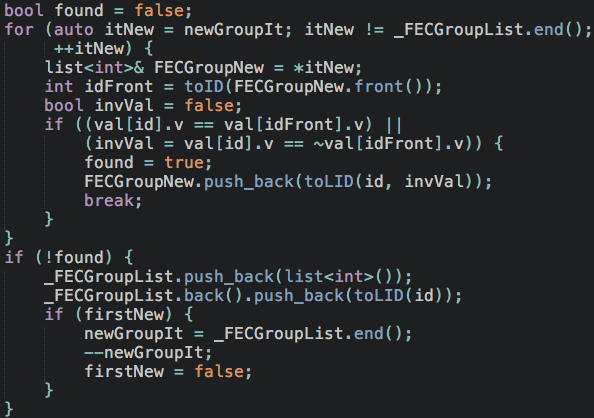
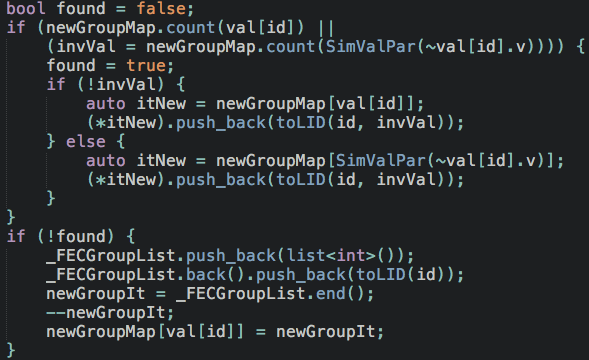
實際上 (1) 的部分不需要遍歷整個FECGroupList，而可從第一個新增的群（*firstNewGroup*）開始找起，因為新的群總是FECGroupList的最後幾個連續的Group。

用流程圖說明可參考下圖：



事實上不難發現，尋找相同SimVal是一個匹配的動作，說到匹配就會想到HashMap，所以利用HashMap可以避免上面的做法中尋找相同SimVal為O(n) 時間，而可以達到O(1) 的匹配查詢。故最後的做法中改用的是HashMap<SimVal, list<list<int>>::iterator> 來配合新增Group的動作，而不是利用for做線性的查找。

程式碼比較如下（左為原作法使用for、右為新作法使用HashMap）：

到這邊為止cirsim的效能已經可以稍微超過ref程式，例如在sim13.aag這個有超過八萬個AIG的電路中，ref執行cirsim -f pattern.13的時間約4.8秒，而我改用HashMap的程式執行時間約4.2秒。利用diff檢查兩個程式cirp -fec 的輸出也可以確認找到的FEC Pair是相同的，故在效能上確實有差別。

1. 優化FRAIG

在FRAIG中，不一定要對FEC Group中所有Gate都進行SAT Prove，經過測試可以用幾個簡單的原則來執行FRAIG，增加FRAIG的執行效率：

1. 跳過已經被merge的Gate
2. 一旦「連續失敗n次」，則停止繼續嘗試對這個Gate進行SAT Prove。

經過測試後，發現其實這個原則相當簡單，但是可以在效能和減少Gate數量之間取得平衡，重點是n的取值問題，這方面可以由實際實驗得出。

1. 其它優化細節
2. 利用bitwise operation避免除法（/）、模（%）運算、重複計算。

在電腦上要計算除法以及模運算相對其它運算要來得慢。但在這次的FRAIG中常常會需要進行ID的轉換。

比如說LID轉換成ID是LID/2，!ID轉換成LID是ID\*2+1等等。至於避免這些運算的方法也很簡單，只要利用位元（bitwise）運算操作數字即可，程式碼如下：



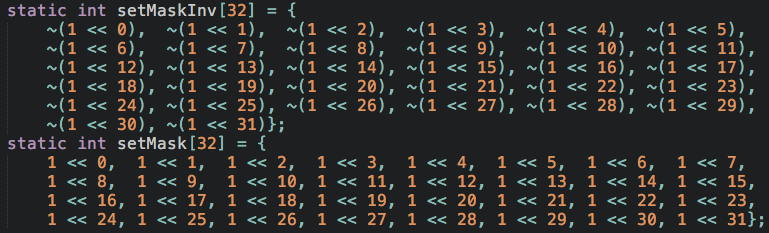
基本上就是利用 x<< 1 當做乘以2；x>> 1 當做除以 2 等方法來避免進行實際的除法運算。

此外先前提到class SimVal中使用的setMask和setMaskInv是用來避免重複計算位移運算的陣列，精神上可以想成是一種preprocessing的方式。

由於在operator[] 中每呼叫一次，若在不優化的情況下要得到該bit就要呼叫一次位移運算，如下：



但是實際上我們知道i的範圍在 [0,31]，故可以使用預處理，也就是如程式碼中的setMask陣列。另外在set(size\_t, bool) 函數中，將一個bit設為0需要的是 ~setMask，實際上也可以先預處理出另一個陣列setMaskInv，如程式碼所示：



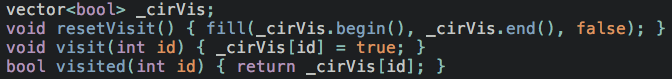
1. 避免重新宣告visit等陣列

在DFS的過程中，會用到vector<bool> visit陣列，大小和gate的數量相同。但是若每次DFS都要重新宣告一個visit陣列可能會降低程式的效能。

故在CirMgr中，宣告一個通用的陣列 \_cirVis，當需要DFS的時候呼叫resetVisit() 函數，將 \_cirVis的值重新變回false，如此可以避免不斷呼叫vector<bool> 的constructor/destructor，減少Memory allocation的次數。只要在CirRead後呼叫vector<bool>::resize() 一次即可。

將值重設為false的動作使用std::fill() 函數。

介面如下：



當然函數的命名上也可以增加可讀性。