在 ns-2 網路模擬器上改進 ELP 的效能

Improving the Performance of the ELP Approach over the Ns-2 Network Simulator

研 究 生：何庭緯　　Student：Ting-Wei Ho

指導教授：王協源　　Advisor：Shie-Yuan Wang

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

June 2008

Hsinchu, Taiwan, Republic of China

中華民國九十八年六月

在 ns-2 網路模擬器上改進 ELP 效能

Improving the Performance of the ELP Approach over the Ns-2 Network Simulator

研 究 生：何庭緯　　指導教授：王協源

國立交通大學

資訊科學與工程研究所

摘 要

近年來，由於半導體的技術極限，提升處理器 (CPU)的運算速度是變的越來越難以達成，因此目前處理器製造商都開始走向多核心的架構來達到提升總體校能的目的，具有多核心處理器的桌上型與筆記型電腦也越來越普及，因而如何有效利用此一架構來增進程式的效能，便是一個相當重要的問題。事件層平行運算 (Event-Level Parallerism) ，簡稱 ELP，即是針對多核心架構而設計的網路模擬技術。本篇論文旨在提出一個新的方法來改進 ELP 的效能。在本篇論文中，我們將會分析 ELP 的各個重要部份，並根據結果來建立新的 ELP 架構。最後，我們將這新方法實作到 ns-2 網路模擬器上，並測量它在不同網路情形下的模擬效能。

關鍵字：ELP、ns-2、網路模擬器、多核心處理器、多執行緒、平行模擬

Improving the Performance of the ELP Approach over the Ns-2 Network Simulator

student：Ting-Wei Ho　　Advisor：Prof. Shie-Yuan Wang

Institute of Computer Science and Engineering

National Chiao Tung University

ABSTRACT

Multi-core computers have been ubiquitous in the current market. On such a computer, efficiently using the computing power of all cores (CPUs) to finish a task becomes an important and challenging issue. It is difficult for an application (including a network simulator) to automatically gain performance speedups on multi-core systems because the application process can only be run on a single CPU at any given time. To gain performance speedups, an application program needs to be made "multi-threaded" so that its threads can be run on multiple CPUs simultaneously. However, turning an application program to be "multi-threaded" is not trivial and does not necessarily achieve good performance speedups.

To solve this problem, an novel "Event-level Parallelism (ELP)" approach was proposed in [7]. In this thesis, we first evaluated the performances of the ELP approach proposed in [7], identified its drawbacks, and proposed a new architecture for the ELP approach. We studied and compared the performances of the original ELP architecture and our proposed new ELP architecture over the ns-2 network simulator. Our results show that our proposed new ELP architecture outperforms the original ELP architecture on simulation speedups in most of the evaluated network conditions.

Keywords: ELP, ns-2, network simulator, multi-core, multi-thread, parallel simulation

誌謝

在這兩年的研究生涯，首先我要感謝指導教授王協源老師與實驗室的學長、同學與學弟，給予了我許多的幫助，並讓我學到許多寶貴的實務經驗與研究方法，同時提升自己的學業與專業技能，學弟也在忙碌時幫忙處理實驗室的事務，和有關實驗室開發軟體的問題，讓我在最後能專心完成論文的撰寫。

感謝系上計算機中心所有的同學、學長與學弟，在我任職的這兩年間給予我不少幫助與指導，學會在計中任職所需的各項技能與經驗，不僅在這段時間熟悉到許多系統管理方面的專業技術，更讓我學習到更進階機房運作的相關知識。

最後我要感謝家人的支持，讓我能無後顧之憂的完成學業，並在我感到困惑、挫折時，能給予我鼓勵與支持，使我在求學階段能順利的度過各個關卡。

目　錄

表目錄

圖目錄

Chapter 1 Introduction

近年來，提升單顆CPU的處理速度變的越來越難以達成，因此目前市面上的CPU都開始走向多核心的架構來達到提升總體效校能的目的，具有多核心處理器的桌上型與筆記型電腦也越來越普及，因而如何有效利用此一架構來增進程式的效能，便是一個相當重要的問題。

對於原本執行在單顆核心的應用程式當中，要同時使用多核心是件困難的事，最主要的原因是因為這些程式在執行時都只有用到一顆CPU的資源，完全沒有考慮到在多核心下不同核心之間的競爭問題，為了使其能在多核心的CPU上能獲得效率提升，勢必需將程式改寫成「多執行緒」(multi-thread)的架構，使的不同執行緒可以使用不同的CPU資源做運算。

隨著網路環境的日益複雜，模擬一個網路環境所需的時間也越來越久，因此有許多學者希望可以透過多核心的資源來達到模擬效能的提升，其方法主要分為兩大類--保守法和樂觀法，無論那一種方法，使用者都必需修改原有的模擬器程式，使其分割成數份可交由不同CPU同步執行的程式，以避免錯誤的模擬結果。保守法在實作起來較簡單，不過一般而言也難有太大的效率提升，而樂觀法則可以取得較好的效率提升，可是實作上也更為複雜。

而平行模擬目前並不流行，主要原因有兩個：一是目前現有的模擬器做完平行化後效率提升都不好，或是只支援特定幾種case，第二個原因是使用者若不了解平行模擬的概念和做法，沒有將要模擬的網路分割好，就沒辦法得到好的效率。

對大多數的使用者來說最希望的應該就是能夠不用動任何設定，就能夠提升模擬效能並維持模擬的正確性。因此在這篇論文中，我們提出了新的「事件層平行模擬方法」(Event-level parallelism)方法，它不會改變原有循序執行的方式，但同時亦保有平行運算的能力，其想法源自於應用於計算機架構和編譯器的「指令層平行方法」(instruction-level parallelism)，並不需要考慮到上層應用程式的特性，只需考慮兩個指令是否有相依關係，藉此送到不同的CPU來提升效能，在傳統的平行模擬中，需要將網路分割，並進行虛擬時間的同步，而且分割不好的話還會造成各CPU的loading

分擔不平均，使模擬效能低落。而用 ELP 的話，使用者只需要照原本的方法來操作，模擬器就會自動去找尋安全事件並送到 worker thread 去執行。

本篇論文組成如下所述，在第二章中將呈現傳統的平行和分散式模擬、介紹 ns-2 模擬平台、以及在 ELP 中如何判斷二個事件互為安全事件，第三章將簡單介紹舊有的 ELP 架構，第四、五章將分析 ELP 的架構和理想上的效率提升上限，在第六章中將根據分析結果改進架構，第七章則是改進後的效能呈現，最後則是對事件層平行方法未來的擴充性和結論。

Chapter 2 Background

在此章節中，我們將簡單介紹所使用的網路模擬器平台 ns-2，傳統的平行和分散式模擬方法，以及 ELP 在實作上的基本概念。

2.1 The ns-2 Network Simulator

ns-2，是 Network Simulator－Version 2 的簡寫。在 1989 年首先由 REAL Network Simulator 改版而來，當時由 VINT Project 這個團體所維護，現在由 SAMAN 和 CONSER 負責。它是一種以 C++和 OTcl 寫的非連續-事件引發 (discrete-event driven)及物件導向 (object oriented)的網路模擬器，提供了在有線或無線的模擬環境上跑 TCP、UDP、或 Routing protocols。ns-2 的效能取決於它要處理的事件數，事件越多模擬的速度就越慢。

Figure 2.1 表示的是在 ns-2 裡一個具有三個節點的無線網路拓樸和其內部的架構，可看出 ns-2 是採用模組化的設計 (protocol module based)。如 Figure 2.2，每一種型態的網路都使用此架構來建構各種協定模組，因此當有新的網路出現後，只需實作其內部的協定模組，並更換進 ns-2 內，就可開始新網路的模擬和測試。

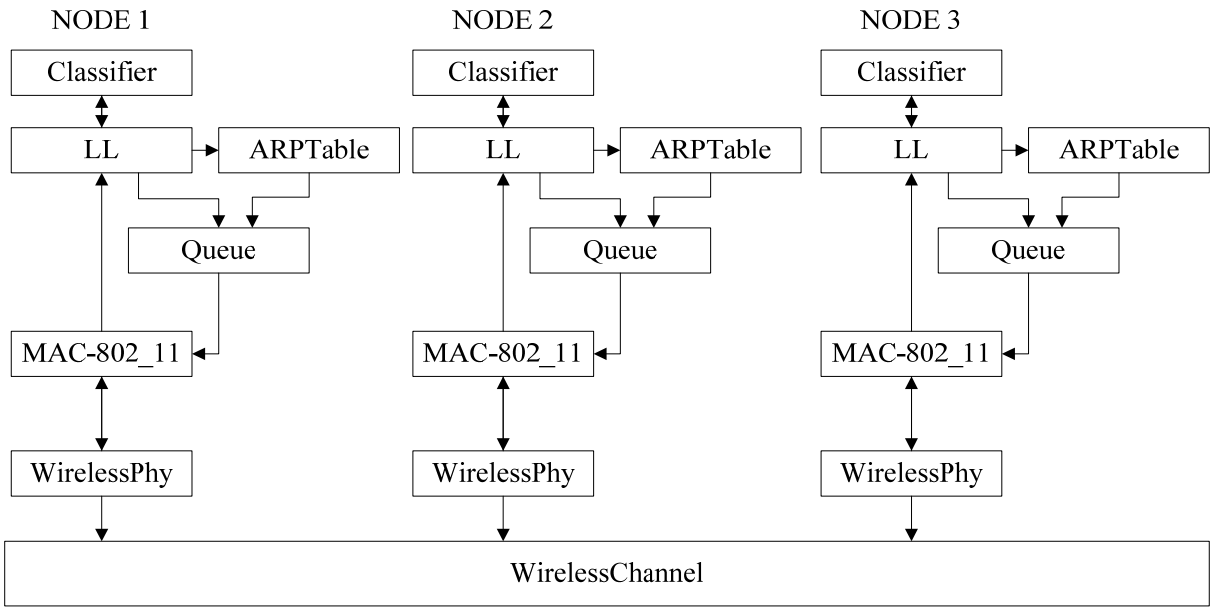
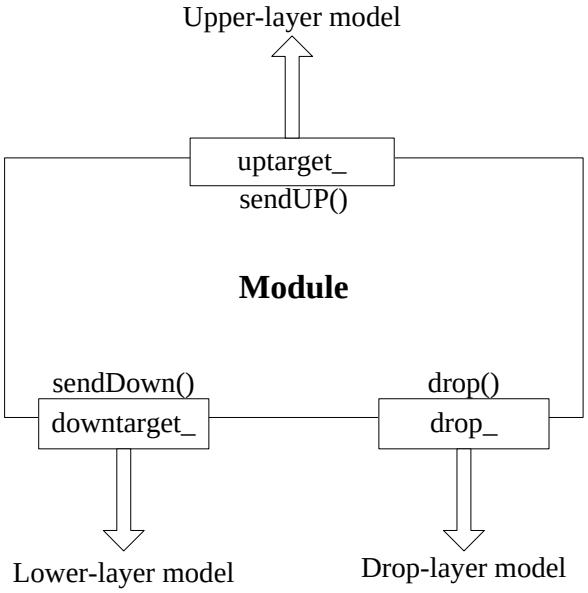


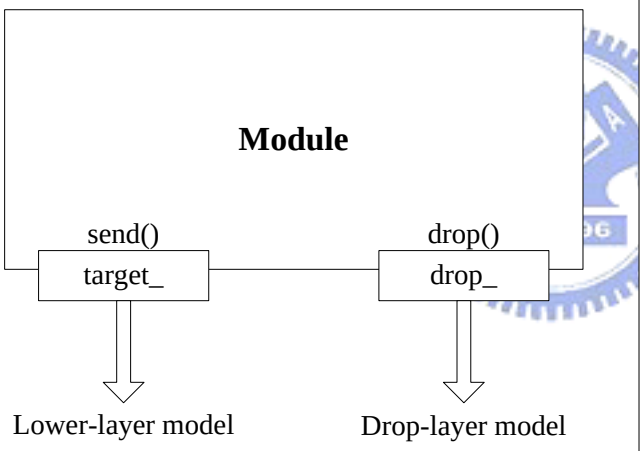
Figure 2.1 The module-based platform of the ns-2 network simulator




```
class BiConnector : public NsObject
{
public:
        NsObject* uptarget();
        NsObject* downtarget();
        virtual void drop(Packet* p);
        ...

protected:
        char *name;
        NsObject* uptarget_;
        NsObject* downtarget_;
        NsObject* drop_;
        ...
};
```

(a) The Connetcor-model skeleton




```
class Connector : public NsObject
{
public:
        virtual void send(Packet* p, Handler* h);
        virtual void drop(Packet* p);
        ...

protected:
        char *name;
        NsObject* target_;
        NsObject* drop_;
        ...
};
```

(b) The bi-connetcor-model skeleton

Figure 2.2 The module skeleton used in the ns-2 network simulator

ns-2 在模擬送封包時，是採用 discrete event 的方式，每一個動作都是由一連串的連續事件來完成。當某個事件在處理時，模擬器會將時間改成事件的時間，並改變系統的狀態。因此所有的事件所採用的時間是虛擬的，和真實世界中所用的時間不同。

2.2 Related Work

現在由於網路越來越大，要模擬一個網路行為所需的時間也越來越長，因此為了達到縮短模擬時間的效果，如何運用多臺電腦或多核心CPU來加速網路模擬的方法也紛紛被提出，在[1]中，作著用Parallel discrete event simulation (PDES)來改進ns-2網路模擬器[2]的排程機制，使它能平行運算事件，雖然效能上有提升，不過只能應用在OSI-7 layer架構的網路環境下。在[3]中，作者著重在無線網路下效能的提升，並在文章中提出新的技術，他從三個不同層面尋找可以同時運算的事件，分別是 1) event level, 2) node level, 3) group level。不同層面由於lookahead大小不同，找到可平行執行的事件也不相同，不過仍需要將原有循序執行的網路模擬器切割成不同的邏輯程序「logical process (LP)」，不同的區塊透過IPC的方式交換訊息，在區塊完成自己執行的事件後，都必須等到所有區塊都完成事件才能繼續執行新的事件，這是為了避免沒發現有訊息在傳遞中，使的判斷可平行執行的事件發生錯誤，而造成不正確的模擬結果，因此要使用global barrier的方式來暫停已完成的LP，這樣便限制了效能的提昇，另外，使用者仍需具備平行模擬的概念，才能適當分割模擬網路使不同CPU的負載相等，達到最佳的效能。

在[4]中，作者提出GloMosim來支援大型無線網路的平行模擬，它是使用了PARSEC[8] (C-base的平行模擬語言)來開發的函式庫。在[9]中，作者發表了Ted/GTW，Ted是一個小的程式語言拓展，將要模擬的網路對應到可平行模擬的核心Georga Tech Time Warp (GTW)，這個方法在Lookahead值大時其拓展性好，可是在Lookahead值小時，其效能卻比循序模擬差很多，因此在[10]中，為了解決在Lookahead值小所造成的效率不佳問題，作者提出了在LP內進行平行化的作法。在[11]中，作者提出了可拓展的模擬框架 (SSF)以及它的平行版本 (DaSSF)，採用C++與Jave來開發，它定義了五個基本的物件與SSF的API，讓使用者可發展自己的協定模組。在這些方法中，有些要使用者修改要模擬的程式使其能平行模擬。有的要使用者了解這些方法使用的程式語言或函式庫來發展自己的通訊協定模組，並要妥善的切割模擬網路到不同的LP，使其達到良好的效能提升。

在[15]中，作者採用分散式平行模擬發表了Aurora方法，它是設計用來操作大量的電腦來模擬網路。Aurora在電腦之間的通訊量大，因此適合用在client端有大量計算

的應用程式。就像是其它保守法的平行網路模擬，Aurora是把要平行模擬的程式組成LP。不過和保守法不同的地方是它的LP和CPU是一一對映，且用peer-to-peer的方式做模擬時間的同步，並採用master/worker的架構。在Aurora中，LP和相關的資料結構都包成「work units」，並由master分配給worker來平行執行。Aurora用的是集中式的方式來同步，決定那個LP(work unit)中的事件在client中可以被平行執行。

在 Aurora 中，有幾點和我們的 ELP 方法不同。首先，在 Aurora 中需要將模擬網路分到各個 LP 中，每個 LP 都有自己的模擬時間。第二，因為採用保守法的關係，Aurora 需要同步 LP 中的模擬時間，因此當 lookahead 值小的話，模擬速度會比循序模擬慢。而在 ELP 架構中，因為不需要切割模擬網路，也不需要同步模擬時間，所以即使在 lookahead 小的情況下，模擬速度也不會變慢。第三，Aurora 是跑在分散式的電腦上，而 ELP 是跑在多核心的機器上。第四，在 Aurora 上是以 LP 為最小的工作單位，ELP 則是以 event 為最小的工作單位，因為比 Aurora 劃分的更細，因此可以更好的平衡負載。第五，Aurora 溝通所需的負擔比 ELP 大，因為在 Aurora 中，需要對和 LP 相關的資料結構和狀態進行 encode/decode，packed/unpacked，並藉由網路傳輸，而對照之下，ELP 則是用 light-weight threads 來避免這麼高的負擔。

2.2.1 Parallel and Distributed Simulation Overview

如何確保在多顆 CPU 下所運行的結果是正確的，是平行與分散式模擬最主要的問題。一般來說循序模擬會從事件串列中選取時間最小的來執行，但在多核心的架構下，因為可能會有多個事件同時被執行，所以常會造成有較大時間的事件反而先執行的情況發生，如果沒有判定哪些事件之間會互相影響，就會造成模擬結果的不正確，這種情況稱為「因果錯誤 (causality errors)」。

為了解決「因果錯誤」的問題，在平行與分散式系統中會將網路拓樸切出許多「邏輯程序」(Logical process)，它們彼此為互相獨立的個體，負責完成自己事件串列下的所有事件，為了確保每一個 LP 是在 time stamp non-decreasing ordering 的狀態下完成各自負責的模擬事件，LP 透過 IPC 交換帶有時間的訊息來獲得「local causality constraint」(Events within each logical process must be process in time stamp

order)，雖然在 local causality constraint 下可以保證不會有「因果錯誤」的發生，不過違背它也不一定會發生「因果錯誤」，這是因為不同的兩個事件可能是互相獨立的，即使不按時間前後關係完成也不會造成結果不同。

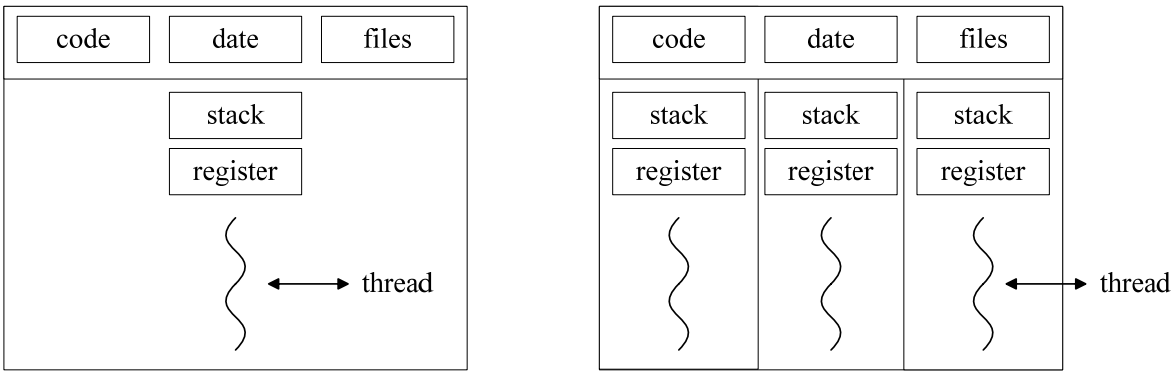
對於使用平行與分散式模擬來說，同步機制是非常重要的。為了達到同步，最有名的兩個方法為保守法和樂觀法。保守法的話，每一個 LP 都遵守 local causality constrain，它們將被阻擋直到確定在事件串列中的元素都是安全的才會開始執行。因為每一個 LP 都是照時間戳記的大小依序完成。所以不會有「因果錯誤」發生。而樂觀法則沒有完全遵守 local causality constrain，因此在使用時，必須配合額外的因果錯誤機制。

2.3 The Multi-threaded Evolution

一個跑在多核心下的多執行緒程式需要作業系統和使用者執行緒函式庫的支援。接下來我們將會介紹關於作業系統對多執行序提供那些支援，以及使用者執行緒函式庫的演進。

2.3.1 Thread Design in the Linux Kernel

執行緒是 CPU 排程的基本單元。一支行程最少擁有一條執行緒，由同一支行程所創建的執行緒會共用相同的記憶體空間。如圖 Figure 2.1 所示，每一個方框代表一支行程。如圖 Figure 2.1 (a)所示，如果行程只擁有一條執行緒，這樣的程式稱為單一執行緒應用程式，另一方面，如果行程內部擁有超過一條執行緒時，則這程式稱為多執行緒應用程式，如 Figure 2.1 (b)。



(a) Single-threaded Process　　(b) Multi-threaded Process

Figure 2.3 Mapping between Threads and Processes

在作業系統中，執行緒可分為兩大類，使用者執行緒 (user thread)和核心執行緒 (kernel thread)，使用者執行緒只在 user-level 的程式中運作，它主要是透過使用者執行緒函式庫創建，而核心執行緒只存在於 kernel space。

在早期的 Linux 作業系統中，一支多執行緒的應用程式，無論是被創造、處理、排程都是透過使用者執行緒函式庫來完成。多執行緒的應用程式裡每條執行緒在核心中都對應到同一份行程描述器結構，因此只要有一條執行緒正在存取 linux 核心資源時，其它的執行緒將被暫停。

而核心執行緒基本上與行程 (process)相似，許多行程上的特性，核心執行緒也具備。核心執行緒不同於使用者執行緒的地方，在於它可以獨立的被核心排程。在 Linux 2.6 核心以前，因為並沒有核心執行緒的設計，所以，過去行程與核心執行緒代表的是相同層面的架構，也就是說核心執行緒指的就是行程。從 Linux 2.6 核心開始，才開始支援核心執行緒，之後又被稱做輕量級行程 (LWP) 。

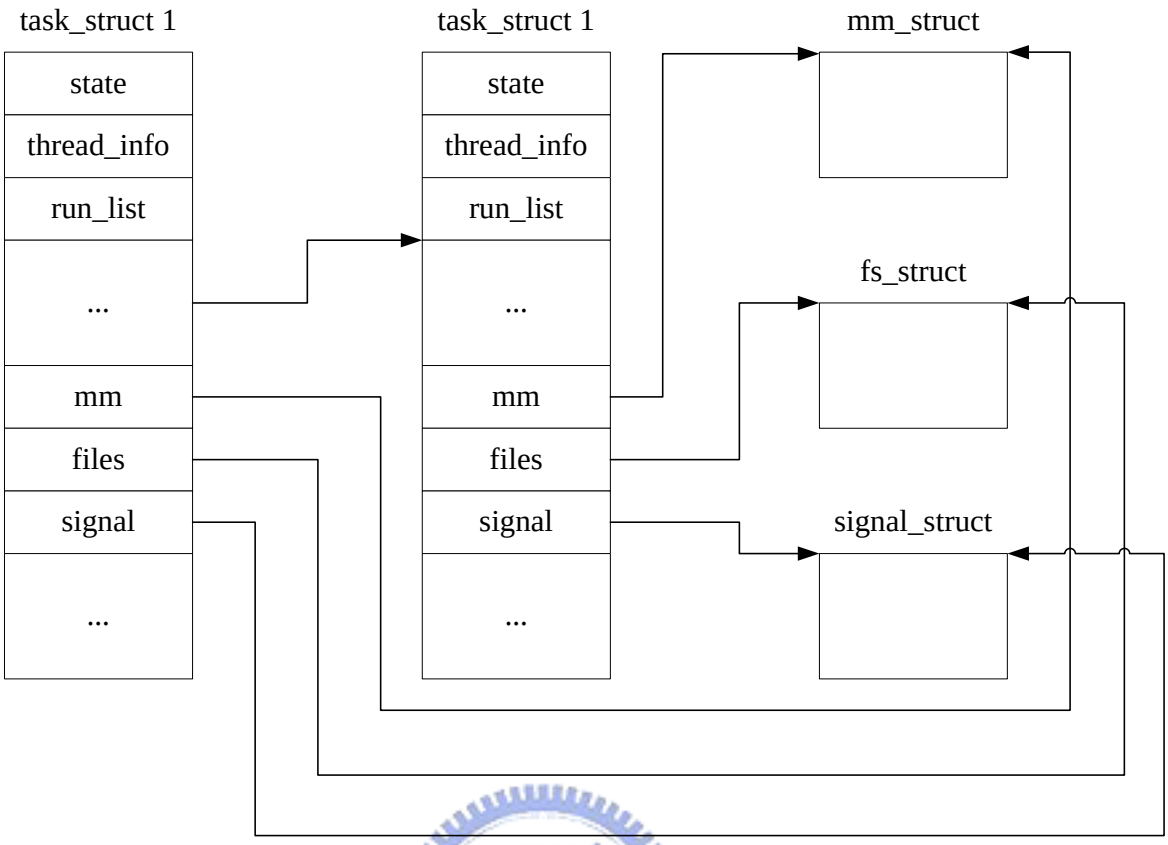


Figure 2.4 Light-weight Process Architecture

輕量級行程的架構如 Figure 2.4 所示，每一個輕量級行程，都有一塊獨立的行程描述器空間，但是輕量級行程與它的父行程仍共用相同的核心資源，這樣的設計使的當某個輕量級行程行程更改核心資源時，與其共用的執行緒將會立即反應。

一個最簡單的多執行緒應用程式設計，就是讓每一支使用者執行緒都關連到一個輕量級行程。由於一個輕量級行程可被核心獨立排程，因此每一條使用者執行緒在核心中都可被視為獨立的個體。

目前的 Linux 作業系統已支援 SMP 架構，可以為每一顆 CPU 排定一個輕量級行程，目前已整合到 Linux 2.6 的核心內。

2.3.2 Thread Design in the User-Level

在此節中，我們將從最早的 LinuxThread，到目前比較廣範使用的 NGPT 和 NPTL 等使用者執行緒函式庫提出它們的演進過程，以及彼此的優缺點。

2.3.2.1 The LinuxThread Library

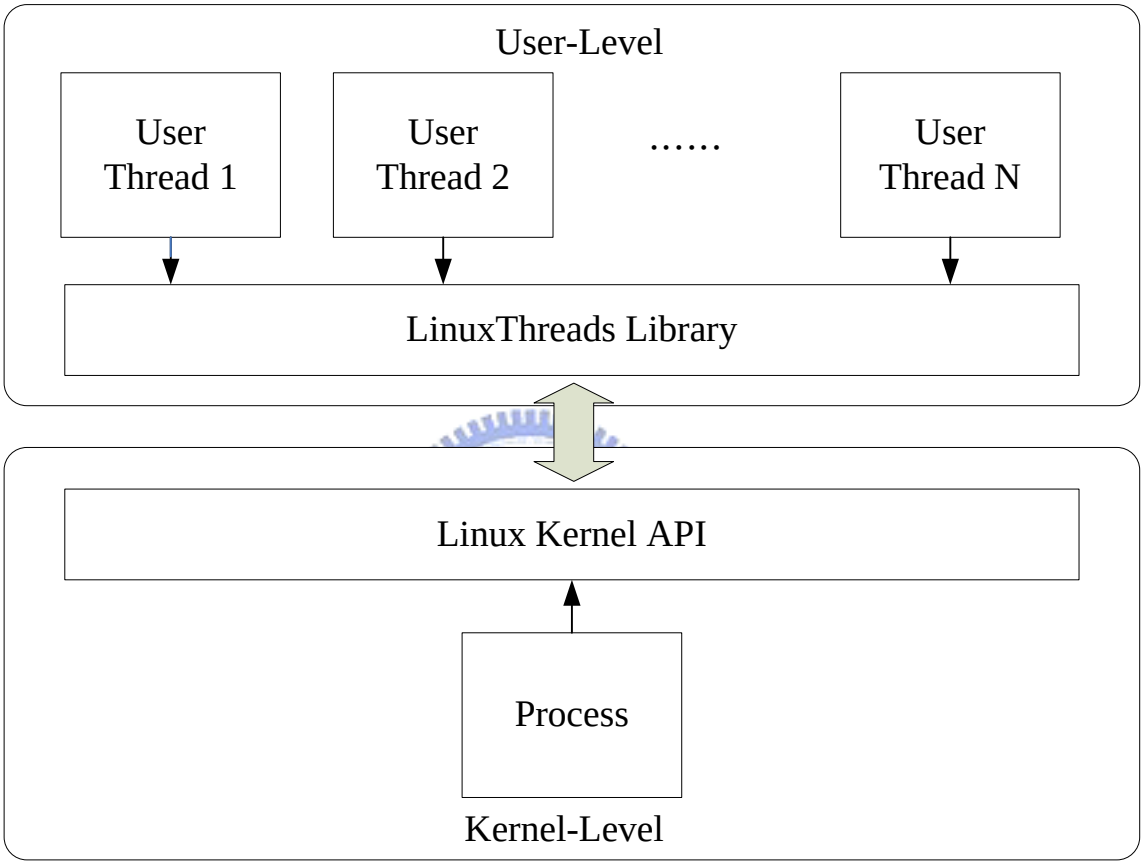


Figure 2.5 The Nx1 model of the LinuxThread library

LinuxThread Library 是最早被發展用來支援多執行緒應用程式的使用者執行緒函式庫。它是以 Nx1 的模型來做為多執行緒應用程式的架構，如 Figure 2.3 所示。Nx1 模型代表的是在同一支程式內，所有的使用者執行緒在核心中都只對應到一支行程，因此當某一條執行緒正存取核心資源時，其他執行緒也會一起暫停，使得這種做法並不能有效的發揮多執行緒的好處。而為了改進 LinuxThread Library 的缺點，因此才有後來的 NGPT 和 NPTL 的產生。

2.3.2.2 NGPT

NGPT (Next Generation POSIX Thread)是由IBM的工程師團隊發展的，在[5]中提到，NGPT的效能是LinuxThread的兩倍快，不過還是沒有比NPTL突出，因此最後NGPT在2003 年就被放棄繼續發展。

2.3.2.3 NPTL

NPTL (Native POSIX Thread Library)是由 Red Hat 發展的使用者執行緒函式庫，使用與 LinuxThread 相似的方法，藉由系統呼叫「clone」去完成多執行緒應用程式的實作，不同的是，NPTL 創建出來的是輕量級行程，而非一般的行程。因此 NPTL 並非完全是 user-level 的函式庫，它需要核心的支援，使的每一支在多執行緒應用程式內的使用者執行緒都可以獨立排程。

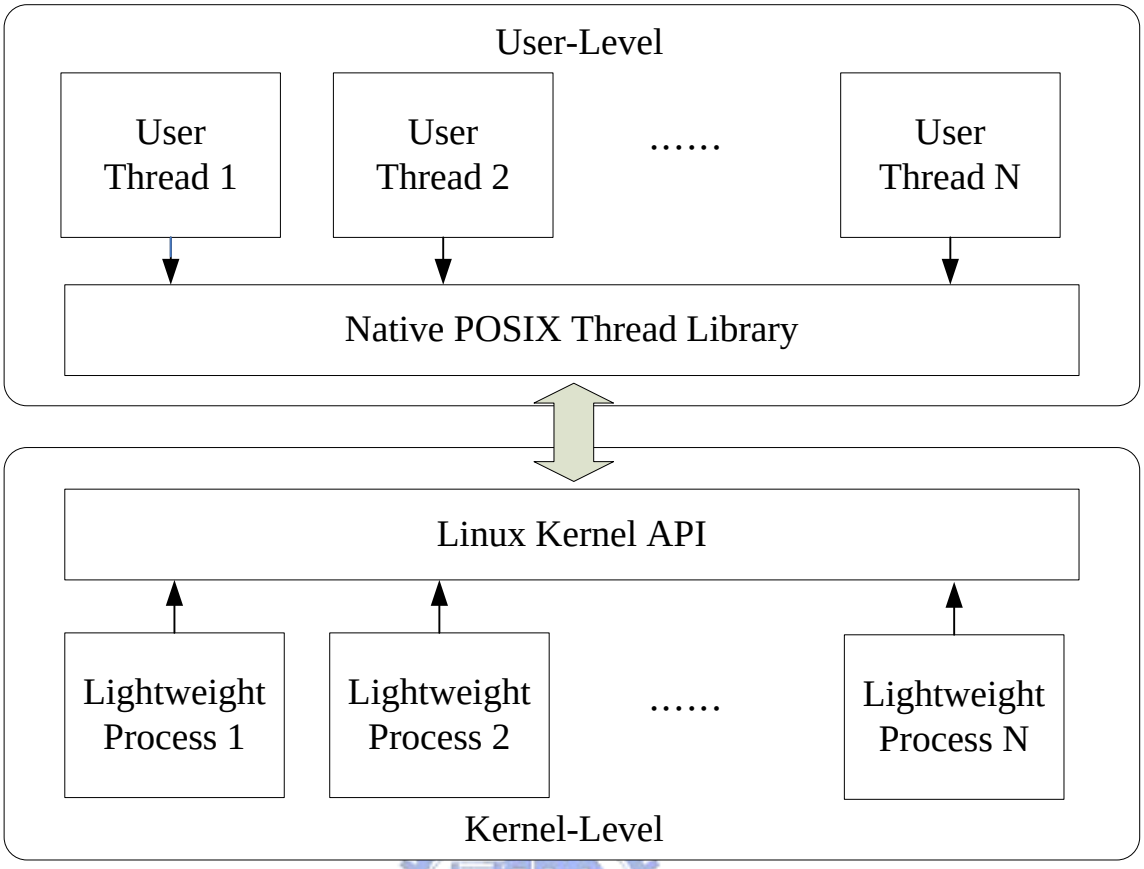


Figure 2.6 1x1 Multi-Thread Model in NPTL Library

NPTL 函式庫提供兩類不同的多執行緒應用程式的實作模型。第一種如 Figure 2.4 所示，對每一條使用者執行緒都提供一支輕量級行程來對應，稱之為 1x1 架構。使用這種架構的多執行緒應用程式在核心中都可以被獨立排程，因此每條執行緒都可同時存取核心資源。

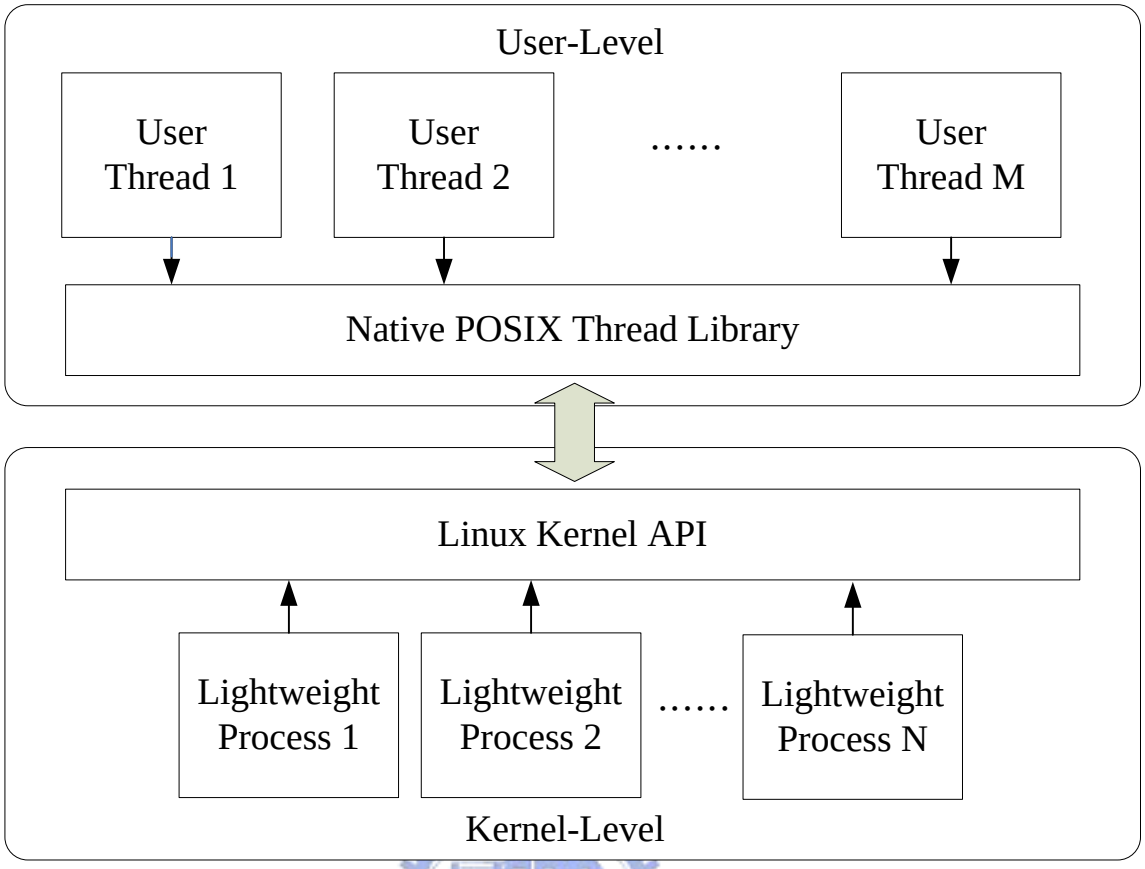


Figure 2.7 MxN Mutli-Thread Model in NPTL Library

第二種實作模型則是 MxN 架構，如 Figure 2.5 所示。它是將 M 條使用者執行緒對應到 N 支輕量級行程，因此只能在 user-level 下實作執行緒排程。由於 MxN 架構使得執行緒間的切換必需經由函式庫才能完成，造成其開發多執行序應用程式的複雜度大增。所以現今的 Linux 平台都採用 1x1 的架構來支援多執行緒程式。

目前有關多執行緒應用程式的開發，所使用的函式庫都已改為 NPTL。在 Linux 作業系統內，NPTL 已經成為 POSIX-compliant library 的標準，而且它已經整合進 GUN C library 內。

2.4 Event-Level Parallelism Approach

在平行化事件層網路模擬下，首先最要緊的是要先知道如何找出可以平行執行的安全事件，以下將會介紹我們所分析出來找安全事件的規則。

2.4.1 The Event Relationship

在這小節中，我們將說明如何找尋安全事件給 worker thread 執行。能否找出安全事件的關鍵取決於 lookahead 的大小，因此，我們簡短說明安全事件與 lookahead 的關係。假設一個網路模擬器正準備處理其時間為 T 的事件，而且與其它的 logical process 交換訊息得知因執行而產生的新事件最少要 L 時間後才會被執行，那麼落在 T+L 之間的所有事件都可與它同時執行，可被平行執行的事件就叫安全事件，且 L 即是指 lookahead。

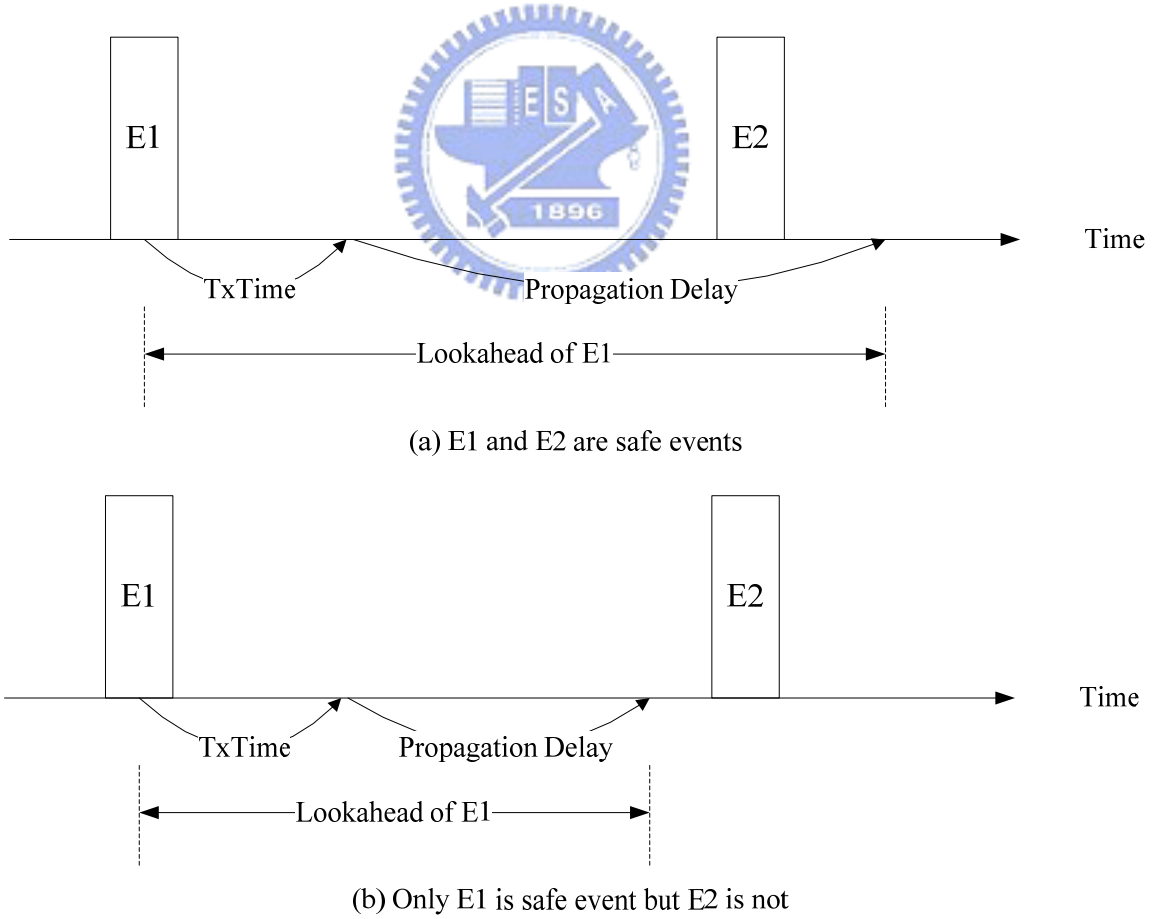


Figure 2.8 Determination of lookahead values

在網路模擬環境下，lookahead 可視為封包在鏈結 (link)上傳輸所花費的時間。這個值是訊號傳送時間 D 加上封包流出網卡所需的傳輸時間 Tx 的總合，其中在固定的兩個節點中 D 是定值，而 Tx 則會與封包大小、線路頻寬有關。如 Figure 2.8 (a)所示，假設目前模擬時間為 T1 且正在執行事件 E1，E1 為一個正從來源節點送到目的節點的封包，若目的節點有一事件 E2 會在時間 T2 時執行，且 T2 大於 T1，只要 T1+D+Tx > T2，則保證 E1 不會對 E2 造成影響，這兩個互為安全事件，彼此可以平行執行。相反的，如果 T1+D+Tx < T2，如 Figure 2.8 (b)所示，那麼在 E1 到達目的節點後，便會改變目的節點的狀態，可能會影響到 E2 的執行，因此為了確保 E2 執行時的正確性，E1、E2 便要照順序完成，故只有 E1 可為安全事件。

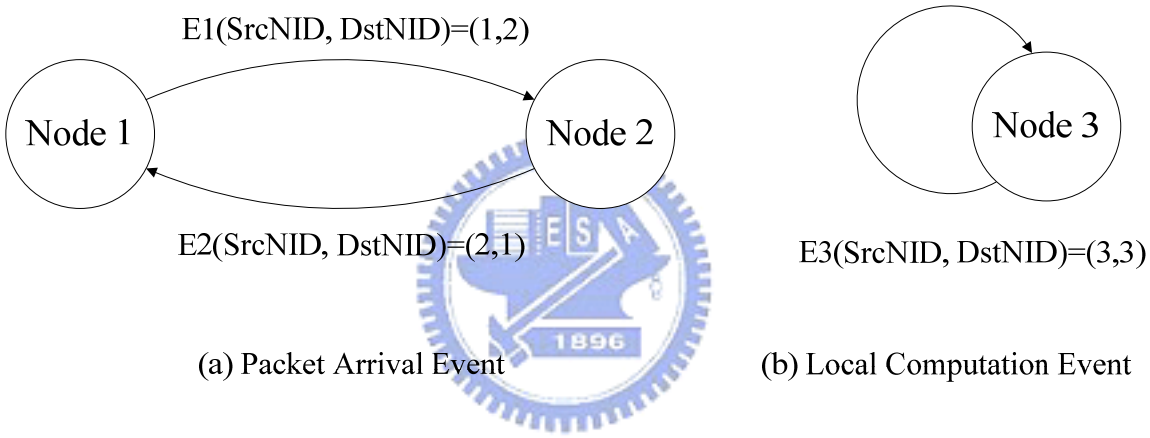


Figure 2.9 Two typical events

接下來我們將探討當兩個節點並不直接相連時，一個較早執行的事件 E1 會不會影響到另一個節點上較晚執行的事件 E2，並提出一套規則來檢驗它。在事件層平行模擬方法下，對每一筆事件都會額外儲存它是被哪個來源點 (SrcNID)所排程，以及作用在那一個目的點 (DstNID)上。舉例來說，假設有一個事件是表示某個封包從 Ni 送到 Nj 的行為，那麼事件的 SrcNID 將被設成 i 而 DstNID 則被設為 j，如 Figure 2.9 (a)。另一方面，若事件並非代表封包傳送的話，那麼這事件一定是本地端運算的事件，它只會改變執行這個事件所在節點的內部狀態，並不會影響到其它節點的運作。對一個本地端運算的事件而言，假設這個事件在 Ni 上執行的話，那麼它的 SrcNID 和 DstNID 都會被設成 i，如 Figure 2.9 (b)。在模擬過程中，這些值一但被設定後就不會再更改。

在 ns-2 網路模擬器的架構下，不同的網路型態都是由一連串的協定模組所組成，

每一個模組代表的是各個不同網路的運作行為。當要模擬一個從應用程式送出的封包時，上層模組先會排定一個事件，代表從應用程式產生出一筆新的封包。在這事件到達協定模組最下面的實體層模組之前，它仍在同一個節點上執行，並不會影響到其它節點的狀態，因此會被視為本地運算事件。最後則是由實體層模組來模擬一個封包在鏈結上的傳輸行為，送到目的端的實體層模組去。

在接下來的小節，我們將討論在有線網路上，如何判斷一筆事件的執行，是否會影響到另一筆在別的節點上執行的事件。

Rules	Condition
Rule 1	SrcNID1≠SrcNID2 and DstNID1≠DstNID2
Rule 2	SrcNID1＝SrcNID2 and DstNID1≠DstNID2
Rule 3	SrcNID1≠SrcNID2 and DstNID1＝DstNID2
Rule 4	SrcNID1＝SrcNID2 and DstNID1＝DstNID2

Table 2.1　Rules for Determining Safe Events

2.4.1.1 Packet Arrival Event

在這小節中，我們將以傳輸事件為主來探討。假設目前有兩個傳輸事件 E1 與 E2，E1 是由 SrcNID1 送至 DstNID1，E2 是由 SrcNID2 送至 DstNID2。比較來源點與目的點的異同，可得到四組不同的條件，如 Table 2.1 所示。

我們將使用 Figure 2.9 來解釋如何使用這四組條件來檢驗事件彼此是否互相獨立。在這張圖中，每一個節點都給定一個名字 Ni，I 代表的是節點的 ID，圖中的鏈結(link)則表示為 Lij，i 與 j 分別代表此鏈結兩端點的 ID。每一個節點都與其相鄰的節點透過鏈結相連，且每一條鏈結都各自擁有一份 output buffer。圖中，Ea、Eb、Ec、Ee、和 Ef 都是代表傳送封包的事件 (Packet Arrival Event)，而 Ed 則是代表本地運算的事件 (Local computation Event)。每一筆封包傳送事件使用箭頭指示符號表示它是由哪一個來源點送至哪一個目的點。不同於封包傳送事件，本地運算事件則是使用一個自旋箭頭代表來源點與目的點相同。

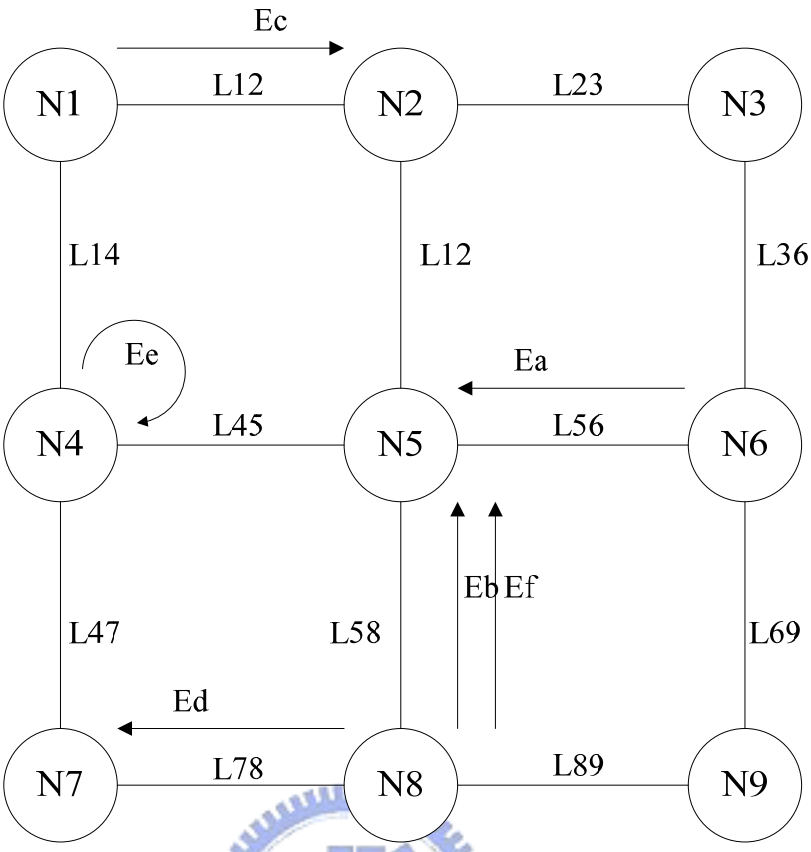


Figure 2.10 Example of Packet Arrival Events

符合條件 1：SrcNID1≠SrcNID2 且 DstNID1≠DstNID2 的事件只要 DstNIS1 到 DstNID2 的時間夠長，就有可能成為安全事件。以 Figure 2.9 的 Ea、Ec 為例，假設 Ea 被執行的時間小於 Ec，雖然 Ea 的目的端 N5 和 Ec 的目的端 N2 並不相同，可是若存在一條 N5 到 N2 的路徑，則 Ea 有機會影響到 Ec。

為了能夠確定一個在節點 Nsrc 上較早執行的事件 E1 是否會影響到在另一個節點 Ndst 上較晚發生的事件 E2，必需要計算從 Nsrc 到 Ndst 的最短路徑所花的時間。這是因為 Nsrc 也許會排定一個目的地為 Ni 的事件，當 Ni 收到後或許又會排定下一個事件，其目的地為 Nj，…，最後有可能 Nk 接受到這事件，並排定一個事件到 Ndst。如果 Nk 排定到 Ndst 的事件其時間小於 E2，那麼 E1 就會影響到 E2。

由上述說明可得知，要決定在N5 上的事件Ea是否會影響到N2 上的事件Ec，必須要計算從N5 到N2 的最短路徑所耗費的時間。它可以透過all-pairs shortest path Dijkstra's algorithm事先計算，並儲存所有節點間的最短路徑所花費的時間。假設事

先計算得到N5 到N2 所需時間為PLA_{52}，Ea的執行時間為Ta且Ec的執行時間為Tc。若$Tb+PLA_{52}>Tc$，則Ea不會影響到Ec，彼此可以平行的運算。

符合條件 2：SrcNID1＝SrcNID2 and DstNID1≠DstNID2 的事件，其驗證方法和條件 1 一樣。在此用Figure 2.9 的Eb與Ed來舉例，假設Eb被執行的時間小於Ed，如同條件 1 所描述的，假設從N5 到N7 最短路徑所花費的時間為PLA_{57}，Eb的執行時間為Tb且Ed的執行時間為Td，如果$Tb+PLA_{57}>Td$，那麼Eb不會影響到Ed，彼此可以平行的運算。

符合條件 3：SrcNID1≠SrcNID2 and DstNID1＝DstNID2 的事件都被視為不安全的事件。以 Figure 2.9 的 Ea、Eb 為例，假設 Ea 被執行的時間小於 Eb，因為一筆封包傳輸事件可能會改變接收點的內部狀態，所以很明顯的可以證明 Ea 會影響到 Eb，為了確保模擬結果不會出現「因果錯誤」，必需確定 Ea 事件完成後，Eb 才可開始執行。

符合條件 4：SrcNID1＝SrcNID2 and DstNID1＝DstNID2 的事件都不會是安全事件。以 Figure 2.9 的 Eb、Ef 為例，假設 Eb 被執行的時間小於 Ef。雖然一張網卡同一時間只能接收一個封包，不過當鏈結有較大的延遲且高頻寬的條件下，會使一個封包的傳輸時間小於鏈結造成的延遲，變成同時有許多封包在此鏈結上傳輸，這些封包就會有相同的 SrcNID 和 DstNID。很明顯的，這些封包到達接收端的順序必需被維持，所以它們彼此之間不應該同時執行，否則會造成結果的不正確。

2.4.1.2 Local Computation Event

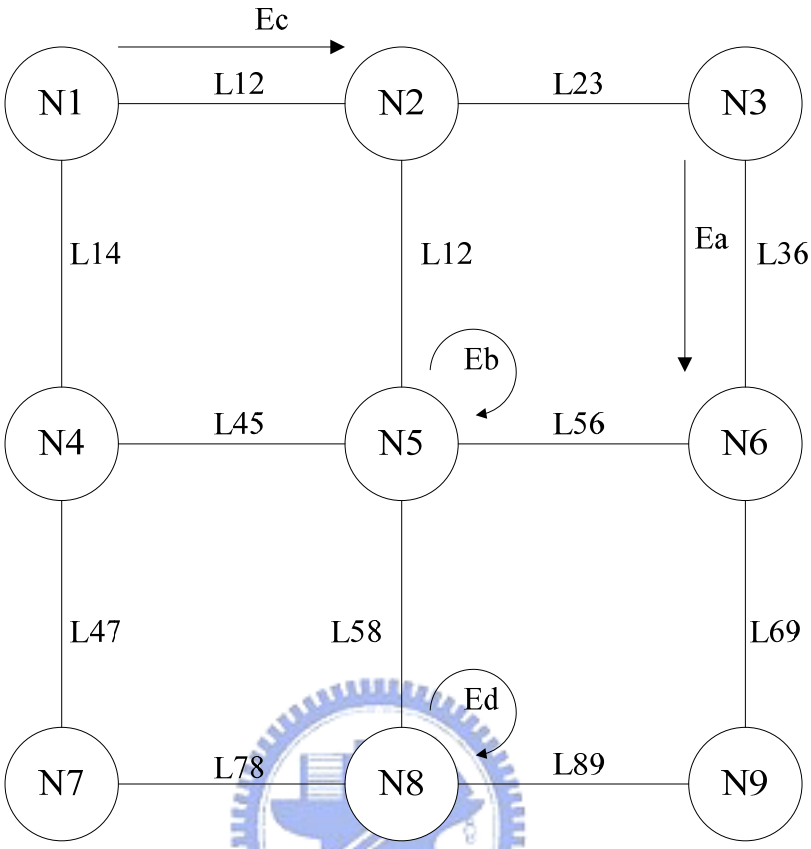


Figure 2.11 Example of Local Computation Events

在這章節中，我們將考慮有本地計算事件的情況，對於一筆本地運算事件，它的 SrcNID 與 DstNID 是相同的，因此只要比較這兩個值就能判斷是不是本地運算事件。

如同前一小節所提到的，無論 E1 和 E2 是何種事件，只要符合 DstNID1＝DstNID2 的事件都將會被視為不安全的事件。我們將使用 Figure 2.10 來探討 DstNID1≠DstNID2 的情況。Figure 2.9 和 Figure 2.10 是相似的，在圖中 Ea 和 Ec 代表的是封包傳輸事件，而 Eb 和 Ed 代表的是本地計算事件。

對於E1 和E2 都是本地計算事件時，以Figure 2.10 的Eb、Ed為例，假設Eb被執行的時間小於Ed，雖然Eb的目的端是N5，Ed的目的端是N8，只要中間存在一條從N5 到N8 的路徑，那麼Eb就有可能影響到Ed。相反的，若路徑不存在，則Eb無法影響到Ed的執行。假設這條路徑存在，如前一節所提到的，必需計算經過這條路徑所需的最小時間，在此

稱之為PLA_{58}。假如Eb執行的時間加上PLA_{58}大於Ed所執行的時間，則Eb與Ed都是安全事件，反之，則只有Eb為安全事件。

對於E1 是本地計算事件，而E2 是封包傳送事件的情況下，以Figure 2.10 的Ea、Eb為例，假設Eb被執行的時間小於Ea。Eb的目的端是N5，Ea的目的端是N6。如同前面所描述的，假設從N5 到N6 的最短路徑所花費的最小時間為PLA_{56}，如果Eb執行的時間加上PLA_{56}大於Ea所執行的時間，則Eb不會影響到Ea，反之Eb將影響Ea。

對於E1 是封包傳送事件的情況，而E2 是本地計算事件下，以Figure 2.10 的Ea、Eb為例，假設Ea被執行的時間小於Eb。Ea的目的端是N6，Eb的目的端是N5。如同前面所描述的，假設從N6 到N5 的最短路徑所花費的最小時間為PLA_{56}，如果Ea執行的時間加上PLA_{56}大於Eb所執行的時間，則Ea不會影響到Eb，反之Ea將影響Eb。

2.4.2 ELP for Wireless Network

在之前的章節，我們探討的都是有線網路，在有線網路的環境下，每條鏈結都是雙向的，因此封包從 Ni 送到 Nj 與從 Nj 送到 Ni 都是互相獨立的，並不會互相影響。但在無線網路的環境下，每一筆封包都是廣播的方式傳送，在傳輸範圍下的所有節點都會收到此封包的訊號，因此網路模擬器必須為所有在傳輸範圍內的節點排程一筆封包傳送事件。無線網路並不像有線網路，其封包從 Ni 傳給 Nj 與從 Nj 傳給 Ni 是會互相影響的，因為它們是共用相同的無線通道（wireless channel），所以在同一個節點上無法同時做收送的動作。

Table 2.1 的條件 1：SrcNID1≠SrcNID2 且 DstNID1≠DstNID2，在無線網路的環境下，必需要增加 SrcNID1≠DstNID2 和 SrcNID2≠DstNID1 的條件，才能確保找安全事件的正確性，這是因為無線網路不像有線網路上傳和下載是獨立的，兩邊會互相影響。在此使用一個簡單的例子說明，若有兩個節點 Na、Nb 分別發送 E1、E2 的傳輸事件給對方，試著觀察 E1 是否會影響 E2。假設 E1 的執行時間為 T1，E2 的執行時間為 T2 且 T1 加 lookahead 的時間大於 T2。原本在循序模擬下，應為 Na 發送出封包，而忽略 Nb 的封包，這時若只用有線所設的條件來判斷，便有可能讓 E2 先做，而使的 Na 因為收到

Nb 的封包而暫停發送，使得模擬結果不同。所以在無線網路中增加 SrcNID1≠DstNID2 和 SrcNID1≠DstNID2 的條件是必要的。

為了將事件層平行模擬方法應用在無線網路，像是點對點網路（ad-hoc network）上，鏈結的延遲便非常重要，因為 lookahead 的大小將會影響到安全事件的多寡。因此在無線網路裡，首先必需決定每一個節點的傳輸範圍內含蓋哪些節點，如果 Ni 在 Nj 的範圍內，那就像有一條有線的鍊結從 Ni 連到 Nj。無線網路的 lookahead 指的是無線鏈結上的延遲（訊號從 Ni 送到 Nj 所需的傳遞時間)加上封包從網卡出去的傳輸時間。重覆執行上面的敘述，在檢測完所有節點間的關係後，我們可以將無線網路視為有線網路來處理。所以在有線網路中用來檢測安全事件的條件，在無線網路中也可適用。對於一個會移動的點對點網路（ad-hoc network)來說，由於移動的關係會導致網路拓樸無時無刻在改變，相對的也改變了節點之間的延遲，因此為了模擬的正確性就必須要隨時更新 lookahead 的值。所以本篇論文並不考慮點對點的網路。

Chapter 3 原有的 ELP 架構

在這個章節中，我們將介紹原本所設計的 ELP 整體架構，如 Figure 3.1 所展示，並以這個架構為基礎來做之分析。

3.1 The Overview of the Original ELP Architecture

在原本的 ELP 架構中，主要由四個元素所組成，下面將分別描述它們在 ELP 中所辦演的角色。

- Master Thread

  主要的任務是負責尋找目前有那些事件是可在不同 thread 上平行執行，且不會造成「因果錯誤」的發生。

- Worker Thread

  執行由 master thread 找到的安全事件，並把執行過程中所產生新的事件放到 global event queue 內。

- Global event queue

  所有執行過程中所產生的事件都會先放到這裡，等待 master thread 處理，可能會被 master thread 和 worker thread 存取。

- Safe event queue

  存放所有由 master thread 找到的安全事件，等待 worker thread 來執行，safe event queue 也有可能會被 master thread 和 worker thread 存取。

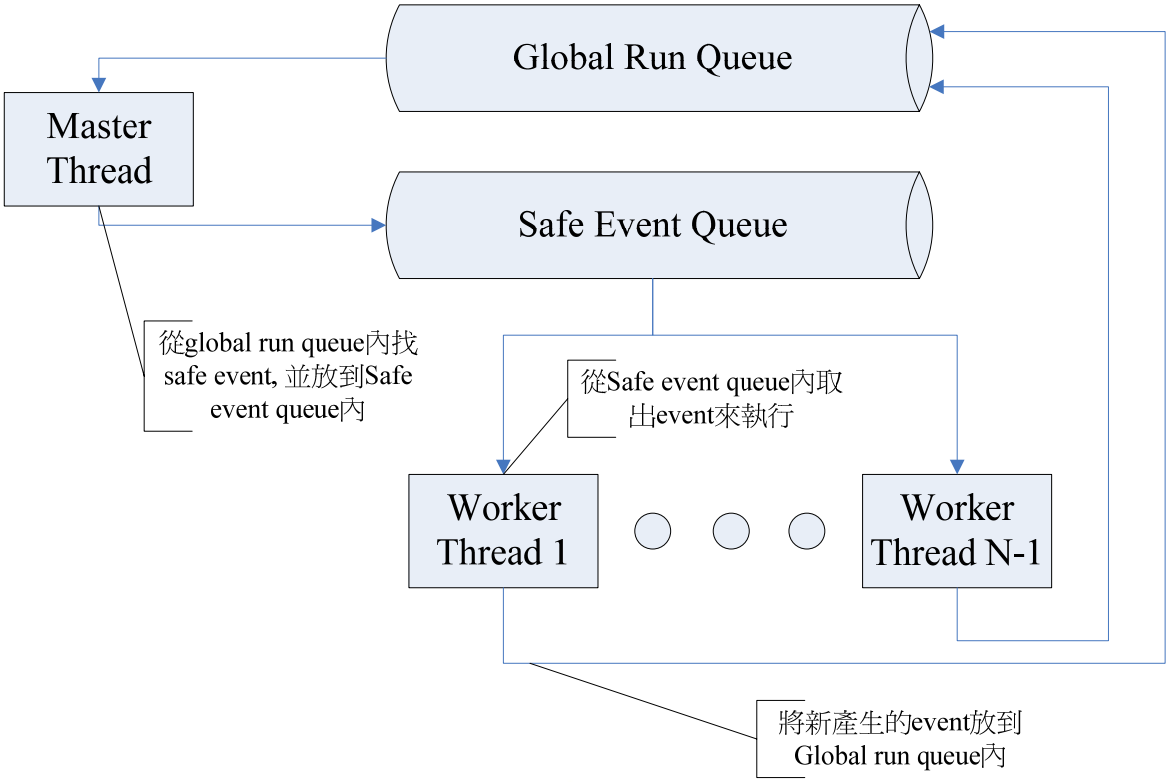


Figure 3.1 原有的 ELP 架構

對於 N 核心的系統來說，ELP 會產生 N 個 thread，其中一條為 master thread，其他的則為 worker thread。Master thread 主要負責安全事件的尋找，而找到的安全事件將會放到 safe event list 內，由於這些安全事件是彼此相互獨立的，因此不同的 worker thread 平行執行並不會造成模擬結果的錯誤。

如 Figure 3.2 所示為 master thread 的流程圖，當 master thread 找到足夠多的安全事件時，便會叫醒 worker thread 去處理，當 safe event queue 滿或是找不到 safe event 時，master worker thread 就會進入沉睡狀態，直到 worker thread 發現 safe event queue 為空，主動叫醒 master thread 為止。當 master thread 醒來後，它會繼續尋找新的安全事件，並重覆此流程直到模擬結束為止。

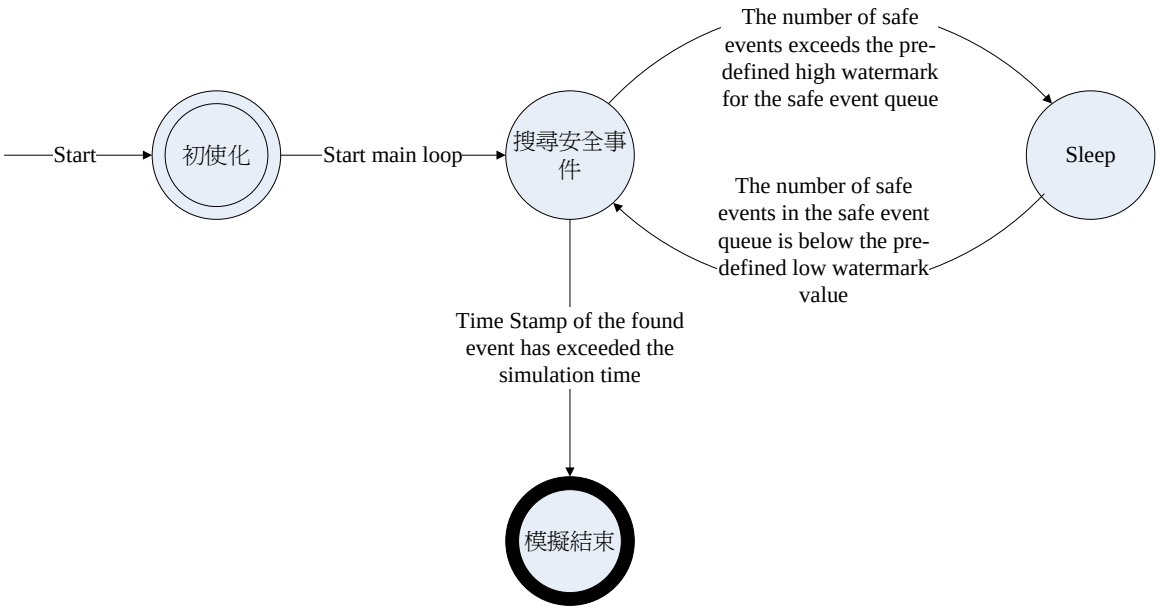


Figure 3.2 Master Thread 的執行流程圖

在任何的時間點下，至少會有一條 worker thread 正在執行，這是因為事件層平行模擬是按照時間順序來找事件執行，而擁有最小時間戳記的事件在整個模擬行進中，一定是安全事件。因此，假設目前只剩下一條 worker thread 正在執行安全事件，當它完成事件後，由於目前 safe event list 為空，所以 master thread 必需找尋新的安全事件放到 safe event list 讓 worker thread 執行，此時所找到的安全事件其必擁有最小的時間戳記。在新的安全事件加到 safe event list 後，worker thread 又可以開始執行，詳細的狀態圖如 Figure 3 所示，當 worker thread 啟動時，會去 safe event queue 抓取安全事件來執行。如果 safe event queue 為空時，worker thread 會去叫醒 master thread 來找安全事件，自己則進入休眠狀態等待 master thread 完成安全事件的搜尋，所以 master thread 可以確保在每次醒來後，safe event queue 內不會有任何安全事件，必須再從 global run queue 內重新尋找。

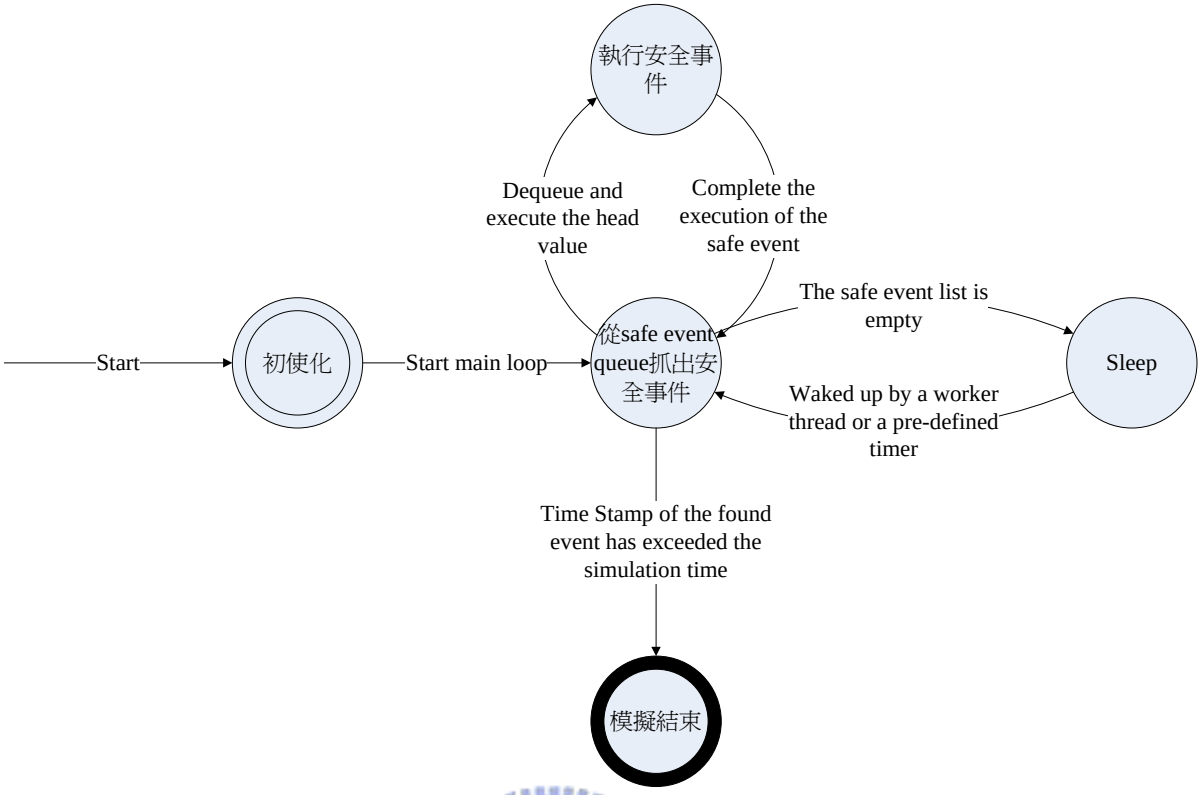


Figure 3.3 Worker Thread 流程圖

Global run queue 內放的都是被等待執行的事件，這些事件是根據時間做為排序的準則，時間則是表示事件被執行的時間，並確保不會有亂序執行的情況發生。循序模擬器建購在單一處理器的系統上，依序處理 global run queue 內的事件，避免「因果錯誤」的發生。在執行過程中產生的新事件也將按照順序放入 global run queue 內等待執行。

事件層平行模擬方法，會隨著系統使用的處理器個數決定 worker thread 的數量。假設目前有 N 顆處理器，則會創出 N-1 顆 worker threads 加上一條 master thread。只要 worker threads 可以一直處理安全事件，沒有閒置的情況發生，那麼理想上，隨著 CPU 數的增加，效能也可以呈現線性提升。而 master thread 最重要的部份在於能否找到足夠多的安全事件，使得 worker thread 可以隨時從 safe event queue 內拿到安全事件來執行。

因為 master thread 和 worker thread 都會同時存取 global event list 和 safe event list，因此為了確保資料結構的一致性，必須在存取這兩個資料結構時使用 lock

的方式進行保護。除了這兩個資料結構外，如果程式中仍有一些全域的變數或資料結構有可能同時被不同的 threads 存取，則它們就需要使用 lock 保護。如果只是要取出它們的值，而不會改變它的話，那麼就可以不用 lock。由於每條 worker thread 可以同時執行 safe event queue 內的安全事件，而每筆安全事件所帶的時間都代表目前模擬所處的時間，因此正在執行事件的時間必須存放到不同變數內，讓 worker thread 可以在執行的過程中，在有需要時可以透過存取這個變數，來獲得當前的正確的時間。現階段每條 worker thread 都擁有一個屬於自己的資料結構，用來儲存個別的值，只需利用 thread ID 的對應就可存取這個資料結構。

3.2 Inter-thread Communication

Master thread 與 worker threads 間存在一種溝通的方式，使得 master thread 可以通知 worker thread 去 safe event queue 內抓取安全事件來執行，而 worker thread 也能喚醒 master thread，要求它去搜尋新的安全事件。

POSIX 執行緒函式庫提供許多函式給使用者，其中有兩種函式可以完成上述功能，分別為 pthread_cond_signal ()與 pthread_cond_broadcast ()。pthread_cond_signal ()僅會通知等待相同條件而處於休眠中的其中一條執行緒，而 pthread_cond_broadcast ()會以廣播的方式通知所有等待相同條件群組中的所有執行緒。

當 master thread 想要通知 worker threads 去執行 safe event queue 內的安全事件時，它可以使用 pthread_cond_broadcast ()去叫醒所有的 worker thread。可是若目前找到的事件數並不足以供所有的 worker threads 工作，那麼被叫醒的 worker threads 有可能因為找不到安全事件可以執行而馬上進入休眠狀態，這反而是浪費 CPU 的時間。

因此，當找到的安全事件其數目小於 worker threads 個數時，master thread 將不使用 pthread_cond_broadcast ()，而是用 pthread_cond_signal ()去個別叫醒相等個數的 worker threads。

另一方面，worker threads 也會通知 master thread 去尋找新的安全事件來提供它們執行，因為在整個模擬過程中最多只會有一個 master thread，因此它們總是用

pthread_cond_signal ()去叫醒 master thread。

3.3 Precomputation of the Minimum Path Lookahead

Master thread所能找到多少安全事件與lookahead的大小有關，因此我們必需事先計算封包在點與點間傳送所需要花費的最小時間。為了計算所有點的最短路徑，我們先取得相鄰點之間鏈結上的延遲。使用一個二維陣列存放相鄰點之間鏈結上的延遲後，便利用Floyd-Warshall演算法事先運算並將結果存入一個二維陣列內。計算封包從一個點傳到另一個點所需花費的最短時間，需要O（N^2)的時間複雜度，而為了儲存結果，也需要O（N^2)的記憶體空間。Master thread可以很容易的用來源點ID和目的點ID來取得陣列中任一元素的資訊。

Chapter 4 架構分析

事件層平行模擬為了能在多核心的系統上正常模擬，它增加了許多額外的負擔，包含執行緒間共用的變數必需上鎖，尋找可平行運算的安全事件，以及 thread 之間的 IPC 通訊等。上述因素均會影響到事件層平行模擬的效能。在這個章節中，我們將分析此 ELP 架構的各個部份，希望藉此來找出可以改良的地方。

Parameter Name	Value
ns-2 Version	2.31
CPU model	Intel Quad-Core
Main memory	2 GBytes
Simulated Grid Network Topology	3x3, 4x4, 5x5, 6x6, 7x7, 8x8, 9x9, *10x10
Number of threads	1, 2, 3, 4
Simulated Time (s)	60
Traffic Type	CBR UDP, TCP
Interface Output Queue Length (pkts)	50
Event Computation Empty Loop Count (x1000)	0, 10, 20, 30, 50, 60, 70, 80, 90, 100
Link Bandwidth (Mbps)	1, *10, 100, 1000
Link Delay (ms)	1, *10, 100, 200, 300
Transmit range (m)	250

Table 4.1 The used parameter settings

模擬用的環境參數如 Table 4.1 所示，我們列出了所有使用的系統參數，不同的參數代表不一樣的網路環境。表中用星號標記的文字代表模擬中所使用的預設值。我們建立了 8 種大小的 Grid 網路拓樸，包括 9、16、25、36、49、64、81、100 個 node。

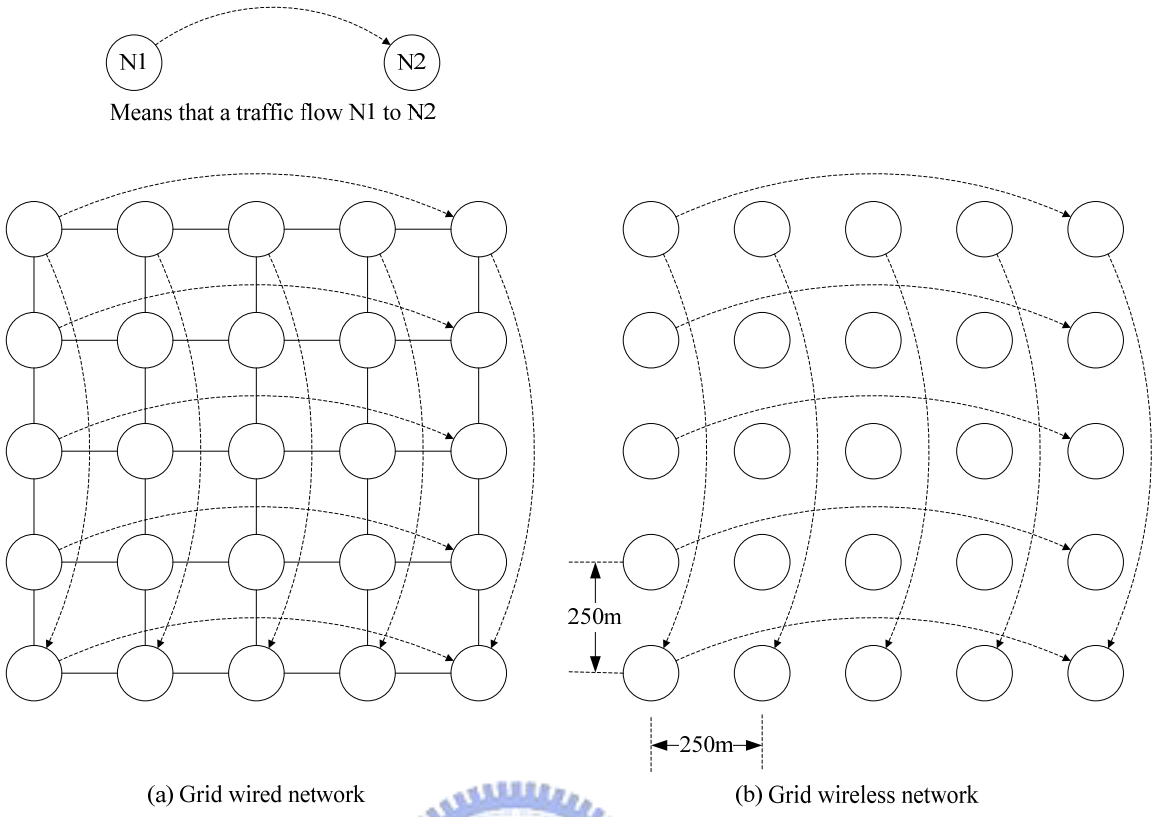


Figure 4.1 The 5x5 grid wired and wireless networks

以 5x5 大小的模擬網路拓樸為例，如 Figure 4.1 (a)所示為 Grid wired network 的相連方式，每個節點都與上、下、左、右相鄰四點做連接。而 Figure 4.1 (b)所示為 Grid wireless network 的放置方式，每個節點和上、下、左、右相鄰四點的間距都為 250 公尺，使封包剛好可以送到。所有測試的案例，不管是使用 TCP 或是 UDP 協定模組的應用程式，都會依照拓樸大小來決定應用程式的多寡。所有在網路最上排的節點都放置了程式來產生封包送往最下排的節點上的程式，網路最左邊的節點，也都會產生封包送往最右邊的節點。

Event computation loop 所表示的是一個事件所需額外執行的空迴圈次數，如 for(i=0;i<N;i++);，我們藉由這個值來模擬封包在 physical layer 做編碼和解碼所花費的時間，其送端和收端的模組都有一個對應的迴圈存在，我們測量的值為零到十萬，當其值為零時表示沒有附加額外的 overhead。

4.1 影響 ELP 效能的因素

根據 ELP 的想法歸納出以下幾個會影響到平行化模擬效能的原因。

- worker thread 的個數

  因為 worker thread 的數量決定了最大可以同時執行的事件數，因此 speedup 最大不會超過這個值。

- 模擬節點的個數

  在網路模擬當中，若是以收端的角度來看，不管是 timer event 或是 physical layer event，所有它處理的事件一定是依照時間的先後次序來執行，正因為如此，只有不同節點在等待處理的第 1 個封包才有機會判定為安全事件，所以 speedup 上限不會超過模擬的節點數。

- 處理事件的時間與找安全事件的時間的比例

  為了避免 race condition 發生，因此找安全事件時要 lock run queue，確保同一時間只能有一個執行緒去存取，這樣的話，若是想要達到 N 倍的 speedup，找安全事件所需時間就需要是執行事件所需時間的 N 倍快。

- 找到安全事件的機率

  在平行化網路模擬中，當兩個事件互為安全事件的機率越高，master thread 越不用花多餘的時間去找其它安全事件來給 worker thread，使得 worker thread 可以更快的拿到安全事件執行，增進效能。

依據以上所分析的影響平行化模擬效能的因素，因此我們設計了以下的實驗，將會使用不同的執行緒數目、traffic type、event computation loop count 來模擬。

4.2 Overheads of finding Safe Events

當找安全事件所花的時間越小，代表要平行化網路模擬器所額外付出的 overhead 就越少，其 speedup 就會越高，因此計算找出一個安全事件平均所需的時間，可有助於了解使用這個架構額外付出的 overhead。在檢查安全事件時，因為要存取 global run queue，需要用 lock 來做保護，因此，我們利用 gettimeofdays () function 來分別取得 lock 前和 unlock 後的時間，依據其差值得出所花的時間。

Number of Worker Threads	1	2	3	4
average required time	1.341316	4.446133	5.157505	5.641939
standard deviation	0.483141	4.622712	4.619523	4.110412

Table 4.2 Time required for finding safe events

根據模擬的結果，找安全事件所需的時間只和執行緒數目有關。由 Table 3 可明顯看出找安全事件所需的時間是相當集中的，當執行緒數量由 1 增加到 2 時，可看到所需時間大幅上升，原因在於多了要同步不同 CPU 之間的 cache 的負擔，而當 worker thread 數目增加時，找安全事件所需的時間也會繼續上升，這是因為當我們在找安全事件時，在 worker thread 內執行的事件也要一併檢查，以避免發生因果錯誤，因此當 thread 數越多，找安全事件所花的時間就會越長。

4.3 The Proportion of Physical-layer Events to All of the Simulated Events

在網路模擬器當中，通常 physical layer event 會要花比較多的時間來處理，如進行封包的編碼、解碼、計算 send power、receive power 等等，而在 ns-2 裡並沒有實作進去，因此我們便在相對應的位置加入 event computation loop，藉此模擬這功能。當這種類似 physical layer event 之類的 heavy weight event 越多時，代表平均起來

執行安全事件所需的時間就越長，平行化後就越容易得到效率的提升。因此我們計算在不同的模擬環境下 physical layer event 的比例，並統計成下表

Traffic Type	Wired TCP	Wired UDP	Wireless TCP	Wireless UDP
Physical-layer Event	28.16%	96.93%	2.39%	18.14%

Table 4.3 Physical-layer event 與總合 event 的比例

從 Table 4.3 來看，UDP 的 physical layer event 比例比 TCP 高，原因是 TCP 需要許多 timer event 來幫忙處理 retransmit、congestion control 等，而 802.11 的 CSMA/CA 機制，也同樣需要大量的 timer event，因此在 wired 中 physical-layer event 的比例也明顯比 wireless 多。另外根據實測的結果發現，就算模擬的網路拓樸大小不同，對 physical layer event 所佔比例沒有影響。

由此可看出 Wired UDP 因為有大量的 physical-layer event，當 event computation loop count 增加時，其模擬速度上升的效率會是裡面最好的，Wired TCP 和 Wireless UDP 則是其次，而 Wireless TCP 因為 physical layer event 稀少，可能難以有良好的結果。

4.4 Overheads of Executing a Safe Event

藉由執行安全事件平均所需的時間和找安全事件所需的時間比，可用來推算出理想上效率提升的最大值。因此我們在 worker thread 執行安全事件時，利用 gettimeofdays () function 來分別取得執行安全事件前後的時間，並依據其差值得出所花的時間。

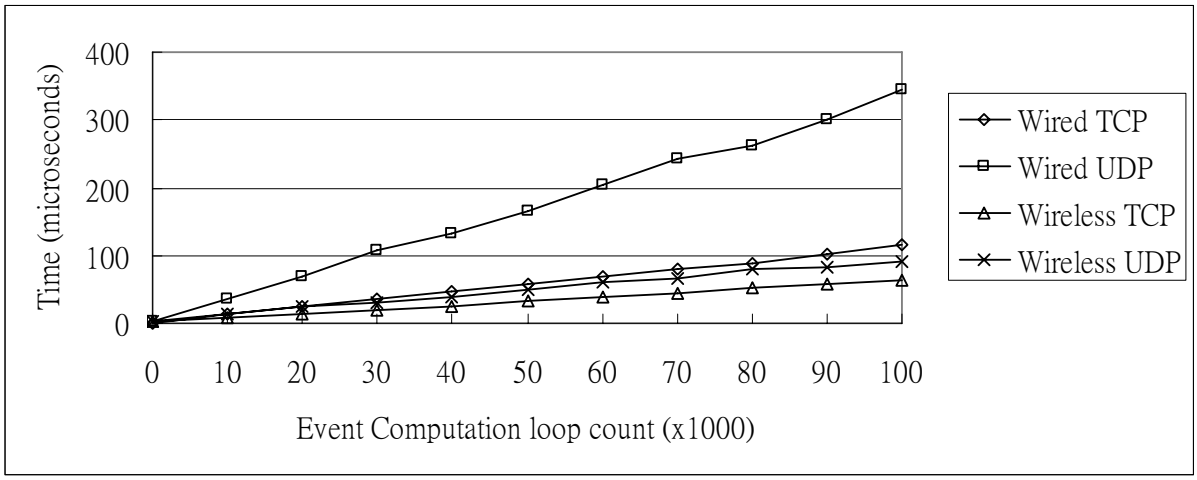


Figure 4.2 Event Processing time

Event Computation Loop count 代表對 physical-layer event 額外加的 overhead，如 Figure 4.2 所示，當 event computation loop count 增加後，平均處理事件所需的時間也會上升。其模擬結果也和上一小節的 physical-layer event 的比例相呼應，當 event computation loop count 增加，physical-layer event 比例越高的網路拓樸其處理安全事件所需時間的上升越快。

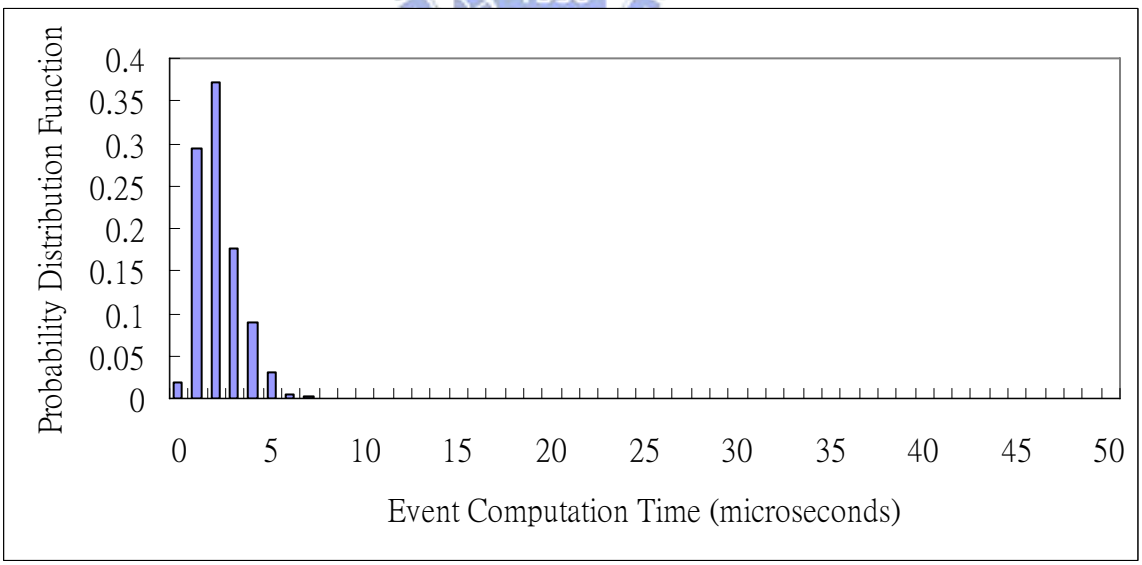


Figure 4.3 Event computation loop count 為零的 Wireless UDP 處理事件時間分佈圖

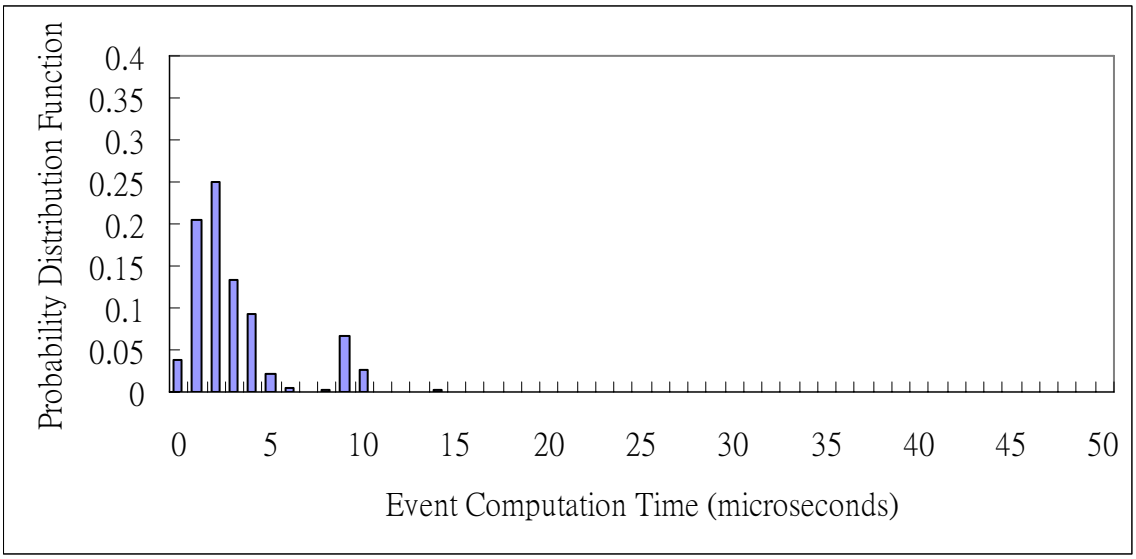


Figure 4.4 Event computation loop count 為十萬的 Wireless UDP 處理事件時間分佈圖

以上為處理事件的時間分布圖，我們以 10x10 的 Wireless UDP 的為例。兩張圖相比較的結果顯示，沒有隨 event computation loop count 上升而增加處理時間的都是 timer event，很明顯的可以看出，這些事件所需的處理時間都相當小，而在多執行緒的架構下，找安全事件所需的時間平均也要 4 microseconds，這樣的話平行處理這些 timer event 不僅不會有任何好處，反而會降低模擬的效率。

4.5 Overheads of Thread Switching and Global Variable Access

在多執行緒的應用程式中，各執行緒間共享多個變數，一起去修改它是常有的事，而在平行模擬中，最常被存取的就是 global run queue，因為原始的平行化網路模擬器架構較為複雜，並不利於測量，所以另外寫一個小程式，用來估算 thread switching 所需最小 overhead。

先建立 2 個執行緒來搶同個 lock，在取得 lock 的同時執行緒會讀取同個變數，並寫入特定的數值，因此當發現此變數改變時，就表示曾切換到另一個執行緒過。藉由計算此變數改變的次數，便可得知執行緒切換的次數。而由那個執行緒搶到 lock 則是由

Linux kernel 來決定的，因此我們跑了數次結果，將執行此程式所花的時間和執行緒切換的次數做成如下圖表。

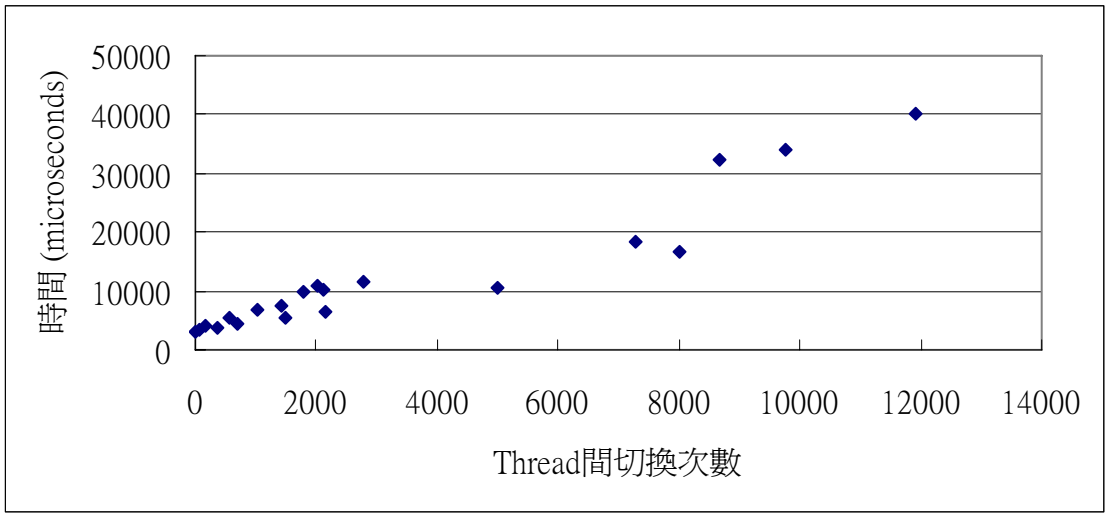


Figure 4.5 Thread switching overhead

根據此圖，可發現大致上點是分佈在一條直線上的，因此我們用最小平方法計算出其 linear equation 為”y=2.818x+2767.279”，則執行緒間切換所造成的 overhead 約為 2.818 microseconds，而網路模擬器在執行緒切換時要同步的變數遠比此測試程式多，因此實際上所需的 overhead 相信會比這值大。

Chapter 5 新的 ELP 架構

在這個章節，我們將依據之前分析的結果，提出新的架構來改進舊有的 ELP 平行化模擬，使其達到更好的效率。

5.1 設計目標

- 為了讓 CPU 的使用更有效率，儘量讓所有的 thread 隨時有工作可以做，因此取消了 master thread，而讓 worker thread 自行去找 safe event 來執行，使其平均分攤工作量，且增加 worker thread 可使模擬器同時執行更多安全事件，增加最大的 speedup。
- 當某個事件執行所需的時間大於找安全事件所需的時間時，去平行執行它反而是降低效能的事，因此只需要針對執行時間長的事件做平行處理，如 physical layer event，至於執行時間短的，像是 timer event 這些就以循序的方式處理。

5.2 實作

在此新架構中，我們移除了 master thread 和 safe event queue，原本 master thread 的工作則分攤給 worker thread 處理，並讓 worker thread 直接從 global run queue 內找安全事件。

- Worker Thread

從 Global run queue 內檢查第一個事件是否為 safe event，是的話就取出並執行，並將產生的事件放到 Global run queue 內。

- Global run queue

所有 worker thread 產生的事件都存放在此並等待被執行。

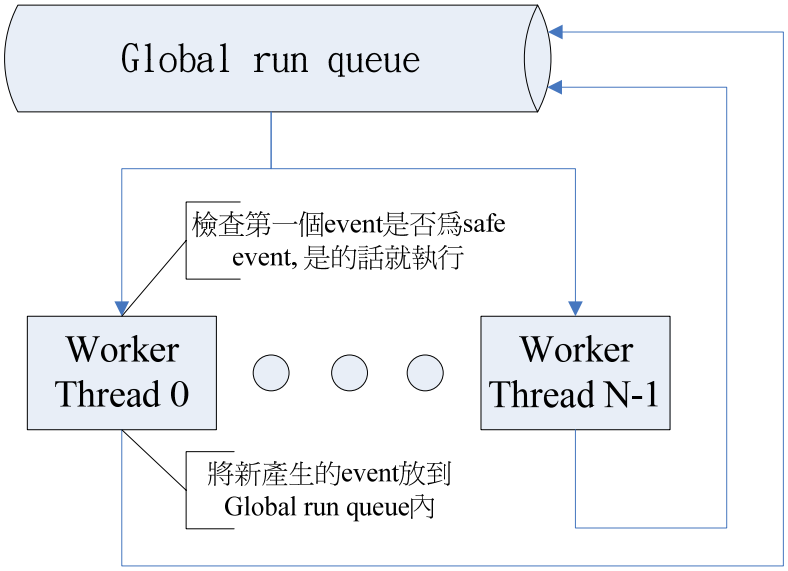


Figure 5.1 The architecture of a parallel network simulator using the proposed ELP approach

Figure 5.1 為新 ELP 的架構圖，在 N 核心的系統中，此新架構會產生 N 個 worker thread，每個 worker thread 都是平等的，隨時會從 global run queue 找安全事件來執行，新產生的事件也會放到 global run queue 內。

Worker thread 的流程圖如 Figure 5.2 所示，基本上與舊架構類似，在模擬時，worker thread 會先檢查 event queue 的第一個事件和其它 worker thread 正在處理的事件是否互為安全事件，如果有任何一個事件不能和它安全的平行執行的話，此 worker thread 就進入休眠狀態直到有其它 worker thread 叫醒它。如果都是互為安全事件的話，就取出這個 event 並執行它，若執行時發現這是個 physical layer event 的話，才去叫醒其它進入休眠狀態的 worker thread，這是因為通常這種 event 都要執行較久，這時平行處理才有利益可得，而在 ns-2 的話因為架構問題，只能在模擬時才能判斷事件是否為 physical layer event，因此目前是改用在 processing loop 時才叫醒 worker thread。為了確保在同一時間最多只會有一個 worker thread 去執行 timer event，我們額外多加了一個 lock，只有取得 lock 的 worker thread 才能去 global run queue 內找安全事件，而只有在完成事件的執行或是確定這個事件是 physical layer event

時才會釋放 lock，因此能確保同一時間最多只會有一個 timer event 在執行，而其它 worker thread 不是在執行 physical layer event，就是進入休眠狀態。

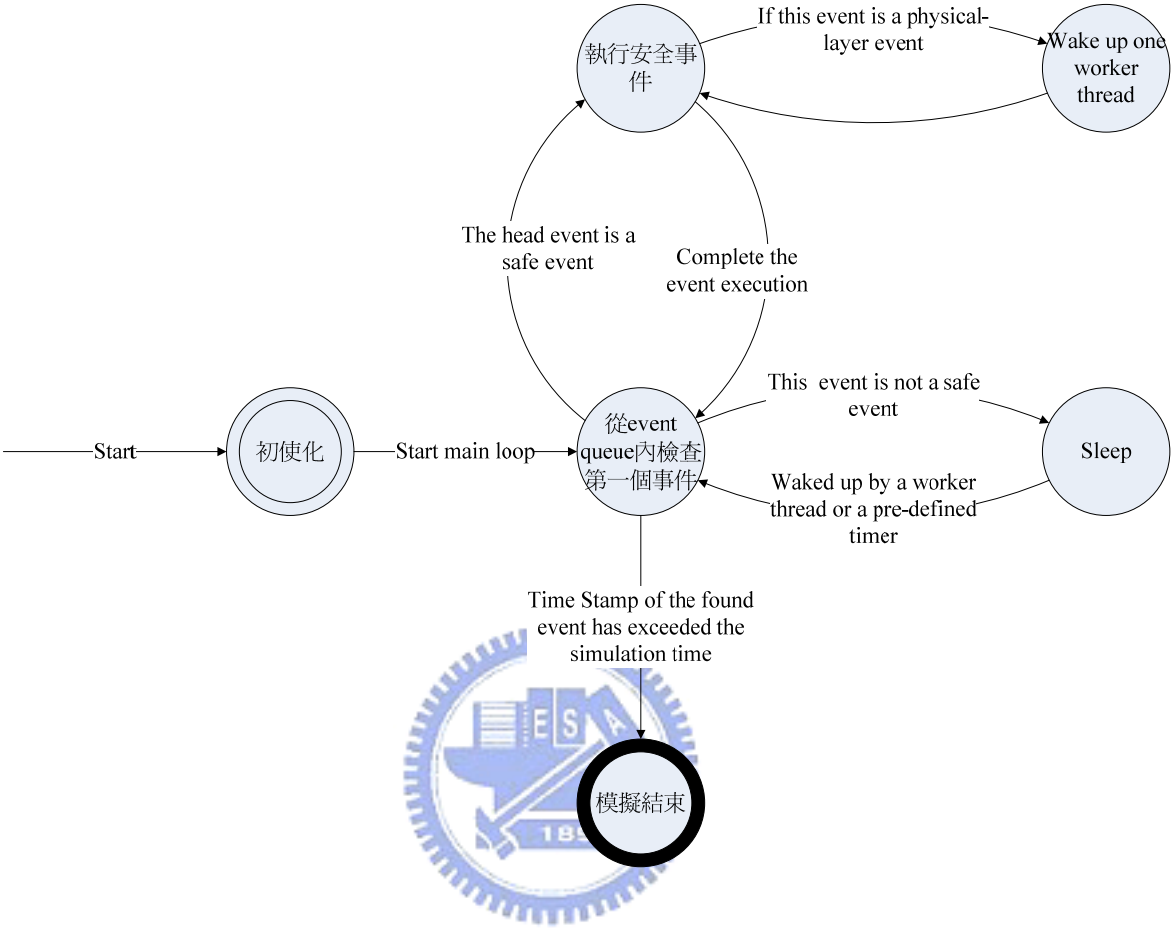


Figure 5.2 worker thread State Diagram

根據此架構，因為不需要額外的 master thread 來幫忙找安全事件，因此可以比舊的 ELP 架構多一條 worker thread 來處理，而沒有去平行化處理 timer event 也減少了些 CPU 資源的浪費，理論上也可得到較好的效率提升。

Chapter 6 Performance Evaluation of the Two ELP Architectures using Ns-2

在這個章節中，我們會進行效能量測，模擬所用的參數同 Table 4.1，主要採用 10 x 10 的 grid network，分別以有線、無線和 TCP、UDP 的組合，配合 event computation loop count 來測試新的 ELP 架構的效能，並在使用相同的模擬環境和參數設定下，比較新、舊架構的效能差異。

$$\text{其中效能提升 (Performance Speedup)} = \frac{\text{循序模擬執行時間}}{\text{事件層平行模擬執行時間}}$$

在模擬過程中，可能會有許多非網路模擬器的應用程式請求CPU資源，為了減少這種情況發生，降低效能量測的誤差，因此要在模擬開始前要將不必要的程式關閉。然後，為了測量使用不同機制 (循序模擬、舊的事件層平行化網路模擬、新的事件層平行化網路模擬)下模擬所花的時間，我們將使用由GNU project[6]提供的「time」指令來觀察。

6.1 The Impacts of Safe Event Search Depths on Required Simulation Time

在新架構中，影響效率的因素除了執行緒的數量外，還有找到安全事件的機率。當 worker thread 對 global run queue 的搜尋深度越深時，找到安全事件的機率就越大，相對的，worker thread 就能更快的去執行安全事件。

其中P_E = 安全事件的個數 / 檢查安全事件的次數

6.1.1 有線網路

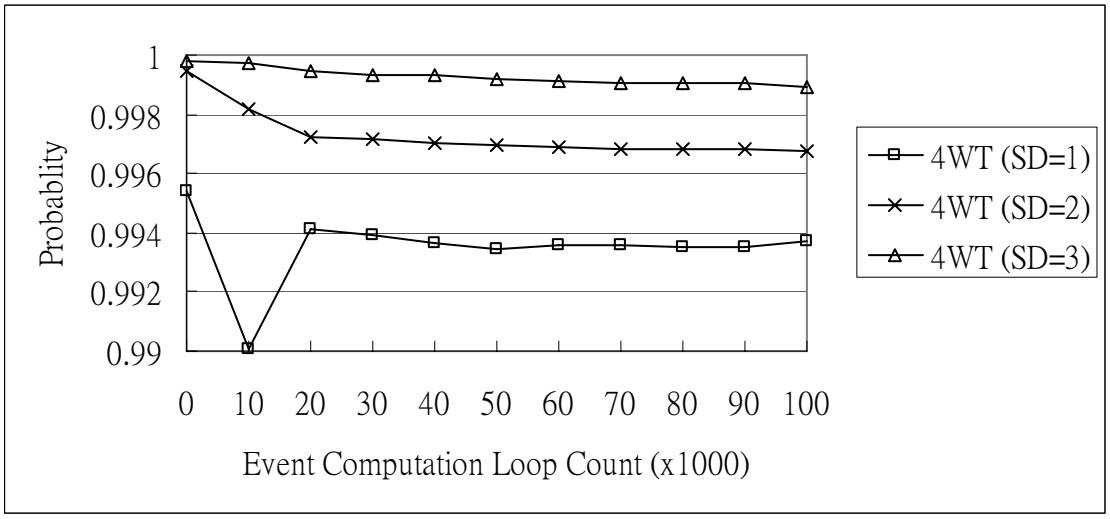


Figure 6.1 The Wired TCP P_E result over different event queue search depth

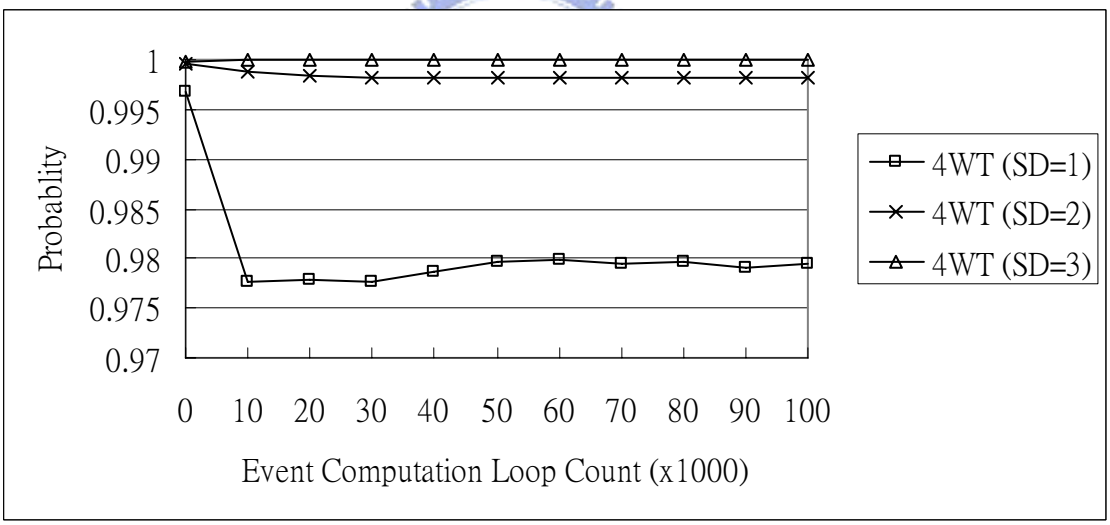


Figure 6.2 The Wired UDP P_E result over different event queue search depth

首先我們從有線的部分來看，不管是 TCP 或 UDP 的模擬環境，在有線的部份原本當搜尋深度為 1 時，找到安全事件的機率就已經很高了，所以再往後搜尋其找到安全事件的機率只有些微上升，因此對模擬時間來說也沒有影響，如 Figure 6.3 所示。

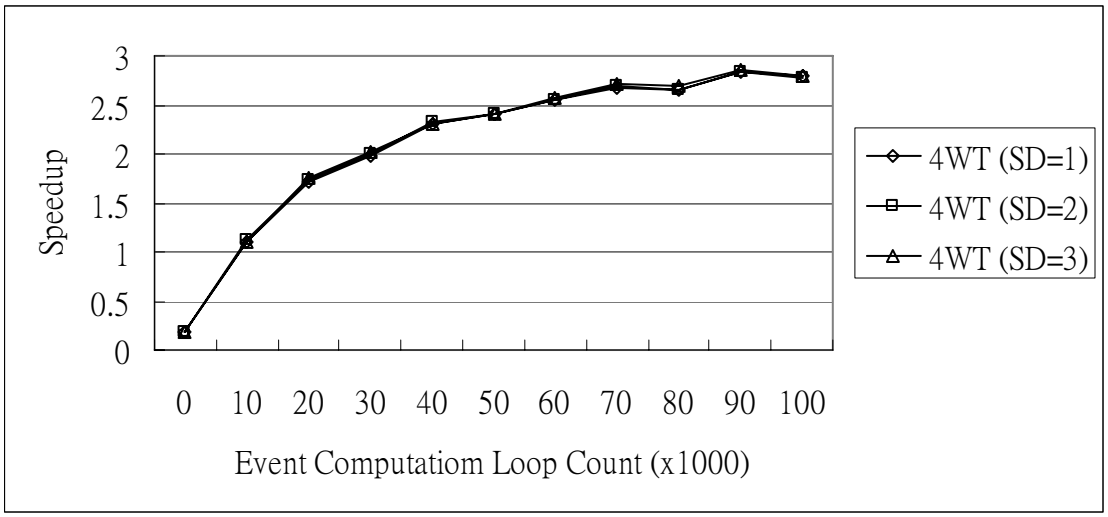


Figure 6.3 The Speedup Results over different event queue search depth in Wired TCP

6.1.2 無線網路

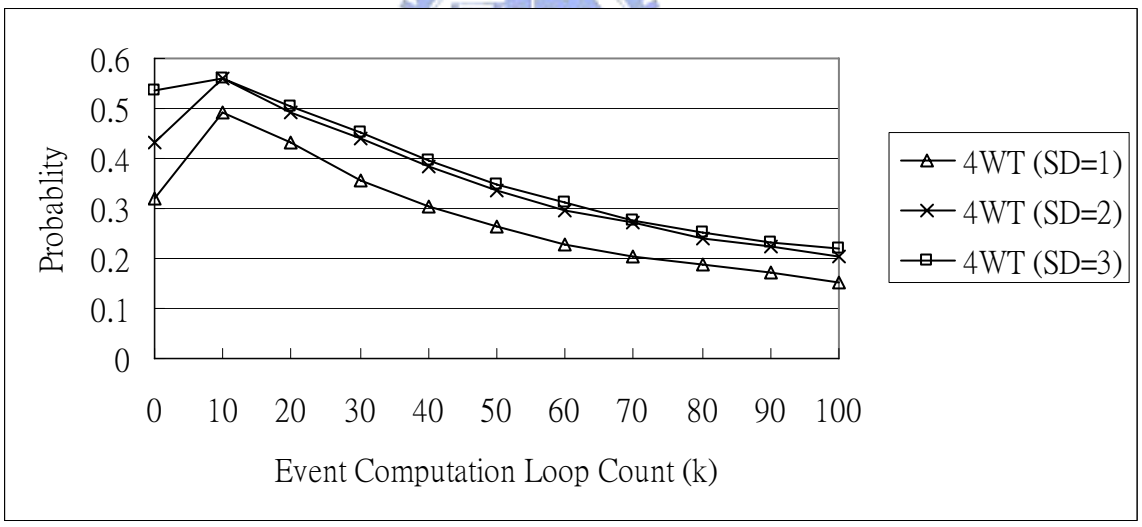


Figure 6.4 The Wireless TCP P_E result over different event queue search depth

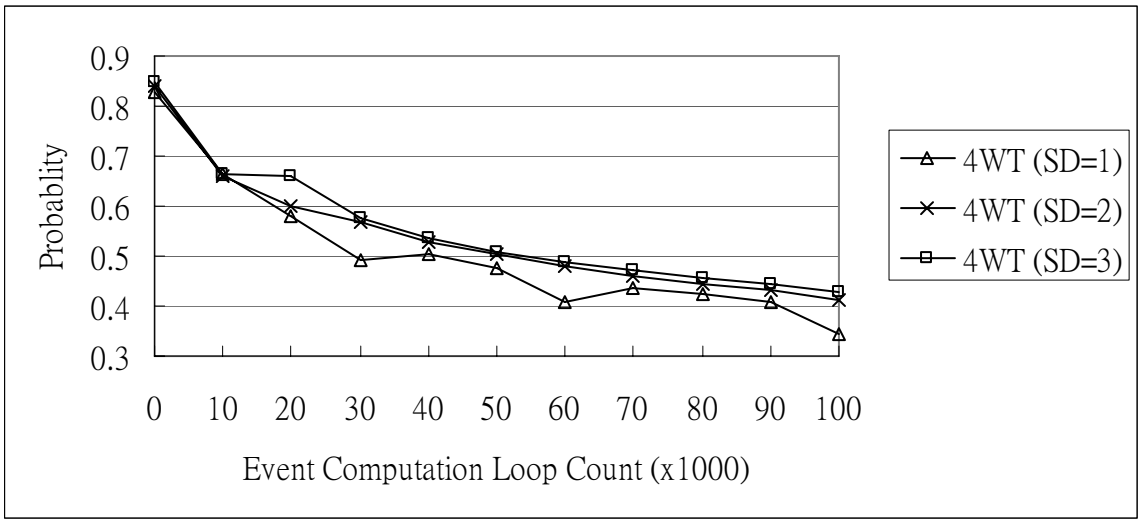


Figure 6.5 The Wireless UDP P_E result over different event queue search depth

在無線的部份，找到安全事件的機率比有線低很多，因此當 global run queue 的搜尋深度上升時，可以很明顯的看出找到安全事件的機率會增加，不過從效能圖來看也是和有線類似的情況，對效能並沒有明顯的影響，如 Figure 6.5 所示。

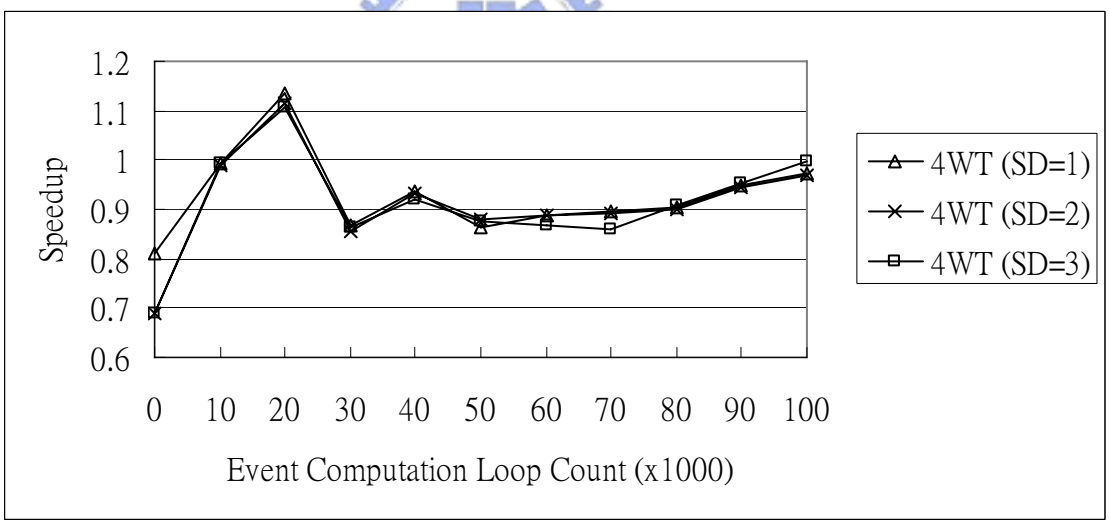


Figure 6.6 The Speedup Results over different event queue search depth in Wireless TCP

在 global run queue 內增加安全事件的搜尋深度，雖然對找到安全事件的機率有利，效能卻沒有因此提升，猜測是因為搜尋的深度增加的同時也會增加找安全事件所花的時間，反而會影響效能，且即使在較後面的安全事件沒有被找出來，遲早也會判定是安全事件而和其他事件平行執行，因此後面我們將只跑搜尋深度為 1 的結果。

6.2 Performance Evaluation of Wired Networks

在這章節我們將展示新架構在有線網路下的效能，其網路拓樸所使用的參數參考 Table 4.1。

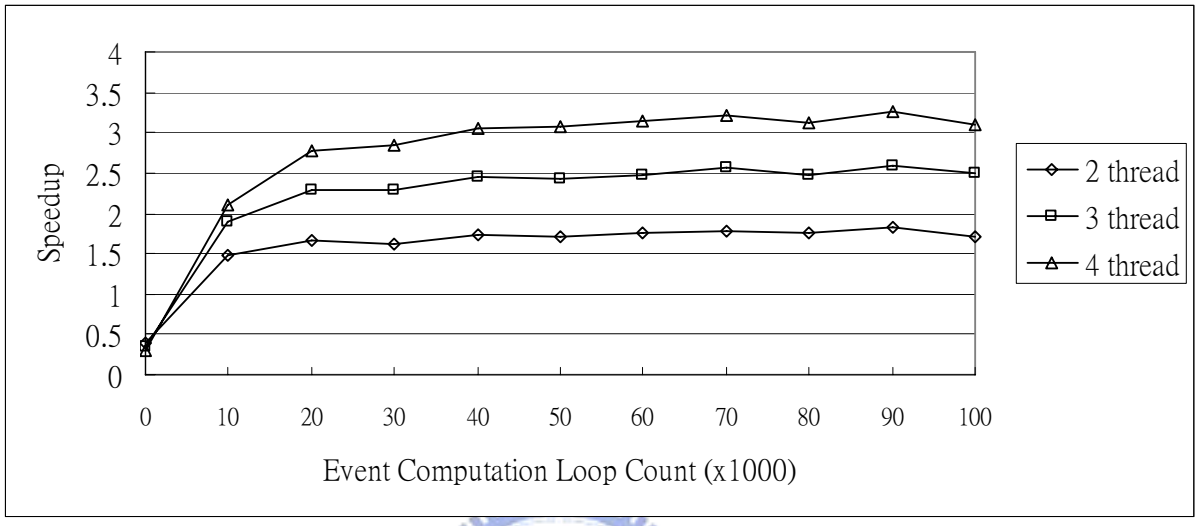


Figure 6.7 The Speedup Results of ELP in the Wired TCP case

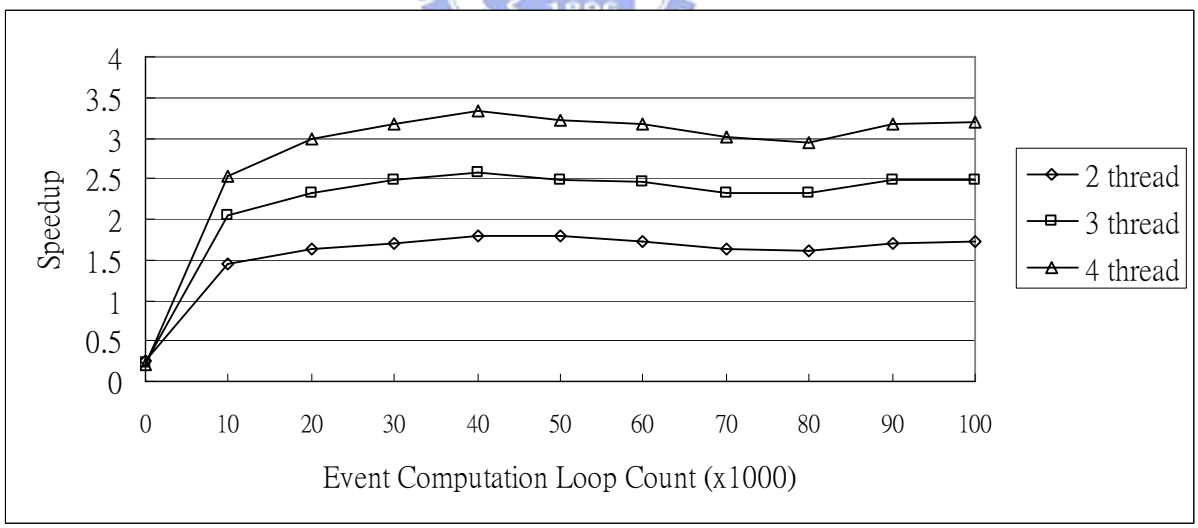


Figure 6.8 The Speedup Results of ELP in the Wired UDP case

如圖 Figure 6.5、Figure 6.6，分別表示的是有線的 TCP 和 UDP 的模擬結果，在執行緒多的情況下，可以明顯的看出 UDP 的 speedup 比 TCP 升的還快，這是因為 UDP 的

physical layer event 比例較 TCP 多，不過當 event computation loop count 夠大時，效能提升就差不多了，而因為找安全事件、thread switching 等 overhead，使得 speedup 最高只到 3.2 倍左右。

6.3 Performances of ELP in Wireless Networks

在這章節我們將展示新架構在有線網路下的效能，其網路拓樸所使用的參數參考 Table 4.1。

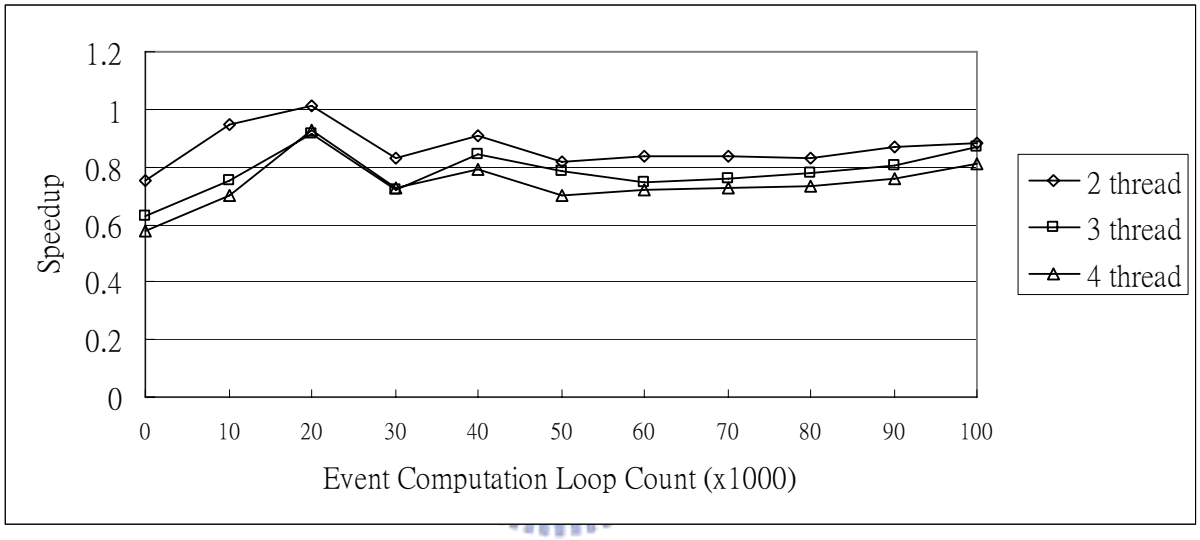


Figure 6.9 The Speedup Results of ELP in the Wireless TCP case

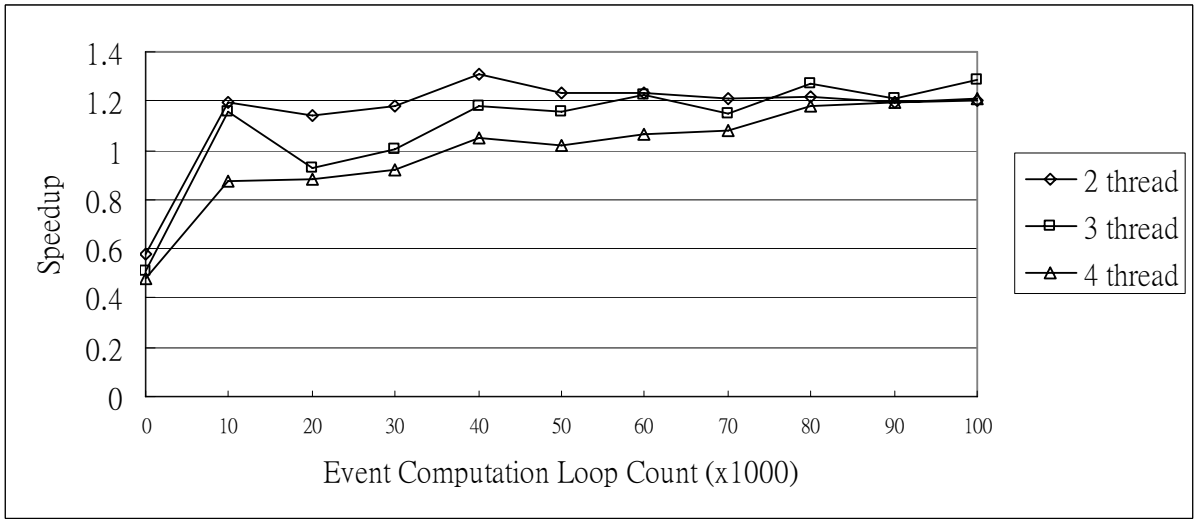


Figure 6.10 The Speedup Results of ELP in the Wireless UDP case

如圖 Figure 9、Figure 10，分別表示的是 wireless 的 TCP 和 UDP 的模擬結果，和有線的結果相比，可以很明顯的看出效率比有線差很多。UDP 其 speedup 最高可以升到 1.3 左右，不過 TCP 在多執行緒的情況下並不大好，speedup 只能到 0.8，這是因為無線網路的 physical layer event 並不多，即使用 event computation loop 增加其執行的時間，還是因為小的 timer event 太多而只能循序處理，得不到好的效果。

6.4 和舊 ELP 架構的效能比較

在本小節中，我們使用舊的 ELP 架構跑相同的網路拓樸，藉此比較和新架構的效能差異。

6.4.1 有線網路的效能比較

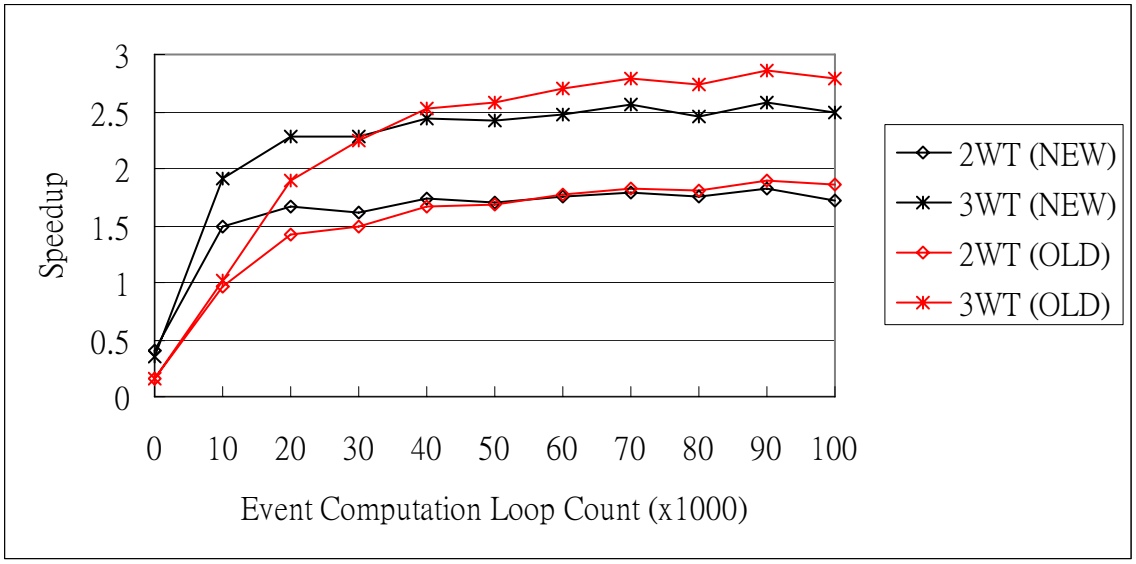


Figure 6.11 The Speedup Results of the Original and New ELP Architecture in Wired TCP Case

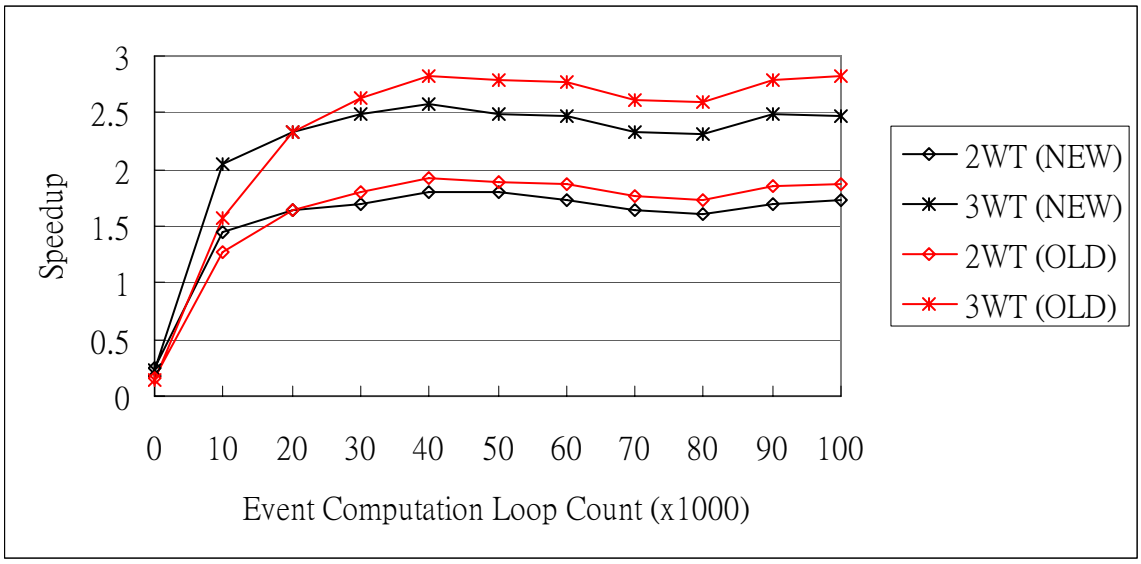
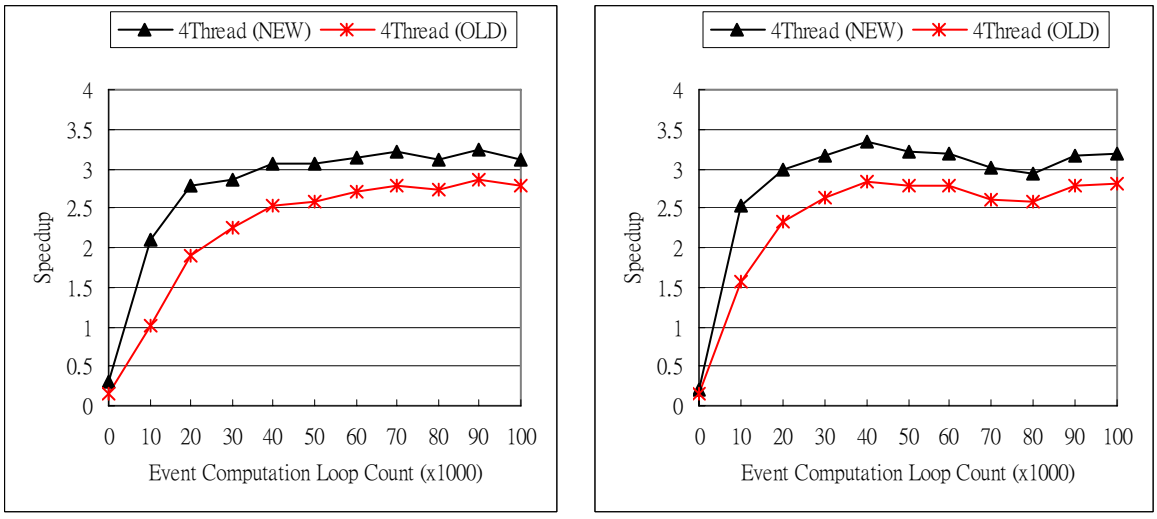


Figure 6.12 The Speedup Results of the Original and New ELP Architecture in Wired UDP Case

在這一節我們比較有線的結果，在同樣 worker thread 數量的情況下，如上圖 6.11 和圖 6.12，發現不管是 TCP 還是 UDP，新的架構在小的 event computation loop count 下有較好的 speedup，這是因為我們並沒有平行化處理時間短的 timer event，使得這部分所造成的浪費減少，而當 computation loop count 夠大，使得執行安全事件所需的時間足以支付找安全事件的 overhead 時，因為新架構的 worker thread 還要額外付出找安全事件的時間，因此會較舊架構慢。



(a) Wired TCP　　(b) Wired UDP

Figure 6.13　在同執行緒數量下，新舊 ELP 的效能比較

不過就總體上來說，因為新架構並不需要 master thread 幫忙找安全事件，所以在同樣的環境下新架構比舊架構多一條 worker thread 來執行安全事件，如 Figure 6.11 所示，可以很明顯的看出新架構在有線的網路可以達到更好的效能。

6.4.2 無線網路的效能比較

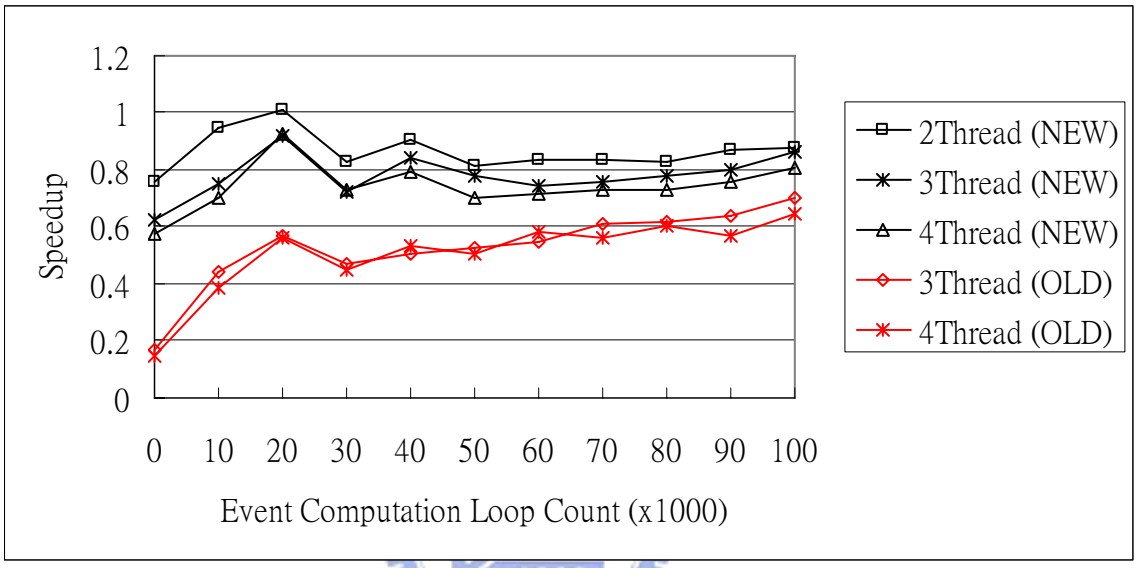


Figure 6.14 The Speedup Results of the Original and New ELP Architecture in Wireless TCP Case

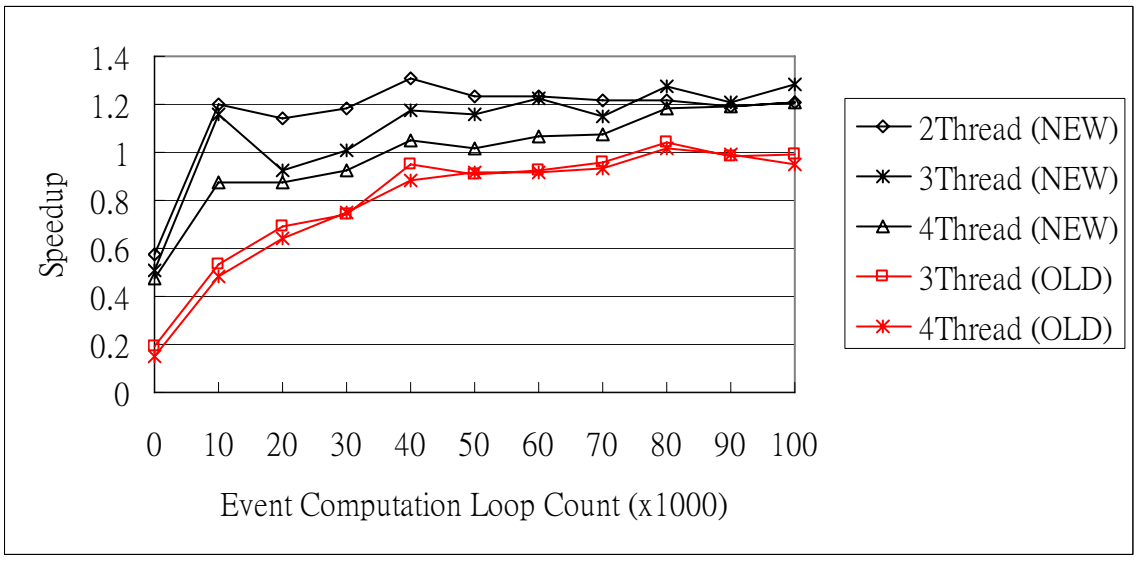


Figure 6.15 The Speedup Results of the Original and New ELP Architecture in

Wireless UDP Case

Figure 6.14 和 Figure 6.15 為無線的比較結果，在 TCP 的環境下，新舊架構的效率都不好，不過新架構可以比舊架構更接近循序模擬，這是因為從之前的結果可知，Wireless UDP 的 physical-layer event 數很少，當 event computation loop count 上升時，其找安全事件所需的時間還是大於執行安全事件的時間。而在 UDP 的環境下，新架構最高可以到 1.3 倍的效率提升，而舊架構即使在大的 event computation loop count 下其效率也只是接近循序模擬，可能要更大的 event computation loop count 才能超過。

Chapter 7 Future Work

在多執行緒的架構下，平行化網路模擬在處理事件時需要額外多花時間來找安全事件，而將新產生的事件放回到 global run queue 內也會產生額外的負擔，因此額外的 overhead 為：

search event overhead + insert event overhead > 4.446133 + 2.818 = 7.264 (ms)

要被平行處理的事件其執行時間起碼要大於 7.264 microsecond 才會有實質上的效率提升，而使用的執行緒增加的話，其所需的值還會上升，而在 N 核心的架構下，要被平行處理的事件更需要有 N 倍的執行時間，才能完全發揮其效能。

所以未來的目標可以朝向

- 降低找 safe event 所花的時間。
- 計算出各種事件執行所需的時間，在新增事件時加入參數，用以辨別這個事件可以在多少核心下進行平行處理，並將執行時間過短的事件設成循序處理，避免浪費 CPU cycle。
- 減少 Thread switching 的次數，雖然在有線網路上，找安全事件的機率大於九成，可是在無線網路其機率卻不高，若是能在確定能執行到安全事件的情況下才去叫醒 worker thread，那麼就能有效減少 Thread switching 的次數，加快無線的模擬。不過為了確保不會執行到執行時間過短的事件，需要先做完上述事件執行所需時間的標記。

Chapter 8 Conclusion

ELP 平行化模擬不同於以往的方法，需要將原有的網路模擬器架構做大幅度的修改，而是從事件層面分析，找出可獨立執行的事件，並分給不同的執行緒平行處理。因此使用這套方法並不需要了解複雜的協定架構，只需在原有的事件架構下做小修改即可達到平行運算的效果，這也讓使用者不需了解平行化的概念，就可使用並享受良好的效能提升。在本文中我們分析了舊的 ELP 架構，找出各個重要步驟所花的時間，並藉由分析結果來調整 ELP 架構，增進其平行模擬的效能。歸納出在 ELP 架構下，其最大平行度與模擬節點數和使用執行緒的數目之間成正比。我們將新舊 ELP 架構實作到 ns-2 網路模擬器上並比較兩者的效能差異。模擬結果顯示，在相同的模擬環境下，新的 ELP 架構的效能會比舊的 ELP 架構好，且可在大多數的網路情形下以較少的時間完成一個模擬 CASE。模擬結果同時也顯示，ELP 此類型的模擬架構較適合用於 heavy-load event simulation。

Reference

[1] Yui Li, and Depei Qian. "A Practical Approach for Constructing a Parallel Network Simulator" in the Proceedings of Parallel and Distributed Computing, Applications and Technologies, 2003. PDCAT'2003. Proceedings of the Fourth International Conference. 27-29 Aug. 2003.

[2] ns-2 http://www.isi.edu/nsnam/ns/.

[3] Thoppian, M., Venkatesan, S., Vu, H., Prakash, R., and Mittal, N. Improving "Performance of Parallel Simulation Kernel for wireless Network Simulations" in Military Communications Conference, 2006. MILCOM 2006.

[4] Bagrodia R. X. Zeng and M. Gerla. "GloMoSim: A library for the parallel simulation of large wireless networks". Workshop on Parallel and Distributed Simulation 1998, 154-161

[5] Kerneltrap http://kerneltrap.org/node/429?PHPSESSID=cf32edae30f893bafebba80631cf738a.

[6] GNU http://www.gnu.org/.

[7] Y.S. Tzeng. "The Performance of the NS2 Network Simulator using the Event-level Parallelism Approach". 2007.

[8] Bagrodia, R., Meyerr, R., and et al. 1998. "Parsec: A parallel simulation environment for complex system". IEEE Computer Magazine 31, 10 (October), 77 - 85

[9] Bhatt, S., Fujimoto, R., Ogielski, A., and Perumalla, K. 1998. "Parallel simulation techniques for large-scale networks". IEEE Communications Magazine.

[10] Lin, S., Cheng, X., and Lv, J. 2008. "Micro-synchronization in conservative parallel network simulation" in Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation (PADS'08). 195-202.

[11] Cowie, J., Nicol, D., and Ogielski, A. 1999. "Modeling the global internet". Computing in Science and Engineering 1, 1 (January), 42-50.

[12] Creeger, M. 2005. "Multicore cpus for the masses". ACM Queue 3, 7, 63-64.

[13] Das, S., Fujimoto, R., Panesar, K., Allison, D., and Hybinnette, M. 1994. "Gtw: A time arp system for shared memory multiprocessors" in Proceedings of the 1994 Winter Simulation Conference.

[14] Fujimoto, R. M. 2000. "Parallel and Distributed Simulation Systems". John Wiley & Sons, Inc.

[15] Jones, K. G. and Das, S. R. 2000. "Parallel execution of a sequential network simulator" in Proceedings of the 2000 Winter Simulation Conference.

[16] Park, A. and Fujimoto, R. M. 2006. "Aurora: An approach to high throughput parallel simulation" in Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation (PADS'06).

[17] Riley, G., Ammar, M. H., Fujimoto, R. M., Park, A., PeruMalla, K., and Xu, D. 2004. "A federated approach to distributed network simulation". ACM Transaction on Modeling and Computer Simulation 14, 2 (April), 116-148.

[18] Riley, G., Fujimoto, R. M., and Ammar, M. H. 1999. "A generic framework for parallelization of network simulations" in Proceedings of the 1999 Modeling, Analysis and Simulation of Computer and Telecommunication Systems Conference.

[19] Wu, H., Fujimoto, R. M., and Riley, G. 2001. "Experiences parallelizing

a commercial network simulator" in Proceedings of the 2001 Winter Simulation Conference.

[20] Loyd Case. "Multicore Processors Transform at Rapid Pace". What's New@IEEE in Computing, Vol. 8, No. 1, January 2007.

[21] Bader, D.A., Kanade, V., nd Madduri, K. "SWARM: A Parallel Programming Framework for Multicore Processors" in the Proceedings of Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International. 26-30 March 2007

[22] W.G. Huang. "The Performance of the NCTUns Network Simulator using the Event-level Parallelism Approach". 2008.