# 行政院國家科學委員會補助專題研究計畫成果報告

## 多執行緒程式之漏電耗能管理方法

## (Leakage Energy Management for Multithreaded Programs)

計畫類別：■ 個別型計畫　　□ 整合型計畫
計畫編號：NSC 97－2218－E－009－043－MY3
執行期間：　97 年 08 月　01　日至　100 年　7 月　31　日

計畫主持人：游逸平
共同主持人：
計畫參與人員：　羅世融、何柏瑲、田燦榮、蔡羽軒、王深泓、吳翰融、
陳思捷、林聖偉、莊睦昂

成果報告類型(依經費核定清單規定繳交)：□精簡報告　■完整報告

本成果報告包括以下應繳交之附件：
□赴國外出差或研習心得報告一份
□赴大陸地區出差或研習心得報告一份
□出席國際學術會議心得報告及發表之論文各一份
□國際合作研究計畫國外研究報告書一份

處理方式：除產學合作研究計畫、提升產業技術及人才培育研究計畫、列管計畫及下列情形者外，得立即公開查詢
■涉及專利或其他智慧財產權，□一年■二年後可公開查詢

執行單位：國立交通大學資訊工程學系

中　華　民　國　100　年　10　月　29　日

# 目　　錄

頁次

一、　中文摘要 .......... 5

二、　英文摘要 .......... 5

三、　計畫簡介 .......... 6

3.1　計畫緣起及背景 .......... 6

3.2　研究目的 .......... 10

四、　研究方法 .......... 10

4.1　各年度研究簡述 .......... 10

4.1.1　第一年度研究 .......... 10

4.1.2　第二年度研究 .......... 11

4.1.3　第三年度研究 .......... 12

4.2　Predicated-Power-Gating Mechanism (PPG)： .......... 13

4.3　Component Activity Driven Parallel Execution Graph (CADPEG)： .......... 15

4.4　May-Happen-in-Parallel Component-Activity Data-flow Analysis (MHP-CADFA)： .......... 16

4.5　Multithreaded Power-Gating Analysis (MTPGA)： .......... 20

4.6　並行編程模型 (Concurrent Programing Models)： .......... 24

4.7　編譯器與作業系統協同合作電源管理 (Collaborative Operating System and Compiler Power Management)： .......... 27

4.8　多核心架構電源管理(Power Management in Multi-core

**Architecture）： ...... 29**

**五、 結果與討論 ...... 33**

**六、 參考文獻 ...... 36**

## 圖 目 錄

**Figure 1. Trends in power ...... 8**
**Figure 2. Leakage power growth ...... 8**
**Figure 3. Power gating controls for single-thread programs ...... 9**
**Figure 4. Three concurrent threads and their utilization statuses ...... 9**
**Figure 5. A possible run-time scenario ...... 10**
**Figure 6. Power-gating mechanism ...... 13**
**Figure 7. Power-gating mechanism with predication support ...... 14**
**Figure 8. Predicated-power-on for Candidate 1 ...... 14**
**Figure 9. Predicated-power-off for Candidate 1 ...... 14**
**Figure 10. Initialization ...... 14**
**Figure 11. A Component Activity Driven Parallel Execution Graph (CADPEG) example ...... 15**
**Figure 12. May-Happen-in-Parallel Component-Activity Data-flow Analysis equations ...... 17**
**Figure 13. The algorithm of Sequential Component Activity Analysis phase ...... 18**
**Figure 14. The algorithm of Concurrent Component Activity Analysis phase ...... 19**
**Figure 15. Two threads in a may-happen-in-parallel region and their utilization statuses of the power-gating candidates ...... 21**
**Figure 16. A probable placement of predicated-power-gating instructions upon the MTPGA results for Figure 15 ...... 21**
**Figure 17. A illustration of the cost model for MTPGA ...... 22**
**Figure 18. The algorithm of Multithreaded Power-Gating Analysis ...... 23**
**Figure 19. Fork-Join-Model ...... 25**
**Figure 20. CUDA Memory Hierarchy ...... 25**
**Figure 21. OpneCL Memory Hierarchy ...... 27**
**Figure 22. The phases of the collaborative between compiler and operating system ...... 28**

**Figure 23. OpenMP programming model in multicore system.................... 29**
**Figure 24. Parallel region ........................................................................ 30**
**Figure 25. A example of the distribution of the for loop's iterations........... 30**
**Figure 26. The idea of power saving in parallel for loop .............................. 31**
**Figure 27. The example for the idea's problem ............................................. 32**
**Figure 28. A compilation flow of power management for multithreaded programs ........................................................................................... 34**
**Figure 29. The worst, best and average normalized power consumption of thread combinations for DSPstone programs ....................................... 34**
**Figure 30. Component turn-off rate and power consumption of four concurrent executed threads ................................................................ 35**

# 一、 中文摘要

在現代半導體技術中，漏電功率在整體功率中所佔的比例越來越高，原因是電晶體的大小不斷縮小而速度卻不斷提升。最近，已有研究提出硬體搭配編譯器技術來降低漏電的消耗，其作法是透過編譯器分析程式並在程式中適當地安插指令以開啟或關閉特定元件。

然而，在過去的研究中所提出的方法與架構僅適用於單一執行緒的程式，無法針對多執行緒程式進行漏電管理,理由是：執行緒的執行順序在程式執行時才可決定，在編譯時期難以得知執行時期欲進行電源閘控管理元件的使用狀況。

如果直接將過去研究中提出的方法應用在多執行緒程式上，則會產生邏輯上的矛盾：程式中某一個執行緒正準備要使用某個元件時，這個元件已經被另一個執行緒關閉。因此我們需要新的硬體及軟體架構來解決這樣的問題。我們之所以會對多執行緒程式在耗能管理上的議題感到興趣，主因是多執行緒程式設計在現代的程式設計中所扮演的角色愈來愈重要，因為多執行緒可以讓程式充分地使用 CPU 時間，也因此讓程式可以被寫得很有效率，並且，目前正興起的多核心架構也需要多執行緒程式設計來提高它們的效能；況且，多執行緒是處理以事件驅動軟體的最佳選擇，例如現今許多的嵌入式環境、分散式環境、網路環境等。

有鑑於此，在本計畫中我們將以過去相關的研究為基礎，針對在單核心與多核心架構處理器架構下的多執行緒程式提出相關的硬體及編譯器技術以降低處理器的漏電耗能，同時，我們也把電源管理的範圍擴大到編譯器與作業系統間的協同合作，替多執行緒程式提供一套完整的漏電耗能管理方案，以因應低耗能、高效能產品應用的需求。

關鍵詞：編譯器，多執行緒程式，漏電耗能管理

# 二、 英文摘要

Leakage power constitutes an increasing fraction of the total power consumption in modern semiconductor technologies due to the continuing size reductions and increasing speeds of transistors. Recent studies have attempted to reduce leakage power using integrated architecture and compiler power-gating mechanisms. This approach involves compilers inserting instructions into programs to shut down and wake up components as appropriate. However, the approaches proposed in the early studies are only applicable to single-threaded programs. When it comes to multithreaded programs, there should be an alternative hardware and/or software mechanism(s) to deal with the additional concern of the interaction among threads. The uncertainty of the execution sequence of threads makes it difficult for compilers to analyze the utilization status of the power-gating candidates of a multithreaded program. Simply applying power-gating management for multithreaded programs with the previous work will most likely break the logic that a unit must be in the active state, i.e., powered on, before being used for processing since the unit might be powered off by a thread while other concurrent threads are still using or about to use it. The reason why we are

concerning about the energy management for multithreaded programs is that multithreading is becoming an increasingly important part of modern programming. One reason for this is that multithreading enables a program to make the best use of available CPU cycles, thus allowing very efficient programs to be written. Moreover, the rising multi-core architectures need multithreaded programming to boost their performance. Another reason is that multithreading is a natural choice for handling event-driven code, which is so common in today's embedded, highly distributed, and networked environments. In view of this, in this project we propose compiler techniques based on the previous work on power-gating control and multithreading analyses with hardware supports to manage leakage energy consumption for multithreaded programs on both single-core and multi-core processors. Also, we enlarge the management to the collaboration between compilers and operating systems. We provide a comprehensive solution for leakage energy management of multithreaded programs in order to cope with the demand of low-power and high-performance applications.

Keywords: Compilers, Multithreaded programs, Leakage energy management

# 三、 計畫簡介

## 3.1 計畫緣起及背景

低耗能是目前世界的趨勢。廣義地說低耗能可以是為了節省地球上僅有的資源，為後代子孫保存生存的權力；若將焦點拉近到現實生活的應用上，低耗能這個名詞卻成為電子產品重要的指標。尤其對於現今普及的攜帶式電子產品，低耗能代表的是更長的使用時間，因為它們的動力全來自於電池，若耗電量愈小，使用時間自然增加，而隨之衍生的能源消耗問題則成為其銷售成效的主要關鍵。由於嵌入式系統處理器生命週期短、程式碼高效能及低耗能的需求，使得編譯器在嵌入式系統這個領域逐漸扮演關鍵性的角色，要有完整的開發環境，才能較快地產生產品。

電子元件中一般功率消耗的來源，主要為互補金氧半導體(CMOS)電路中因為漏電流(leakage current)損失導致的靜態功率消耗(static power dissipation)，或因為切換暫態電流(switching transient current)和負載電容之充放電所產生的動態功率消耗(dynamic power dissipation)。由 A. Chandrakasan[19]的研究中我們可以得知一個 CMOS 元件的電源消耗模式如下所示：

$$P = \alpha \cdot CL \cdot V_{dd} \cdot f_{clk} + \alpha \cdot Q_{SC} \cdot V_{dd} \cdot f_{clk} + I_{leak} \cdot V_{dd}$$

上式中 P 為功率消耗，第一項為動態功率耗量，即為電路節點充電與放電所需的電源消耗，其中 α 為節點的切換機率(switching activity)，*CL* 為節點的電容量，$V_{dd}$ 為供應電壓，$f_{clk}$ 為運作時脈，這個功率消耗來自於電晶體輸出端從 1 到 0 或從 0 到 1 變化時，負載電容充放電所導致之功率消耗。第二項為短路功率耗量，是當輸出轉變時電流流經節點到接地所消耗的電源，此電流又稱短路電流，其中 $Q_{SC}$ 為每次轉換時短路電流所攜帶的電荷量，這部份功率消耗係指在元件內部的功率消耗，它包刮當 Gate 在做切換時，在 $V_{dd}$

Gnd 之間的短路電流所造成的功率消耗。第三項為靜態功率耗量(亦稱作漏電率耗量)，是由漏電電流 $I_{leak}$（leakage current）所造成的電源消耗，這個功率消耗是由於介於擴散級(diffuse)和基級(substrate)反偏壓的 PN 界面間的漏電流索引起，當電路沒有啟動時，漏耗電功率占消耗功率的大部分。

在採用設計良好的邏輯閘製出的 VLSI 電路裡，有 90%的耗電都是由切換動作所造成的，亦即動態功率耗量，因此減少程式執行的切換動作也成為低耗電的關鍵。然而，隨著互補金氧半導體電路設計的進步，先進的製程技術已經達到深次微(deep-submicron)的階段，靜態功率耗量佔了整個消耗功率的比率逐漸增加，在原有 1.0μm 製程下，靜態功率耗量僅有動態功率耗量的萬分之一，但是到了 0.13μm 以下的製程技術，靜態功率耗量已逐漸逼近動態功率耗量，成長的速度更是動態功率耗量的一百倍，如 Figure 1 及 Figure 2 所示。另外，電路的工作溫度也影響著功率的消耗，從 V. De and S. Borkar 的文章中可得知，在愈高的工作溫度下，靜態功率耗量所佔整個消耗功率的比例愈大。靜態功率消耗 $P_{static}$ 的功率消耗公式為：

$$P_{static} = V_{dd} \times N \times k_{design} \times I_{leakage}$$

其中，$V_{dd}$ 表示電晶體之輸入電壓、$N$ 表示電晶體數目、$k_{design}$ 表示系統設計常數，及 $I_{leakage}$ 則表示積體電路由反向偏壓(reverse bias leakage)造成的漏電流。

由於處理器或系統晶片中的電晶體具有上述靜態的功率耗損因素，降低系統晶片耗電量的重要性與日俱增，而透過低功率電路的設計與動態耗能管理(dynamic power management)，可使系統晶片在正常運算下，不會時常處於高溫甚至過熱的狀態，減少了散熱的問題，且廠商也不用為了替過燙晶片散熱付出更多的成本在晶片封裝上，使線路的可靠性提高，並同時增加晶片的使用壽命。為降低處理器或系統晶片漏電功率消耗，許多以編譯器技術搭配電源閘控(power gating)方法的研究論文被相繼提出，例如 2002 年 LCPC 及 CC 會議、2003 年 DATE 會議、2005 年 EMSOFT 會議及 2006 年、2007 年 ACM TODAES 期刊中等論文中，提供了以電源閘控指令(power-gating instruction)的方式對給定的程式進行電源閘控指令佈置來降低各處理元件的漏電能量消耗。上述方法皆使用編譯器技術分析程式在處理器中的使用情況，並搭配硬體的電源閘控機制，適當地安插電源閘空指令(power-gating instruction)以關閉空閒的處理元件。如 Figure 3 所示，經由編譯器分析後可得知程式在處理器中某一個功能元件(function unit)的使用狀態。假設黃色區域代表功能元件在時間軸上的使用狀態，立體區塊代表該元件正在被使用中，而平面區塊代表該元件處於空閒狀態，因此編譯器則在該元件被使用前後的時間點分別安插開啟(紅色箭頭)與關閉指令(藍色箭頭)。

然而，前述這些耗電耗能管理方法僅適用於單一執行緒程式（single-thread program），對於多執行緒程式（multithreaded programs）則因為編譯器無法掌握執行緒間的互動關係、無法精確得知各處理元件的使用狀況而束手無策。舉一個簡單的例子來說，Figure 4 所顯示的是某多執行緒程式中三個可能同時執行的執行緒（分別為 Thread 1、Thread 2 及 Thread 3），若我們直接將前述的研究成果應用至此多執行緒程式，亦即針對每個執行緒分析功能元件的使用狀態，並在

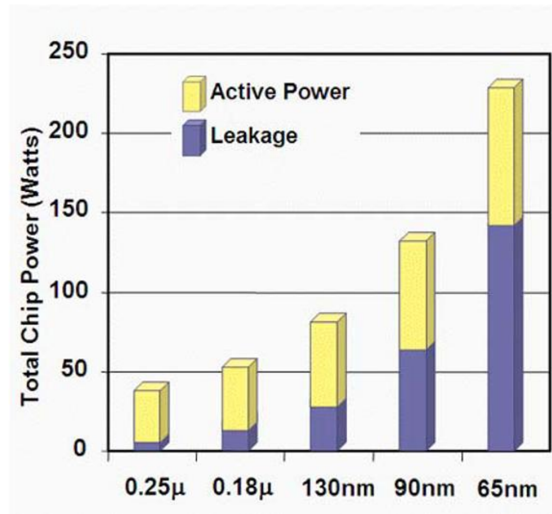


Figure 1. Trends in power

每個執行緒中功能元件的使用前後分別安插開啟與關閉指令，並假設 Thread 1 與 Thread 2 這兩個執行緒在 run-time 的執行順序為 Thread 1(尚未執行結束)→Thread 2(尚未執行結束)→Thread 1(執行結束)→Thread 2(執行結束)，即 Figure 5 中所顯示的執行順序，我們可得知在 run-time 時功能元件的開、關時機如下：

1. 程式的執行權為 Thread 1，Thread 1 在使用元件前先將元件開啟。
2. 執行權跳到 Thread 2，Thread 2 在使用元件前又開啟了元件。
3. 執行權回到 Thread 1，Thread 1 在使用元件後將元件關閉。
4. 執行權回到 Thread 2，Thread 2 準備繼續使用元件，但元件為關閉狀態。

我們會發現：開啟或關閉指令的執行時機違反了正常邏輯，即元件在被使用前的狀態為關閉狀態。因此，在這個計畫中，我們將提出一套搭配軟、硬體方法來解決無法有效管理多執行緒程式漏電耗能的窘境。

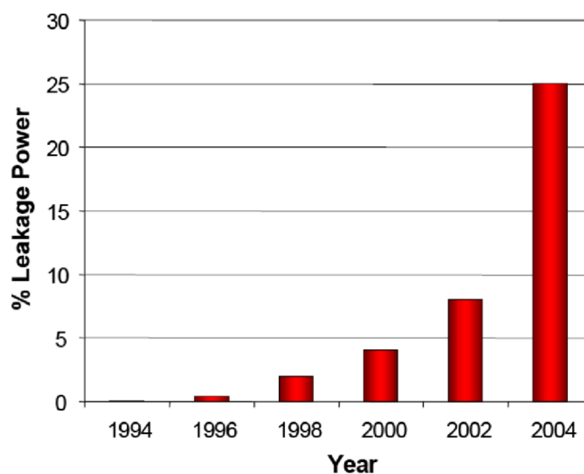


Figure 2. Leakage power growth

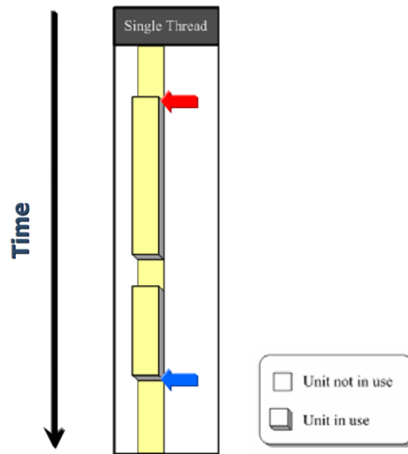


Figure 3. Power gating controls for single-thread programs

另外，隨著資訊、電子科技的發展，嵌入式系統在多媒體、遊戲、通訊等相關市場上佔有越來越大的地位，以單一處理器為主系統已無法滿足強大的運算需求。在消費性嵌入式系統產品這樣諸多考量皆大於效能的領域，將成本投資在一顆昂貴的而耗電的高效能處理器上，遠不如將工作分配給多顆處理器，以平行處理的方式來達到相同效能且降低成本，這樣的多核心處理器(multi-core processor)在市面上已成趨勢，如 Intel Xeon 六核心處理器以及使用可九個核心的 IBM Cell 處理器等。而隨著多核心處理器問世之後，若要讓每個核心達到最佳的效能，軟體發展人員在規劃設計應用程式時，將必須有並行處理(concurrency)的思維，以便設計出可發揮多核心系統架構威力的軟體。並行處理意指把單一任務分解城多個模組，並分配至不同的核心同時進行處理及計算，再將各個模組的結果進行統整，予以組合得到其處理結果。因此，開發人員們也必須使用多執行緒才能從多核心處理器中獲益，因為方有使用多執行緒來撰寫軟體才能有效發揮多核心架構的長處。這意味著多執行緒

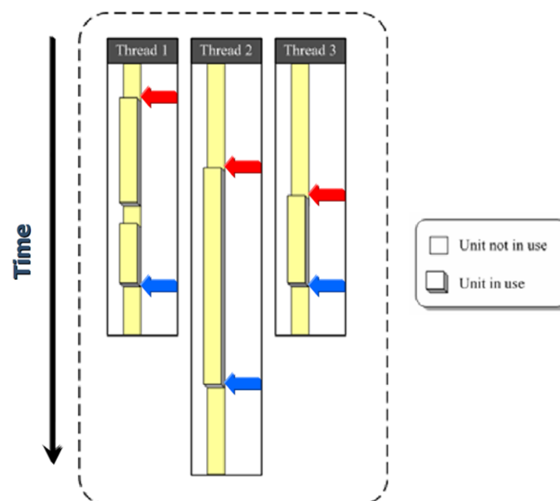


Figure 4. Three concurrent threads and their utilization statuses

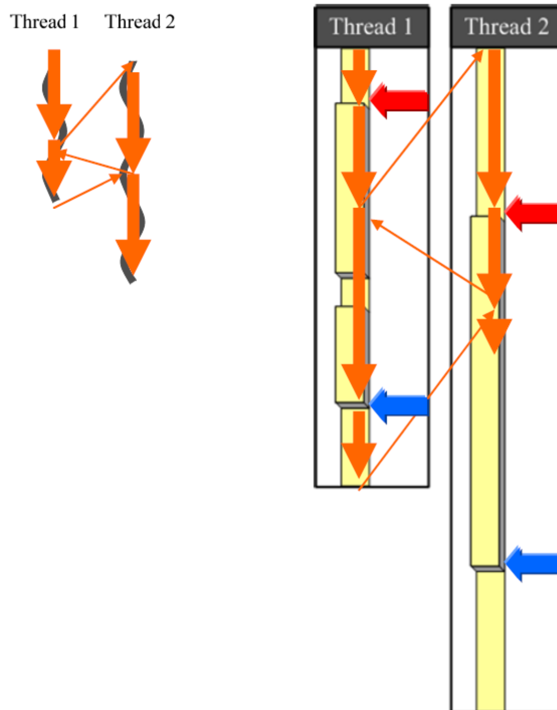


Figure 5. A possible run-time scenario

的應用將更為廣泛，並且大部分的應用將會是在講求低耗能的嵌入式系統上，而本計畫正為解決多執行緒電源管理而生，計畫的成果也可應用於多核心的處理器中。除此之外，我們也把電源管理的範圍擴大到編譯器與作業系統間的協同合作，替多執行緒程式提供一套完整的漏電耗能管理方案，以因應低耗能、高效能產品應用的需求。

## 3.2 研究目的

在現代半導體技術中，漏電功率在整體功率中所佔的比例愈來愈高，原因是電晶體的大小不斷縮小而速度卻不斷提昇。最近，已有研究提出硬體搭配編譯器技術來降低漏電的消耗，其作法是透過編譯器分析程式並在程式中適當地安插指令以開啟或關閉特定元件，然而，在過去研究中所提出的方法與架構僅是用於單一執行緒的程式，無法針對多執行緒程式進行漏電管理。因此我們需要新的硬體及軟體架構來解決這樣的問題。我們之所以會對多執行緒程式在耗能管理上的議題感到興趣，主因是多執行緒程式設計在現代的程式設計中所扮演的角色愈來愈重要，因為多執行緒可以讓程式充分地使用 CPU 時間，也因此讓程式可以被寫得很有效率，並且，目前正興起的多核心架構也需要多執行緒程式設計來提高它們的效能；況且，多執行緒是處理以事件驅動軟體的最佳選擇，例如現今許多的嵌入式環境、分散式環境、網路環境等。

# 四、 研究方法

## 4.1 各年度研究簡述

### 4.1.1 第一年度研究：

在第一年度計畫中，我們提出了一套新的條件式電源閘控(Predicated Power Gating)硬體機制。並且利用編譯器技術

及已知的 MHP (May-Happen-in-Parallel)技術分析程式可能的並行區間，進而提出一套適用於條件式電源閘控機制的管理方法，稱為多執行緒電源閘控分析(Multithreaded power-gating analysis, MTPGA)方法。我們為多執行緒程式的漏電耗能管理提出了一套完整的解決方法，並在下列各主要環節上取得豐碩的成果：(1) 對於多執行緒的可能並行運算分析做了深入的研究，了解現有分析技術能力，並將其應用至本計畫中。(2) 提出條件式電源閘控硬體機制的概念，以解決舊有機制無法應用至多執行緒程式漏電耗能管理的問題。(3) 利用 MHP 及過去研究的分析資訊，提出一套多執行緒電源閘控分析方法，用以管理及佈置條件式電源閘控指令(Predicated-power-gating instructions)。以下將對第一年度的各項研究成果做一簡述。

**(1) May-Happen-in-Parallel (MHP) Analysis：**

使用文獻中的技術分析多執行緒程式中可能並行的程式區間，我們在文獻中找到在 MHP 分析技術領域中重要的研究論文，並加以研讀，作為本項研究的 MHP 分析基礎。

**(2) Predicated-Power-Gating Mechanism (PPG)：**

條件式指令(predicated instruction)的概念主要是透過參考指令中的條件執行的 predicate 值是 True 或 False。若 predicate 值為 True，則正常地執行該指令；但若 predicate 值為 False，該指令將不會被執行且不會對於系統的狀態造成任何的更動。我們將條件式執行(predicate execution)的機制引進至電源閘控裝置(power-gating device)上，以便能在多執行緒環境下正確地應用電源閘控技術。

**(3) Multithread Power-Gating Analysis (MTPGA)：**

MTPGA 是以 CADPED、MHP-CADFA 所收集而得的各個元件使用的資訊作為依據及參考，並再加上 PPG 硬體技術的協助來預估多執行緒程式環境下的電源使用的情形以及插入條件式電源閘控指令(Predicated-Power-Gating instructions)以達到省電的效果。而在這其中我們也針對了標準的電源閘控架構、條件式電源閘控架構說明了兩者的總體耗能，並且對於總體耗能進行比較，進而建立了一個判斷式來決定是否要使用 MTPGA，並將整個 MTPGA 的流程寫成演算法形式。

**4.1.2 第二年度研究：**

第二年度計畫的主要目標在於多核心系統架構的漏電耗能管理，我們延續第一年成果，並針對現有常用的並行編程模型(concurrent programming model) 進行深入探討，以便在不同的並行編程模型及多核心系統架構下進行漏電耗能管理。在本年度計畫中，我們在下列各主要環節上取得豐碩的成果：(1)對於並行編程模型做深入研究，了解現有模型能力。(2)延續第一年度的成果，提出 may-be-used-in-parallel component-activity data-flow analysis (MUP-CADFA) 分析方法。以下將對第二年度的各項研究成果做一簡述。

**(1) 並行編程模型深入研究**

(Concurrent Programming Models)：

針對現行常用三種並行編程模型(OpenMP、CUDA、OpenCL)進行深入研究，並了解各模型優、缺點及其特性。

**(2) Component Activity Driven Parallel Execution Graph (CADPEG)：**

CADPED 是由 Parallel Execution Graph (PEG) 作為基礎，為了能讓後續的 MHP-CADFA 更容易地分析及收集多執行緒之間元件使用的資訊進而開發出來，而我們也在此定義了元件使用資訊的表示法、Event producing node (P node)、Event consuming node (C node) 以及為了收集上述兩種 nodes 所需的集合。

**(3) May-Happen-in-Parallel Component-Activity Data-Flow Analysis (MHP-CADFA)：**

MHP-CADFA 透過使用 CADPED 來保守地分析及收集多執行緒之間可能的並行區間 (May-Happen-in-Parallel region)的元件使用資訊，以供 MTPGA 作為分析的依據及參考。其中，收集資訊的步驟也分為 Sequential Component Activity Analysis phase (SCAA) 及 Concurrent Component Activity Analysis phase (CCAA)兩個階段來分別收集循序階段以及平行階段的元件使用資訊。SCAA 部分主要是收集程式碼循序部分的元件使用資訊；而 CCAA 部分則是以 SCAA 部分所得到的資訊進而再去針對多執行緒之間可能的並行區間收集元件使用資訊。在此我們也將 SCAA 以及 CCAA 兩個部分的整個流程寫成演算法形式。

### 4.1.3 第三年度研究：

在第三年度計畫中，我們針對編譯器與作業系統間互動(interaction)或協同合作以及多核心系統架構系統上的漏電耗能管理進行探討。在本年度計畫中，我們在下列各主要環節上取得豐碩的成果：(1)對於編譯器與作業系統協同合作的電源管理進行探討，了解現有的技術及架構。(2)對於多核心架構提出一個以 OpenMP 為主要語言架構的漏電耗能管理的想法所可能會遇到的問題進行探討。以下將對第三年度的各項研究成果做一簡述。

**(1) 編譯器與作業系統協同合作電源管理 (Collaborative Operating System and Compiler Power Management) 探討：**

針對即時應用程式(real-time application)並且利用 dynamic voltage scaling (DVS)技術作為探討的基礎架構，對於 application model、energy model 及 overhead model 進行探討與分析。而我們也簡述了編譯器與作業系統合作的流程，最後則是比較了利用編譯器與作業系統合作和單獨使用作業系統或單獨使用編譯器的差別。

**(2) 多核心架構電源管理 (Power Management in Multi-core Architecture) 探討：**

我們針對在多核心架構並且以 OpenMP 處理平行處理作為探討的基礎下，對於 OpenMP 在平行 for loop 部份實做省電機制的時機以及可能會遇

到的問題進行探討與分析。

以下我們便開始針對各項研究重點做詳細的說明。

## 4.2 Predicated-Power-Gating Mechanism (PPG)：

Figure 6 顯示的是標準電源閘控機制的結構示意圖。而 Figure 7 則是我們所提出具有條件式(predicate)概念支援的電源閘控機制的結構示意圖。Figure 6 及 Figure 7 中皆假定為是在具有 M 個硬體執行緒的機器上，而在其中有 N 個可電源閘控的元件($C_1, C_2,\ldots,C_N$)。

在 Figure 6 (標準電源閘控機制結構)中，具有兩個重要的元件：

(1) N 個 bits 的電源閘控控制暫存器(power gating control register)，其目的是為了記錄且控制 N 個可電源閘控元件的電源($pgc_1,pgc_2,...,pgc_N$)。

(2) 電源閘控控制器(power gating controller)，其目的是根據電源閘控開關指令來改變儲存在電源閘控控制暫存器的值。

正常電源閘控系統的運作方式有兩種：

(1) 利用 P 型金氧半場效電晶體(PMOS)控制虛擬電壓(virtual $V_{dd}$)。

(2) 利用 N 型金氧半場效電晶體(NMOS）限制漏電電流(leakage curren)。

而 Figure 6 便是將電源閘控控制暫存器連結到 P 型金氧半場效電晶體以改變虛擬電壓。一旦電源閘控控制器更新了電源閘控控制暫存器當前狀態，則對應的可電源閘控的元件便會被立即的開啟或關閉。

Figure 7 (具有條件式概念支援的電源閘控機制的結構)的結構相似於標準電源閘控機制結構，但是有新增三個元件來支援條件式概念的支援：

(1) N 個 bits 的電源閘控條件暫存器(power-gating predicate register；$pgp_1,pgp_2,\ldots pgp_N$)

(2) $N\times\lceil\log_2 M\rceil$個 bits 參考計數暫存器(reference counter register；$rc_1,rc_2,\ldots rc_N$)

(3) N 個條件式電閘

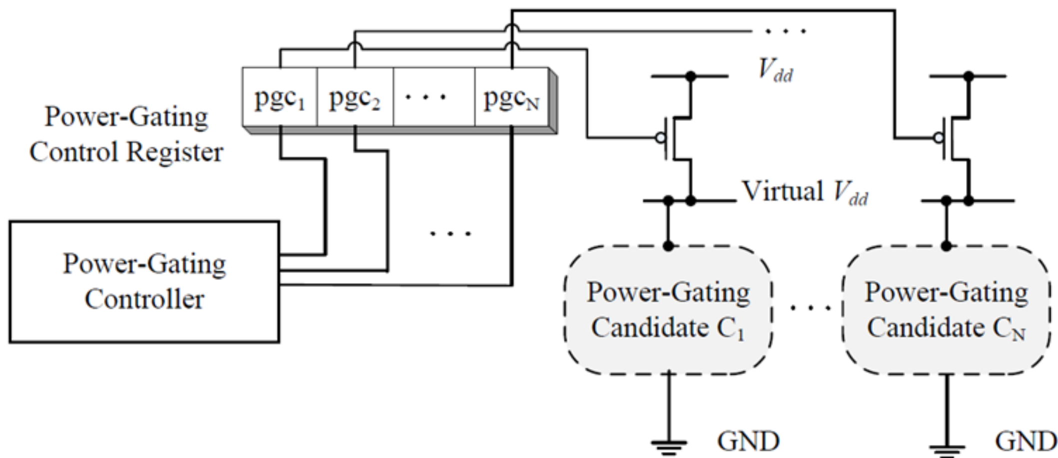


Figure 6. Power-gating mechanism

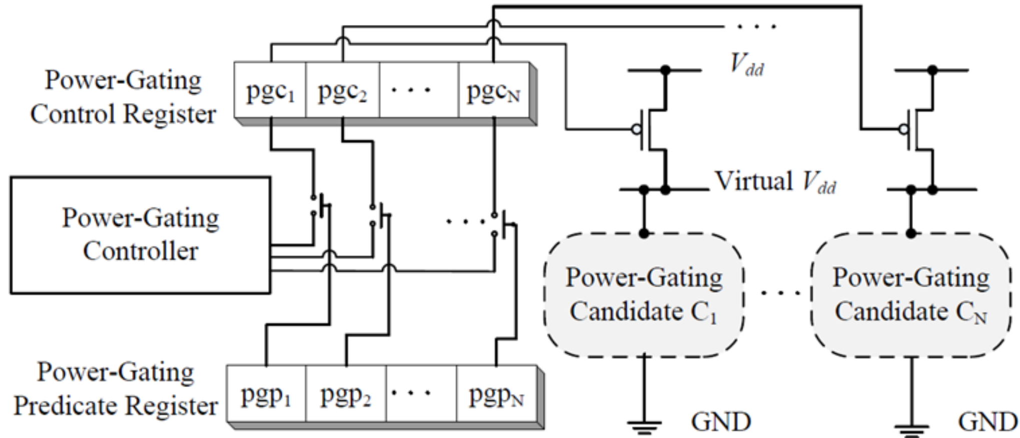


Figure 7. Power-gating mechanism with predication support

接著，我們便要對條件式電源閘控運算做一個詳細的定義及說明：

(1) Predicated-power-on operation：

i. Power on Candidate i if predicate bit $pgp_i$ is set.

ii. Increase the reference counter of Candidate i ($rc_i$), which keeps track of the number of threads that reference the power-gating candidate at this time, by one.

iii. Unset predicate bit $pgp_i$.

(2) Predicated-power-off operation：

i. Decrease the reference counter of Candidate i ($rc_i$) by one.

ii. Set predicate bit $pgp_i$ if reference counter $rc_i$ is zero.

iii. Power off Candidate i if predicate bit $pgp_i$ is set.

(3) Initialization operation：

The initialization operation is designed for cleaning all predicate bits and emptying all reference counters when processor is starting up.

Figure 8 及 Figure 9 分別顯示了針對 Candidate 1 的 Predicated-power-on operation 及 Predicated-power-off operation 的虛擬程式碼區段。Figure 10 則顯示了 Initialization operation 的虛擬程式碼區段。

```
        lock lc1;
(pgpi)  poweron C1;
        rc1 = rc1 + 1;
        pgp1 = 0;
        unlock lc1;
```

Figure 8. Predicated-power-on for Candidate 1

```
        lock lc1;
        rc1 = rc1 - i;
        pgp1 = (rc1 == 0);
(pgpi)  poweroff C1;
        unlock lc1;
```

Figure 9. Predicated-power-off for Candidate 1

```
pgp1 = pgp2 = ... = pgpN = 0;
rc1 = rc2 = ... = rcN = 0;
```

Figure 10. Initialization

## 4.3 Component Activity Driven Parallel Execution Graph (CADPEG)：

為了能夠更容易地分析多執行緒程式之間元件使用的情形，我們以 Parallel Execution Graph(PEG)為基礎，更進一步地發展出 Component Activity Driven Parallel Execution Graph(CADPEG)。以下便對我們發展出的CADPEG做詳細說明，Figure 11 為一個 CADPEG 的範例圖。

在 CADPEG 中，我們對於基本構成做了以下定義：

(1) 一個 node 皆為(object , name , caller)的形式，且各 node 皆有一個獨一無二的編號，例如，$n_1,n_2,\ldots n_N$。而 CADPEG 便是這些 node 來表示它們之間的執行緒層面的行為。

(2) 執行緒以符號 t 表示，例如，$t_1,t_2,\ldots t_N$

(3) 系統中的各個元件則以符號 C 表示，例如，$C_1,C_2,\ldots C_N$.

(4) 件使用狀況亦即以符號 $C_n^m$ 來表示，m 代表 node 號碼，而 n 代表元件號碼。例如，$C_1^{10}$ 代表 node 10 會使用到 component 1。

而 CADPEG 具有一個特性是，它會將多執行緒事件(multithread evevts)分為兩類：

(1) Event producing node，以符號 P 表示

(2) Event consuming node，以符號 C 表示

以 Figure 11 為例，圖中的 creation nodes (* , start , *)及 notify node (* , notify , *)就 被分類為 event producing nodes P，它們會分別產生出對應並行的(* , begin , *)及(* , notified-entry , *)這兩種 event consuming nodes C。以 event consuming node 的觀點來看，event producing node 限制了它們的執行；換句話說，event consuming node 必定會等至 event producing node 執行完後才會執行。

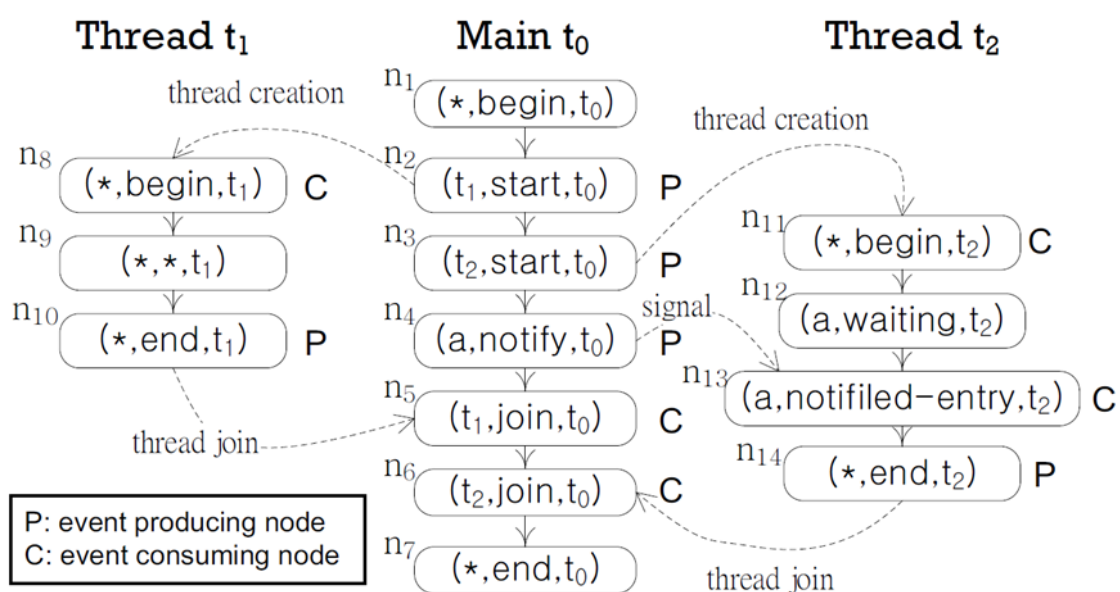


Figure 11. A Component Activity Driven Parallel Execution Graph (CADPEG) example

而對於這樣的 event producing node 及 event consuming node，我們更進一步的再定義兩種集合

(1) cons(n)是收集 event producing node n 的相對應的所有 event consuming nodes。

(2) prod(n)是收集 event consuming node m 的相對應的所有 event producing nodes。

依舊以 Figure 11 為例，$n_3$ 為 event producing node，而 $n_{11}$ 為其對應的 event consuming node，所以 $cons(n_3) = \{n_{11}\}$，$prod(n_{11}) = \{n_3\}$。

## 4.4 May-Happen-in-Parallel Component-Activity Data-flow Analysis (MHP-CADFA)：

藉著 4.3 發展出的 CADPEG，我們就可去分析多執行緒程式之間元件使用的情形，以 May-Happen-in-Parallel (MHP)analysis 作為基礎，更進一步地發展 May-Happen-in-Parallel Component-Activity Data-flow Analysis (MHP-CADFA)，Figure 12 為 MHP-CADFA 所有會使用的數學公式。

MHP-CADFA 主要由兩個部分組成：

(1) Sequential Component Activity Analysis phase (SCAA)：

在 SCAA 此一部分，主要是收集每一個 CADPEG node 的各元件使用情形以及執行緒的資訊，以供 CCAA phase 使用。而為了要記錄這些資訊，我們對於每一個 CADPEG node 皆定義了 $CA_{IN}(n)$, $CA_{OUT}(n)$, $CA_{GEN}(n)$, $CA_{KILL}(n)$, $CA_{Util}(n)$ 這些集合；此外，針對每一個執行緒也定義了 $CA_T(t)$集合。以下便一一對各個集合作說明：

1. $CA_{GEN}(n)$：
當程式執行 node n 時，若是需要用到某個 component，就將之記錄在這個集合中。

2. $CA_{KILL}(n)$：
當程式執行 node n 時，若是不需要用到某個 component，就將之記錄在這個集合中。

3. $CA_{IN}(n)$是紀錄當程式執行至準備進去 node n 時，當下 components 是否有被使用的情形。計算公式是 Figure 12 的(1)式。

4. $CA_{OUT}(n)$是紀錄當程式執行完 node n 時，當下的 components 是否有被使用的情形。計算公式是 Figure 12 的(2)式。

5. $CA_{Util}(n)$是紀錄的資訊和 $CA_{OUT}(n)$ 集合是相同的，但是差別在於 $CA_{OUT}(n)$只記錄 component number；然而，$CA_{Util}(n)$ 集合則多紀錄了 node 的資訊。舉例來說，若在 $CA_{OUT}(3)$中有一個元素是 $C_1$，則在 $CA_{Util}(3)$則會將這筆資訊紀錄為 $C_1^3$。計算公式是 Figure 12 的(3)式。

6. $CA_T(t)$是將所有屬於 thread t 的 CADPEG nodes 的 $CA_{Util}$ 集合做聯集，以便能夠做 component 使用情形的保守估計。計算公式是 Figure 12 的(4)式。

Figure 13 所顯示的是 SSCA phase 的演算法。

$$\mathrm{CA}_{IN}(n) = \bigcup_{m:\text{ a predecessor of } n} \mathrm{CA}_{OUT}(m) \quad (1)$$

$$\mathrm{CA}_{OUT}(n) = \mathrm{CA}_{GEN}(n) \cup (\mathrm{CA}_{IN}(n) - \mathrm{CA}_{KILL}(n)) \quad (2)$$

$$\mathrm{CA}_{Util}(n) = \bigcup C_i^n, \text{ where } C_i \in \mathrm{CA}_{OUT}(n) \quad (3)$$

$$\mathrm{CA}_{T}(t) = \bigcup_{\forall n \in t} \mathrm{CA}_{Util}(n) \quad (4)$$

$$\mathrm{MCA}_{GEN}(n) = \begin{cases} \bigcup_{m \in cons(n)} \mathrm{CA}_{Util}(m) & \text{if } n \in P, \\ \bigcup_{m \in prod(n)} \mathrm{CA}_{Util}(m) & \text{if } n \in C, \\ \emptyset & \text{otherwise.} \end{cases} \quad (5)$$

$$\mathrm{MCA}_{KILL}(n) = \begin{cases} \bigcup_{m \in anc(prod(n))} \mathrm{CA}_{Util}(m) & \text{if } n \in C, \\ \emptyset & \text{otherwise.} \end{cases} \quad (6)$$

$$\mathrm{MCA}_{KILL'}(n) = \bigcup_{m:\text{a predecessor of } n} (\mathrm{MCA}_{KILL'}(m) \cup \mathrm{MCA}_{KILL}(m)) - \mathrm{MCA}_{GEN}(m) \quad (7)$$

$$\mathrm{MCA}_{IN}(n) = \bigcup_{m:\text{ a predecessor of } n} \mathrm{MCA}_{OUT}(m) \cup \begin{cases} \bigcup_{k \in prod(n)} \mathrm{MCA}_{OUT}(k) & \text{if } n \in C, \\ \emptyset & \text{otherwise.} \end{cases} \quad (8)$$

$$\mathrm{MCA}_{OUT}(n) = \mathrm{MCA}_{GEN}(n) \cup (\mathrm{MCA}(n) - (\mathrm{MCA}_{KILL}(n) \cup \mathrm{MCA}_{KILL'}(n))) \quad (9)$$

$$\mathrm{MCA}_{Parallel}(n) = \bigcup C_i, \text{ where } C_i^n \in \mathrm{MCA}_{IN}(n) \quad (10)$$

Figure 12. May-Happen-in-Parallel Component-Activity Data-flow Analysis equations

(2) Concurrent Component Activity Analysis phase(CCAA)：

在 CCAA 此一部分，主要是收集 may-happen-in-parallel component-activity 的資訊。而為了要記錄這些資訊，我們對於每一個 CADPEG node 又再定義了 $\mathrm{MCA}_{IN}(n)$, $\mathrm{MCA}_{OUT}(n)$, $\mathrm{MCA}_{GEN}(n)$, $\mathrm{MCA}_{KILL}(n)$這些集合。在對每個集合做說明之前，我們還得要在定義三個集合用來敘述 CADPEG node 之

**Algorithm 1:** The algorithm of *Sequential Component Activity Analysis (SCAA)* phase

**Input** : A multithreaded program and its relative DACPEG.
**Output**: The DACPEG with component activities information of nodes $CA_{Util}()$ and component utilization information of threads $CA_T()$.

1 **foreach** *basic block* $b \in$ *input program* **do**
2   **foreach** *node* $n \in b$ **do**
3     $CA_{IN}(n) := CA_{OUT}(n) = \emptyset$ ;
4     $CA_{GEN}(n) := CA_{GEN}(n) \cup \{C_n\}$, where $C_n \in$ components that node $n$ will use ;
5     $CA_{KILL}(n) := CA_{KILL}(n) \cup \{C_m\}$, where $C_m \in$ components that node $n$ will not use;
6   **end**
7   **repeat**
8     $CA_{IN}(n) = \bigcup_{n' \in \text{a predecessor of } n} CA_{OUT}(n')$ ;
9     $CA_{OUT}(n) = CA_{GEN}(n) \cup (CA_{IN}(n) - CA_{KILL}(n))$ ;
10   **until** $\forall_{m \in b}$ *$CA_{IN}(m)$ and $CA_{OUT}(m)$ converge*;
11 **end**
   `Collect component utilization information among threads and nodes.`
12 **foreach** *DACPEG node* $n \in$ *thread* $t$ **do**
13   $CA_{Util}(n) = \cup C_m^n$, where $C_m \in CA_{IN}(n)$ ;
14   $CA_T(t) = \cup CA_{Util}(n)$ ;
15 **end**

Figure 13. The algorithm of Sequential Component Activity Analysis phase

間關係：

1. anc(n)是遞迴地去收集 node n 的 predecessors 以及這些 predecessors 的全部祖先。
2. cons(n)是收集 event producing node n 的相對應的所有 event consuming nodes。
3. prod(n) 是收集 event consuming node m 的相對應的所有 event producing nodes。

接著便針對 $MCA_{IN}(n)$, $MCA_{OUT}(n)$, $MCA_{GEN}(n)$, $MCA_{KILL}(n)$這些集合做說明：

1. $MCA_{GEN}(n)$收集的資訊分為三種情形：

<1> 當 node n 是 P 時，主要是收集

**Algorithm 2:** Data analysis algorithm for concurrent component usage

**Input** : A multithreaded program and its relative CADPEG.
**Output**: The program with may-happen-in-parallel component-activity information.

```
1  foreach basic block b ∈ input program do
       Initialization
2      foreach instruction i ∈ b do
3          MCA_IN(i) = MCA_OUT(i) = φ
4          MCA_GEN(i) = MCA_KILL(i) = φ
5          MCA_KILL'(i) = φ
6          if i ∈ P then
7              MCA_GEN(i) = ⋃_{i'∈cons(i)} CA_UTL(i')
8          end
9          else if i ∈ C then
10             MCA_GEN(i) = ⋃_{i'∈prod(i)} CA_UTL(i')
11             MCA_KILL(i) = ⋃_{i'∈anc(Prod(i))} CA_UTL(i')
12         end
13     end
       Iterative approach to compute all sets
14     repeat
15         MCA_KILL'(i) = ⋃_{i':a predecessor of i}
                 (MCA_KILL'(i') ∪ MCA_KILL(i')) − MCA_GEN(i')
16         MCA_IN(i) = ⋃_{i':a predecessor of i} MCA_OUT(i')
17         if i ∈ C then
18             MCA_IN(i) = MCA_IN(i) ⋃_{i'∈prod{i}}
               MCA_OUT(i')
19         end
20         MCA_OUT(i) = MCA_GEN(i) ∪ (MCA(i)
                 − (MCA_KILL(i) ∪ MCA_KILL'(i)))
           Symmetrize all CADPEG component usage
21         foreach C^n_* ∈ MCA(i) do
22             if CA_UTL(i) ⊄ MCA(n) then
23                 MCA(n) = MCA(n) ∪ CA_Util(i)
24             end
25         end
26     until all sets converge;
27     foreach instruction i ∈ b do
28         MCA_Parallel(i) = ⋃ C_m, where C^i_m ∈ MCA(i)
29     end
30 end
```

Figure 14. The algorithm of Concurrent Component Activity Analysis phase

它所產生的 C nodes 的元件使用資訊。

<2> 當 node n 是 C 時，主要是收集它所對應的 P nodes 的元件使用資訊。

<3> 其餘情形皆是空集合。

計算公式是 Figure 12 的(5)式。

2. $MCA_{KILL}(n)$收集的資訊分為兩種情形：

<1> 當 node n 是 C 時，主要是收集它所對應的 P nodes 以及各個 P node 的祖先的元件使用資訊。

<2> 其餘情形皆是空集合。

計算公式是 Figure 12 的(6)式。

而關於那些確定在平行區塊已經不被使用的元件使用資訊，我們使用另外一個集合將它們收集起來，這是為了讓後面的分析可以清楚地知道在平行區塊開始之前各個元件使用的詳細資訊。

計算公式是 Figure 12 的(7)式。

3. $MCA_{IN}(n)$ 是收集 node n 的所有 predecessors 的 $MCA_{OUT}$ 的聯集。

計算公式是 Figure 12 的(8)式。

4. $MCA_{OUT}(n)$主要收集的元件資訊有兩部分：

<1> Node n 本身自己所需要的元件資訊。

<2> 除了 node n 所在的執行緒外，其他執行緒在平行區塊所可能會用到的元件資訊。

計算公式是 Figure 12 的(9)式。

最終，我們將會把每個指令在執行時自身所會用到的元件資訊以及透過 CCAA phase 演算法所得知的其他並行執行的 threads 所可能會用到的元件資訊全部收集在 $MCA_{PARALLEL}(n)$這個集合，如此便完成 MHP-CADFA 收集的資訊的步驟。

## 4.5 Multithreaded Power-Gating Analysis (MTPGA)：

綜合 4.2、4.3 及 4.4 部分提出的研究方法所收集而得的結果，我們以條件式電源閘控機制提出一套適用於多執行緒程式環境的漏電耗能管理方法，稱為多執行緒電源閘控分析 (Multithreaded power-gating analysis, MTPGA)方法。

MTPGA 最主要的概念便是在一個可做電源閘控的元件上，第一次需要開啟時插入 Predicated-power-on operation；而在最後使用完畢後插入 Predicated-power-off operation。我們用以下 Figure 15 及 Figure16 兩張圖來說明這一個概念。

Figure 15 中所顯示的是某個多執行緒程式中兩個可能同時執行的執行緒(分別為 Thread 1 及 Thread 2)，它們在可能並行執行區間中的三個電源閘控的元件($C_1$、$C_2$ 及 $C_3$)的使用情形。

Figure 16 中則顯示了一種根據 MTPGA 的結果所產生的一種條件式電源閘控指令插入位置的可能情況。一個黑色的箭頭代表一個 Predicated-power-on operation；而一個灰色的箭頭代表一個 Predicated-power-off operation。在條件式電源閘控指令的支援下，電源閘控指令只會在該元件第一次被使用時將其打開及最後一次使用完畢後將其關閉，其餘位置

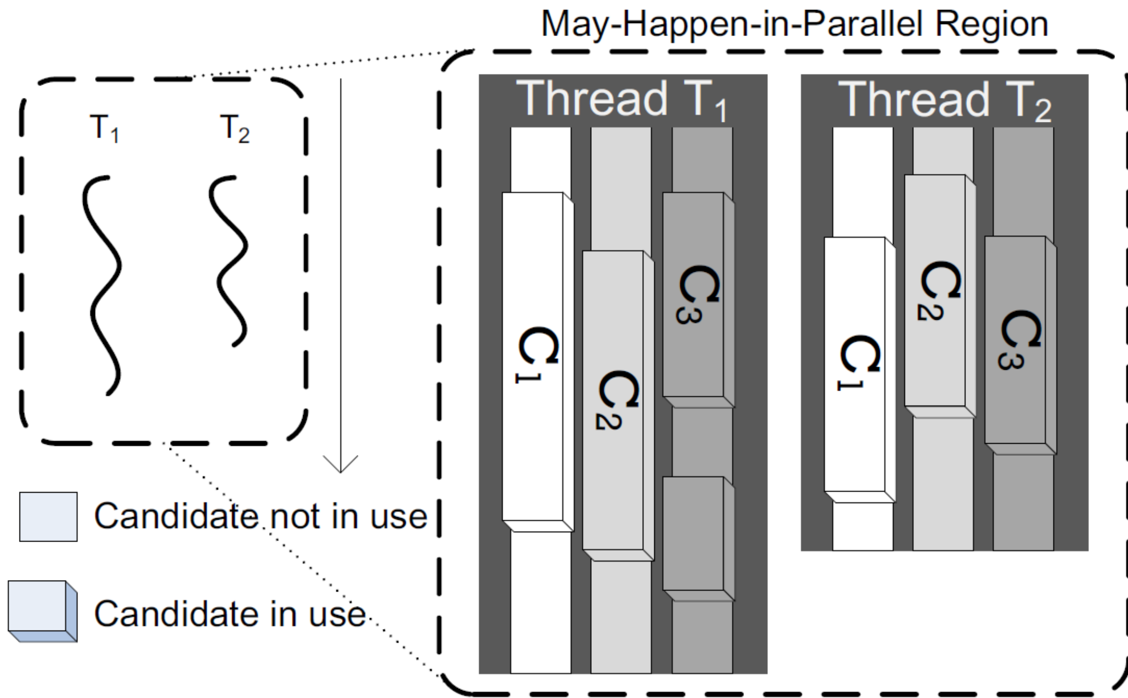


Figure 15. Two threads in a may-happen-in-parallel region and their utilization statuses of the power-gating candidates

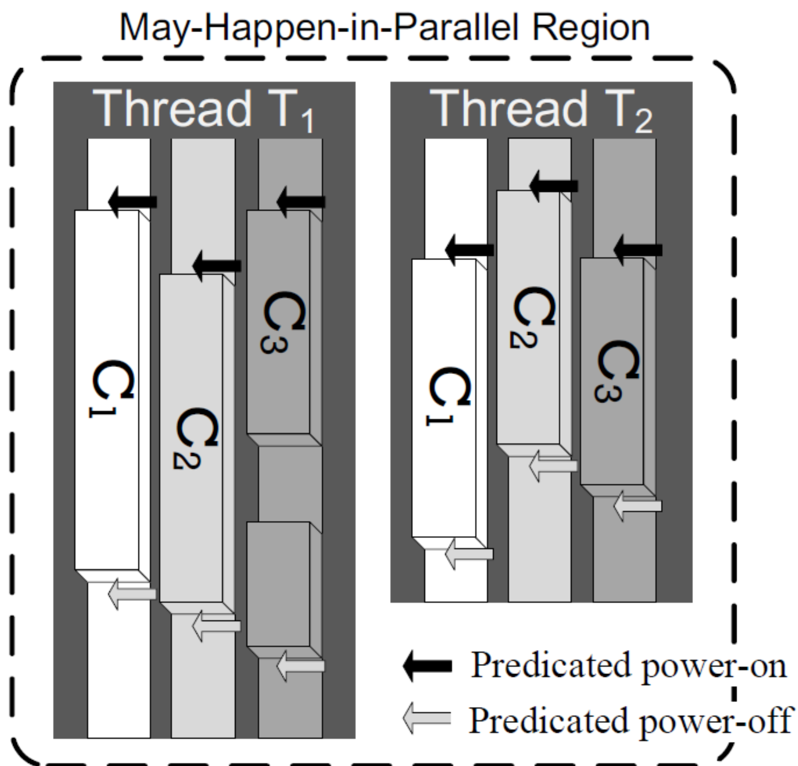


Figure 16. A probable placement of predicated-power-gating instructions upon the MTPGA results for Figure 15

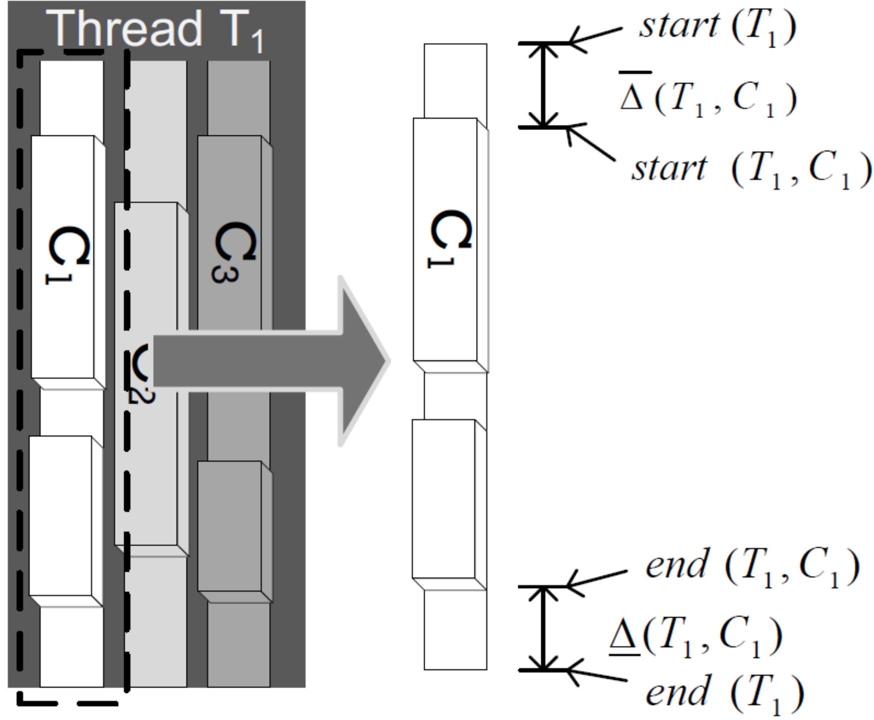


Figure 17. A illustration of the cost model for MTPGA

的電源閘控指令都將不會運作。

假設目前有N個可做電源閘控的元件：$C_1$、$C_2 \cdots C_N$ 以及 K 個可能同時並行執行的執行緒：$T_1$、$T_2 \cdots T_K$。我們必須先定義兩個函式：

(1) $\overline{\triangle}(T_i , C_j) = start(T_i , C_j) - start(T_i)$：

此函式是計算從 $T_i$ 開始執行到第一次 $C_j$ 元件被使用的時間長度。

(2) $\underline{\triangle}(T_i , C_j) = end(T_i) - end(T_i , C_j)$：

此函式是計算最後一次 $C_j$ 元件被使用到 $T_i$ 執行結束的時間長度。

Figure 17 便是顯示以 Thread $T_1$ 中的 Component $C_1$ 為例來說明上述兩個函式在時間軸上的圖示表示。

接著，我們再定義兩個函式：

(1) $\overline{M}(Cj) = MIN_{\forall i}\ \overline{\triangle}(T_i , C_j)$：

在各個執行緒當中，皆有自身的 $C_j$ 元件使用的資訊。此函式便是希望從中找出 $C_j$ 元件是在何時最先被開啟。

(2) $\underline{M}(Cj) = MIN_{\forall i}\ \underline{\triangle}(T_i , C_j)$：

在各個執行緒當中，皆有自身的 $C_j$ 元件使用的資訊。此函式便是希望從中找出 $C_j$ 元件是在何時被最後關閉。

定義上述四個函式後，我們可以推導出條件式電源閘控機制的總耗能為 Equation 1.

$$\mathbb{E}_{on}(C_j) + \mathbb{E}_{off}(C_j) + K_j \times (\mathbb{E}_{pred-on} + \mathbb{E}_{pred-off}) \\ + (\overline{\mathbb{M}}(C_j) + \underline{\mathbb{M}}(C_j)) \times \mathbb{P}_{rleak}(C_j),$$

Equation 1. The total energy consumption of the predicated power-gating mechanism

$$\mathbb{E}_{on}(C_j) + \mathbb{E}_{off}(C_j) + (\overline{\mathbb{M}}(C_j) + \underline{\mathbb{M}}(C_j)) \times \mathbb{P}_{leak}(C_j)$$

Equation 2. The total energy consumption of the normal power-gating mechanism

$$\begin{aligned}\mathbb{E}_{on}(C_j) + \mathbb{E}_{off}(C_j) + K_j \times (\mathbb{E}_{pred-on} + \mathbb{E}_{pred-off}) \\ + (\overline{\mathbb{M}}(C_j) + \underline{\mathbb{M}}(C_j)) \times \mathbb{P}_{rleak}(C_j) < \\ \mathbb{E}_{on}(C_j) + \mathbb{E}_{off}(C_j) + (\overline{\mathbb{M}}(C_j) + \underline{\mathbb{M}}(C_j)) \times \mathbb{P}_{leak}(C_j)\end{aligned}$$

Equation 3. The inequality of the energy consumption

$$\overline{\mathbb{M}}(C_j) + \underline{\mathbb{M}}(C_j) > \frac{K_j \times (\mathbb{E}_{pred-on} + \mathbb{E}_{pred-off})}{\mathbb{P}_{leak}(C_j) - \mathbb{P}_{rleak}(C_j)}$$

Equation 4. The cost model of the MTPGA

**Algorithm 3:** Algorithm of the multithreaded-power-gating analysis.

**Input** : A multithreaded program and its MHP and CADFA with Sink-N-Hoist information.
**Output**: The program with power-gating controls.

```
1  foreach MHP region do
2      foreach power-gating candidate C do
3          if M̄(C) + M̲(C) ≤ THRESHOLD† then
4              Place a power-on and a power-off instruction for
               C at the beginning and the end of the MHP
               region, respectively.
5          else
6              foreach thread do
7                  Place a predicated-power-on and a
                   predicated-power-off operation before/after
                   the candidate operates for the first/last time
                   within the thread.
8              end
9          end
10     end
11 end
```

$$^{\dagger}THRESHOLD = \frac{K_j \times (\mathbb{E}_{pred-on} + \mathbb{E}_{pred-off})}{\mathbb{P}_{leak}(C) - \mathbb{P}_{rleak}(C)}$$

Figure 18. The algorithm of Multithreaded Power-Gating Analysis

Equation 1 中 $E_{on}$ 及 $E_{off}$ 為真正有做條件式電源閘控開關指令所消耗的電能，$K_j$ 為可能同時執行區間中需要用到 $C_j$ 元件的執行緒數目，$E_{pred-on}$ 及 $E_{pred-off}$ 為沒有真正做條件式電源閘控開關指令所消耗的電能(雖然它們並未真正對元件做開關，但在指令的 Fetch 及 decode 等等方面還是需要耗電能)，$P_{rleak}(C_j)$代表的是在元件 $C_j$ 第一次使用開啟之前以及最後一次使用關閉之後的那些時間的功率消耗(此一功率消耗是有被限制的，亦即與一般正常狀態下的功率消耗非相同)。

標準的電源閘控機制的總耗能為 Equation 2，其中 $P_{leak}(C_j)$為一般正常狀態下的每個 cycle 的功率消耗。

明顯地，若是標準的電源閘控機制的總耗能比條件式電源閘控機制還要來的省電，也就是說反而使用條件式電源閘控機制會導致更多的耗電的話，那我們便不使用條件式電源閘控機制。Equation 3 便是在說明此一概念。

而在整理式子之後，Equation 4 便是我們判定是否要做條件式電源閘控機制的 cost model 的結果。

而 Figure 18 所顯示的是 MTPGA 的演算法。

## 4.6 並行編程模型 (Concurrent Programing Models)：

我們針對常用的並行編程模型(concurrent programming model)：OpenMP、CUDA、OpenCL，進行深入探討：

(1) OpenMP：

OpenMP 是以共享記憶體(Shared Memory)的方式去分享所要執行的任務和資料，以主執行緒(Master Thread)為任務主軸並搭配應用分岔-聯結模式(Fork-Join Model)來產生與執行可平行區塊(parallel region)任務所需的執行緒列(threads)。當程式撰寫者撰寫 OpenMP 時，須注意核心要素(core elements)的架構，其為 OpenMP 程式碼中加入任務與資料分享資訊或創造執行緒所要給予編譯器知道的說明。

- 共享記憶體 (Shared Memory)，執行緒形式的平行方式 (Thread Based Parallelism)。OpenMP 是基於共享記憶體內存在多道執行緒的編程規範，共享記憶體程序可組合多道的執行緒。

- 分岔-聯結模式 (Fork-Join Model)：
  OpenMP 是以一個主執行緒 (Master thread) 做為主要的執行路徑，然後循序地執行所要執行的敘述 (Statement)直到遇到可平行的區域 (parallel region)架構。而後於主執行緒中分岔 (FORK)出一組可平行的分支執行緒列，去平行的處理所要處理的敘述。當處理完後，執行聯結 (JOIN) 動作，指的是當一組分支出的執行緒列處理完成要處理的敘述後，便將分支的執行緒列同時的終止與離開只留下主執行緒繼續的循序地執行下去。如圖 Figure 19 所示

- 核心要素(core elements)：
  OpenMP 的核心要素 (core elements)是架構出當在程式撰寫者在撰寫 OpenMP 時所要考量的不同狀況與程式所需加注的地方，如遇到可平行區塊時的執行緒的創造 (thread creation)，工作分享

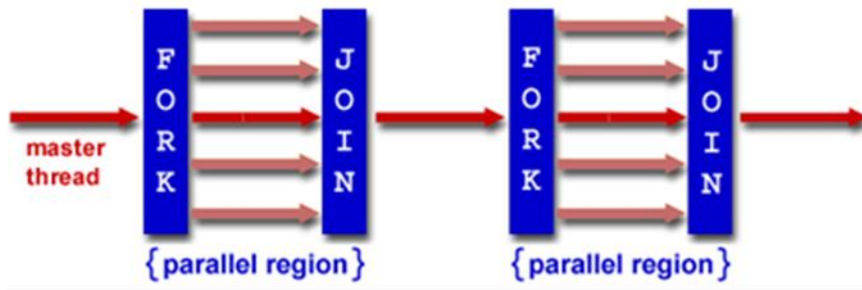

Figure 19. Fork-Join-Model

(work sharing)，資料環境管理(Data-environment management)，執行緒同步(thread synchronization)，使用者位階的執行時的例行程序(user-level runtime routines)與環境變數(environment variables)等情況與類別。

(2) CUDA：

核心函式(kernel function)為 CUDA 呼叫 GPU 執行數值運算的地方，並以執行緒為基本運算單元且將其以若干數量組織不同階層方便管理與應用，每一執行緒對 GPU 的記憶體也有不同階層的存取速度與權限，程式撰寫者在撰寫時也須對其使用謹慎量。

- 核心函式(kernel function)：CUDA C 是衍生的 C 語言，允許撰寫程式者去定義 C 語言形式的函式,去呼叫核心,被呼叫的核心以 N 個不同的 CUDA 執行緒列平行的被執行 N 次。
- 執行緒階層(Thread Hierarchy)：在 CUDA 中，執行緒(thread)為核心中最小的運算單元。且 CUDA 中有多個執行緒階層: 若干個執行緒列 (threads)構成一個執行緒區塊(thread block)，若干個執行區塊構成一個格子(grid)。執行緒列有對應的執行緒 ID 包含於區塊內。核心程式使用執行緒 ID 去選擇工作與定址共享資料。
- 記憶體階層(Memory Hierarchy)：

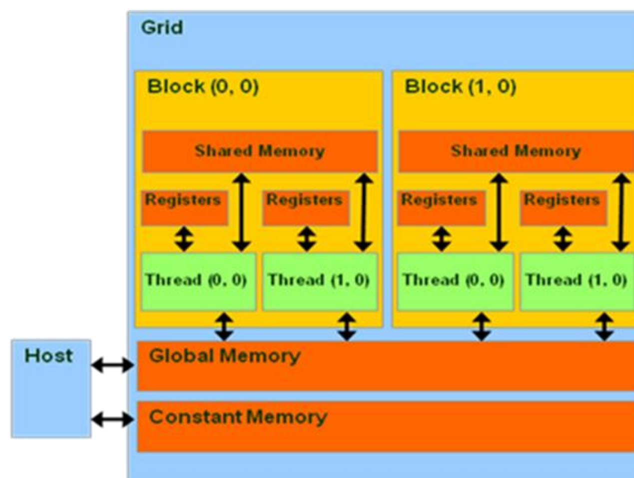

Figure 20. CUDA Memory Hierarchy

| 記憶體 | 說明 | 速度 | 使用範圍 |
|---|---|---|---|
| **暫存器** | 執行緒使用到的一般變數 | 快速 | 執行緒內部 |
| **共享記憶體** | 同一區塊內的執行緒共用 | 快速(幾乎與暫存器速度相同) | 區域內交換資料用 |
| **常數記憶體** | 存放整個程式共用的常數 | 快速 | 全域使用 |
| **全域記憶體** | 顯示卡中的DRAM | 很慢 | 全域使用 |

如圖 Figure 20 所示，CUDA 執行緒於其執行期間可從多層記憶體空間存取資料。每個執行緒擁有各自的暫存器(register)，每個執行區塊 (thread block)，擁有共享記憶體可讓所有的區塊執行緒列看見與同區塊有相同的生命週期。所有的執行緒列可以存取相同的全域記憶體(global memory)。常數記憶體(Constant memory)則是存放整各程式共用的常數。對於 CUDA 的記體體階層，我們以上面的表格比較了他們的差異性。

(3) OpenCL：

OpenCL 為開放式的異質多核心程式語言，其核心為其主要運算執行的地方，並將其平行方式分為資料平行(data-parallel)與任務平行(task-parallel)，前者相似為 CUDA 後者則相似為 OpenMP，其核心的執行順序則以程序設計 (program) 作為依據可分為依序(in-order)或亂序(out-of-order)執行核心。OpenCL 的記憶體階層大致架構與 CUDA 類似。

- 執行模式(execution model)：
  核心 (kernel)，基本的可執行單位程式碼相似於 C 的函式型式，可做資料平行 (Data-parallel)或任務平行 (task-parallel)的平行方式執行；資料平行 (Data-parallel)，在資料平行的模型中，定義了 N 維度的計算域 (N-Dimensional computation domain)，其中也定義了每個獨立的工作項目 (work-item)，相當於 CUDA 中定義的執行緒，若干的工作項目數量則被定義為工作團體(work-group) (相當 CUDA 的 thread block);而在工作團體中的工作項目可以互相溝通，也可以進行同步(synchronize)，而在 OpenCL 中，也可以多個工作團體同時被執行；任務平行 (task-parallel)，在任務平行的模型中，類似於將程序產生大量的多個執行緒列，以此執行不同的任務。在 OpenCL 中，任務平行是專注於排序多個核心使得 OpenCL 執行它們時，可以盡量的使用能夠運用的處理器。
- 程序設計(program)：
  蒐集核心與其他函式，相似於一個

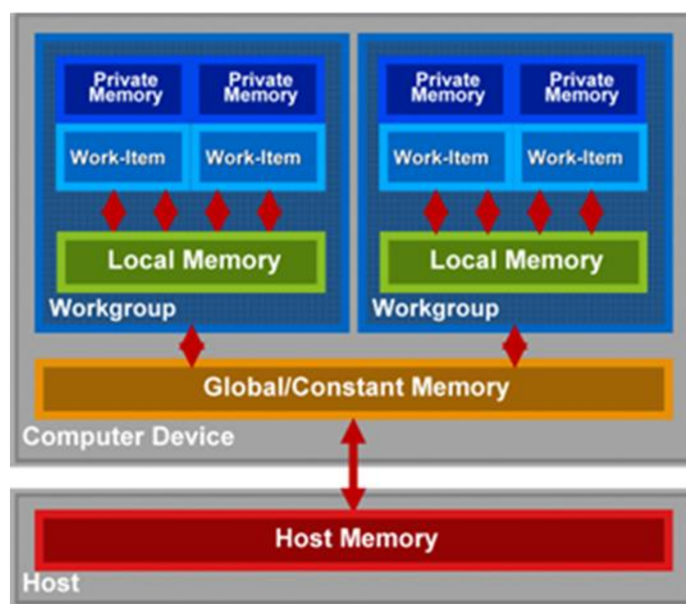


Figure 21. OpneCL Memory Hierarchy

動態函式庫。應用排序核心執行範例 (applications queue kernel execution instances)，對核心依序排序，當執行時的順序可以依序(in-order)或亂序 (out-of-order)來執行。

- 記憶體模式(memory model)：
如 Figure 21 所示，私有記憶體(private memory)，為每個工作項目有各自的私有記憶體，類似 CUDA 的暫存器。區域記憶體 (Local memory)，共享於一個工作團 (約為 16Kb)中 ，類似於 CUDA 的共享記憶體。全域/常數記憶體(global/constant memory)，可共享於每個工作團中，類似 CUDA 的全域記憶體。主記憶體 (host memory)，於 CPU 中的記憶體。資料搬移方式，如將資料從主記憶體搬至裝置(device)中的區域記憶體，其路徑為 (host)->全域 (global)->區域 (local)。

## 4.7 編譯器與作業系統協同合作電源管理 (Collaborative Operating System and Compiler Power Management)：

我們是以即時應用程式(real-time application)作為主要對象，並且利用 dynamic voltage scaling(DVS)技術來進行探討：

(1) Application Model：

一個即時應用程式(real-time application)有著執行時間上的規定期限，以標記 d 來表示此一期限，每一個應用程式也都會有自身所謂的 worst-case execution time(WCET)。假若 d－WCET ＞0，亦即一個應用程式最壞情形的執行時間小於規定期限，表示可以使用 DVS 來降低電壓以改變此一應用程式的執行時間，讓應用程式盡量地在剛好達到 d 的期限內完成工作即可。如此一來，便可在要求的時間內完成工作而又能夠達到省電的效果。而正常地來說，應用程

式的真實執行時間一定會比 WCET 來的短，這就表示在不是最壞情況下我們能夠節省更多的電能。

(2) Energy Model：

在 CMOS 電路中，電源消耗主要由靜態功率消耗和動態功率消耗組成。靜態功率消耗主要是因為漏電流（leakage current）損失導致的。動態功率消耗的公式正比於 $V_{dd}$ 的平方，亦即當調降電壓對於降低整體電源消耗是有指數次方的效果，而非單單是線性的效果。但是調降電壓也會造成處理器的處理速度下降而導致整體效能也跟著下降，所以我們得在效能以及省電之間取得一個平衡。

(3) Overhead Model：

使用 DVS 在效能上會有 overhead，而 overhead 的來源為：

<1> 計算處理器新的頻率所花費的時間

<2> 透過電壓的調整來設定處理器在(1)中所算出的新的頻率

所以當使用 DVS 技術想達到節省電源的同時，也得謹慎考量這兩個原因所產生的 overhead 是否會對於整體的效能造成太大的影響，若是會對效能造成大的影響，可能做了 DVS 反而會比不做 DVS 還要來的耗電，那麼寧可不做 DVS。

在以上三點為基礎考量之下，若要妥善利用 DVS 來達到省電效果，會需要同時利用到編譯器及作業系統。首先，利用編譯器收集程式的行為資訊並在程式碼中加入電源管理提示 (power-management hints)，這些提示主要是紀錄了程式的動態行為，包含了執行路徑以及執行時間的變動等等資訊。而在程式執行時，作業系統便會週期性的去呼叫中斷服務常式(interrupt service routine)並依據在程式碼中加入的 hints 的資訊來適當地改變處理器的頻率以及電壓，這些改變處理器的頻率及電壓的時機點被稱為 power-management points (PMPs)。若是單獨使用 OS，OS 無法察覺到程式執行的行為以及剩餘的時間；而若單獨使用 compiler，compiler 則是無法去控制 PMPs 被呼叫的週期。

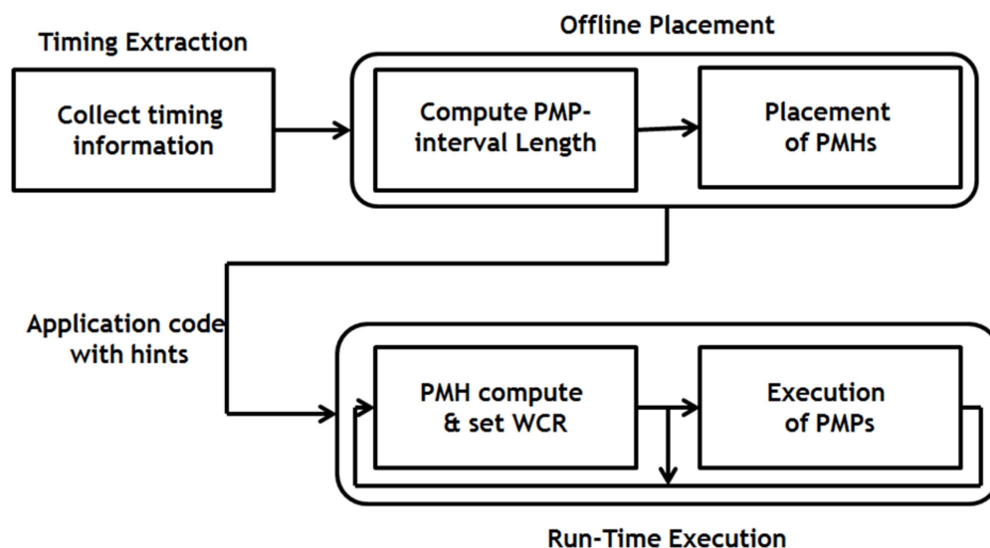


Figure 22. The phases of the collaborative between compiler and operating system

所以作業系統要多久去呼叫一次中斷服務常式來改變處理器的頻率及電壓呢？理想化的情況當然是越頻繁地去呼叫中斷服務常式越能夠依據程式行為的需求來調整處理器的頻率及電壓，以達到既能在時間期限內完成工作又能夠節省耗電能。但是實際狀況下，我們得要考慮上述的 overhead model 中提到的兩種 overhead 的來源可能會對電源節省的部分造成影響，故以多長的周期去呼叫一次中斷服務常式是一個重要的議題。Figure 22 顯示了在即時應用程式及 DVS 的基礎下，編譯器與作業系統合作的整體流程圖。

## 4.8 多核心架構電源管理 (Power Management in Multi-core Architecture)：

由於近年來多核心處理器(multi-core processor)架構在市面上已成趨勢，故我們也將注意力轉移至在如此的架構下，要如何地讓多核心系統能夠省電。延續上述 4.5 的討論，我們將焦點放在以 OpenMP 為主要模型來進行探討。

在多核心架構下，循序的程式區段是以一個核心作為主執行緒來執行程式；在遇到可以平行執行的程式區塊時，便會利用除了作為主執行緒核心外的其他核心來幫忙執行程式的平行區塊。當處理完平行區塊後，又會再度回到只剩下主執行緒的核心繼續執行循序的程式區段，如 Figure 23 所示(圖中以四個核心為例)。

由 Figure 23 中可以看到紅色線條便是作為主執行緒的核心的執行過程，其餘白色線條便是只有當程式遇到可以平行執行區塊才會幫忙執行的其餘核心，而這也符合了 4.6 中提到的分岔-聯結模式(Fork-Join Model)。而 OpenMP 在執行平行區塊中的程式碼時，在平行區塊的尾端會有一個 implicit barrier，如 Figure 24 所示。Implicit barrier 會將已經執行完平行運算工作的 core(s)阻擋住，不讓它(們)繼續執行循序部分的程式碼，直到所有的 core 皆已完成平行運算的工作後，才會繼續往下執行循序部分的程式碼。而 OpenMP 主要是針對 for loop 做平行化運算，我們依然舉一個例子來說明 OpenMP 對於 for loop 是如何地去做平行運算。Figure 25 說明了一個會執行 16 次的 for loop 在四個核心的系統上使用 OpenMP 中的 static scheduling 的方式去做平行運算的示意圖。

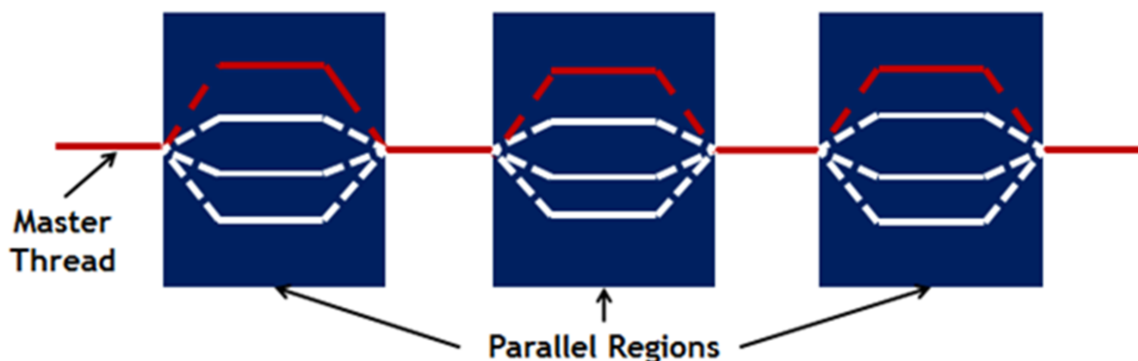


Figure 23. OpenMP programming model in multicore system

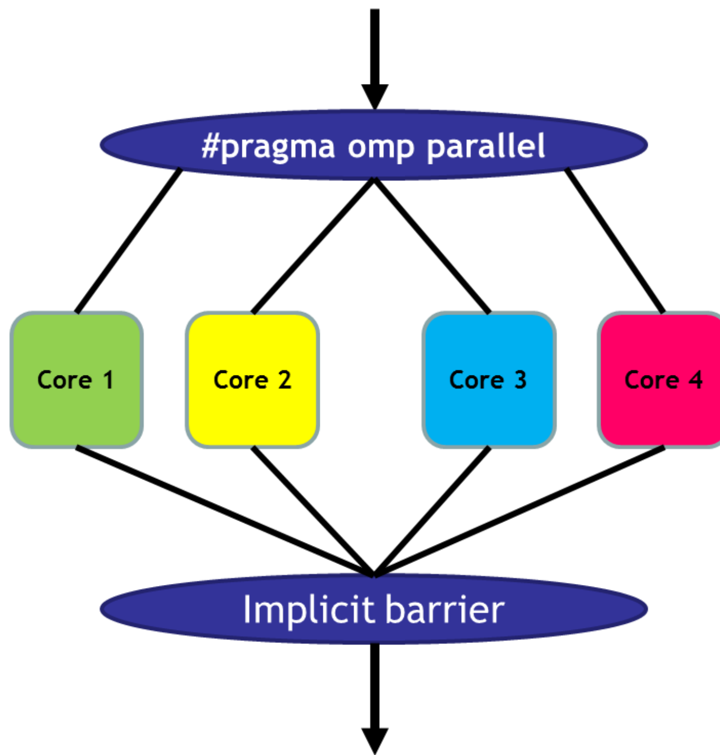


Figure 24. Parallel region

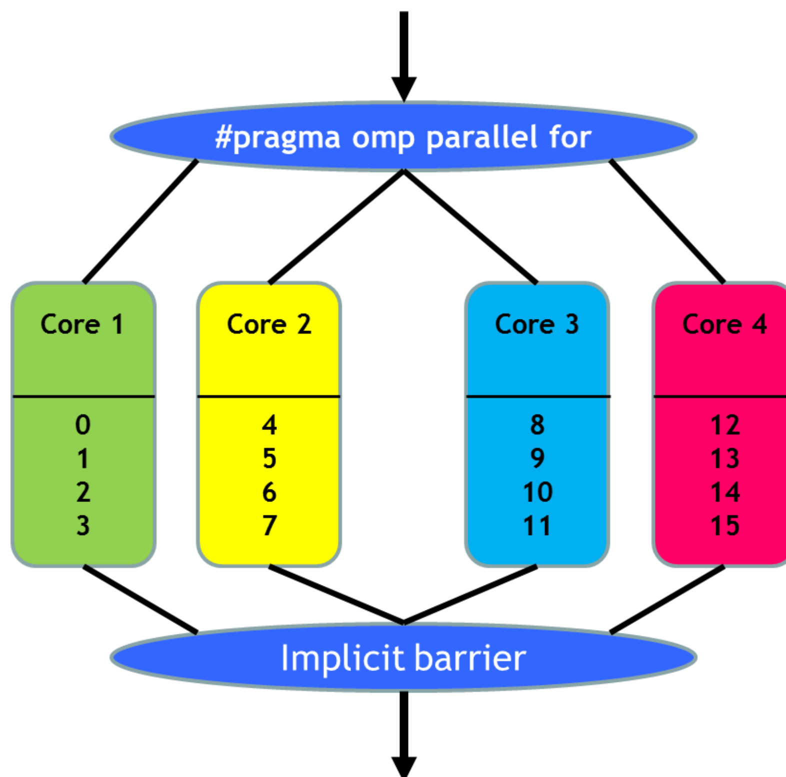


Figure 25. A example of the distribution of the for loop's iterations

OpenMP 將 for loop 中的 iterations 分配到各個核心有以下三種方式：

(1) Static schedule：

將所有的 iterations 按照 core 的數目去平均分配。以 Figure 25 為例，16 個 iterations 除以 4 平均分配 4 個 iterations 給每一個 core。

(2) Dynamic schedule：

在動態排程裡，OpenMP 會以一個參數 chunk 為基數來分配 chunk 數字 iterations 給每一個 core，而當某個 core 完成它所被分配的工作以後，系統便會馬上再分配給它 chunk 數字的 iterations 給它，直到平行區塊的工作被分配完畢。

(3) Guided schedule：

類似動態排程，一樣以 chunk 數為基準，整體一次性地分配小於 chunk 數目的 iterations 給每一個 core，但是在第二次的整體分配時，所分配給每個 core 的 iterations 數不會大於上一次各個 core 所拿到的 iterations 數。

而在平行區塊中，若是每個 core 所被分配到的工作量都相差不多時，會發現其實幾乎沒有什麼機會可以實做 power-gating 或者是 clock-gating 的機制來省電。換句話說，每個 core 所被分配的工作量不是那麼平均的話，我們便可從中找到機會來達到省電。再進一步地想，是什麼原因會造成每個 core 所分配到的 iterations 會有執行時間上的差異呢？我們的想法是：因為在 for loop 中會有判斷式的結構(亦即像是 if、if then else 或是 while 甚至是內層也在有 for loop 也算在內)。這些判斷式結構不會每個 iteration 皆會進入，導致有進入和沒有進入判斷式結構的 iteration 會有時間差，這就使得我們有機會以這樣的想法為基礎來實作 power-gating 或者是 clock-gating 達到省電的目的。Figure 26 顯示了當每個 core 所被分配到的工作量不同，所導致在平行區塊中會有 core 在完成安排的工作以後會有著空閒的時間。

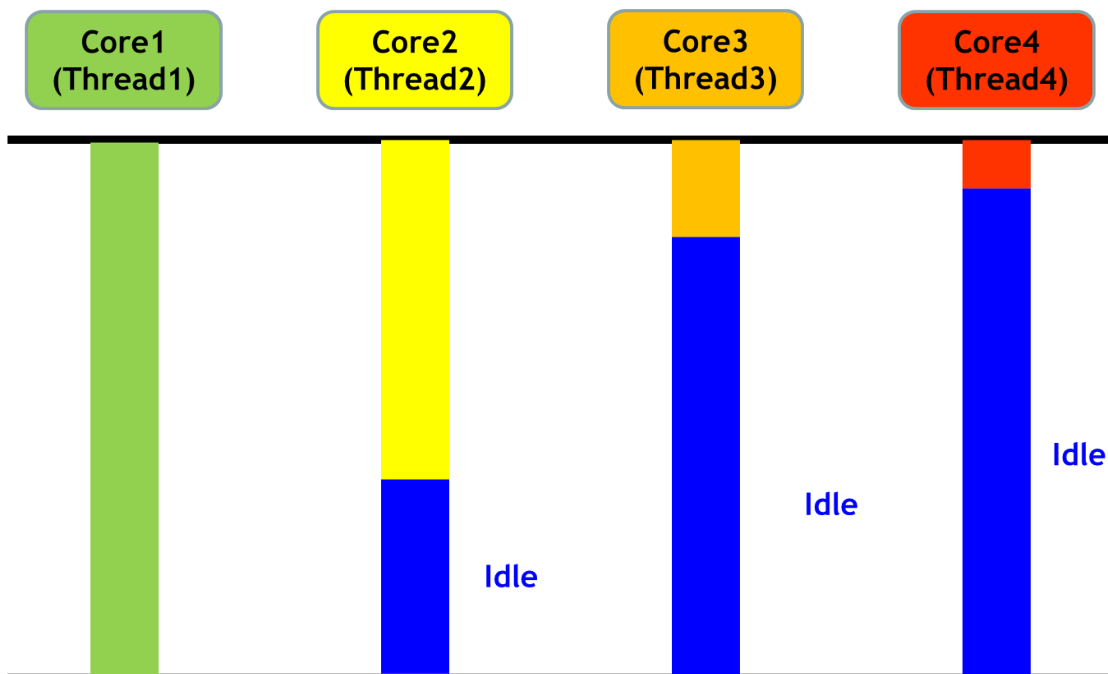


Figure 26. The idea of power saving in parallel for loop

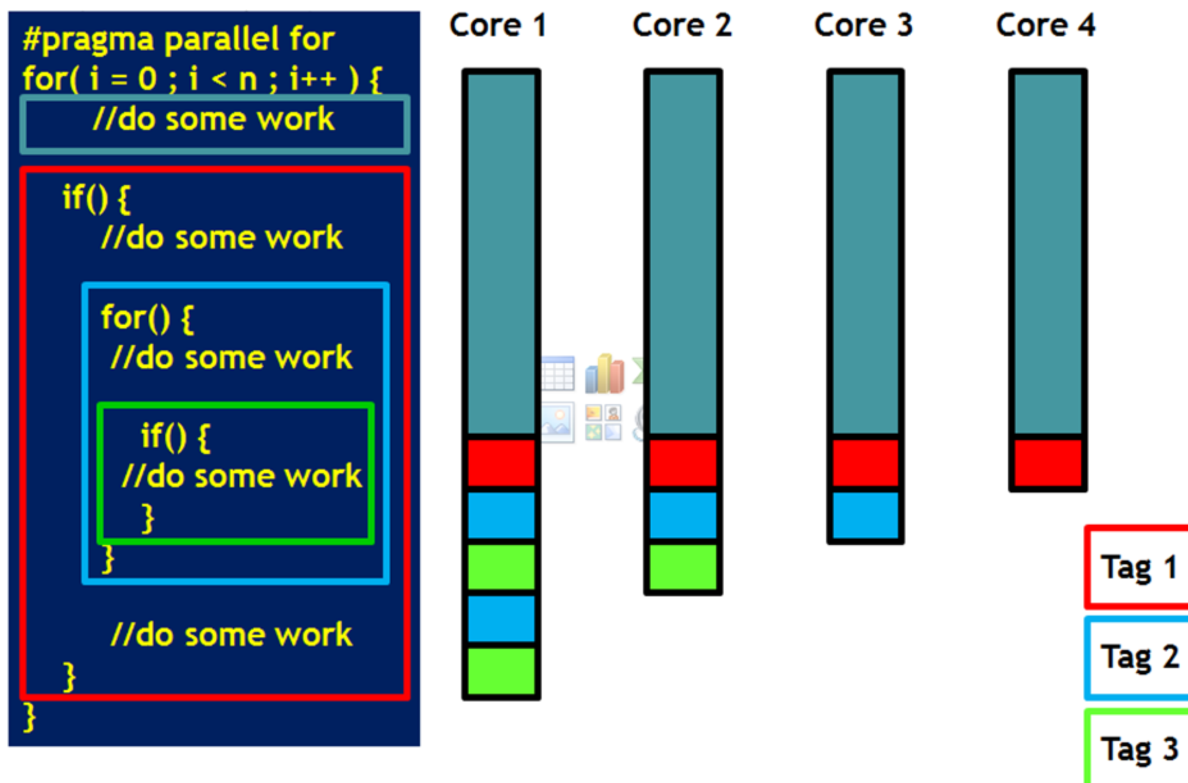


Figure 27. The example for the idea’s problem

Figure 26 中，上面的黑線代表著 parallel for 的開始，下面的黑線代表 parallel for 的結束，亦即是 implicit barrier 的所在。可以明顯的看出，Core 2、Core 3 及 Core 4 皆在尚未到達 implicit barrier 時，便已完成了他們所被分配的工作，所以會導致他們在剩餘的時間內其實沒有在做任何的工作，但是它們依舊是處於開啟的狀態，我們的想法便是要在這段空閒的時間做 power-gating 或者是 clock-gating，也就是將執行時間最長的 core 所花的時間去扣掉其餘提早完成工作的 core 所花的時間，來觀察是否值得做 power-gating 或者是 clock-gating(有考慮到 overhead)。但是這個想法會遇到一個潛在的問題，我們以 Figure 27 來做介紹和說明。

首先是 Figure 27 的左半邊的程式碼部分，分為四個部分：

(1) 藍綠色框框內的程式碼是每個 iteration 皆會執行到的部分。

(2) 紅色框框內且在藍色框框外的程式碼是進入 if 判斷式結構一定會執行到的部分，我們將執行這些程式部份的時間估算以後記錄在 Tag 1 中。

(3) 藍色框框內且在綠色框框外的程式碼是進入此一 for 判斷式結構一定會執行到的部分，我們將執行這些程式部份的時間估算以後記錄在 Tag 2 中。

(4) 綠色框框內的程式碼是進入此一 if 判斷式結構一定會執行到的部分，我們將執行這些程式部份的時間估算以後記錄在 Tag 3 中。

而 Figure 31 的右半邊的圖則是 4 個不同的 iterations 在遇到判斷式結構進入與否所造成的差異現象，我們以 Core 2 為例來做說明。Core 2 在執行時間上有著

一個藍綠色的長條、一個紅色方塊、一個藍色方塊以及一個綠色方塊，這代表著除了每個 iteration 皆一定會執行到的藍綠色框框所花費的時間以外，他還必須加上進入紅色框框、藍色框框以及綠色框框所會花費的時間，而這些時間分別就以紅色方塊、藍色方塊以及綠色方塊來表示。

為了說明方便我們將整個執行流程縮小來看，假設 4 個核心皆只分別做這 4 個 iteration 的話，因為程式只有在 run-time 時，才能確定 iteration 是否有進入判斷式結構，所以當在某個 core 上的 iteration 有進入某個判斷式結構，就會增加相對應時間到該 core。那麼就有可能會產生一個問題是：當 Core 4 結束工作以後，它會以自己結束工作之時的另外三個 cores 中的哪一個當前的時間最長去計算是否可以使用 power-gating 或 clock-gating。但其他三個 cores 皆尚未結束，如此會造成誤判的情形導致無法使用 power-gating 或 clock-gating。

若是將模型放大到正常的情形來觀察，或許上述的問題出現的情況可能就會減少，因為當每個 core 執行 iteration 的數量都很多時，相對地時間差上正常地來說就會有較大的差異，但依舊無法完全排除問題出現的可能性。我們目前針對這個問題的想法是利用 Mohammad Ali Ghodrat [32]中 value range 的概念將判斷式分為三種情況：(1)一定會進入、(2)一定不會進入及(3)未知會不會進入。針對(1)和(2)，我們能夠清楚的知道是否需要增加時間，但是在(3)中我們無法確定是否要增加時間，所以在未知區域我們想到的可能的解決方法有二：

(1) 皆猜會進入，但這樣有可能會造成誤判的情形

(2) 賦予每個判斷式一個機率，而在 run-time 時，根據實際程式執行的過程去動態地調整進入各個判斷式的機率

以上所做的探討是我們在多核心架構下思考的研究想法。

# 五、 結果與討論

(1) Multithreaded Power-Gating Analysis (MTPGA)：

綜合 4.2 至 4.5 所發展的研究方法，我們以 SUIF2、Control Flow Graph(CFG)及 Machine-SUIF 的 machine library 為基礎加以實作。Figure 28 所顯示的便是編譯流程 (compilation flow)。

而為了驗證我們所提出的 Multithreaded Power-Gating Analysis(MTPGA)方法，我們將重心放在研究可能同時執行的區塊 (May-Happen-in-Parallel region)。首先我們以兩個 DSPstone 浮點運算程式當作輸入 (input)來預估算數運算單元(ALU，包含 4 個整數運算單元、4 個浮點數運算單元)及乘法器(multiplier，包含 1 個整數乘法器、1 個浮點數乘法器)的電源消耗情形。實驗的數據主要是以每一支 DSPstone 程式為主，讓這支程式和其餘除了自己以外的 DSPstone 程式做為一個組合去讓兩個硬體執行緒去執行，進而在從中去挑出最好、最差以及平均的情形做成圖表，Figure 29 便是實驗結果的數據圖。由 Figure 29 中可以看出：

<1> 就算是最差的情形，各種組合的平均耗電量也是原本沒有使用 MTPGA 方法的 89.75%。

<2> 所有最好的情形的平均則是原本沒有使用 MTPGA 方法的 53.27%。

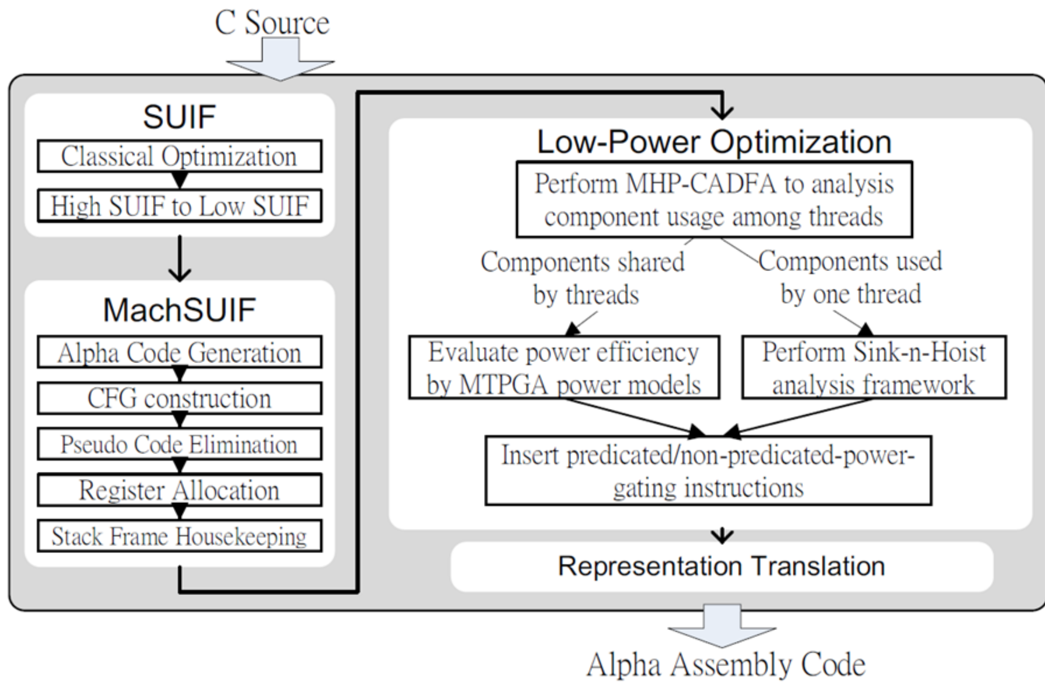


Figure 28. A compilation flow of power management for multithreaded programs

<3> 各種組合平均情形的再平均則是原本沒有使用 MTPGA 方法的 62.75%。

接著我們再做了四個硬體執行緒的情形，我們從各種組合中隨機挑出了7種組合作為數據圖表，Figure 30 便是這7組的各項元件、有無使用 MTPGA 的總耗電量以及改善耗電的數據圖。

在 Figure 30 中，第1行所顯示的是被挑選出的程式，從第2行至第4行是顯示浮點數運算單元(FP ALU)、浮點數乘法器(FP MUL)及乘法器(MUL)針對整個乘法執行它們處於關閉狀態的比例，第5行及第6行則是顯示所有運算單元在有無 MTPGA 方法支援的總體耗電量，而最後一行則是顯示有使用 MTPGA 的總耗電量占了沒有使用 MTPGA 的總耗電量的比例。

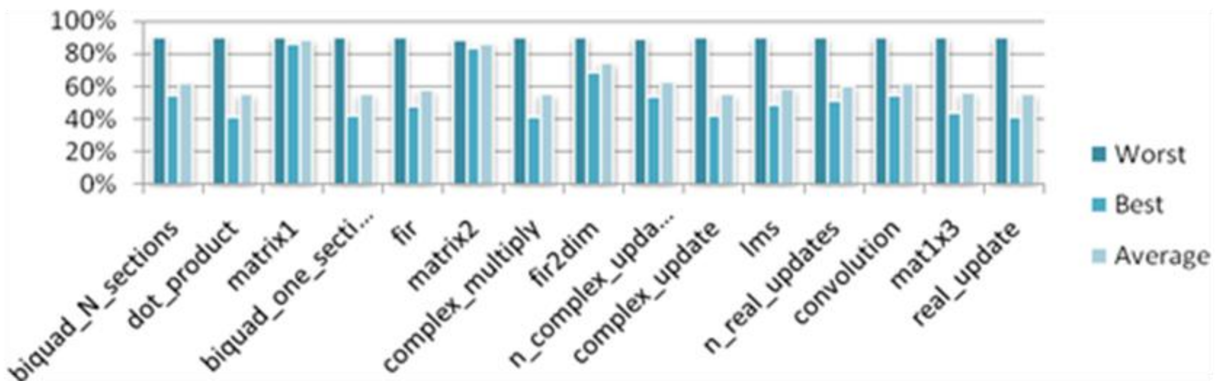


Figure 29. The worst, best and average normalized power consumption of thread combinations for DSPstone programs

| Concurrent Threads | Turn-Off Rate | | | Energy | | |
|---|---|---|---|---|---|---|
| | FP ALU | FP MUL | MUL | w/ PPG | w/o PPG | improvement |
| matrix1,matrix2,fir2dim,biquad_N_sections | 22.17% | 21.55% | 14.71% | 35617.22 | 39152.17 | 90.97% |
| convolution, n_complex_updates, lms, n_real_updates | 68.48% | 80.48% | 59.17% | 7159.10 | 10892.27 | 65.73% |
| fir, complex_multiply, biquad_one_section, mat1x3 | 84.65% | 88.86% | 86.95% | 5078.80 | 8858.30 | 57.33% |
| dot_product, complex_update, real_update, matrix1 | 21.87% | 20.67% | 19.69% | 34807.91 | 38584.68 | 90.21% |
| matrix2, fir2dim, biquad_N_sections, convolution | 22.65% | 30.47% | 15.93% | 31989.53 | 35796.24 | 89.37% |
| n_complex_updates, lms, n_real_updates, fir | 68.63% | 80.95% | 59.11% | 7580.15 | 11320.04 | 66.96% |
| complex_multiply, biquad_one_section, mat1x3, dot_product | 93.59% | 94.31% | 96.39% | 3859.34 | 7497.21 | 51.48% |

Figure 30. Component turn-off rate and power consumption of four concurrent executed threads

(2) 並行編程模型（Concurrent Programming Models）：

針對並行編程模型，我們則將現行常用三種並行編程模型(OpenMP、CUDA、OpenCL)進行深入研究，並了解各模型優、缺點及其特性。下面的表格則是我們深入探討的結果統整。

(3) 編譯器與作業系統協同合作電源管理探討：

我們針對即時應用程式(real-time application)並且利用 dynamic voltage scaling(DVS)技術作為探討的基礎架構，對於 application model、energy model 及 overhead model 進行探討與分析。而我們也簡述了編譯器與作業系

| 語言 | OpenMP | CUDA | OpenCL |
|---|---|---|---|
| 主要支援語言 | C, C++, Fortran | C, C++ | C |
| 適用硬體 | CPU | GPU | CPU, GPU, 其他 |
| 記憶體階層 | 共享記憶體 | 較複雜的記憶體階層 | 較複雜的記憶體階層 |
| 編譯環境 | 完整 | 完整 | 不成熟 |
| 支援版本 | 3.1 | 4.0 | 1.1 |
| 支援的平行方式 | 任務平行 | 執行緒平行 | 任務平行或執行緒平行 |
| 執行迴圈(Loop)與判斷式(if statement)能力 | 優 | 較差 | 依硬體情況<br>CPU 上：優<br>GPU 上：較差 |
| 程式碼公開程度 | 公開 | 半公開(公開使用 Open64 的部分) | 公開 |

統合作的流程，最後則是比較了利用編譯器與作業系統合作和單獨使用作業系統或單獨使用編譯器的差別。

(4) 多核心電源管理探討：

我們針對在多核心架構並且以 OpenMP 處理平行處理作為探討的基礎下，對於 OpenMP 在平行 for loop 部份實做省電機制的時機以及可能會遇到的問題進行探討與分析。

本計畫成果已發表至兩個會議論文中，並已投稿至期刊審查中，另外計畫部分成果也已獲得美國專利：

- **Yi-Ping You**, Jenq Kuen Lee, Kuo-Yu Chuang, and Tsung Hsien Wu, ``Multi-thread power-gating control design," US Patent, No. 7,904,736, Issued Date: 2011/03/08.
- **Yi-Ping You**, Shen-Hong Wang, and I-Ting Lin, ``Energy-aware Code Motion for GPU Shader Processors,'' in *Proceedings of the 17th Workshop on Compiler Techniques for High-Performance and Embedded Computing (CTHPC'11)*, Taichung, Taiwan, June 2-3, 2011.
- **Yi-Ping You** and Yu-Shiuan Tsai, ``Compiler-Assisted Resource Management for CUDA Programs,'' accepted, *the 16th Workshop on Compilers for Parallel Computing (CPC'12)*, Padova, Italy, Janary 11-13, 2012.
- **Yi-Ping You**, and Shen-Hong Wang, ``Energy-aware Code Motion for GPU Shader Processors,'' submitted to *ACM Transactions on Embedded Computing Systems (TECS)*.
- **Yi-Ping You** and Yu-Shiuan Tsai, ``Compiler-Assisted Resource Management for CUDA Programs,'' submitted to *International Journal of Parallel Programming*.
- Wen-Li Shih, **Yi-Ping You**, Chung-Wen Huang, and Jenq-Kuen Lee, ``Compiler for Leakage Power Reduction on Multithreaded Programs,'' in preparation.

# 六、 參考文獻


[1] Steven Dropsho, Volkan Kursun, David H. Albonesi, Sandhya Dwarkadas, and Eby G. Friedman. ``Managing static leakage energy in microprocessor functional units," In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO'02)*, pages 321-332, Istanbul, Turkey, November 2002.

[2] Yen-Hsiang Fan, Yuan-Shin Hwang, Yi-Ping You, and Jenq-Kuen Lee, ``Compiler-based vs. Hardware-based Power Gating Techniques for Functional Units," in *Proceedings of the 6th Workshop on Optimizations for DSP and Embedded Systems (ODES-6)*, pp. 26-35, Boston, MA, April 6, 2008.

[3] Siddharth Rele, Santosh Pande, Soner Onder, and Rajiv Gupta. ``Optimizing static power dissipation by functional units in superscalar processors," In *Proceedings of the 11th International Conference on Compiler Construction (CC'02)*, pages 261-275, Grenoble, France, April 2002.

[4] S. Roy, N. Ranganathan, S. Katkoori, "A Framework for Power Gating Functional Units in Embedded Microprocessors", *IEEE Transactions on Very Large Scale Integrated Systems*, Volume 17, Issue 11, November 2009, Page(s):1640-1649.

[5] S. Roy, N. Ranganathan, S.Katkoori, "Compiler Directed Power Gating in Embedded Microprocessors", in *Proceedings of IEEE International Conference on Computer Design*, October 2009, pages 35-40.

[6] S. Roy, N. Ranganathan, S.Katkoori, "Exploration of Compiler Optimization Techniques for Enhancing Power Gating", in *Proceedings of IEEE International Symposium on Circuits and Systems*, May 2009, pages 1004-1007.

[7] S. Roy, S. Katkoori, N. Ranganathan, "A Compiler Based Leakage Reduction Technique by Power-Gating Functional Units in Embedded Microprocessors", in *Proceedings of 20th International Conference on VLSI Design*, Jan 2007, pages 215 - 220.

[8] Yi-Ping You, Chingren Lee, and Jenq-Kuen Lee, ``Compiler Analysis and Supports for Leakage Power Reduction on Microprocessors," in *Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing (LCPC'02)*, College Park, MD, July 25-27, 2002. (also in *Lecture Notes in Computer Science*, Vol. 2481, Springer-Verlag, Germany, pp. 45-60, 2005.)

[9] Yi-Ping You, Chingren Lee, and Jenq Kuen Lee, ``Compilers for Leakage Power Reduction," *ACM Transactions on Design Automation of Electronic Systems*, Vol. 11, Issue 1, ACM, New York, pp. 147-164, January 2006.

[10] Yi-Ping You, Chung-Wen Huang, and Jenq Kuen Lee, ``A Sink-N-Hoist Framework for Leakage Power Reduction," in *Proceedings of the ACM International Conference on Embedded Software (EMSOFT'05)*, pp. 124-133, Jersey City, NJ, September 18-22, 2005.

[11] Yi-Ping You, Chung-Wen Huang, and Jenq Kuen Lee, ``Compilation for Compact Power-Gating Controls," *ACM Transactions on Design Automation of Electronic Systems*, Vol. 12, Issue 4, Article 51, ACM, New York, September 2007.

[12] Yi-Ping You and Jenq Kuen Lee, ``Compiler Frameworks for Leakage Power Reduction," in Student Poster

Session of *ACM SIGPLAN/SIGBED 2005 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05)*, Chicago, IL, June 15-17, 2005.

[13] W. Zhang, Mahmut T. Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, and V. De. ``Compiler support for reducing leakage energy consumption," In *Proceedings of the 6th Design Automation and Test in Europe Conference (DATE'03)*, pages 1146-1147, Messe Munich, Germany, March 2003.

[14] Wen-Li Shih, Yi-Ping You, Chung-Wen Huang and Jenq Kuen Lee, `` Compiler for Leakage Power Reduction on Multithreaded Programs," Submitted to *International Conference on Embedded Software 2010*.

[15] N. AbouGhazaleh, D. Mossé, B. R. Childers, and R. Melhem, R. "Collaborative operating system and compiler power management for real-time applications," *Transaction on Embedded Computing Systems* 5(1): 82-115, Feb. 2006.

[16] Nevine AbouGhazaleh, Daniel Mosse, Bruce Childers, Rami Melhem, and Matthew Craven."Collaborative operating system and compiler power management for real-time applications," In *Proceedings of the Ninth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'03)*, pages 133-141, Washington, D.C., USA, May 2003.

[17] Nevine AbouGhazaleh, Daniel Mosse, Bruce Childers, Rami Melhem, and Matthew Craven. "Energy management for real-time embedded applications with compiler support," In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems (LCTES'03)*, pages 284-293, San Diego, California, USA, June 2003.

[18] Kemal Aygun, Michael J. Hill, Kimberly Eilert, Kaladhar Radhakrishnan, and Alex Levin. "Power delivery for high-performance microprocessors," *Intel Technology Journal*, 9(4):273-283, 2005.

[19] Anantha P. Chandrakasan, Samuel Sheng and Robert W.Broadersen, "Low Power CMOS Digital Design," IEEE Journal of Solid State Circuits, vol. 27, no. 4, pp.472-484, April 1992.

[20] Matthew Curtis-Maury, James Dzierwa, Christos D. Antonopoulos,

Dimitrios S. Nikolopoulos, “Online power-performance adaptation of multithreaded programs using hardware event-based prediction,” in *Proceedings of the 20th annual international conference on Supercomputing*, pp.157-166, 2006.

[21] V. De and S. Borkar, “Technology and design challenges for low power and high performance," *Proc. of Int. Symp. Low Power Electronics and Design*, pp.163-168, 1999.

[22] Yunsi Fei, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. ”Energy-optimizing source code transformations for OS-driven embedded software,” In *Proceedings of the 17th International Conference on，VLSI Design (VLSID'04)*, pages 261-266, Mumbai, India, January 2004.

[23] Peter Y. T. Hsu and Edward S. Davidson. “Highly concurrent scalar processing,” In Proceedings of the 13th Annual International Symposium on Computer Architecture (ISCA'05), pages 386-395, Tokyo, Japan, June 1986.

[24] Robert Jones, “Modeling and design techniques reduce 90 nm power,” EETimes, URL=http://www.eetimes.com/news/design/features/showArticle.jhtml;jsessionid=43KUZMV5DM11GQSNDLRSKH0CJUNN2JVN?articleID=26806450, August 6, 2004.

[25] Sung Mo Kang and Yusuf Leblebici, “CMOS Digital Integrated Circuits-Analysis and Design,” Tata McGraw Hill, Third Edition, New Delhi, 2003.

[26] Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. “Cache decay: Exploiting generational behavior to reduce cache leakage power,” In *Proceedings of the International Symposium on Computer Architecture*, pages 240-251, Gothenburg, Sweden, June 2001.

[27] Michael Keating, David Flynn, Robert Aitken, Alan Gibsons and Kaijian Shi, “Low Power Methodology Manual for System on Chip Design”, Springer Publications, NewYork, 2007.

[28] Ripal Nathuji, Balasubramanian Seshasayee, and Karsten Schwan. “Combining compiler and operating system support for energy efficient I/O on embedded platforms,” In *Proceedings of the 9th International Workshop on Software and Compilers for Embedded Systems (SCOPES'05)*,

pages 80-90, Dallas, Texas, USA, September 2005.

[29] Massoud Pedram, "Leakage Power Modeling and Minimization," University of Southern California, Dept. of EE-Systems, Los Angeles, CA, ICCAD 2004.

[30] M.D. Powell, S-H. Yang, B. Falsa, K. Roy, and T.N. Vijaykumar. Gated-Vdd: A circuit technique to reduce leakage in deep-submicron cache memories. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 90-95, Rapallo, Italy, August 2000.

[31] Xiaodong Zhang, "High Performance Low Leakage Design Using Power Compiler and Multi-Vt Libraries," Synopsys, SNUG, Europe, 2003.

[32] Mohammad Ali Ghodrat, Tony Givargis andAlex Nicolan, "Equivalence Checking of Arithmetic Expressions using Fast Evaluation," in Proceedings of the 2005 international conference on Compilers, p147-156, September 2005.