國立臺灣師範大學
資訊工程研究所碩士論文

指導教授：黃冠寰 博士

儲存執行狀態來加速
可達性測試的研究

State Storing in Prefix-Based Replay to
Accelerate the Reachability Testing

研究生：　薛光耀　撰

中華民國 98 年

中文摘要

在軟體測試上，我們的 reachability testing[1]主要就是針對並行程式來作分析測試的，它最主要的目的就是要把所有同步事件的 interleaves 找出來以供使用者分析參考，針對這一方面的研究，有提高效能的，也有向外應用的研究，例如 web service、dynamic effective testing、多核心競爭的分析等等。

這項研究主要的目的就是改進 reachability testing 系統的效能，在提高 reachability testing 系統效能上的議題也蠻多人作研究。在現實生活中，我們所測試的並行程式軟體，一定會相當的龐大，在跑 reachability testing 會很耗時，有的時候也許會是五、六個小時，大者甚至一個禮拜左右，像這樣我們一定要提升這個系統的效能，對著此系統作假設、省略和一些演算法的改進。我的研究最主要是針對 reachability testing 裡的 prefix-based replay 作改進，保留目前執行的狀態，到下次執行 prefix-based replay 的時候，利用 restore state 的方法直接到目的地去執行 replay，使用這樣的方式去提高我們 reachability testing 的效能。目前對提高效能的方面已經有相當的成果了，在未來的工作是要跟目前 dynamic effective testing 作整合 ，也可以在裡面利用此方法，來作更進一步的提高效率的功能。

關鍵字：軟體測試、reachability testing、web service、dynamic effective testing、nondeterministic behavior、prefix-based replay。

ABSTRACT

In software testing, our reachability testing [1] for concurrent program is mainly for testing, which main purpose is to find out all the interleaves for user analysis of reference for research in this area, there are improve the performance, but also from outside the application of research, such as web service, dynamic effective testing, analysis of multi-core competition and so on.

The main purpose of this study is to improve the effectiveness of the system reachability testing, reachability testing system to improve performance also were the subject of further study. In real life, we have a concurrent software, and it will be quite large. Running reachability testing will be very time-consuming, and some time may be five or six hours, or even the largest of about a week like this. We enhance the effectiveness of the system, make any assumptions at this system, omitted, and some algorithm improvements. My research is mainly in the reachability testing for prefix-based replay for improvement. It would retain the current status of execution to the next prefix-based replay ,it would jump directly to the destination to do it, and use this method to accelerate reachability testing. At present, this method has been in the fine performance of the experimental results.In future work, it would be with the current dynamic effective testing for integration, it could also use this method to further improve the efficiency they provide.

Keywords: Software testing, reachability testing, web service, dynamic effective testing, nondeterministic behavior, prefix-based replay.

誌謝

當初因為工作上種種的不愉快，而加入到碩士班考試，也很順利的考上師大資訊工程學系，在找指導教授的一方面，因為學生對於資訊工程這方面的知識頗粗淺，所以當初非常的感謝黃冠寰教授願意指導學生這兩年，雖然學生常常惹教授生氣，但是教授還是很有耐心的把學生導向正確的研究方向，我的指導教授對每位學生的付出，有的時候讓我真的很感動萬分，在未來我也會把我的指導教授處事方式當作我學習的方向。

在研究的過程，遇到過很多的瓶頸，很多時後會有種想放棄的心態，甚至有種想逃避的感覺，還好有遇到實驗室裡的林哲生學長，就算私人的生活上、學業的問題上等等，哲生學長都是一個會開導學弟的好良師，在過程中一有問題就會去請教哲生學長，他也是不顧辛勞的幫學弟解決任何問題，還有蕭宇呈學長，假如跟程式有相關不熟的問題，他都會細心的教導學弟，所以在實驗室的環境是非常融合的。也感謝我同屆的同學笠德、招政、文賢、彥佑、李祺，大家互相的幫忙和包容，雖然我們當初沒有二年級的學長，但是我們有非常多的同學可以互相請教學習，所以在實驗室裡面的過程中，假如沒有教授、同學、學長、學弟等等的幫助，學生應該不會順利畢業的，所以非常的感謝大家。

在此也很感謝我的家人對我祝福和期待，我在台北的住宿方面的幫助，讓我可以順利撐過這兩年，雖然中途有點困難，也算是一種歷練，在此希望大家都可以健康和快樂。

薛光耀 誌於

國立台灣師範大學資訊工程研究所

民國九十八年七月

目錄

圖目錄

表目錄

State Storing in Prefix-Based Replay to Accelerate the Reachability Testing

1. 簡介

近年來軟體的功能越來越多元化，軟體的複雜性，卻不是一般人可以去設計的，往往工程師也是花一推時間去除錯，即使上市了一套軟體系統，也會不停的出現更新檔給使用者下載去更新，這樣的錯誤是很難預防的，往往也是因為隱藏的錯誤是很難去找出來，也許軟體執行一萬次、十萬次才會出錯一次，但是很多的行業對於這樣的錯誤是不允許的，因為對他們而言會損失甚重，例如：太空發射系統、提款機系統等等，所以這類型的錯誤只能預防萬一，沒有很好的方式去根除這類的問題，工程師人員也必須持續的追蹤使用者使用情形，了解使用者的需求等等之類的後續情況。所以這些錯誤的追蹤，將是工程師要去著手思考的，這就是為什麼要去作軟體測試的原因。

現在的軟體為了提升更好效能，都是以多程序的方式去實作，作多程序的軟體裡面，往往也隱藏各式各樣的錯誤，這些錯誤的找尋方法通常都會造成工程師一個很大的負擔，慢慢就研發出很多找尋這類型錯誤的方式，所以軟體測試也變成現在很重要研究的一門技術。我們系統是用 Java language 裡面的多執行緒技

術來模擬多程序的情況。

本論文的概述分為以下，章節 1 我會先介紹軟體測試過去幾年的發展以及目前正在進行的研究，章節 2 會去介紹我在研究 State Storing in Prefix-Based Replay to Accelerate the Reachability Testing 之前所參考的一些文獻，章節 3 就是這項研究的實作方式和一些特殊例子的解決，章節 4 就是一些重要例子的實驗數據和此最佳化後和原始的系統比較，章節 5 就是討論和未來要作的事情，及最後的 references。

1.1 Software testing

不論是軟體專案、工程專案或是其它各類的專案，在整個專案進行的過程中或是結案前，都存在著大大小小的測試。我們以軟體專案為例，從元件開發過程中的單元 測試(Unit Test)、到功能測試(Function Test)、然後是整合測試(Integration Test)、最後還有個使用者驗收測試(UAT, User Acceptance Test)等等，一堆的測試。這些測試最終目的就是為了讓整個系統能夠正常順利的運作，只要測試的項目涵蓋範圍夠廣夠完整，那麼就可以精確掌握各種可能的狀況，讓系統出錯機率降到最低。

Software testing 中有 white box testing 與 black box testing 兩大分類，它們主要的差異是在測試時的能見觀點，white box testing 的涵意是設計 test case 時是從軟體內部的觀點來設計，因此他可能使用內部的資料結構來設計 test case，而

black box testing 則是由軟體外部的觀點來設計 test case，這類的測試方式通常是經由程式的輸入與輸出來測驗系統功能性的滿足與否。軟體測試主要分為三大層級，分別為 Module(Unit) test, Integration test, System test，主要是伴隨軟體系統的發展過程而產生，然而隨著軟體專案的複雜程度，又可再細分為更多層級，每個不同階段的軟體開發都伴隨著特有的測試方法，我們將之解釋如下：

1. Module(Unit) test：Module test 主要的目的是找出此元件與他的介面規格描述的輸入輸出是否相符，此階段的測試是由系統開發者所執行，而非使用者。
2. Integration test：Integration test 將已經利用 Module test 測試過的元件來組合成更大的組件並加以測試他們的正確性，此階段也是由系統開發者主導，此階段的軟體測試方法為白盒子測試。
3. Function test：此類型的測試方法是用來測驗軟體程式與軟體規格是否相符，此類型的軟體測試方法是屬於黑盒子測試，因此測驗時必須仰賴前兩種測試方法來保證軟體組件間的正確性，此階段的軟體測試方法為白盒子測試。
4. System test：此類型的測試方法也是屬於黑盒子測試，他建立於測試已經完整的系統，驗證這個軟體系統是否符合軟體規格。
5. Acceptance test：是由使用者來主導測試，使用者經由比較軟體的執行方式與他們所制定的規格相符與否以決定測試成功或失敗，此類型的軟體測

試方法亦屬於黑盒子測試。

6. Installation test：此類的軟體測試是一個不常見的軟體測試，因為他主要目的不是測試軟體是否符合規格，而是用來測試軟體安裝介面的問題。

不過，既然測試那麼重要，但是軟體界一項常見的問題是，軟體產品往往沒有經過足夠的測試。原因在於，不管是人力的成本或是其它相關資源的成本上，測試的成本非常高。雖然有許多的測試工具軟體可以輔助我們做測試的工作，但是要測哪些項目、怎麼測以及有些操作還是得靠人力才行。程式設計師自行測試 會很自然地用他們認為最佳的方式來操作這套系統，這會產生許多盲點，導致成品出現許多錯誤。

本實驗室最主要是針對測試並行程式軟體的研究為主軸，在並行程式裡面就會有共享的資源，透過作業系統對這些並行程式的工作安排，有可能會產生一些不一樣的結果，即對共享資源有著不一樣順序的存取而導致不同的結果我們稱之為 race condition[2]，我們的系統最主要就是對這些會對共享資源存取的事件裡，針對這些事件找出一些可能的交錯組合行徑，且一一的把他找出來再供程式設計師進行分析。

1.1.1 Static analysis

對解決並行程式的 nondeterministic behavior，一些研究人員已經發展針對在並行程式內的異常種類檢測之靜態分析技術[10,11,12]。並不會對程式進行執行，

而是針對程式碼、演算法作一些驗證的動作。 靜態分析常被用來在檢測並行程式裡的 synchronization 和資料用法的錯誤，synchronization 錯誤包括 deadlock 和 “wait forever”， 資料用法的錯誤包括一般性 sequential 資料用法錯誤，例如讀取未初始化的變數，以及平行的資料用法錯誤。

1.1.2 Dynamic testing

動態分析最主要就是程式是有實際在跑的，然後在針對他跑完的結果作分析。針對執行程式的順序不同有著不同的變化，或者針對其值的不同也會造成不同的結果。Dynamic testing 最主要的例子就是在我們實驗室所研究的 reachability testing[1]，此系統可以驗證 non-deterministic 的情況，可以找出每一條 interleave 且都有去實際執行過。

1.2 Approaches to Testing Concurrent Software

測試並行軟體最主要的方法，分為 nondeterministic testing 和 deterministic testing 和他的一些相關研究，這些詳細的資料可以參考[3]。再介紹 reachability testing 之前，我們要了解一些名詞定義，一個 synchronization event 可以為在並行程式裡面會對共享資料作存取的動作稱之，大致上有 message passing(針對 channel 作使用)、semphore(號誌控制的使用)、access to shared memory 和 database transaction(create 和 query 等命令)等等，我們的 reachability testing 主要是使用

access to shared memory 裡面的 read/write 操作來代表它是一個 synchronization event。一個 synchronization sequence (或稱 SYN-sequence) *S* 通常可以說為是一些 synchronization events 交錯組合而成的。

1.2.1 Nondeterministic Testing

Nondeterministic Testing 也是最常見的一種測試，是針對一組輸入對此並行程式一直作重覆的執行，讓它有可能產生不同的結果出來，根據這些結果來判斷此程式是否正確與否，如圖 1 所示。

但是此方式最主要缺點的：

(1) 相同的 SYN-sequence 很有可能會一直執行到的。

(2) 有些 SYN-sequences 很可能永遠沒辦法執行到的。

所以 nondeterministic testing 是永遠沒辦法保證達成 exhaustive testing 的。

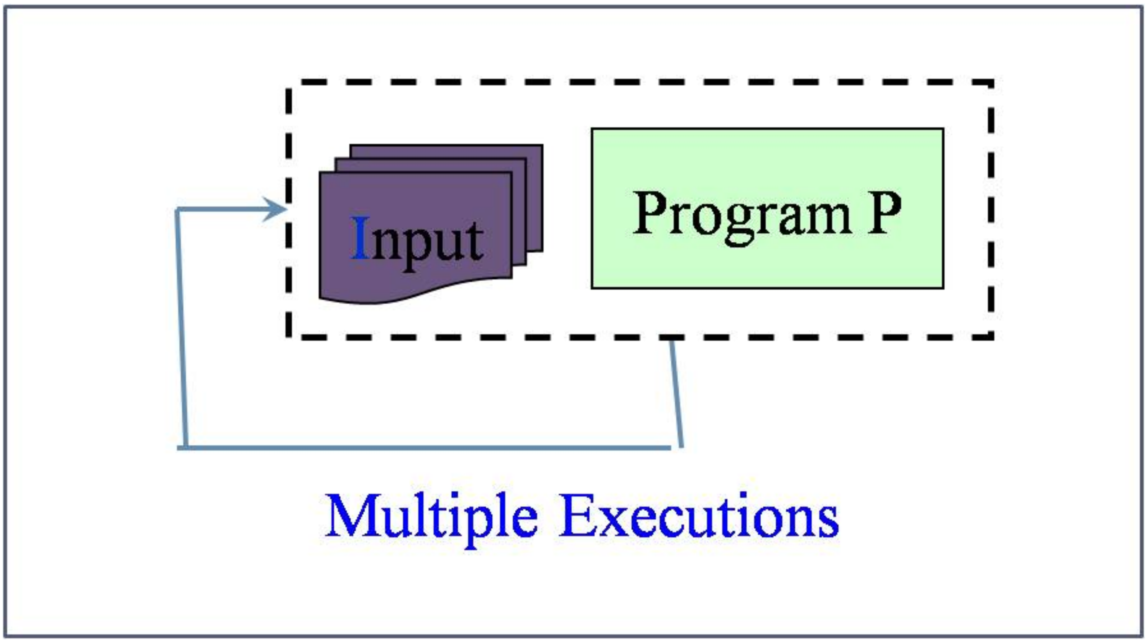

圖.1 Nondeterministic Testing 架構圖

1.2.2 Deterministic Testing

跟先前的 nondeterministic testing 不同之處是會在執行的時候多一項 SYN-sequence *S* 輸入，針對這個 SYN-sequence *S* 作 replay 的動作，這項強迫的執行用來確定 *S* 和輸入是不是符合這項程式 *P* 的，如圖 2 所示。

這項最主要的缺點就是，再取決這個在於選擇那一個 feasible SYN-sequence 和輸入是非常困難的，有些相關問題解決方法的詳細論文在[6][7][8][9]，所以在之後我的指導教授才會研發 reachability testing 來解決 nondeterministic testing 和 deterministic testing 的缺點。

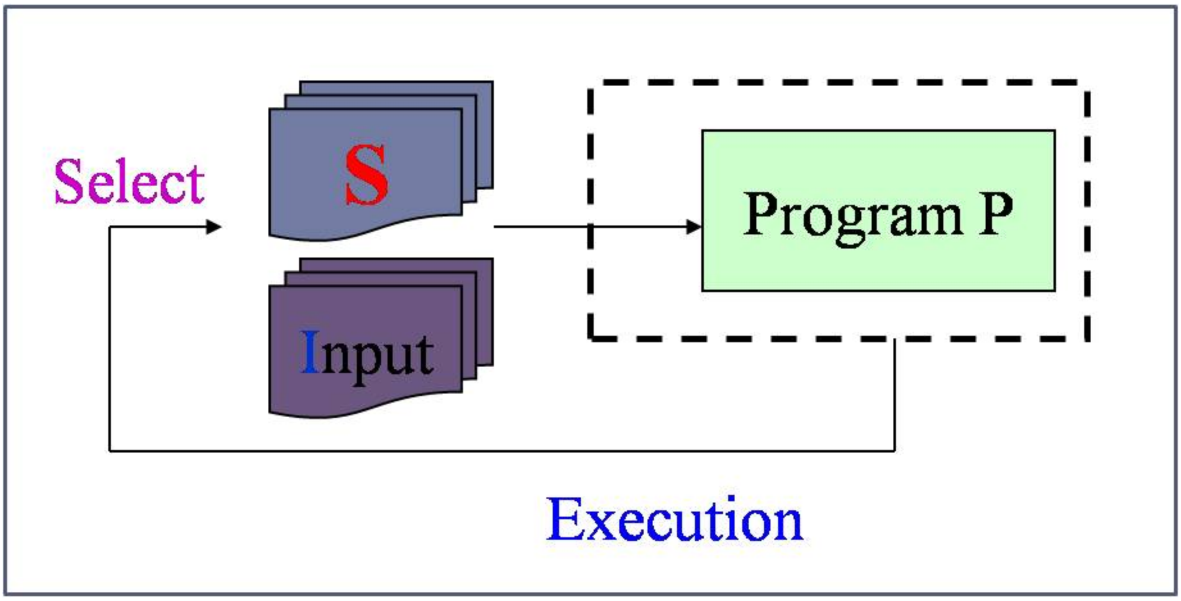

圖.2 Deterministic Testing 架構圖

1.2.3 Reachability Testing

將執行過程中所有 synchronization events 的執行順序記錄下來是進行並行程式測試的第一步。前面提到 synchronization events 之間彼此會競爭特定的執行順序，並進而影響程式執行的結果；因此如果想要在反覆的執行過程中得到相同的執行結果，顯然必須記錄前次執行的 synchronization events 順序並依此控制之後的執行。在並行架構中，如果想要達到控制程式執行、重現前次執行結果的目的，僅需要記錄 synchronization events 間 partially order 的關係，而一次完整執行中所產生的記錄被稱作為 SYN-sequence。Deterministic testing 的概念即是：將 SYN-sequence 作為程式輸入的一部分，利用特定的協定將並行程式中所有的 synchronization events 作包裝，使 synchronization events 能按照輸入的 SYN-sequence 完成執行。這種概念雖然適當地處理了並行程式非決定性行為的

問題，但仍存在另一個盲點：在每次重複的測試過程中，執行的 SYN-sequence 都與先前相同，而 synchronization events 可能的排列情形卻往往總數龐大，因此倚賴 deterministic testing 並無法完整的進行測試。相對於 deterministic testing 指定程式執行時的 SYN-sequence，nondeterministic testing 著眼於讓程式在不受干預的情形下被反覆地執行，因此每次執行的 SYN-sequence 都可能不同。在傳統測試循序程式時，以反覆執行為主體的測試是相當常見的；但在測試並行程式時，由於不確定性行為使得反覆執行的過程中，往往執行的僅是所有可能 SYN-sequence 的一小部分；根據實驗數據[1]顯示：對於一個擁有 420 個相異 SYN-sequences 的並行程式，經過 300 次的重複執行，期間被執行的相異 SYN-sequences 個數僅僅只有 7 個；即使重複執行的次數提高到 1000 次，卻也只有 10 個相異的 SYN-sequences 被執行。從以上的介紹發現，無論採用 deterministic testing 或 nondeterministic testing，對於測試並行程式都有其不足之處。Reachability testing 是一種用來測試並行程式非決定性行為的方法，架構上揉合了 nondeterministic testing 與 deterministic testing 兩者的概念；透過 reachability testing，系統開發人員可以確認並行程式執行的正確與否。

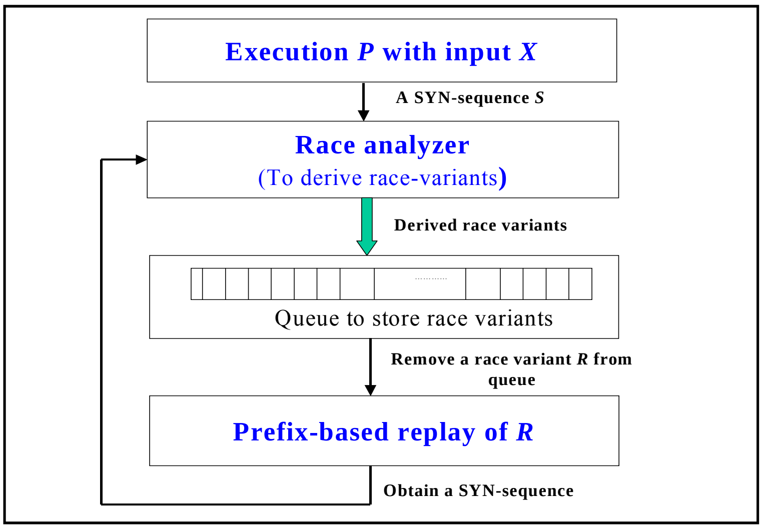

圖.3 Reachability testing 架構圖

Reachability testing 整個測試過程如圖 3 可粗分成兩個階段：monitor phase 與 replay phase。在 monitor phase 中，受測程式的執行如同 nondeterministic testing 一般，是完全不受控制的，系統的排程可自由地組合任何可能 SYN-sequence 並完成執行。對於 reachability testing 整體架構而言，monitor phase 所需要作的僅是將受測程式執行的 SYN-sequence 記錄下來，供作後續的分析與執行需要。顧名思義，replay phase 的目的即在重現前次的執行，它應用 deterministic testing 的概念，受測程式在執行時需要指定 synchronization events 的 partially order 關係，藉由 reachability testing 提供的協定，將所有 synchronization events 按指定的次序執行。很明顯地，單純地實作 monitor phase 與 replay phase 並不能解決 nondeterministic testing 與 deterministic testing 的不足，reachability testing 因此導入了 race analyzer 的機制，將 monitor phase 與 replay phase 作完美的結合。Race analyzer 的主要工作在於從 monitor phase 記錄的 SYN-sequence 中，分析出其他

不同於此 SYN-sequence 但又可能出現的 partially order 關係，稱作 race variant。換言之，race analyzer 可藉由更動 SYN-sequence 中部分 synchronization events 的 partially order 關係用以產生 race variants；由於在被更動點之後的 synchronization events 已無法保證其執行的必然性（例如：經由不同的執行順序，執行過程中可能會產生不同的變數值，而這些變數值可能會作為程式中 conditional branch 的依據，並決定部分程式碼的執行與否）；相對於 SYN-sequence，race variants 將不再代表某次完整的執行順序，而是某個執行順序中最前面的一部分；當要進行受測程式的整個執行時，必須將 replay phase 作一適當的調整。前面提到的 replay phase 僅是應用 deterministic testing 的概念，但由於指定的 partially order 關係──race variant 是一個不完整的執行過程，reachability testing 因此提出 prefix-based replay 的概念，透過在 race variant 中加入一個特殊標記的方式，使 replay phase 時得知 race analyzer 更動的 partially order 的位置，當 replay 的過程中遇到前述的特殊標記，reachability testing 立刻切換到 monitor 模式，改由系統排程自由地產生可能的執行順序，執行受測程式中之後的部分，並將此記錄下來形成另一個不同的 SYN-sequence 再交由 race analyzer 分析、產生其它新的 race variants。由此可見，reachability testing 利用這樣反覆的動作，一開始將第一次執行得到的 SYN-sequence 從頭開始對每一個 partially order 關係作更動，產生一些 race variants，並標示更動的位置；在後續的 race analyzer 進行過程中，特殊標記前的 partially order 將不再被更動，因此隨著 reachability testing 中每一輪的執行，標記

的位置會逐步地向受測程式結尾處靠近。當特殊標記到達程式結尾，race analyzer 將不再產生任何新的 race variants，整個 reachability testing 過程也在執行完所有的 race variants 後結束。最後，為了達到完整測試的目的，race analyzer 所採用的演算法必須能保證所有受測程式可能產生的 race variants 都會在測試過程中產生，另外由於效率上的考量，演算法同時也要儘量避免產生重複的 race variants。

1.2.4 Model Checking

Model checking 是現在軟體測試中蠻流行的一種工具，主要的方式就是要先將測試的原始碼，轉換成他們的模型去運作來進行分析。這方面的研究也相當多，而且種類也很多，一般來說我們最常看見的 model checking 有 SPIN[13]、NuSMV[14].、JPF[15]等等。Model checking 最主要的缺陷且一值無法很有效地解決的問題就是狀態爆炸的問題，只能儘可能的去簡化模組來解決這類的問題。Model checking 是利用數學的方式去表示的，和有限狀態機再搭配邏輯運用，例如 LTL[16]與 CTL[16]，來作驗證的動作。

1.2.5 Dynamic Effective testing(DET)

雖然 reachability testing 已實證能完整地測試並行程式的非決定性行為，並在效能上取得長足的進步，但其基礎架構上卻存在一嚴重的不足。在前面提到，reachability testing 在 monitor 的過程中並不對受測程式的執行進行任何干預，因

此當程式中出現無窮迴圈或忙碌等待迴圈(busy-waiting loops)時，reachability testing 很容易因為執行無法順利結束，導致 prefix-based replay 的過程中失敗。

"a" is a shared variable.

Process 0	Process 1
... $S_{0,0}$ **a = 1 ;** $S_{0,1}$ **DO { ... }** $S_{0,2}$ **WHILE (a==0)** ...	... $S_{1,0}$ **a = 0 ;** ...

圖.4 造成 reachability testing 問題的受測程式之一

圖 4 是一個由兩個程序 process 0 與 process 1 組成的並行程式，在 process 0 中有三行程式碼：S0,0, S0,1 與 S0,2，而 process 1 中由一行程式碼 S1,0 組成。這邊很明顯地可以看到：程式的執行將會在 S0,1 與 S0,2 不斷地來回，{S0,0, S1,0, S0,1, S0,2, …}是其中一個可能產生的執行順序。就如前面所敘述 reachability testing 的架構，執行時產生的順序將會被記錄下來，提供後續的 race analyzer 作分析的動作；但在以{S0,0, S1,0, S0,1, S0,2, …}為首的執行順序時，程式將會進入無窮的等待迴圈之中，不斷地記錄的執行順序，最終耗盡系統所有可用的記憶體空間。

"a" is a shared variable and its initial value is 0.

Process 0	Process 1
...	...
$S_{0,0}$ **DO { ... }**	$S_{1,0}$ **a = 1 ;**
$S_{0,1}$ **WHILE(a==0)**	...
...	

圖.5 造成 reachability testing 問題的受測程式之二

另外一個潛在的問題是：有時程式的執行並沒有進入無窮迴圈之中。以圖5為例：受測程式最快可能只執行三行的程式碼{S1,0, S0,0, S0,1}就結束，也有可能多在迴圈裡執行一次，得到{S0,0, S0,1, S1,0, S0,0, S0,1}這樣的執行順序。很明顯地，process 0 中的迴圈S0,0, S0,1 必須不斷地執行忙碌等待迴圈直到process 1 中的S1,0 獲得執行；有別於無窮迴圈圖 6圖 的例子最終必定會因為操作系統排程給process 1 而執行結束。但是在reachability testing的架構之下，race analyzer為達到完整測試的目的，會將所有程式可能的執行順序全部產生，因此無論是執行迴圈S0,0, S0,1 一次、兩次，甚至是無限多次的執行順序，都會因為race analyzer視作相異的race variants而逐一地付予執行。我們可以把這些產生的race variants表達成{(S0,0, S0,1,)i S1,0, S0,0, S0,1}的形式，其中i可以是大於或等於零的任意整數，而這些race variants將使得存放race variants的佇列永遠無法被清空，整個reachability testing將因而沒有結束的時候。

前面以一個最簡單的例子說明非無窮迴圈可能的問題，其中反覆執行的部分

S0,0, S0,1 都在同一個程序中，圖 6 是另一個較複雜的例子，反覆執行的部份將橫跨兩個不同的程序。在圖 6 造成中最短的可能執行序列是{S1,0, S0,0, S0,1, S1,1}，和前述例子類似，這個例子中所有可能的執行序列可以表達成：{(S0,0, S1,0, S0,1, S1,1,)i S1,0, S0,0, S0,1, S1,1}，其中 i 是大於或等於零的任意整數；可能重複執行的部分為 S0,0, S1,0, S0,1, S1,1，分別屬於 process 0 與 process 1。

"a" is a shared variable.

Process 0	**Process 1**
...	...
$S_{0,0}$ **DO { a = 1;}**	$S_{1,0}$ **DO { a = 0;}**
$S_{0,1}$ **WHILE(a==0)**	$S_{1,1}$ **WHILE(a==0)**
...	...

圖.6 造成 reachability testing 問題的受測程式之三

Multi-core 系統的測試驗證有以下困難點： Shared variables 的存取造成 race condition 除了、具迴圈，還有 Task priority，分成以下三個部分來進行研究：

處理迴圈程式可能造成的問題，在 prefix-based replay 時要能將已陷入無窮迴圈的受測程式強制結束，並回報可能造成無窮迴圈的部分程式碼與特定的執行順序。其次，在 race analyzer 的部分應發展能處理忙碌等待迴圈的演算法，避免產出過多的 race variants，導致無窮的測試。

1.2.6 State Storing in Prefix-Based Replay

在原來的 reachability testing 系統裡面，主要有兩個部分來達成我們的 reachability testing 系統的功能：

i. prefix-based replay

ii. race analzer

在每次所推出來的 race variant，在送去作 prefix-based replay，在一開始作 replay phase 的動作（如圖 7)，往往都會從頭開始執行，但是這樣子的方式是很沒有效率的，因為在作 replay phase 的時候，執行此 SYN-sequence 會利用 entry protocol 和 exit protocol 去控制 SYN-sequence 的執行順序，讓並行程式裡的 replay synchronization events 到達@符號（@（界限）：由 race analyzer 在推導 race variants 所定義出來的，在作 prefix-based replay 的時候，此點就是各程序所依據的 replay phase 終點線)，要達成這個目的，就需要在 entry protocol 和 exit protocol 的程式碼裡面，利用到 semaphore 的操作去達成控制，此 context switch 負擔是很重的，所以才會造成 reachability testing 效能上的不足，之後也會對提升 reachability testing 效能上研究著手。

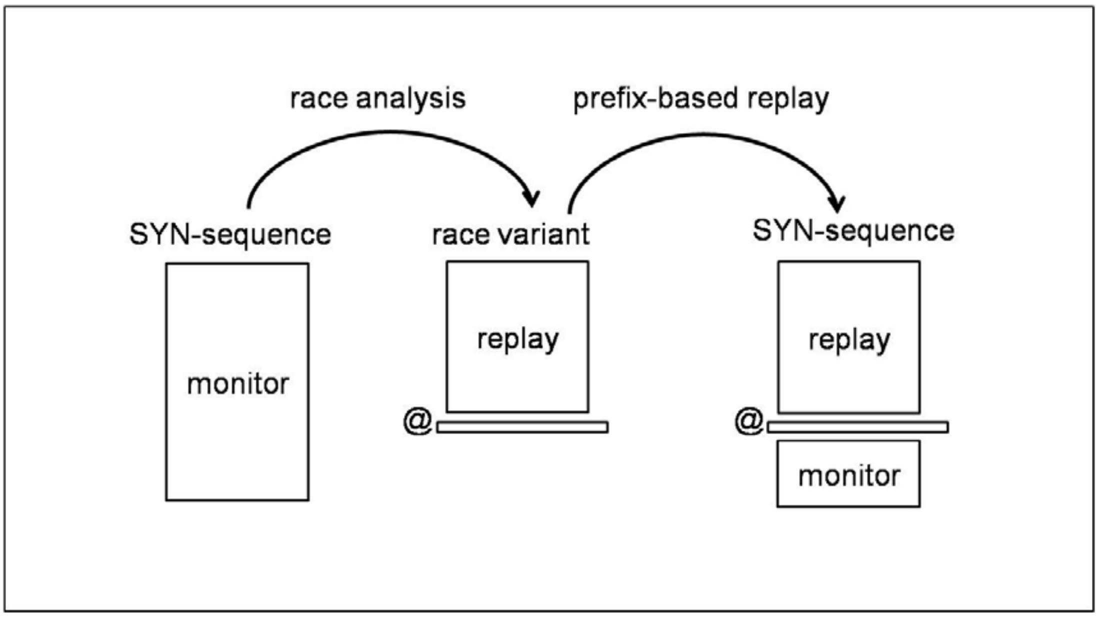

圖.7 SYN-sequences 在 race analyzer 和 prefix-based replay 的運作

經過教授的指導,學生才會去研究 State Storing in Prefix-Based Replay 方法，希望作到忽略某些要作 replay phase 的 synchronization events，這樣子每次在作 prefix-based replay 的時候,所謂在 replay phase 得從頭執行的方式就會有所變更，而改成跳至程式的某一點去執行即可，也可以說是忽略掉某些 replay mode 的 synchronization events(如圖 8)，所以在這裡就必須要用到 state 的觀念來達成，此 state 又有兩種方式來表示動作，restore state 和 store state，這個就是最主要控制整個流程的方法，用來達成跳躍某些程式碼而到達目的地的方式；最後精神所在就是要在甚麼時間點要去作 store state 和 restore state 的動作跟一些狀況的除錯，在之後的作法裡會再加以詳細介紹。

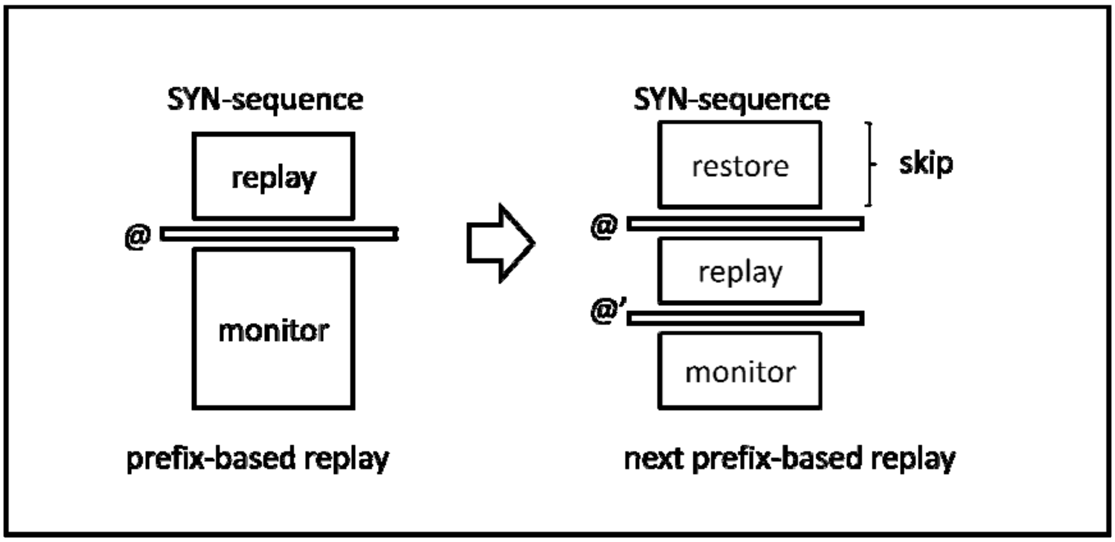

圖.8　在作 prefix-Based Replay 所省略的 synchronization events 的概略圖

2. 文獻探討

再作這項研究之前，我們不外乎一定要對 reachability testing[1]先了解其內容，在針對他之後的應用跟改進，是由我們黃冠寰　博士在 2005 年所發表的那篇 Technology for Tesing Nondeterministic Client／Server Database Applications[4]論文可以探討，reachability testing 目前也正向 dynamic effective testing 的方向去做研究，之後我在碩士生崖裡面閱讀 Testing Concurrent Programs using Value Schedules[17]的論文，透過這篇了解別人的如何去處理一般在 branch 前端是如先暫存一些變數數值，以供系統去作分析的，才引進我們 state 的概念，於是就開始著手這個題目，利用 restore state 的觀念來提高 reachability testing 的效能。

2.1 Reachability Testing

此系統是由黃冠寰　博士率先在 1995 年的時候發表出來的[1]，主要的功能是測試並行程式而想出來的，因為在執行並行程式裡面可能會有很多 synchronization events 的 interleave，在某些 interleave 的 pattern 下，他執行出來的結果可能會有錯，我們最主要就是要把這些 interleave 給找出來，然後去測試再看哪些是錯誤的狀況發生供使用者了解。Reachability testing 主要是結合 nondeterministic testing 和 deterministic testing 兩種方式去實作的。

這是本系統的第一篇論文，所以當初在 race analyzer 裡面，先提出了利用建構 RV-diagram 的方式，並以 shared-memory 與 semaphore 兩種存取機制為例子。

RV-diagram 是一個 n-ary 的樹狀結構，其中 n 代表整個受測程式中並行程序的個數。由 RV-diagram 的 root 到樹狀結構中任一個節點的路徑都分別代表了此受測程式中一部分 synchronization events 間 totally order 的關係。在 reachability testing 執行過程中，每個相異的 SYN-sequence 都分別會產生各自的 RV-diagram，並在展開的過程中標記出代表 race variant 的節點，如圖 9 所示。

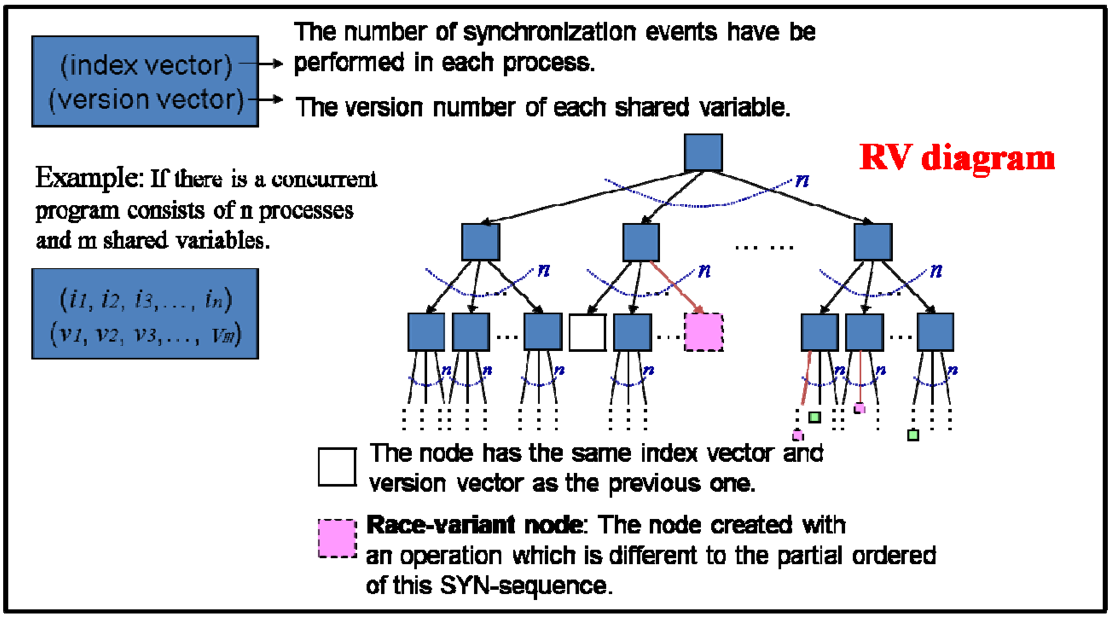

圖.9 RV diagram 的推導 race variants 概略圖

2.2 Race Graph

之後黃冠寰 博士在 2004 年於 IEEE Transaction on Software Engineering 發表以 race graph 的方式[4]，直接更改 Synchronization events 間 partially order 的關係。在 race graph 中每一個節點都代表程式中某一個 synchronization event，synchronization events 間的 partially order 關係則用 directed edge 表示。產生 race variants 時，僅需要分別就某一節點的 in-edge 作反向即可；若某一節點一共有 k

個 in-edge，則在此節點的位置上，至多會產生(sk-1)個 race variants，如圖 10 所示。實驗上相較於 RV-diagram，race graph 的方法能大幅提高效能，對於某個 SYN-sequences 總數為 117 的並行程式，RV-diagram 的方式一共需要 3,537 次的執行，而 race graph 僅僅需要 152 次。那是因為 RV-diagram 所產生的 race variants 容易造成重複的測試，即產生相同的 SYN-sequence。

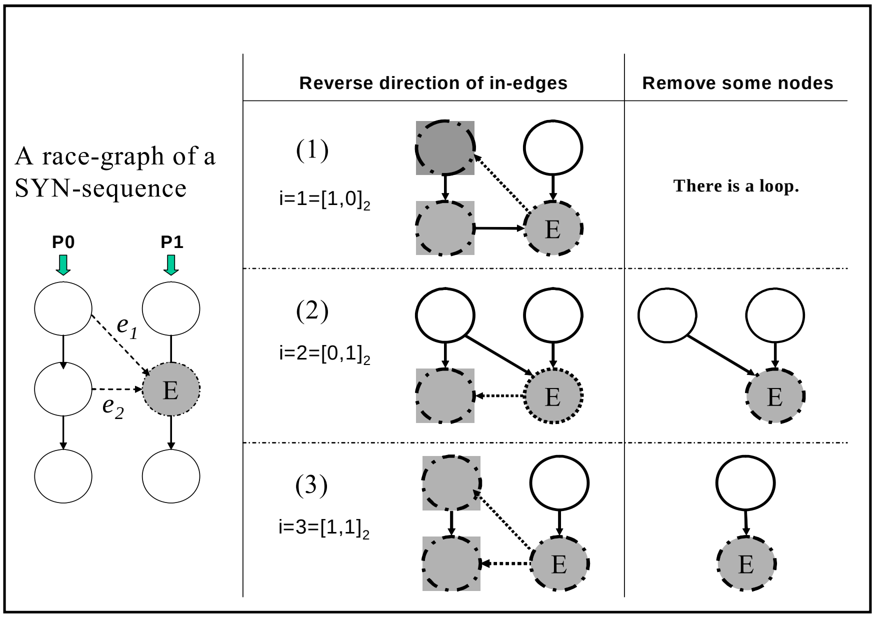

圖.10 利用 race graph 產生 race variant

2.3 Testing concurrent programs using Value Schedules

這篇論文[17]是由 Jun Chen 和 Steve MacDonald 在 2007 年 ACM Intl. Conf. ASE 所發表出來的，這個研究的第一個目標就是把所有 critical concurrent events 的 sequences 找出來，另外一個目的就是利用他的整套系統的架構能夠測試並行程式，根據他所產生出來的 sequence 會輸出其結果，在這樣的 interleave 下是否正確。

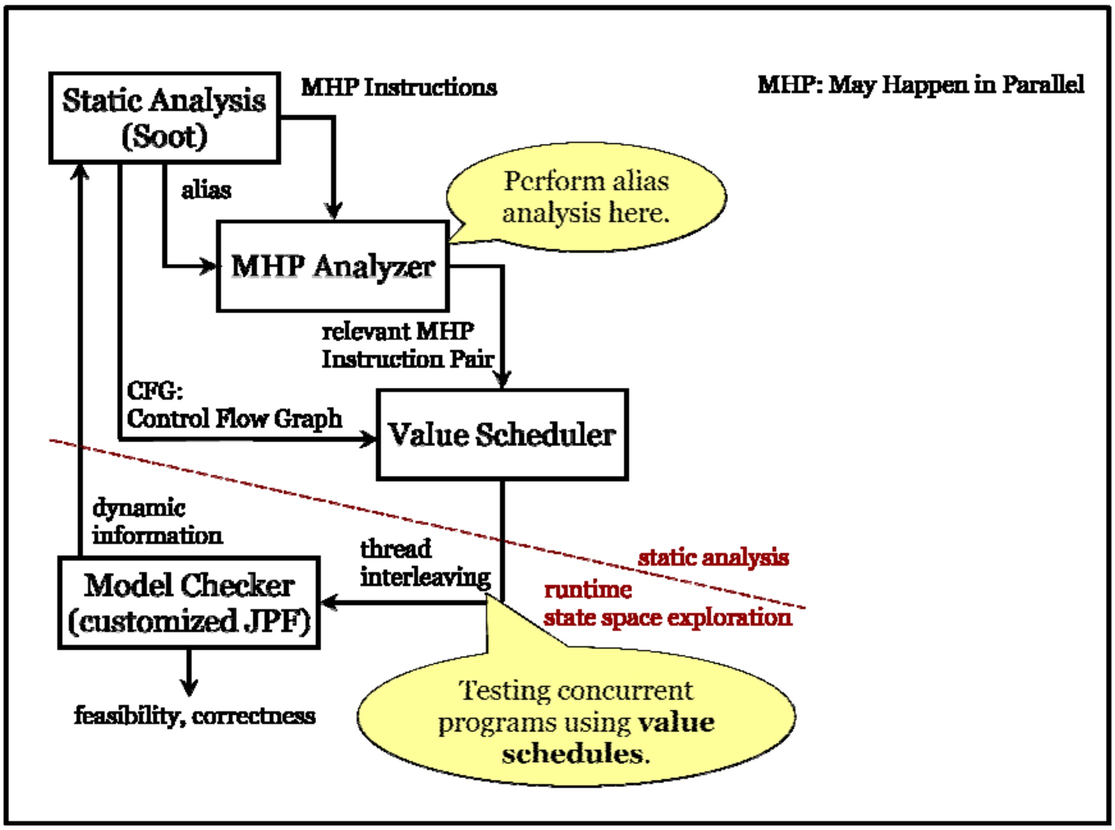

圖.11 整個 Value schedules 的架構圖

根據圖 11 此系統的測試方式主要是分為 static analysis 和 runtime state space exploration，static analysis(Soot)[18]會將 Java byte code 作分析且會控制並行程式的 interleaving 並以 control flow graph 形式分析出來，MHP(May Happen in Parallel)最主要就是要把 critical events 作配對(read-write 和 write-write)，產生一推 pairs，會把這些 pair 串起來，產生一個像 SYN-sequence 的形式，此系統可以保證這個決定的 interleave 也只會被執行一次，Value Scheduler 會去 traversal CFG，且分析 CFG 兩兩 critical events 的 conflicting 情況，最後再把他丟到 model checker(JPF 套件)[15]裡面去分析它的正確性，此為 runtime state space exploration，他的研究

主要的貢獻，就是修改了 JPF 套件，讓它更符合它所要的需求。

針對這篇對我研究上的啟發是在 Value Scheduler 送出的執行緒 interleave，再送到 JPF 裡所作的一些處理，根據圖 11 Value Scheduler 會去 traversal CFG，產生一個 path tree，在分岔點上有去儲存 program counter 於 program state 中，每次 JPF 去執行這樣的 interleave，會去分析此 interleave 是不是 feasibility 和 correctness，在下次 JPF 執行這樣的 interleave，他並不會再重頭執行，而去載入分岔點所儲存的 program state，從那個分岔點去執行，我們就是把這個方式加入到我們的 reachability testing 系統中，讓我們利用這樣的方法來提高我們系統的效能。

3. 實作方法

我們使用本實驗室的 reachability testing 系統來修改和新增，將我的想法加入於此系統中，以便達成我所要得到的效能。先前在圖 7 之中，我們在原始 reachability testing 裡面，以一個 SYN-sequence 去跑 race analyzer 和 prefix-based replay 互動的過程，可以很明確的看出這個 SYN-sequence 的變化，如今我們為了要改進這樣的功能，就開始去把 state 的功能加入到 reachability testing 裡面，所以會像圖 12 裡面的功能一樣，我們會先設計一個 state，且 state 的內容有三個 LocationCounterVector[] = $\{LC_0,LC_1,LC_2,\ldots\ldots\}$，為 synchronization events 的 location counter，ShareVariableVector[] = $\{Val_0,Val_1,Val_2,\ldots\ldots\}$，為共享變數的值，ShareVarVersion[]=$\{Ver_0,Ver_1,Ver_2,\ldots\ldots\}$，為共享變數的版本號。大致上比較重要的就是這三個數值，我們隨便選一個 SYN-sequence 送進去 race analyzer，這個時候就會帶著一個 state0，裡面是由上次 prefix-based replay 所推算出來的 state，到進去作 race analyzer 的時候，會把上一次 mode 為 replay 的 synchronization events 設定為 restore，也推導他的某一個 race variant，即產生一個新的@′符號，接下來，繼續再去作 prefix-based replay，一開始就會先載入 state 的內容，然後會省略掉 mode 為 restore 的 synchronization events 而跳到 mode 為 replay 的 synchronization events 去執行，這樣我們很明確的就省略掉一些指令不去執行它，就是利用這樣的觀念去加速效能，一直到並行程式 replay 到@′，就會去推算它新 state 的內容，即為 state1，就這樣一直遵循原本的 reachability testing 去執行，一直到結束都會

遵循著圖 12 的步驟去作，達成我們的目的。

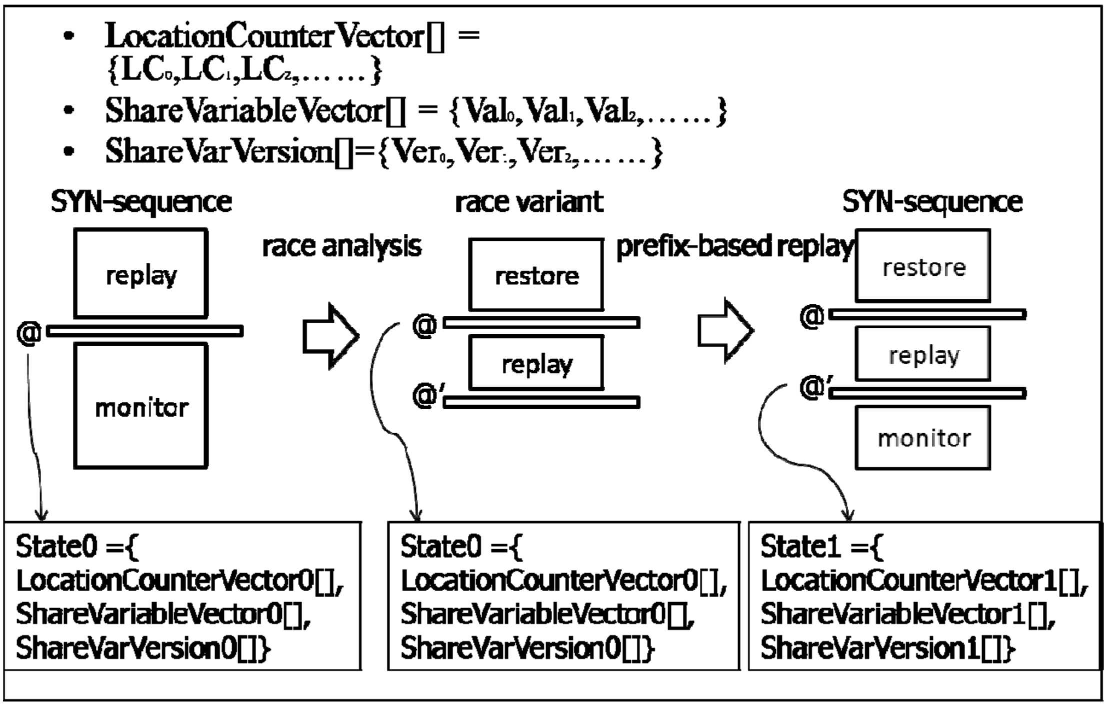

圖.12 加入 state 後 SYN-sequences 在 race analyzer 和 prefix-based replay 的運作

我們在實作上通常都是利用 Java 執行緒來模擬多程序處理的方式，在本章節我們會依序的介紹如何去實作這樣的功能，一開始會先介紹 state 的內容跟設計，接下來會介紹對 entry 和 exit protocol 的修改，restore 和 store state 的時間點，prefix-based replay 的修改和一些特殊例子的處理。

3.1 設計 state

當然我們會在 reachability testing 系統添加一個 restore mode，讓系統認識這個 mode，就像 replay mode 和 monitor mode，此 restore mode 所要作的事情就是將來再作 prefix-based replay 的時候，會省略掉這些 synchronization events，而達

成提高 reachability testing 系統效能的目的。

既然我們今天省略掉這些 restore mode 的 synchronization events，此系統再作 prefix-based replay 的時候，會因為變數的版本號、location counter 和變數的值沒紀錄到，而導整個 reachability testing 系統無法跑完 prefix-based replay，此時我們就會加入 state 的觀念進來，讓整個 prefix-based replay 能夠正確無誤的執行下去，所以我們就先設計好這個 state 的內容。此 state 最主要是要能抓住這三個值，變數的版本號、location counter 和變數的值，程式碼如圖 13 所示：

```
public class SaveState {

    private Integer[] ShvVector;
    private Integer[] VerVector;
    private Integer[][] LovVector;
    private Integer[] LocctrVector;
    private Integer PrcsNum;
    private Integer LovNum;
    private Integer ShvNum;
    private int State_No; // 用來紀錄Syn-sequence的編號

    public SaveState(int PrcsNum, int ShvNum, int LovNum) {
        this.PrcsNum = PrcsNum;
        this.LovNum = LovNum;
        this.ShvNum = ShvNum;
        this.ShvVector = new Integer[ShvNum];
        this.VerVector = new Integer[ShvNum];
        this.LovVector = new Integer[PrcsNum][LovNum];
        this.LocctrVector = new Integer[PrcsNum];
        for (int i = 0; i < PrcsNum; i++)
            this.LocctrVector[i] = i * 1000 + 1000;
        for (int i = 0; i < ShvNum; i++) {
            this.ShvVector[i] = 0;
        }
```

```
        for (int i = 0; i < ShvNum; i++) {
            this.VerVector[i] = 0;
        }
        for (int i = 0; i < PrcsNum; i++)
            for (int j = 0; j < LovNum; j++)
                this.LovVector[i][j] = 0;

        State_No = -1;

    }
```

圖.13 Design state 的資料結構

利用一維陣列的方式去儲存共享變數、版本號和 location counter 的值，也會利用二維陣列去紀錄區域變數的值，即為哪個執行緒的哪個變數之意，在建立這個 state 的時候，會預設一些初始值，然後再把他包成一個物件丟到集合裡面去作儲存，再把這個集合加入到 race graph 裡面去，以至於在作 race analzer 和 dynamic effective testing 不會遺失這個狀態。像後面儲存區域變數的值，類似這種擴充的方式，我們將來也可以擴充這個 state 來去作一些額外的工作，因為此方法有點像是暫存器的樣子，再作完 prefix-based replay 後，之後還可能會作 dynamic effective testing，我們也可以利用它來儲存一些 joint point 等等應用，目前暫不討論這個，只著重於提高 reachability testing 系統的效能。真正在設定 restore mode 給 synchronization events 的地方就是在 race analyzer，再 RaceVariant.jave 開始處來設定它，如下圖 14 所示。

```
    public static int Generator(RaceGraph theRG, RaceGraphSet RVSet)
{

        int genRVNum = 0;
        HashSet<EventNode> theEvnts = theRG.Nodes;
        for(EventNode tmpEvnt : theEvnts) {
            if(!tmpEvnt.Mode.equals("monitor"))
                continue;
             ·················//Original
```

```
public static int Generator(RaceGraph theRG, RaceGraphSet RVSet) {
        int genRVNum = 0;
        theRG.print();
        for(EventNode tmpEvnt : theRG.Nodes){
            if(tmpEvnt.Mode.equals("replay")){
                tmpEvnt.Mode = "restore" ;
                continue;
            }
            else{
            }
        }
            …………………// Optimization
```

圖.14 設定 mode 為 restore 的程式碼

把上次標記為 replay mode 的 synchronization events 一個一個拿出來設定為 restore mode，在 race analyzer 下我們也可以把它想成是一個定義。

3.2 針對 entry 和 exit protocol 的修改

之前我們設計好 state 了，但是在 prefix-based replay 的運用上就必須要先修

改主要控制程式流程的 entry 和 exit protocol，原始我們再作 prefix-based replay 裡 replay phase 的階段，我會利用變數版本號跟其值在加上 entry 和 exit protocol 來控制整個並行程式的流程，會在每個 synchronization events 之前加上 entry protocol，其用意主要是用來確認此 synchronization event 是不是該執行了的控制，假如不該此 synchronization event 執行的話，即會停住，此時作業系統就會去作 context switch 切到另一個執行緒去執行，來達成控制並行程式的執行順序，也就是 replay phase 在作的事情，在設計上就如圖 15。

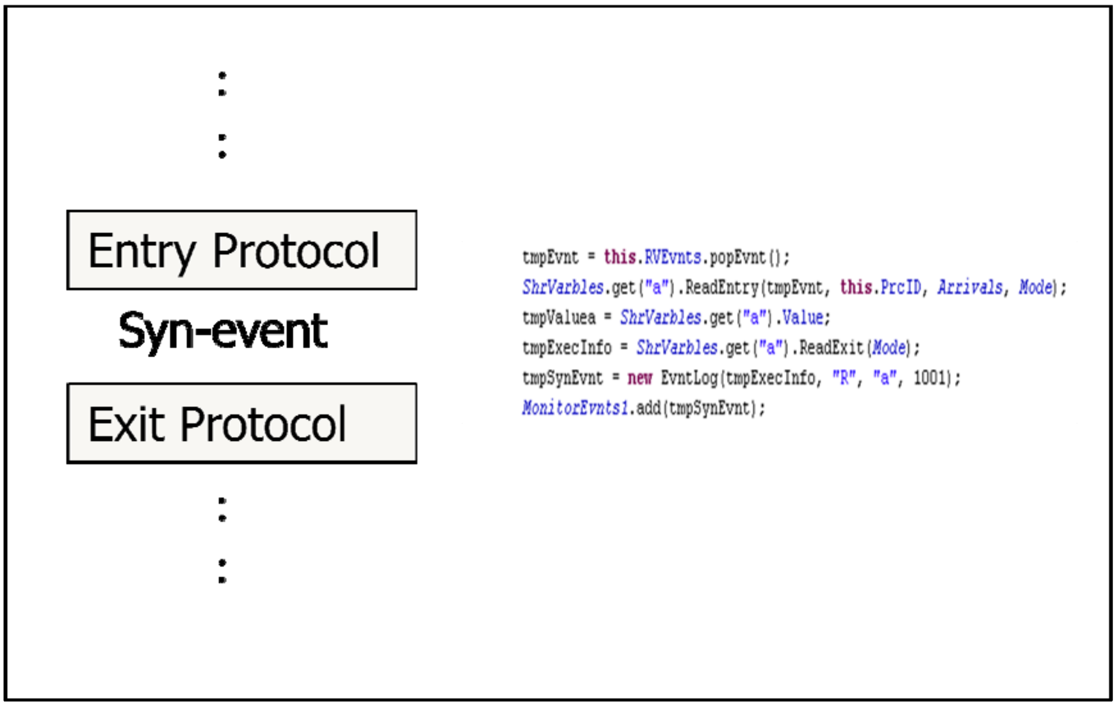

圖.15 原始 entry/exit protocol 的設計

我們就要去修改它，讓我們再作 prefix-based replay，也可以把 restore mode 的概念加入於其中，也要利用 entry protocol 來判斷是否要去執行此 synchronization event，若此 synchronization event 是 restore mode 就會忽略掉此

synchronization event，要的話我們才會去執行它，如圖 16 所示。

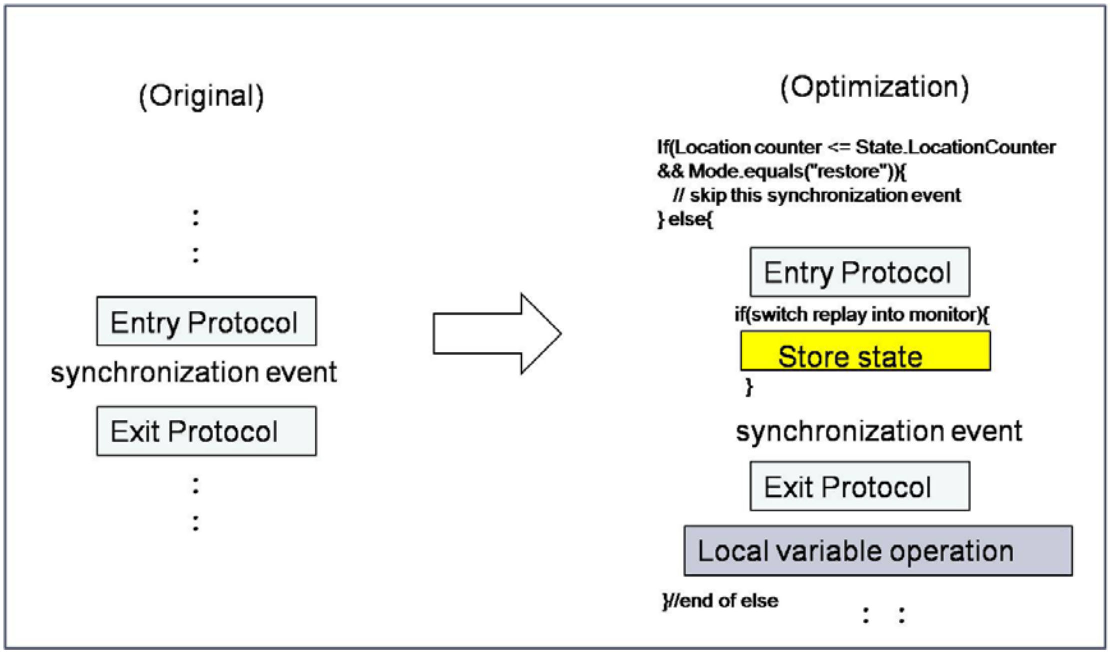

圖.16 將 entry/exit protocol 修改過後

我們利用 if else 的觀念來把 entry 和 exit protocol 包起來，在進入之前 entry protocol 之前，我們會先判斷兩個條件，判斷執行 mode 是否是 restore 和此 location counter 是否會比此 state 裡存的 location counter 要來的小，假如成立，我們就會忽略掉這個 synchronization event，也就是這個 synchronization event 是為 restore mode，反之，我們就會去執行它，也就是它可能是 replay 或 monitor mode 的 synchronization event，這裡暫時先只敘述此項方式，下面我們還會再介紹 restore 和 store state 的時間點。

3.3 Restore 和 store state 的時間點

我們大致上已經在受測程式裡面每一個 synchronization event 的 entry 和 exit protocol 作了修改了，但是這樣還少了一樣事情就是我們甚麼時候要去 store state 和 restore state 的時間點，這個相當重要的關鍵，也是我們整個 prefix-based replay 不會因為所得到的資料不合而無法執行下去的重點，時間點的規劃，會如圖 17 所示，在受測的並行程式中，每個執行緒要執行之前，我們都會將它的 state 作載入的動作，理論上應該是 restore mode 切換到 replay mode 的時候要作此項 state 的更新，因為為了方便讓使用者使用，我們一律都放在開頭，此項也不會影響到程式的執行，也比較好了解。何時要去 store state？就是在 replay mode 切換 monitor mode 的時候，因為我們要記住最後一個 mode 為 replay 的 synchronization event 的資訊，此項就是我們的新 state 的內容，將來在再送去作

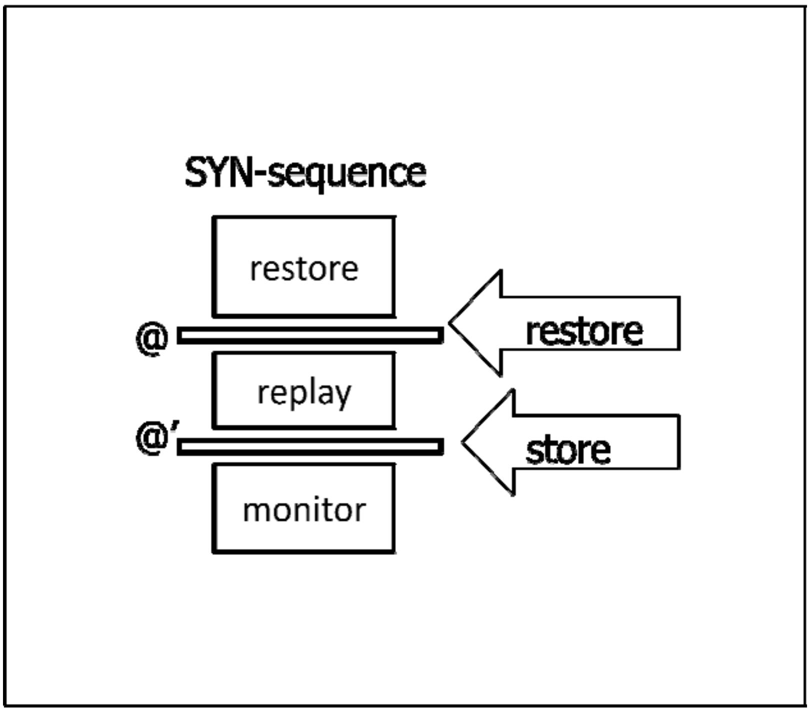

圖.17 Restore 和 store 插入的時間點

race analyzer 的時候，會將 replay mode 的 synchronization event 設成 restore mode，此項 state 就是它的依據，以提供將來作 prefix-based replay 忽略一些 restore synchronization events 用的。圖 16 裡面當進入 entry protocol 的時候，我們也會傳入一個布林值來判斷是否為切換 replay mode 為 monitor mode 的時間點，就是這樣的去判斷是否要去儲存新的 state 內容。

圖 18 即為我們一開始就要在受測程式碼之前 restore state 的程式碼，利用執行 mode 是不是為 restore 和 RestorOne 來判斷，RestorOne 的用途，因為我們是針對共享變數作處理，所以我們只要 restore state 一次內容就好了，而不需要各個執行緒都要去作 restore 的動作，針對區域變數的處理則就另行討論。

```
    if (Mode.equals("restore") && RestorOne ) {
            RestorOne = false;
            ShvVabletemp[0] = ShrVarbles.get("a").Value = LastState
                    .getShvValue(0);
            ShvVabletemp[1] = ShrVarbles.get("b").Value = LastState
                    .getShvValue(1);
            ShvVertemp[0] = ShrVarbles.get("a").Version = LastState
                    .getShvVersion(0);
            ShvVertemp[1] = ShrVarbles.get("b").Version = LastState
                    .getShvVersion(1);

        }
```

圖.18 Restore state 的程式碼

在每個受測程式一開始的使用者的設定上，就是要載入之前的 state 內容，主要是為了之後用這個 state 的內容來判斷是否要省略此項 synchronization event。

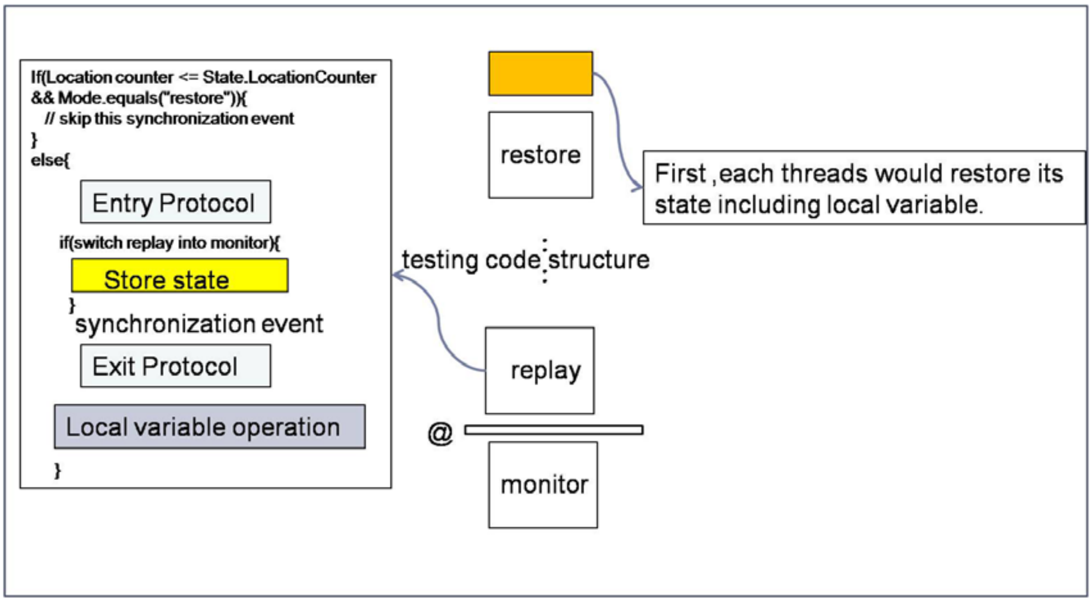

圖.19 受測程式的架構

圖 19 是整個受測並行程式單一執行緒的程式架構，整個 SYN-sequence 可能會有三種 mode，每一個 synchronization events 的處理方式，會像圖右邊一樣，在受測程式的一開始我們就會先作載入 state 的動作。

```
tmpEvnt = this.RVEvnts.popEvnt();
ShrVarbles.get("a").ReadEntry(tmpEvnt, this.PrcID,
                               Arrivals,Mode);
tmpValuea = ShrVarbles.get("a").Value;
tmpExecInfo = ShrVarbles.get("a").ReadExit(Mode);

tmpSynEvnt = new EvntLog(tmpExecInfo, "R", "a", 1001);


 MonitorEvnts1.add(tmpSynEvnt);
```

圖.20 未修改前的一個 synchronization event 程式碼

圖 20 裡是原始我們再設計一個 synchronization event 的內容，在他之前所設

計的 entry protocol 和之後的 exit protocol 的程式碼，在 entry protocol 主要是要用來監督此 synchronization event 是否該執行與否，我們是利用 semaphore 的方法來控制的，假如此 synchronization event 是可行的話，之後 exit protocol 會對這個 event 作記載變數值和版本號還有 location counter 的紀錄，讓後面並行程式裡面各 synchronization events 在 entry protocol 都會依造這個紀錄來去控制這些 synchronization events 的執行順序，這些最主要的控制其實是在作 prefix-based replay 功能。

```
if(1001<=LastState.getLocctr(PrcID)&&Mode.equals("restore")) {
        } else {
            if (Resfinal) {
                ResBerrialcount++;
                while (ResBerrialcount < PrcsNum);
                Mode.set("replay");
                Resfinal = false;
            }
            tmpEvnt = this.RVEvnts.popEvnt();
            RRMtemp  =  ShrVarbles.get("a").ReadEntry(tmpEvnt,
                    this.PrcID,ReplayArrivals, Mode, RRMtemp);
            if (RRMtemp[PrcID] == true) {
                if (Update < PrcsNum) {
        saveState_0.setShrValue(0,ShvVabletemp[0],ShvVertemp[0]);
        saveState_0.setShrValue(1,ShvVabletemp[1],ShvVertemp[1]);
                    Update++;
                }
                saveState_0.setLocctr(PrcID, 1000);
                RRMtemp[PrcID] = false;
            }

            ShvVabletemp[0] = ShrVarbles.get("a").Value;
```

```
            tmpExecInfo = ShrVarbles.get("a").ReadExit(Mode);
            tmpSynEvnt = new EvntLog(tmpExecInfo, "R","a", 1001);
            MonitorEvnts1.add(tmpSynEvnt);

                                         }
```

圖.21 Optimization code 裡 a synchronization event 的程式碼

在圖 21 裡面，當我們要將 State Storing in Prefix-Based Replay 的方式引進來，就必須要改變這個部分，就像右圖所示，我們會針對一開始所載入的 state 的內容來判斷：

```
if (1001 <= LastState.getLocctr(PrcID) && Mode.equals("restore"))
```

我們利用此 synchronization event 的 location counter 和 state 內容裡面的 location counter 來作比較，當 state 裡面發現到的 location counter 大於此 synchronization event 的 location counter，且執行 mode 也為 restore，則代表現在這些 synchronization events 的 mode 都為 restore，代表此 synchronization events 是可以省略不作的。

接下來我們就討論假如這些條件都不符合的話，就會跳到後面的 else 之執行的地方，

```
if (Resfinal) {
        ResBerrialcount++;
    while (ResBerrialcount < PrcsNum);
        Mode.set("replay");
        Resfinal = false;
    }
```

當然 mode 為 restore 的 synchronization events，跟之前 replay phase 和 monitor phase 之間的設計上的觀念也是一樣的，我們也是要先把各執行緒裡 mode 為 restore 的 synchronization events 必須跑完才可以去切換執行 mode 為 replay，假如此執行緒裡面的 mode 為 restore 的 synchronization events 都已經作完了，我們就會把此執行緒先卡在 while 迴圈裡面，讓別人也達到 mode 為 restore 的 synchronization events 都已經作完的狀態下，就會把執行 mode 設為 replay，大致上就是這樣來達成當我們再作 prefix-based replay 這個方法，都可以保證各執行緒裡 mode 為 restore 的 synchronization events 都已經作完了，才可以作之後的 replay phase 和 monitor phase。

```
RRMtemp = ShrVarbles.get("a").ReadEntry(tmpEvnt, this.PrcID,
          ReplayArrivals, Mode, RRMtemp);
if (RRMtemp[PrcID] == true) {
            if (Update < PrcsNum) {
//              System.out.println("開始store state");
                        saveState_0.setShrValue(0, ShvVabletemp[0],
                                ShvVertemp[0]);
                        saveState_0.setShrValue(1, ShvVabletemp[1],
                                ShvVertemp[1]);
                        Update++;
                    }
                saveState_0.setLocctr(PrcID, 1000);
                RRMtemp[PrcID] = false;
            }
```

針對 entry protocol 裡面設計，我們會傳入一個布林值，用來控制是否要去作 store state，此階段也執行 mode 將 replay 切到 monitor 的狀態。因為我們再設計 state 的時候，在執行 prefix-based replay 的時候，是各執行緒裡面都會存有自己的一個 state 內容，也透過他自己來去更新自己 state 的內容，在共享變數上面，也確

保 state 裡面的更新數目絕對不會大於執行緒的數目，因為並行程式的關係，有的時候會造成同步上的問題，所以各執行緒都要自己去儲存一個 state 的內容，在 prefix-based replay 之後，我們再去作組合的動作，看誰的共享變數裡面的版本號較新，就會把他給留下來，這樣我們所推算出來的 state 內容才是正確無誤的，程式裡面 Update 變數的用法就式來確保這個動作的。所以在這裡使用者設計各執行緒裡面大致上就是這樣去設計的，也合乎我們的觀念。

3.4 Prefix-Based Replay 主程式裡面的修改

在受測程式中，這個部分也是要自己去作的，所以我們會改進它的觀念，如圖 21 所示。

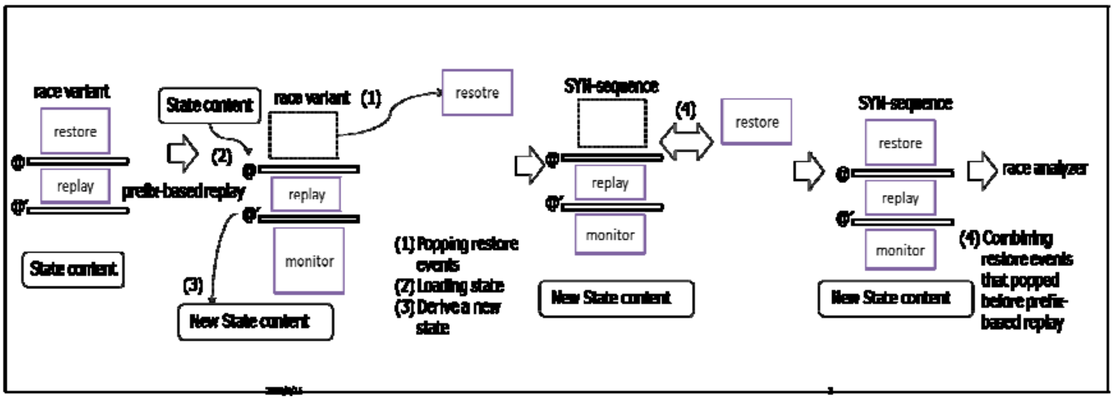

圖.22 Prefix-based replay 修改後的程式流程

我可以知道當一個 SYN-sequence 進入要去作 prefix-based replay 的時候，我們會先把原來的 SYN-sequence 裡面為 restore mode 的 synchronization events 給拆卸出來，真正送進去作 prefix-based replay 的時候，其實只有 replay mode 和 monitor mode 的

synchronization events，這些作的事就跟之前 reachability testing 系統裡 prefix-based replay 的動作是一樣的，只是很巧妙的接上 3.3 裡面所講的忽略掉 restore mode 的 synchronization events，來達到此效果，之後再作完 prefix-based replay 的時候就會在和先前所拆卸出來的 restore mode 的 synchronization events 作組合，即為一個完整的 SYN-sequence，此 SYN-sequence 裡面也包含著更新完後的 state，再送去 race analyzer 裡去推導 race variants。在此想法之前，其實我們是直接刪除掉 restore mode 的 synchronization events，一開始本來是以為這樣也可以節省一些記憶體的空間，但是這樣會出現三個問題，第一個問題點是在送去作 race analyzer 的時候，其實還有一個動作是在作 dynamic effective testing(state analyzer)的，如圖 22 所示，當你傳進去的 SYN-sequence 並非是完整的 SYN-sequence 的時候，此項分析會出現錯誤，第二個問題就是最後程式所收集到的 SYN-sequence 都是不完整的，會違反 reachability testing 找出所有 synchronization events 的 interleaves 原則，第三個問題在某些有迴圈的並行程式裡面會有問題發生，這些問題都是我們為什麼不直接刪除 restore mode 的 synchronization events 的原因，圖 23 裡面的程式碼主要是針對圖 22 去實作的一些細節。

。

```
public class Demo1_PrefixBasedReplay1 {

    public static RaceGraph perform(RaceGraph RaceVariant) {

        ReplayEvnts Evntlist1, Evntlist2;
        RaceGraph tmpRG = new RaceGraph();
        RaceGraph MntrRG = new RaceGraph();
        LinkedHashSet<EventNode> MonitorEvntstemp = new
```

```
         LinkedHashSet<EventNode>();
        Evntlist1 = RaceVariant.getEvntlist(0);
        Evntlist2 = RaceVariant.getEvntlist(1);
         SaveState SS1 = new SaveState(Demo1Thread.PrcsNum,
                         Demo1Thread.ShrNum,Demo1Thread.LocNum);
        SaveStateSet SSet = new SaveStateSet();

        Demo1Thread1 r1 = new Demo1Thread1(Evntlist1);
        Demo1Thread2 r2 = new Demo1Thread2(Evntlist2);

        r1.setShrVarbles(RaceVariant);
        r2.setShrVarbles(RaceVariant);

        int SynEng0 = 0;
        for (EventNode EN : RaceVariant.Nodes) {
            if (EN.Mode.equals("restore") && EN.EvnID.PrcID == 0) {
                SynEng0++;
            }
        }
        for (int i = 0; i < SynEng0; i++) {
            EventNode tmpEvnt = RaceVariant.getEvent(0, i);
            RaceVariant.Nodes.remove(RaceVariant.getEvent(0, i));
            Evntlist1.popEvnt();
            MntrRG.addEvent(0, tmpEvnt.EvnID.EvnSeq, tmpEvnt.RWOP,
                    tmpEvnt.VarName, tmpEvnt.VarVersion, mpEvnt.Mode,
                    tmpEvnt.LOCCTR);
        }

        int SynEng1 = 0;

        for (EventNode EN : RaceVariant.Nodes) {
            if (EN.Mode.equals("restore") && EN.EvnID.PrcID == 1) {
                SynEng1++;
            }
        }
        for (int i = 0; i < SynEng1; i++) {
            EventNode tmpEvnt = RaceVariant.getEvent(1, i);
            RaceVariant.Nodes.remove(RaceVariant.getEvent(1, i));
```

```
            Evntlist2.popEvnt();
            MntrRG.addEvent(1, tmpEvnt.EvnID.EvnSeq, tmpEvnt.RWOP,
                tmpEvnt.VarName,tmpEvnt.VarVersion,tmpEvnt.Mode,
                    tmpEvnt.LOCCTR);
        }

        tmpRG.Nodes.addAll(MonitorEvntstemp);
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);
        if (!RaceVariant.saveState.isEmpty()) {
            Demo1Thread.LastState = RaceVariant.saveState.getState();
        }

        t1.start();
        t2.start();

        while (Demo1Thread.Finished < Demo1Thread.PrcsNum)
            ;
        MntrRG.InvStates = RaceVariant.InvStates.myClone();
        MntrRG.Varbles = RaceVariant.Varbles.myClone();

        for (int i = 0; i < Demo1Thread.MonitorEvnts1.size(); i++) {
            EvntLog tmpSynEvnt = Demo1Thread.MonitorEvnts1.get(i);
            MntrRG.addEvent(tmpSynEvnt, 0, i + SynEng0);
            if (tmpSynEvnt.RWOP.equals("W")) {
                // tmpSynEvnt.print();
                MntrRG.Varbles.addValue(tmpSynEvnt.VarName,
                        tmpSynEvnt.VarVersion, tmpSynEvnt.VarValue);
            }
        }

        for (int i = 0; i < Demo1Thread.MonitorEvnts2.size(); i++) {
            EvntLog tmpSynEvnt = Demo1Thread.MonitorEvnts2.get(i);
            MntrRG.addEvent(tmpSynEvnt, 1, i + SynEng1);
            if (tmpSynEvnt.RWOP.equals("W")) {
                // tmpSynEvnt.print();
                MntrRG.Varbles.addValue(tmpSynEvnt.VarName,
                        tmpSynEvnt.VarVersion, tmpSynEvnt.VarValue);
```

```
            }
        }
        SSet.add(Demo1Thread.saveState_0);
        SSet.add(Demo1Thread.saveState_1);
        MntrRG.saveState.add(SS1.Composite(SSet));
        if (RaceVariant.saveState.getState() != null)
            MntrRG.saveState.getState().setState_NO(
                RaceVariant.saveState.getState().getState_NO());
        return MntrRG;
    }
}
```

圖.23 Optimization code 裡 prefix-based replay 程式碼

3.5 特殊的並行程式

經過以上的實作後，我們在一般的並行程式例子，確實是可以依造這個觀念去執行的，不過我們有可能會遇到一些比較艱難的問題，有區域變數和迴圈的例子，我們的方法還可行嗎？

(1) 我們先從區域變數的例子來看：

假如我們在並行程式裡面的某一個執行緒裡面添加區域變數，則在原本的方法裡面，當我們省略某些 synchronization events 的時候，對於在這些忽略掉的 restore mode 的 synchronization events 中間可能穿插著區域變數，但是假如你沒有對這些區域變數作處理的話，這樣將會出現區域變數的值並未更改到的狀況發生，這樣可能會導致程式的錯誤。所以針對這樣的方式，我們也要將區域變數做 store 的方法，也要加入到 state 裡面去，像我之前設計的 state 內容一樣，我會用

二維陣列的方式去儲存，之所以用二維主要是用來分辨他是哪個執行緒裡面的那個區域變數之內容作儲存。一般來說我們會將這個區域變數的操作 hold 住在 synchronization event 之後，如下面的例子：

Initial shared variables: a:=0, b:=0;
Initial local variables: i:= 0;

	Thread0
0	read a ;
1	read b ;
2	a = 13;
3	i++;
4	b = 14;

	Thread1
0	a = 21 ;
1	b = 22 ;
2	read a;
3	read b;

程式為兩個執行緒的並行程式，兩個共享變數 a、b 初始值為 0，thread0 的區域變數 i 初始值為 0，當程式在跑的時候，我們對 line 2 的 synchronization event 作 store state 的方法，我們也會把這區域變數的操作加入到之後，如下圖 24。

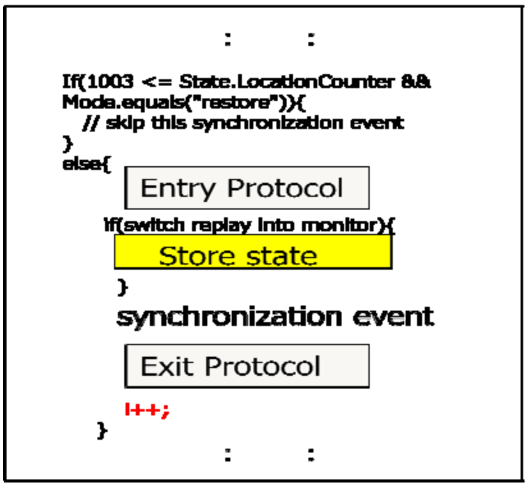

圖.24 區域變數指令處理所放置的位置

store state 裡面會包含對區域變數裡的儲存，這樣也可以確保程式也對區域變數store 到的狀況，在程式一開始的時候，在作 restore state 動作，也一併對區域變數也作 restore，但是區域變數的 restore 並不是跟共享變數一樣的，它是屬於獨立再一個執行緒裡面活動的變數，所以這裡要個別的去儲存，這樣子就可以確保資料無誤了。

(2) 迴圈程式的處理方式

有了先前的區域變數的處理方式，我們對迴圈處理上就有了很大的幫助，程式裡面迴圈的例子，我們大致上分為 do while、while 和 for 的語法，因為三個語言其實是可以互通的，所以我們針對 whlie 迴圈來探討。

迴圈的程式裡面，我們主要是要處理當我們要跳到程式碼的某個迴圈裡面的某個 synchronization event 的時候，我們該怎麼去正確的找出省略的迴圈數，這個就是重點之一。

在 3.4 裡面，我們介紹了 prefix-based replay 的實作方式，其實在進行 replay 的時候，我們把 mode 為 restore 的 synchronization events 抽出來是有意義的，再搭配(1).所提的區域變數的處理，這樣我們再一開始載入區域變數（此時區域變數是紀錄上次所儲存的迴圈數），因為我們再作 replay 的時候，已經沒有 mode 為 restore 的 synchronization events，這兩步都作到了，該程式作 Prefix-base replay

的時候就會直接跳入該圈裡面去作執行了。

我們用以下的迴圈程式例子跑一次 prefix-based replay 的處理，跟我們怎麼處裡迴圈例子的形式：

eq.

Initial shared variables: a:=0, b:=0;
Initial local variables: i:= 0;

Thread0	
	While(i<2) {
0	read(a) ;
1	a := 12;
2	b := 13
3	i++;
	}

Thread1	
0	a = 21 ;
1	b = 22 ;
2	read(b);

大致上 thread0 會跑兩次 iteratives，當我們再處這樣迴圈的例子時，我們利用某一次 race variant 來作 prefix-based replay 的方式來說明，可以看到我之前在 prefix-based replay 裡的設計，如圖 25。

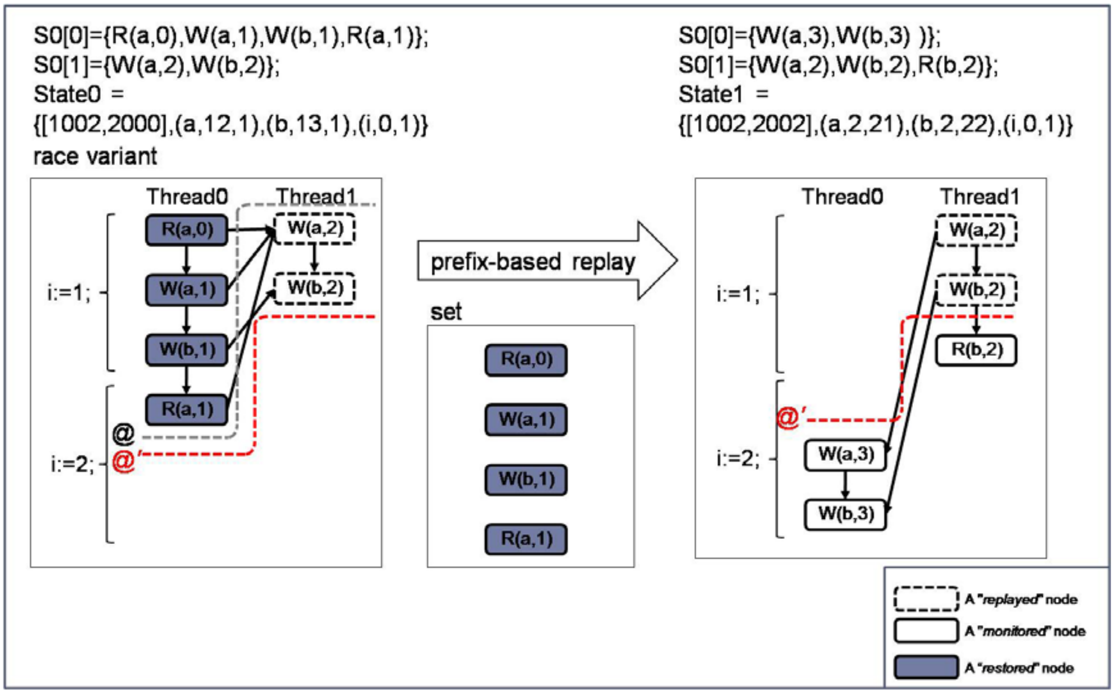

圖.25 Prefix-based replay 裡實作迴圈的例子

在圖 25 裡我們可以解釋圖 22 裡面的抽象圖意，假設我們傳入某一個 race variant，如圖中左邊那個 race graph 的形式，他會包含一個 state0，在要送去 prefix-based replay 之時，他會將 restore synchronization events 抽出來，如圖中間那四個 synchronization events，之後我們就實際開始跑 prefix-based replay 的動作，如圖右所示，我們是從 i:=2 那個 iterative 開始執行的，所以只要再一開始的地方有載入 state 的內容，就可以對上修改過後 prefix-based replay 這個部分，來達成我們跳到迴圈裡面的某一個 synchronization event 執行的目的了。最後在之前所移出的 restore synchronization events 作組合，即為完整的 SYN-sequence，如圖 25，之後就會再送去 race analyzer 裡面分析他的 race variants，整個大框架就照著 reachability testing 裡的架構去跑，內部的修改並不會影響到大框架。

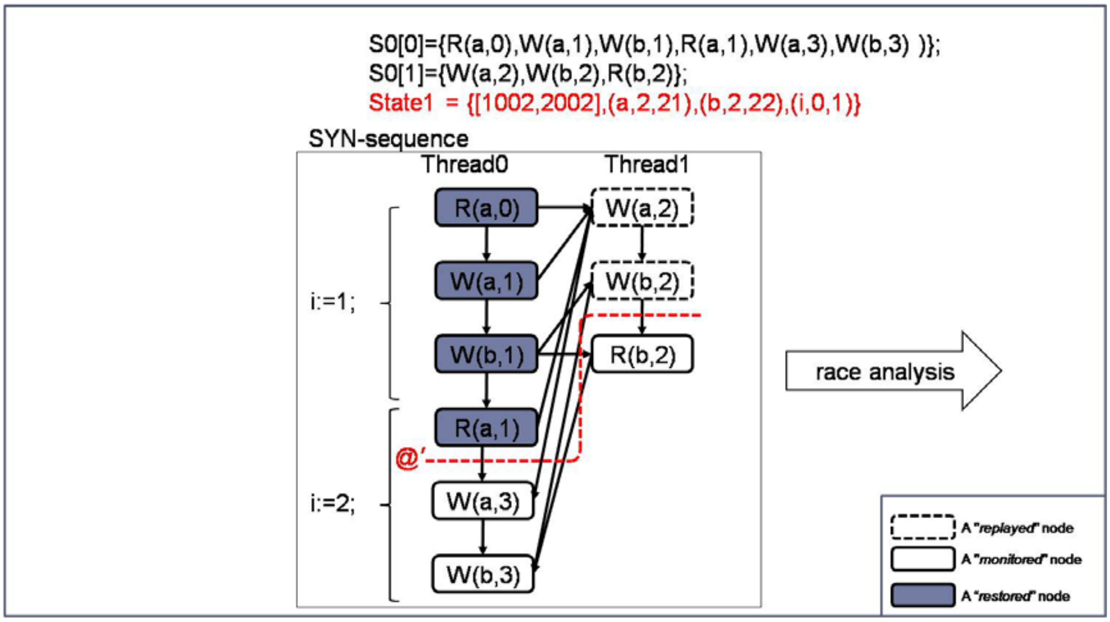

圖.25 Prefix-based replay 作完後所形成的 SYN-sequence

4. 實驗結果

我們利用的電腦配備 Intel(R) Pentium(R) 4 CPU 2.80GHz , 1.00GB 的 RAM。所使用的作業系統Microsoft Window XP Professional Version 2002 Service Pack 3，reachability testing 是用 SUN Java SDK1.6.0_13 去實作的，所使用的 IDE 介面是 Eclipse version 3.4，利用以上的環境去跑以下並行程式的例子：

1. 我利用一個簡單的例子來跑最佳化過的 reachability testing 系統：

Initial shared variables: a:=0, b:=0;

eg.1

Thread0		Thread1	
0	read(a) ;	0	a = 21 ;
1	read(b) ;	1	b = 22 ;
2	a = 13;	2	read(a);
3	b = 14;	3	read(b);

eg.2

Thread0		Thread1	
0	Read(a) ;	0	a = 21 ;
1	read (b) ;	1	b = 22 ;
2	a = 13;	2	read(a);
3	b = 14;	3	read (b);
		4	a = 25;

eg.3

Thread0		Thread1	
0	Read(a) ;	0	a := 21 ;
1	read (b) ;	1	b := 22 ;
2	a = 13;	2	read(a);
3	b = 14;	3	read (b);
		4	a := 25;
		5	a := 26;

eg.4

Thread0		Thread1	
0	Read(a) ;	0	a := 21 ;
1	read (b) ;	1	b := 22 ;
2	a = 13;	2	read(a);
3	b = 14;	3	read (b);
		4	a := 25;
		5	a := 26;
		6	a := 27;

eg.5

Thread0		Thread1	
0	Read(a) ;	0	a := 21 ;
1	read (b) ;	1	b := 22 ;
2	a = 13;	2	read(a);
3	b = 14;	3	read (b);
		4	a := 25;
		5	a := 26;
		6	a := 27;
		7	a := 28;

eg.6

Thread0		Thread1	
0	Read(a) ;	0	a := 21 ;
1	read (b) ;	1	b := 22 ;
2	a = 13;	2	read(a);
3	b = 14;	3	read (b);
		4	a := 25;
		5	a := 26;
		6	a := 27;
		7	a := 28;
		8	a := 29;

eg.7

Thread0		Thread1	
0	Read(a) ;	0	a := 21 ;
1	read (b) ;	1	b := 22 ;
2	a = 13;	2	read(a);
3	b = 14;	3	read (b);
		4	a := 25;
		5	a := 26;
		6	a := 27;
		7	a := 28;
		8	a := 29;
		9	a := 210;

eg.8

Thread0		Thread1	
0	Read(a) ;	0	a := 21 ;
1	read (b) ;	1	b := 22 ;
2	a = 13;	2	read(a);
3	b = 14;	3	read (b);
		4	a := 25;
		5	a := 26;
		6	a := 27;
		7	a := 28;
		8	a := 29;
		9	a := 210;
		10	a := 211;

此實驗結果可以跑出以下的實驗數據，每個例子跑個五次 run 1 ~ run 5：

eg.1(Optimization)	Time Consumed	SYN-sequences
run1	2455ms	21
run2	2316ms	21
run3	2514ms	21
run4	2451ms	21
run5	2453ms	21
eg.1(Original)	Time Consumed	SYN-sequences
run1	2210ms	21
run2	2015ms	21
run3	2032ms	21
run4	2204ms	21
run5	2201ms	21

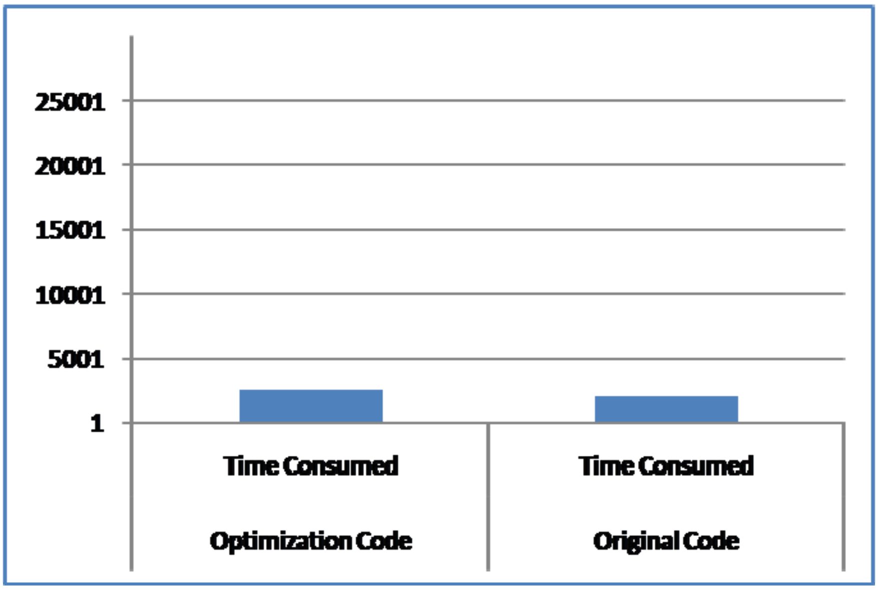

Table.1 8 個 synchronization events 的實驗數據長條圖

在 eq.1 裡面的例子很明顯，並未能達到提高效能的效果，後來經過分析，原因是在於受測的程式太小了，並非會有很明確的效率，接下來我們就開始作實驗了，以 eq.1 的例子為我們的基準點(base)，兩個執行緒各四個 synchronization events，

我們為了跑出明顯的效能時，就必需要把程式加大，也就是 synchronization events 的數量加大，讓它產生的 SYN-sequences 數目就會變比較多。我們就一個一個 synchronization events 一直加，所加的 synchronization events 動作皆為 write，因為 write synchronization event 比較會影響到整個 racing 的情形，所以所產生的 SYN-sequences 數量比較大，比較可以看出效能。我們會在 thread1 的最後再增加一個 write synchronization event，即為 eq.2，增加兩個 write synchronization events，即為 eq.3，以此類推，每個例子跑個三次 run 1~ run 3。經過實驗，我們的實驗數據就如下表。

	Optimization		Original	
eq.2	Time Consumed	SYN-sequences	Time Consumed	SYN-sequences
run 1	4313ms	36	5015ms	36
run 2	4213ms	36	4952ms	36
run 3	4165ms	36	5122ms	36
Average	4230ms	36	5030ms	36

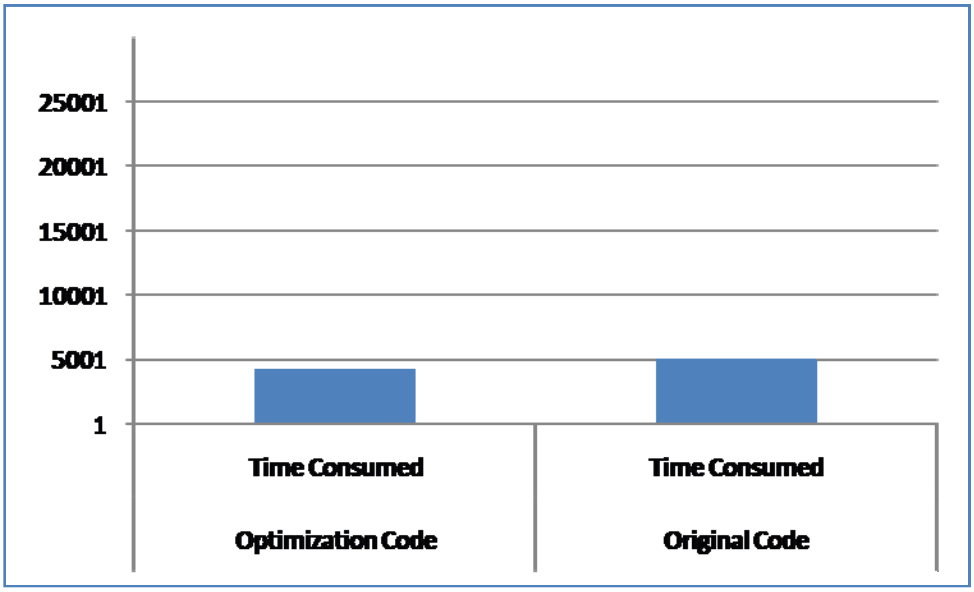

Table.2 9 個 synchronization events 的實驗數據長條圖

	Optimization		Original	
eq.3	Time Consumed	SYN-sequences	Time Consumed	SYN-sequences
run 1	6345ms	51	7500ms	51
run 2	6231ms	51	7432ms	51
run 3	6421ms	51	7612ms	51
Average	6332ms	51	7515ms	51

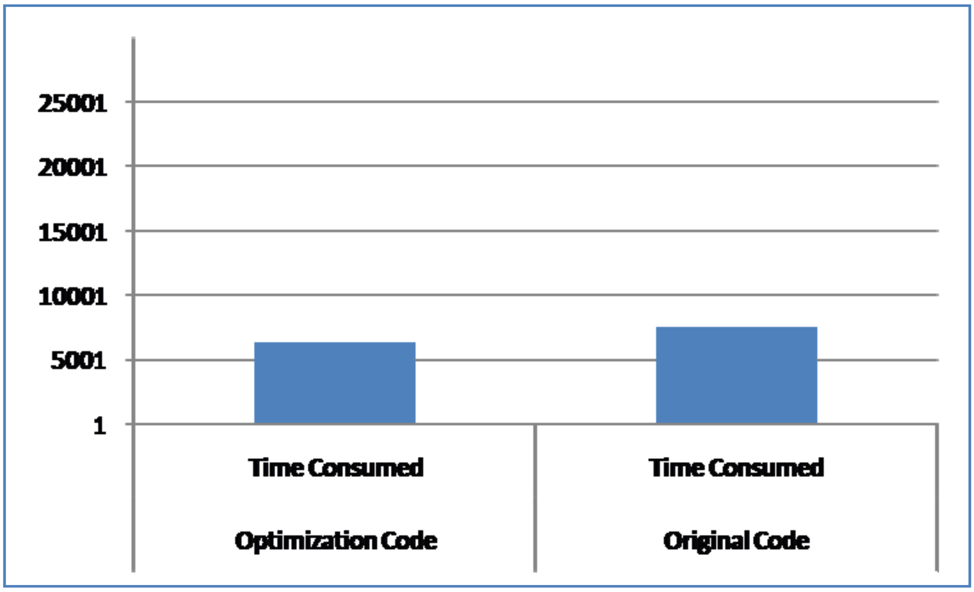

Table.3 10 個 synchronization events 的實驗數據長條圖

	Optimization		Original	
eq.4	Time Consumed	SYN-sequences	Time Consumed	SYN-sequences
run 1	9063ms	68	10218ms	68
run 2	9125ms	68	10122ms	68
run 3	8093ms	68	10321ms	68
Average	8760ms	68	10220ms	68

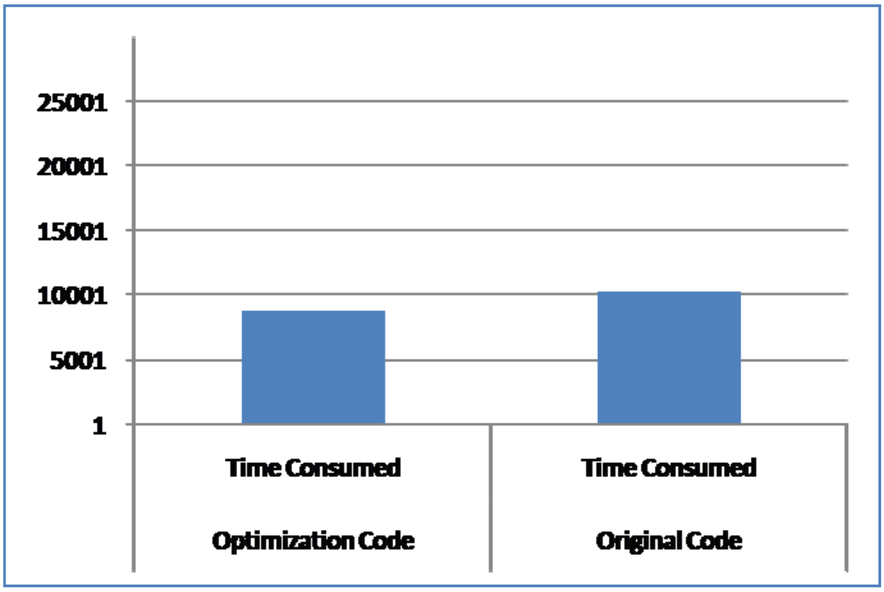

Table.4 11 個 synchronization events 的實驗數據長條圖

	Optimization		Original	
eq.5	Time Consumed	SYN-sequences	Time Consumed	SYN-sequences
run 1	11593ms	87	13154ms	87
run 2	11647ms	87	13453ms	87
run 3	11741ms	87	13233ms	87
Average	11660ms	87	13280ms	87

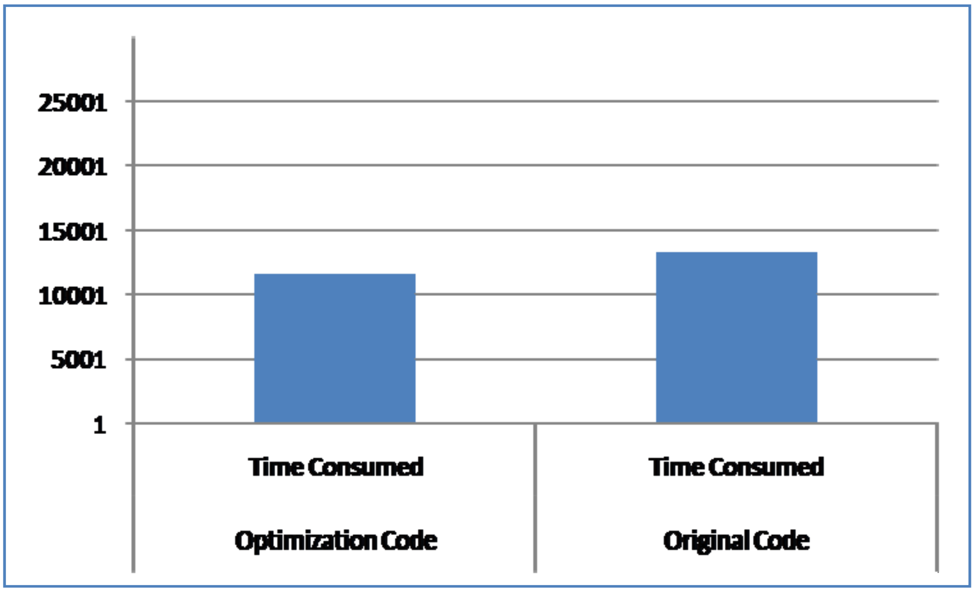

Table.5 12 個 synchronization events 的實驗數據長條圖

	Optimization		Original	
eq.6	Time Consumed	SYN-sequences	Time Consumed	SYN-sequences
run 1	13752ms	108	17015ms	108
run 2	13121ms	108	16987ms	108
run 3	13400ms	108	17125ms	108
Average	13424ms	108	17042ms	108

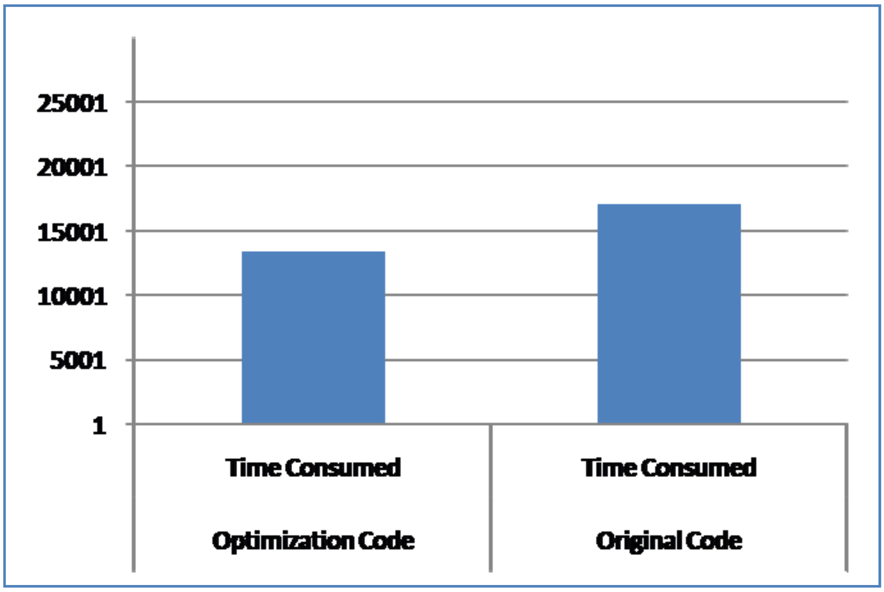

Table.6 13 個 synchronization events 的實驗數據長條圖

	Optimization		Original	
eq.7	Time Consumed	SYN-sequences	Time Consumed	SYN-sequences
run 1	16265ms	131	21626ms	131
run 2	15991ms	131	21345ms	131
run 3	16120ms	131	22100ms	131
Average	16125ms	131	21690ms	131

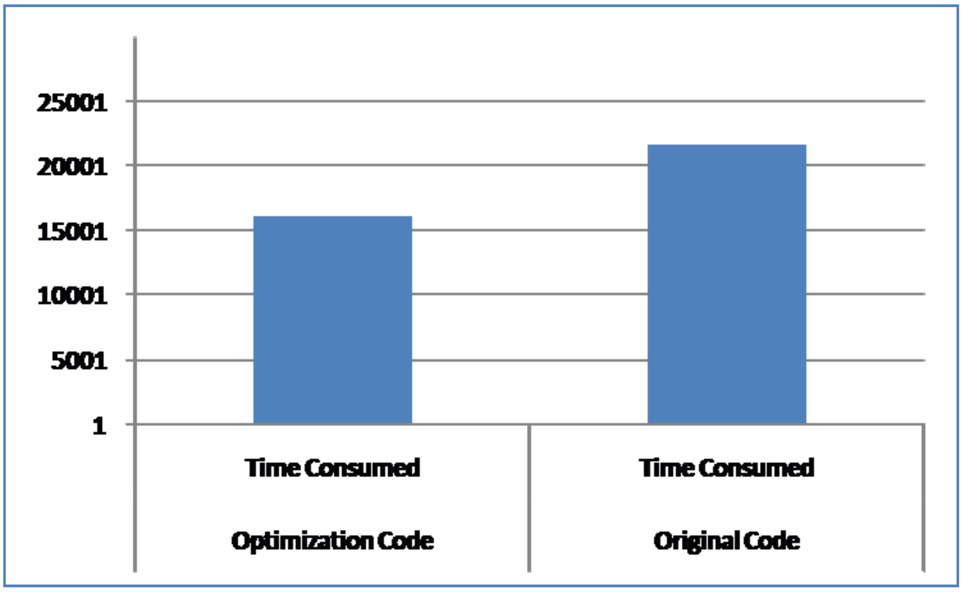

Table.7 14 個 synchronization events 的實驗數據長條圖

	Optimization		Original	
eq.8	Time Consumed	SYN-sequences	Time Consumed	SYN-sequences
run 1	19672ms	156	25938ms	156
run 2	19113ms	156	26078ms	156
run 3	19875ms	156	25797ms	156
Average	19553ms	156	25938ms	156

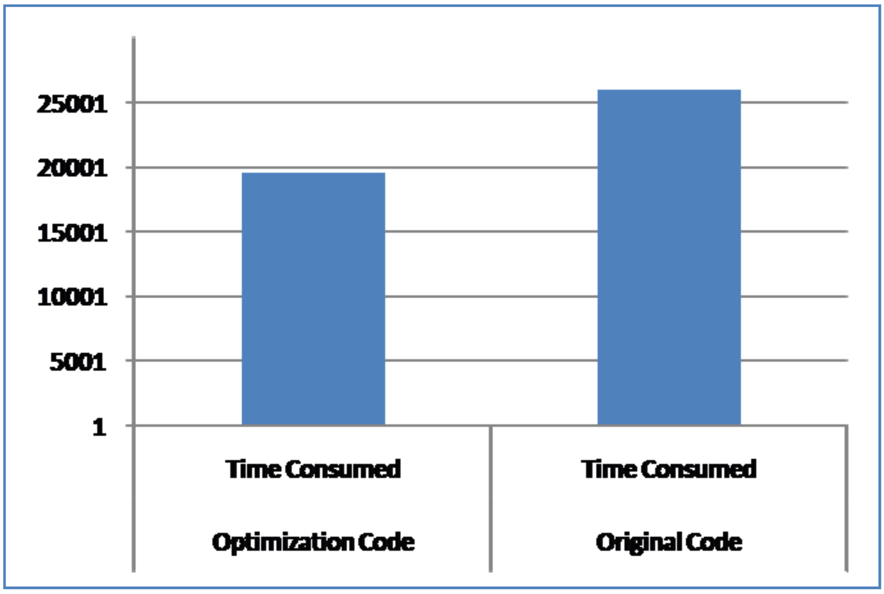

Table.8 15 個 synchronization events 的實驗數據長條圖

	Synchronization events	SYN-sequences	Optimization Time Consumed	Original Time Consumed
eq.1	8	21	2469ms	2025ms
eq.2	9	32	4230ms	5030ms
eq.3	10	45	6332ms	7515ms
eq.4	11	60	8760ms	10220ms
eq.5	12	77	11660ms	13280ms
eq.6	13	96	13424ms	17042ms
eq.7	14	117	16125ms	21690ms
eq.8	15	140	19553ms	25938ms

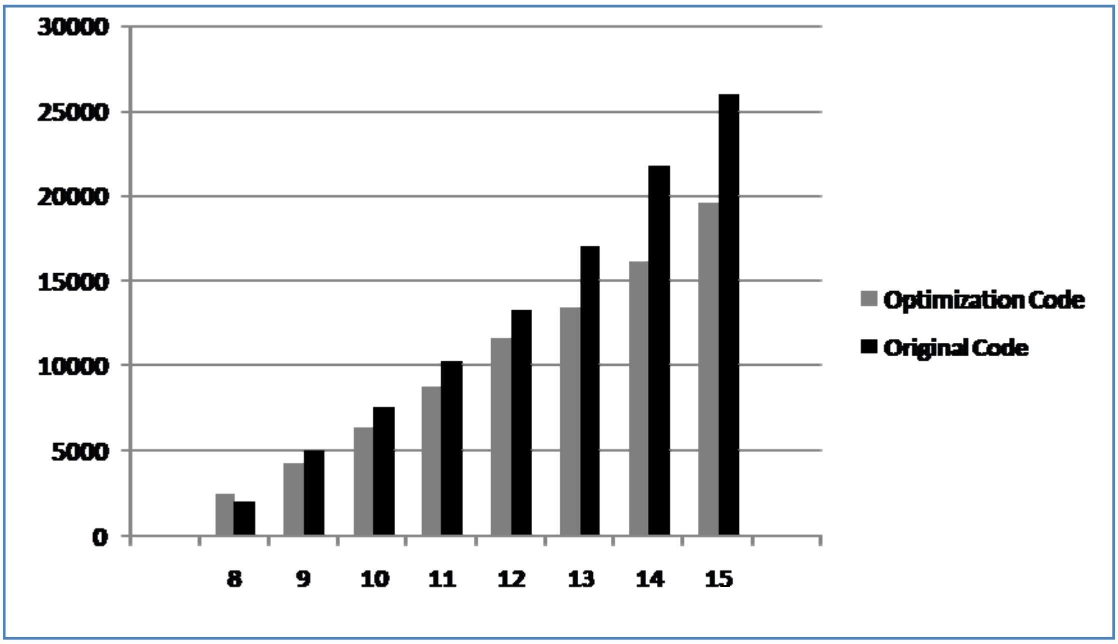

Table.9 增加 synchronization events 效能長條圖

我們將這八個實驗結果總結起來放到 table 9 分析，很明顯在 synchronization event 數目差不多在 9 個以上，由實驗數據可以看出 optimization code 已經穩穩的提高效能了。我們在以 SYN-sequences 的角度去看，在 table10 我們會由實驗得知當程式也大概是在產生 36 個以上的 SYN-sequences 的時候，會開始明顯的有效率

許多，我們在用圖表的方式呈現，如 table10。

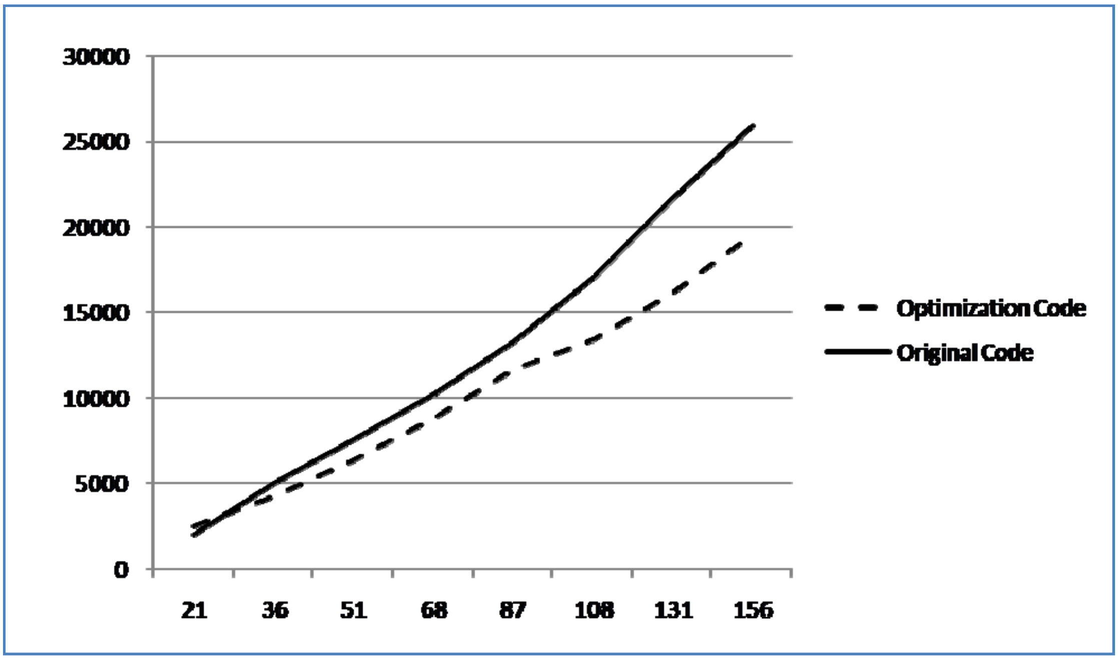

Table.10 SYN-sequences 數量影響效能表

2. 迴圈的例子測試：

eq.9

Initial shared variables: a:=0, b:=0;
Initial local variables: i:= 0 , loopcount = n;

Thread0	
	While(i<loopcount) {
0	read(a) ;
1	read(b) ;
2	a = 13;
3	b = 14;
4	i++;
	}

Thread1	
0	a = 21 ;
1	b = 22 ;
2	read(a);
3	read(b);

Loopcount 代表一個迴圈數目，由人工輸入去跑我們在 thread0 裡面所要跑的圈數來達到我們測試的效果，我們實驗 n:=1~2，在 n=3 的時候，改進的系統已經有提高效能的成果了，所以我們就只舉例 n 到 3，且每個例子都跑五次 run 1~run 5。

	Optimization		Original	
loop(n=1)	Time Consumed	SYN-sequences	Time Consumed	SYN-sequences
run 1	2750ms	21	2437 ms	23
run 2	2735 ms	21	2547 ms	22
run 3	2600 ms	21	2564 ms	21
run 4	2455 ms	21	2522 ms	21
run 5	2734 ms	21	3062 ms	21
Average	2655 ms	21	2626 ms	22
	Optimization		Original	
loop(n=2)	Time Consumed	SYN-sequences	Time Consumed	SYN-sequences
run 1	12115ms	79	11958ms	78
run 2	12046 ms	76	14843ms	79
run 3	12144 ms	79	12328ms	78
run 4	12211 ms	80	13266ms	80
run 5	12031 ms	78	12343ms	79
Average	12103 ms	78	12948 ms	79
loop(n=3)	Time Consumed	SYN-sequences	Time Consumed	SYN-sequences
run 1	23328ms	129	25016ms	139
run 2	24313ms	132	24547ms	138
run 3	24063ms	134	25101ms	140
run 4	23110ms	139	24987ms	137
run 5	24547ms	137	24555ms	136
Average	23872 ms	134	24841ms	138

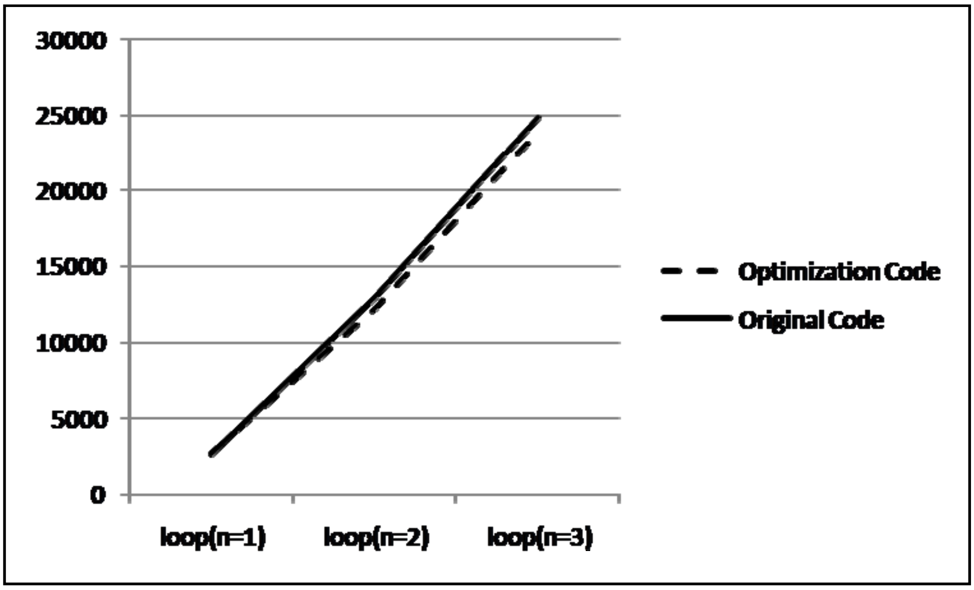

Table.11 Loop 例子的實驗數據表

3. Peterson's Algorithm ：

我們利用作業系統課本[2]裡面的有名例子之一，來去跑 reachability testing 系統，

我們程式的例子如下：

Eq.10

Initial shared variables: trun:=0, flag0:=0,flag1:=0;

Thread0	
0	flag0:=1;
1	turn:=1;
	do{
2	read(flag1);
3	read(turn);
	}while(flag1==1&& turn!=0)
	C.S.
4	flag0:=0;
	R.S.

Thread1	
0	flag1:=1;
1	turn:=0;
	do{
2	read(flag0);
3	read(turn);
	}while(flag0==1&& turn!=1)
	C.S.
4	flag1:=0;
	R.S.

可以大概知道一個執行緒裡面會有五個 synchronization events，program counter 0 和 1 的 write synchronization events，和 do while 迴圈裡面的兩個 read synchronization events，因為這兩個共享變數要去作判斷的動作，所以會先作讀取的動作，這裡有可能會有 busy waiting 的狀態，這個問題可能就會依靠 dynamic effective testing 去作分析，我所關注的是 optimization code 和 original code 在跑這個程式是否有提高效能的狀況，實驗的數據如下：

	Optimization Code		Original Code	
Peterson's	Time Consumed	SYN-sequences	Time Consumed	SYN-sequences
run 1	1563ms	5	1203ms	4
run 2	1594ms	5	1265ms	4
run 3	1414ms	4	1375ms	4
run 4	1594ms	5	1401ms	5
run 5	1875ms	5	1234ms	4
Average	1608ms	5	1296ms	4

Table.12 Peterson’s Algorithm 例子的實驗數據表

很明顯地，因為在 prefix-base replay 和 race analyzsis 中間有經過 state analysis 的分析，所以我們在這個例子所產生的 SYN-sequences 會變比較少一點所以沒辦法很明確的看到系統效能有提高許多， 雖然效果沒有提升，此類的問題很明顯就是程式的例子並不夠大，所以所跑出的 SYN-sequences 並沒有很多，所以再跑我 storing state 的方法會比較沒有省到效能的感覺，但是可以看出這個例子是可以去跑我所修改過的程式碼。

5. 討論和未來工作

因為在目前實驗室裡面 reachability testing 系統中，要處理在並行程式可能有無窮迴圈或忙碌等待迴圈(busy-waiting loops)時，reachability testing 可能沒辦法執行結束的，所以 reachability testing 目前已經是往 dynamic effective testing 發展了，如圖 26，再送去作 race analyzer 的時候，會再經過 state analyzer 這個階段，這裡最主要的就是去找出 state invariant 的動作，來處理一些讓 reachability testing 沒辦法結束的問題。

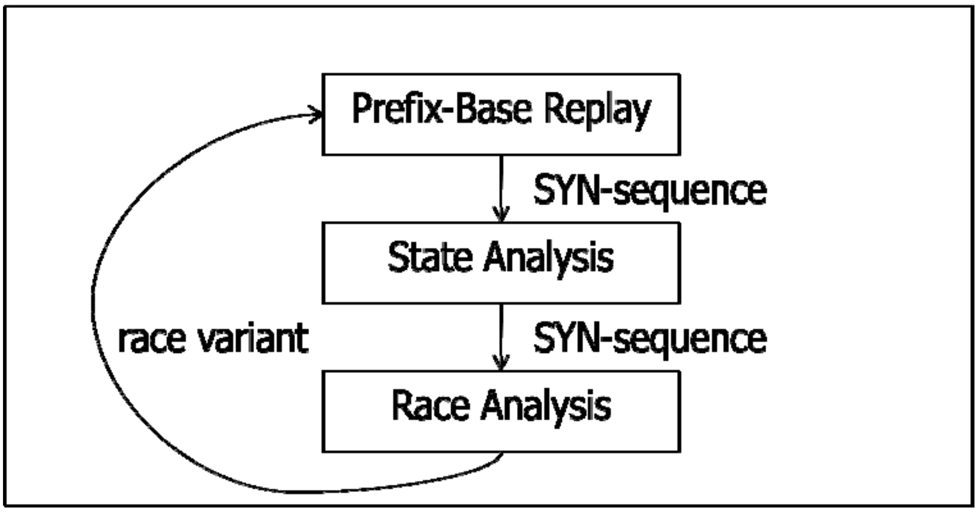

圖.26 Dynamic Effective Testing 新增的架構圖

將來的工作，也是希望可以和 dynamic effective testing 作結合，也可以把我們的 store state 來當作是一些暫存器，可以儲存一些 state analysis 需要用到資料的內容，也可以利用 store state 的觀念來一並的提高此系統的效能。

6. REFERENCES

[1] G. H. Hwang, K.C. Tai, and T.L. Huang, "Reachability Testing: An Approach To Testing Concurrent Software", Int'l J. Software Eng. and Knowledge Eng., vol. 5, no. 4, pp. 493-510, Dec. 1995.

[2] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne,"Operating System Concepts", John Wiley & Sons, ISBN: 0471417432, 6th edition (June 26, 2001).

[3] K.C. Tai and Richard H. Carver,"Testing of Distributed Programs", Chapter 33 in Parallel and Distributed Computing Handbook, editor A. Y. Zomaya, McGraw-Hill, 1996.

[4] Gwan-Hwan Hwang, Sheng-Jen Chang, and Huey-Der Chu, "Technology for Testing Nondeterministic Client/Server Database Applications", IEEE Transaction on Software Engineering, Volume 30, Number 1, pp. 59-77, Jan., 2004.

[5] L. Peterson, "Myths About the Mutual Exclusion Problem", Information Processing Letters, Volume 12, Number 3 (1981).

[6] S. Y. Hsu and C. G. Chung,"A heuristic approach to path selection problem in concurrent program testing", Proc. 3rd IEEE Workshop on Future Trends of Distributed Computing System(1992) pp. 86-92.

[7] K. C. Tai, R. H. Carver, "Use of sequencing constraints for specifying, testing and debugging concurrent programs", Proc. 1994 Int. Conf. Parallel and Distributed Systems (December 1994) pp. 280-287.

[8] R. N. Taylor, D. L. Levine, and C. D. Kelly, "Structural testing of concurrent programs", IEEE Trans. Softw. Eng. 18,3(1992) 206-215,

[9] R. D. Yang and C.G. Chung, "Path analysis testing of concurrent programs",Information and Software Technology 34, 1(1992) 43-56.

[10] Richard N. Taylor, "A General-Purpose Algorithm for Analyzing Concurrent Programs", Communication of the ACM, Volume 21, Number 7, July 1978.

[11] Michael Young and Richard N. Taylor, "Combining Static Concurrency Analysis with Symbolic Execution", IEEE Transaction on Software Engineering, Volume 14, Number 10, October 1988.

[12] Charles E. McDowell, "A Practical Algorithm for Static Analysis of Parallel Programs", Journal of Parallel and Distributed Computing, 6, 515-536(1989).

[13] G.J. Holzmann. The Model Checker SPIN. IEEE Trans. Soft. Engin., Vol.23, No.5, pages 279–295, May 1997.

[14] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic

model checking: 1020 states and beyond. In IEEE Symposium on Logic in Computer Science, pages 428-439, 1990.

[15] JavePathFinder page, http://javapathfinder.sourceforge.net/.

[16] A. Pnueli. A temporal logic of concurrent programs.

[17] Jun Chen, Steve MacDonald,"Testing Concurrent Programs using Value Schedules",ASE’07.November 5-9,2007.

[18] Soot, http://www.sable.mcgill.ca/soot/