行政院國家科學委員會補助專題研究計畫 □ 成 果 報 告
■期中進度報告

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：　98 年 8 月　1　日至　99 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、何柏琀、田璨榮、蔡羽軒、王深泓

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　99　年　5　月　25　日

一、 中文摘要

本計畫為多執行緒程式之漏電耗能管理方法計畫之第二年計畫，主要目的在於設計用於多執行緒程式中的漏電耗能管理方法。第二年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)延續第一年成果，提出 may-be-used-in-parallel component-activity data-flow analysis (MUP-CADFA) 分析方法。我們將繼續此研究議題，並將此研究擴展至多核心系統架構，並探討編譯器與作業系統間互動(interaction)或協同合作。

關鍵詞：編譯器，多執行緒程式，多核心處理器，漏電耗能管理

二、 Abstract

This is the second year of our three-year project, called Leakage Energy Management for Multithreaded Programs. The objective of the second-year project is to design a methodology for leakage energy management for multi-core systems. In this project, we extended the achievement of the first year and made in-depth survey on common concurrent programming models in order to manage leakage on multi-core systems for different concurrent programming models. In the second year of the project, we achieved some significant results: (1) we have deeply investigated into concurrent programming models on the literature. (2) We extended the achievement of the first year and proposed may-be-used-in-parallel component-activity data-flow analysis (MUP-CADFA). We will keep doing research on this subject and extend our research to multi-core architectures, and we will also research on the collaboration between compilers and operating systems.

Keywords: Compilers, Multithreaded programs, Multi-core processors, Leakage energy management

三、 研究目的

在現代半導體技術中，漏電功率在的整體功率中所占的比例愈來愈高，原因是電晶體的大小不斷縮小而速度卻不斷提升。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。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。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 Core 2 Duo 雙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成多個模組，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。

四、 文獻探討

為降低處理器或系統晶片的漏電功率消耗，許多以電源閘控(power gating)方法的研究報告及發明被相繼提出，例如 2002 年 LCPC (Workshop on Languages and Compilers for Parallel Computing) 會議中所發表名稱為”Compiler analysis and supports for leakage power reduction on microprocessors”，或 2002 年在 CC (Conference on Compiler Construction)會議中所發表名稱為” Optimizing static power dissipation by functional units in superscalar processors”，或 2003 年在 DATE (Design Automation and Test in Europe Conference)會議中所發表名稱為”Compiler support for reducing leakage energy consumption”或 2005 年在 EMSOFT (International Conference On Embedded Software)會議中所發表名稱為” A Sink-N-Hoist Framework for Leakage Power Reduction”或 2006 年在 ACM TODAES (ACM Transactions on Design Automation of Electronic Systems) 期刊中所發表名稱為” Compilers for Leakage Power Reduction”或 2007 年在 ACM TODAES (ACM Transactions on Design Automation of Electronic Systems) 期刊中所發表名稱為”Compilation for Compact Power-Gating Controls”等論文中，提供了以電源閘控指令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。其中，電源閘控指令(power-gating instruction)係包含電源啟動指令(power-on instruction)與電源關閉指令(power-off instruction)。電源啟動指令係告知處理器針對某處理元件進行啟動的動作，電源關閉指令係告知處理器針對某處理元件進行關閉的動作。2009 年 Soumyaroop Roy 等人在 IEEE International Symposium on Circuits and Systems 會議中發表了”Exploration of Compiler Optimization Techniques for Enhancing Power Gating”，其論文主要探討特定編譯器最佳化

技術對上述方法的影響，並歸納出合適的編譯器最佳化步驟，以達最佳的節省漏電效果。

然而，前述這些漏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；並且這些方法也僅適用於單核心處理器，對於多核心處理器而言，仍未有文獻關注此議題。

五、 研究方法

■ 並行編程模型(concurrent programming 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：

（一） OpenMP

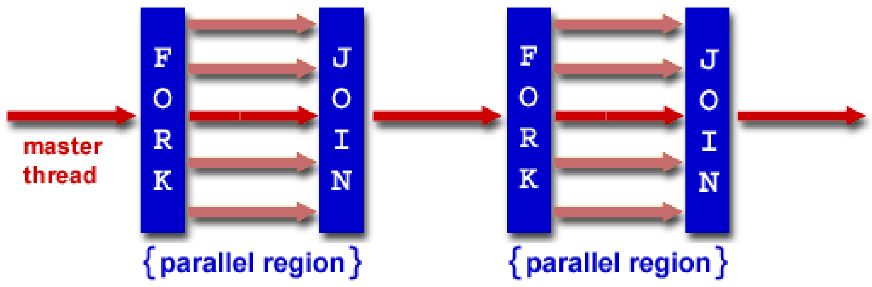
OpenMP是以共享記憶體(Shared Memory)的方式去分享所要執行的任務與資料，以主執行緒 (Master Thread)為任務主軸並搭配應用分岔-聯結模式(Fork-Join Model)來產生與執行可平行區塊(parallel region)任務所需的執行緒列(threads)。當程式撰寫者撰寫OpenMP時，須注意核心要素(core elements)的架構，其為OpenMP程式碼中加入任務與資料分享資訊或創造執行緒所要給予編譯器知道的說明。

● 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。如下圖所示：



● 核心要素(core elements)

OpenMP 的核心要素 (core elements)，如圖表2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 (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。

（二） 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考量。

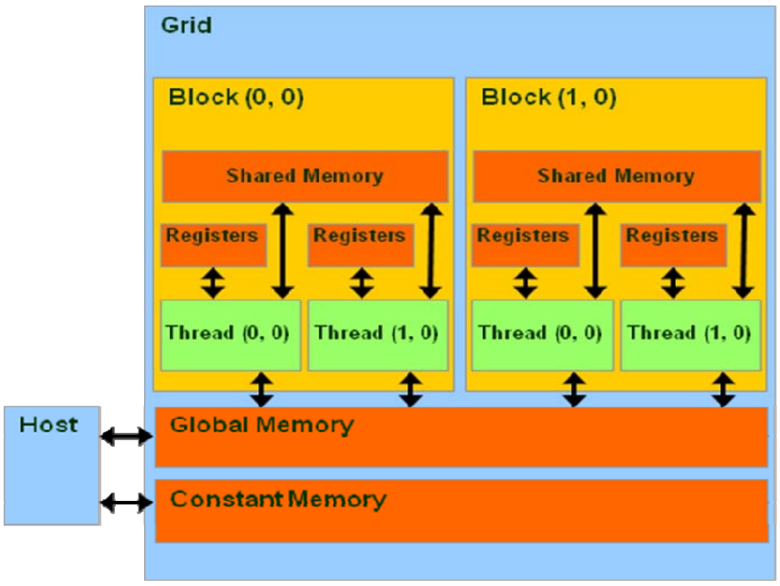
- 核心函式 (kernel function)

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)



如上圖所示，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的記憶體階層，我們做了以下比較：

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)	區塊內交換資料用
常數記憶體	存放整個程式共用的常數	快速	全域共用
全域記憶體	顯示卡中的 DRAM	很慢	全域共用

（三） 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)

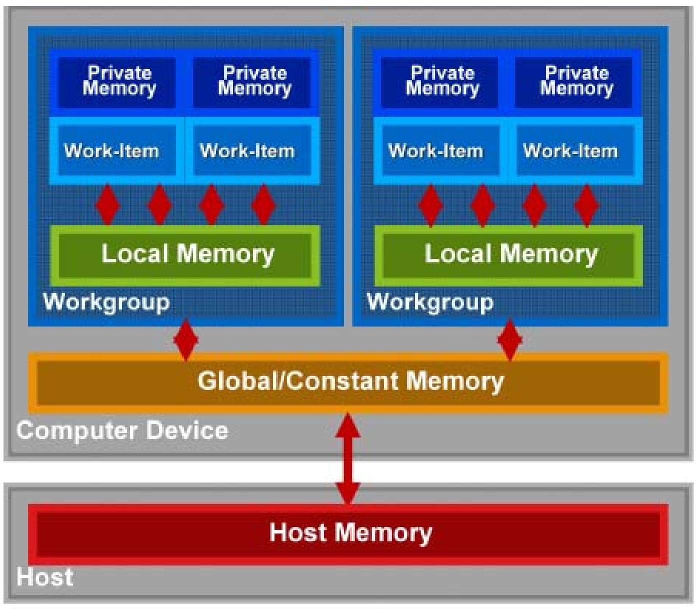
核心 (kernel)，基本的可執行單位程式碼相似於C的函式型式，可做資料平行(Data-parallel)或任務平行 (task-parallel)的平行方式執行；資料平行(Data-parallel)，在資料平行的模型中，定義了N維度的計算域 (N-Dimensional computation domain)，其中也定義了每個獨立的工作項目 (work-item)，相當於CUDA中定義的執行緒，若干的工作項目數量則被定義為工作團體(work-group) (相當CUDA的thread block);而在工作團體中的工作項目可以互相溝通，也可以進行同步 (synchronize)，而在OpenCL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)

如下圖所示，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)。



- May-be-Used-in-Parallel Component-Activity Data-Flow Analysis (MUP-CADFA)

$$C_{IN}(n) = \bigcup_{m\text{: a predessor of } n} C_{OUT}(m) \tag{1}$$

$$C_{OUT}(n) = C_{GEN}(n) \cup (C_{IN}(n) - C_{KILL}(n)) \tag{2}$$

$$C_{Util}(n) = \bigcup (n,c)\text{, where } c \in C_{OUT}(n) \tag{3}$$

$$TC(t) = \bigcup_{\forall n \in t} C_{Util}(n) \tag{4}$$

$$MayC_{GEN}(n) = \begin{cases} \bigcup_{m \in Cons(n)} C_{Util}(m) & \text{if } n \in P \\ \varnothing & \text{otherwise} \end{cases} \tag{5}$$

$$MayC_{KILL}(n) = \begin{cases} \bigcup_{m \in ancestor(Prod(n))} C_{Util}(m) & \text{if } n \in C \\ \varnothing & \text{otherwise} \end{cases} \tag{6}$$

$$MayC(n) = \begin{cases} \bigcup_{m \in Prod(n)} MayC_{OUT}(m) - TC(T(n)) & \text{if } n \in C \\ \bigcup_{m\text{ a predessor of } n} MayC_{OUT}(m) & \text{otherwise} \end{cases} \tag{7}$$

$$MayC_{OUT}(n) = MayC_{GEN}(n) \cup (MayC(n) - MayC_{KILL}(n)) \tag{8}$$

$$C_{Parallel}(n) = \bigcup c\text{, where } (*,c) \in MayC(n) \tag{9}$$

We define $C_{Util}(n)$, $C_{IN}(n)$, $C_{OUT}(n)$, $C_{GEN}(n)$, and $C_{KILL}(n)$ for each PEG node n to gather the component-activity data-flow information. We say that a component activity is generated at n if a component is required for this execution, symbolized as $C_{GEN}(n)$, and that it is killed if the component is released for this execution, symbolized as $C_{KILL}(n)$. The flow equation (1) follows from the observation that $C_{IN}(n)$ is the union of component activities arriving from all the predecessors of B. Equation (2) formulates the component activities after node n is executed—the component activities are either generated in node n or not released yet.

We create a data-flow equation which computes the component utilization $C_{Util}(n)$ in Equation (3). It computes the component utilization by collecting component activities c that is still active from nodes before n or generated by node n. The $C_{Util}(n)$ consists of two-dimensional vectors, (n, c), where n is a PEG node that use c components. Unlike $C_{OUT}(n)$ set, $C_{Util}(n)$ records not only the component-activity c for node n but also node n itself. The paired component-activity and node information is important to MUP-CADFA, because we need it to trace component-activity along MHP nodes. Another important information to MUP-CADFA is the relationship of threads and nodes. To symbolize this relationship, we say that a node n belongs to thread t as $T(n) = t$. The component activities of a specific thread t, symbolized as $TC(t)$, is gathered by traveling a possible nodes of a thread as in Equation (4).

We use an iterative approach to compute the desired results of $C_{Util}(n)$, $C_{IN}(n)$, and $C_{OUT}(n)$ after $C_{GEN}(n)$ and $C_{KILL}(n)$ has been computed for each nodes.

For gathering the may-be-used-in-parallel component activity information, we define four set $MC(n)$, $MC_{OUT}(n)$, MC_{GEN}(n), and $MC_{KILL}(n)$ for each PEG node n. We also define a *ancestor*(n) set which gathers node n's predecessors and their predecessors recursively. The computation of may-be-used-in-parallel execution is by tracking paired component-activity and node information. When a node n is a normal CFG node without communication with other threads, its MHP nodes would be completely the same as its predecessor, and it would continue passing all the information to its successor. We say that a component activity is generated to a producer node n when n triggers its consumer node *Cons*(n), which makes the successor of *Cons*(n) be able to executed concurrently with n. Equations (5) and (7) describe the case above. On another hand, we say that a component activity is killed from a consumer node m when its producer *Prod*(m) triggers m, which means that the ancestor of *Prod*(m) would not executed with m concurrently, so the component activity of *ancestor*(*Prod*(m)) could be killed in Equation (6).

After computing MC and MC_{OUT} sets, we have to take a step to symmetrize all nodes in MC. The symmetry step is done by adding all $C_{Util}(m)$ into $MC(n)$ if $(n, *) \in MC(m)$. Since the change of MC set would affects MC_{OUT} set, all nodes that have been symmetrized should recompute their MC_{OUT} set until all sets are saturated.

Finally, we could compute themay-be-used-in-parallel component activity $C_{Parallel}$ by unpacking the vectorized component utilization information in MC sets as shown in Equation (9).

六、 結果與討論

本年度計畫中，我們的研究重點包括：

（一） 並行編程模型深入研究

我們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。

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	3.0	3.0	1.0
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 CPU 上:優 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

（二） May-be-Used-in-Parallel Component-Activity Data-Flow Analysis (MUP-CADFA)

We incorporated the low-power optimization phase just before code generation; that is, after all traditional performance optimizations are performed. Hence, the additional phase has little or nearly no influence on performance; it only inserts power-gating instructions or predicated-power-gating instructions and thus barely affects execution behavior. The implementation was based on SUIF2 and the Control Flow Graph (CFG) and Machine libraries from Machine-SUIF. Figure 1 shows the compilation flow.

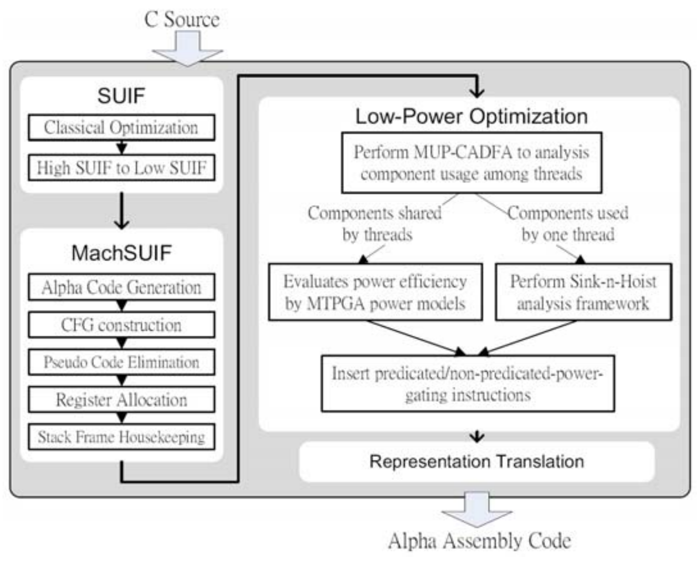

C Source
SUIF
Classical Optimization
High SUIF to Low SUIF
MachSUIF
Alpha Code Generation
CFG construction
Pseudo Code Elimination
Register Allocation
Stack Frame Housekeeping
Low-Power Optimization
Perform MUP-CADFA to analysis component usage among threads
Components shared by threads
Components used by one thread
Evaluates power efficiency by MTPGA power models
Perform Sink-n-Hoist analysis framework
Insert predicated/non-predicated-power-gating instructions
Representation Translation
Alpha Assembly Code

Figure 1 A compilation flow of power management for multithreaded programs

To verify our proposed MTPGA algorithm and PPG mechanism, we focus on investigating component utilization in MHP region. We first apply two floating point DSPstone programs to each hardware thread and measure the power consumption of ALUs and multipliers, including four integer ALUs, an integer multiplier, four floating point ALUs, and a floating point multiplier. For exploring the efficiency of our MTPGA and PPG mechanism, we experiment every possible combination of two threads and report the best, worst, and average power consumption for every DSPstone program. Figure 2 shows the normalized power consumption in all function units with PPG mechanism to one without PPG mechanism. The average normalized power consumption of worst, best, and average results are 89.75%, 53.27%, and 62.75%, respectively.

Figure 3 shows our experiment result of random selected six groups of DSPstone programs and their power consumption of ALUs and multipliers. The first column shows the selected programs. From column two to column four we presents the component turn-off rate (i.e., component turn-off cycle count/execution cycle count) of floating point ALU, floating point multiplier, and integer multiplier, respectively. The fourth and fifth column shows the energy consumption of all ALUs with or without PPG mechanism. The last column reports the power consumption reduced by PPG mechanism. The average of normalized power consumption is 73.15%. Compared to the two-thread version, the experiment result of power saving slightly decays, but it still shows that our MTPGA and PPG mechanism are practicable in reducing leakage power at various multithreading environments.

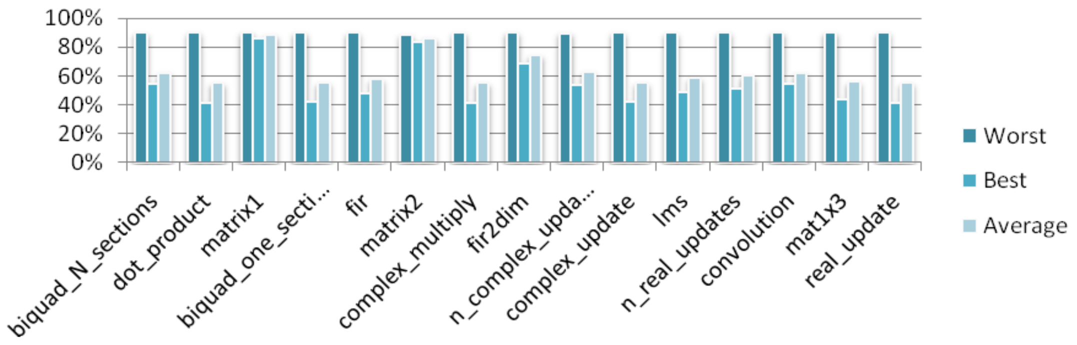
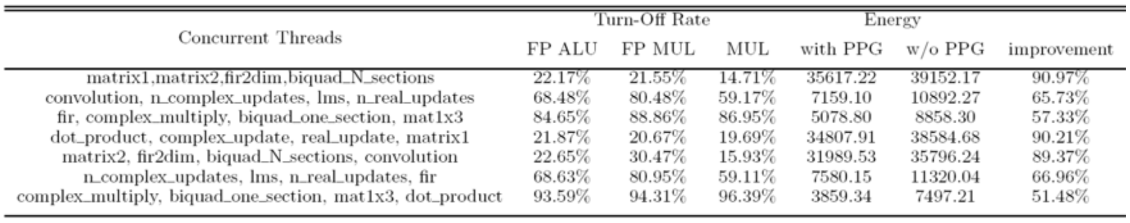


Figure 2 The worst, best, and average normalized power consumption of thread combinations for DSPstone programs



Concurrent Threads	Turn-Off Rate			Energy		
	FP ALU	FP MUL	MUL	with 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 3 Component turn-off rate and power consumption of four concurrent executed threads

七、 計畫成果自評

根據以上這些成果，我們將可以對往後的研究進行更進一步地瞭解與改良。並藉由一些初期實驗的測試，得知我們研究的方法及結果確實對多執行緒程式的漏電耗能管理有所助益，並且研究成果已投稿至 2010 International Conference on Embedded Software (EMSOFT’10)[14]，符合計畫達成的目標及進度。未來本實驗室將持續針對多執行緒程式之漏電耗能管理相關設計進行廣泛與深入之研究與實作。

八、 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*.