行政院國家科學委員會專題研究計畫　成果報告

子計畫一:嵌入式即時作業系統核心程式之設計與實作

計畫類別： 整合型計畫
計畫編號： NSC91-2218-E-002-005-
執行期間： 91 年 08 月 01 日至 92 年 07 月 31 日
執行單位： 國立臺灣大學電機工程學系暨研究所

計畫主持人： 王勝德

計畫參與人員： 張致良、吳國賓、王暉文、謝榮明、樊乃維、陳宏明、黃乙丞

報告類型： 完整報告

處理方式： 本計畫可公開查詢

中　華　民　國 92 年 10 月 31 日

行政院國家科學委員會補助專題研究計畫

成果報告

eHome: 電子家庭雛型之設計與實作—子計畫一：嵌入式即時作業系統核心程式之設計與實作(3/3)

計畫類別：□ 個別型計畫 ☑ 整合型計畫

計畫編號：91-2218-E-002-005

執行期間： 91年8月1日至92年7月31日

計畫主持人：王勝德

計畫參與人員：張致良、吳國賓、王暉文、謝榮明、樊乃維、陳宏明、黃乙丞

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報告各一份

□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：國立台灣大學電機系

中 華 民 國 92 年 8 月 31 日

中文摘要

當今許多嵌入式系統皆用於可移動或攜帶方面的運用，電源管理方面的控制也已普遍的存在於新一代的CPU當中。在本報告中將對嵌入式即時系統的電源管理技術做一番討論。其中包含了動態電壓管理(DPM，Dynamic Power Management)與(DVS，Dynamic Voltage Scaling)這兩方面，並且在即時核心中運用動態電壓調節技術(Dynamic Voltage Scaling)於RT-Linux 3.0並以康伯電腦的iPaq 3630做實驗的平台，特別針對調節時期的電壓與頻率的轉換耗損方面的平衡，提出一個電源管理導向(Power Aware)以Linux為基礎的嵌入式即時作業系統。

關鍵字: 嵌入式作業係統、Linux、電源管理， 即時系統， 能量耗損

Abstract

Embedded systems have existed in many hand-held devices and the power-saving mechanism has been supported by many microprocessors. DVS (Dynamic Voltage Scaling) is a power-saving technique to calculate and adjust the voltage and frequency at run time, according to the workload requirement of OS. In this project, we developed DVS algorithm and implemented in RT-Linux 3.0 running on iPaq 3630. In this project, we addressed the issues and considered the management overhead of power transition, which has not been considered in the previous research. We proposed an algorithm that can reduce the power transition overhead up to 30% of the original one at most, and it will not damage the real time property. Finally, we implement the algorithm in the proposed Linux-based embedded operating systems.

Keywords: Embedded Systems, Linux, Power Management, Real Time System, Energy Consumption

目錄

第一章 緒論

1.1 動機

近些年來手持式或行動式的嵌入式系統設計面臨到許多系統本身資源分配相關方面的問題。 其中這些裝置大都使用”電池”供給系統操作的電源，但是在使用上電池的蓄電量往往供不應求。目前流行的這類裝置包括:無線通訊和影音方面的應用，都是具有比過去傳統裝置更耗電的趨勢。因此，最近有些系統設計的研究論點是以電源管理為中心取代了單純以CPU為中心的設計觀念也逐漸的增加[1]。 所以如何克服電源的瓶頸已成為目前熱門的研究主題之一。此外，俗話說:無法開源，只得節流。省電與有效運用能源的研究在各不同的領域中都有相當多的研究與討論。 事實上這也是手持式系統設計裡最重要的考慮因素之一。 省電的設計不只能夠延長電池的使用時間，並且能夠降低此類裝置的維修成本與延長裝置的使用壽命。因為低耗電意味著不需常常將系統做開關機的動作、拆卸與更換電池或者需要常常充電，這樣一來當然就能減少人力物力的介入維護，並且有些系統因低耗電所以較不會產生熱量，因此機構方面也可以採取較為簡單的設計而且零件的壽命也會較長。如此一來整個系統的成本也可以下降。

從IC設計和作業系統等計算機科學方面已有許多低耗電或省電方面的研究[2，3]。 在攜帶型電腦方面，各家軟硬體製造商都遵循相關的電源管理的規格，例如:(ACPI，Advanced Configuration and Power Interface)[4]，它能夠讓作業系統控制若干不同的工作模式或關閉系統的電源[5]以達到省電的目地。 此外，目前許多電源控制的研究都在即時系統方面的應用，因為大多數有電源限制的應用都具有回應時間的要求限制。例如:行動通訊和手持式的數位影音撥放裝備。

然而過去都使用固定的電壓與頻率來設計系統，所以沒有辦法對相關的零件(尤其是CPU)做電源的管理。近來因為半導體科技與電源設計技術的進步使得當系統運作時能夠即時改變CPU的運作頻率和電源電壓。 所以動態電壓調節技術(DVS， Dynamic Voltage Scaling)是就因此而產生，DVS是將電壓降到一個適當的操作範圍並且能夠滿足相對的工作表現，其所降低電壓所導至能夠節省能量的消耗為平方比。一般而言在即時系統(Real-Time OS)中控制和選擇適當的操作頻率與電壓準位是由排程(scheduler)單元負責，基本上藉由系統內部的資料讓電壓排程依據每一個工作(task)的啟動時間(release

time or arrived time)與執行時間的限制或週期等資料計算出合適的操作參數，也就是說針對目前的運算能力和預測未來所需的運算能力做一番調整。

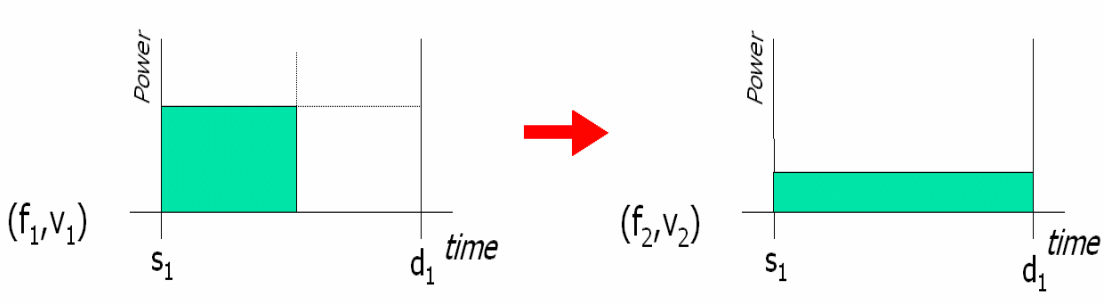


圖 1.1 DVS 示意圖

基於前文所提到的資料，DVS技術已廣範的運用在許多的系統上。那麼，這項技術的使用過去已有許多的研究，然而，卻少有關注於轉換過程的內涵。 這內涵包括了，計算和處理DVS所產生的時間延遲和能量耗損。因此，本報告將針對硬即時系統(Hard Real Time)中的電壓排程對於這些耗損做深入的研究， 包含:系統運作時動態調節會損及即時性的分析，還有這些耗損的原因有那些。 此外，在本文中將提出減少動態電壓調排程時所產生的耗損且精確的模擬出此方法的成效。

1.2 CPU的省電模式

CPU的耗電管理主要是調整CPU的供給電壓和頻率。目前市面上一些新型的CPU大都具有上述兩種省電的模式。即使有些CPU並不是為攜帶型產品所設計的也都會具備省電的功能，只是節省的程度不一。因為目前有些桌上型的CPU運作時會產生非常驚人的熱量，這些熱量容易導致電子元件壽命的縮短，因此能夠具備省電模式是解決的方法之一。

在頻率調整方面，當CPU沒事做或只有少許的工作量時，可以降低運作的頻率來節省耗電。在這方面，近來的CPU具備許多的模式來達成這樣的效果。例如:Intel的Mobile Pentium Ⅲ具有7種模式:Normal， Stop Grant， Auto Halt， Quick Start，HALT/Grant Snoop， Sleep， Deep Sleep[6]，其中Deep Sleep是最省電的模式，但是需要花費30us才能回到正常的工作模式，若是從Auto Halt返回正常的工作模式則需要10 bus clock才能完成。因此，這相關的時間消耗必須加以注意。此外，還有的省電措施包含內部功能模組的選擇，此方式的作用為:將不需要使用的功能關閉以節省耗電量，例如:不需使

用浮點運算時，可將FPU(Floating Point Unit)關閉不用或者是內含的周邊，像是串列埠(serial port)關閉。

1.3 DVS技術

DVS (Dynamic Voltage Scaling)是指CPU可以依照其原設計的規格給予不同的工作電壓,且改變工作電壓不需關閉電源重新啟動的一種技術,例如:Transmeta的Crusoe™ CPU, AMD的Mobile K6和Athlon的CPU， Intel Xscale系列的RISC。

在降低電壓方面，CPU在較低的電壓底下工作可以節省電源和減少能量的消耗，一般而言，CPU的耗電量與這個式子 V^2f 相對應的，V是工作電壓，f指頻率。

所以DVS是一種以CPU的耗電量與供給電壓間的關係是平方比為基礎的省電技[7]，進一步說明:CPU運作頻率的降低意味著CPU的電壓也可以跟著降低以節省耗電[8]。 以這種情況來說電壓降低所導致執行時間的延長，大約是呈平方的反比。 其中有一方面需特別注意;就是用來處理調節CPU速度時軟硬體所要付出的代價， 這代價包含了電源損耗與時間的延遲。 有些DVS相關的研究對電壓調節時所需的代價是比較樂觀的[9]或忽略不計。因此，在第四章中將詳細討論這方面的問題。

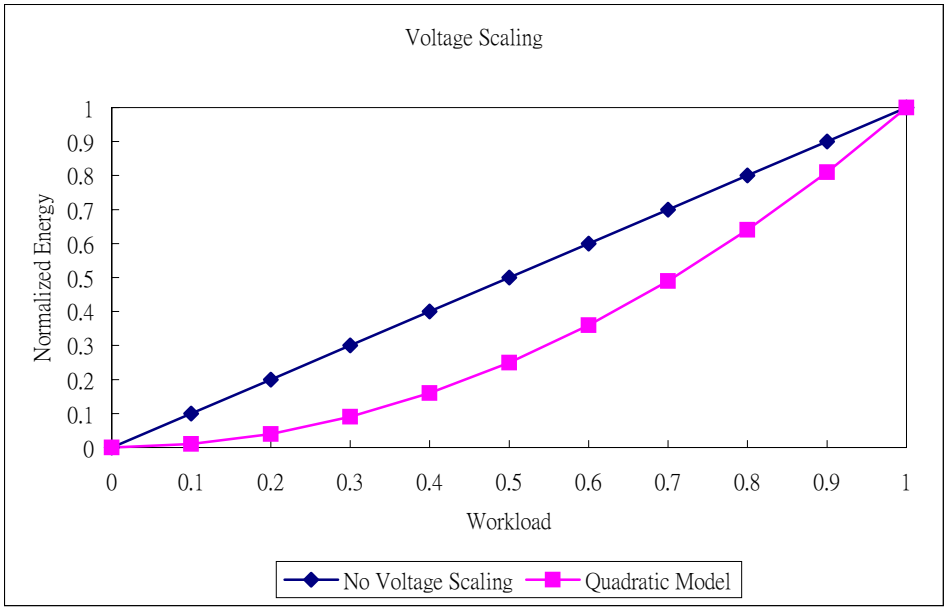


圖 1.2 Voltage Scaling

在Hard Real-Time系統使用DVS技術的電壓排程中，有兩種主要的方法:Intra-Task和Inter-Task。

其中最主要的差別為:這兩種類的演算法所計算或預測出來的空閒時間(slack)是使用在目前的Task還是下一個Task上。Inter-Task的電壓調節點是介於Task與Task之間，而Intra-Task則是在Task內部置入調節點。因此，Inter-Task所產生的slack是給下一個Task使用，而Intra-Task使用的Task則是用予目前的Task。

1.3.1 Intra-Task DVS

在即時系統中每個Task最差情況下的工作時間為WCET(Worst Case Execution Time)。然而在一個Task內可能有許多執行的分支與路徑，因為執行的路徑不同所以造成執行時間的變化。所以當執行路徑不是WCEP(Worst Case Execution path)時工作將產生slack。在這種情況下，Intra-DVS為利用slack的時間，可以立即進行調整以節省能量。Intra-Task的演算法有下面兩種:

1. 路徑法

路徑法是依程式執行的路徑去決定工作電壓和執行頻率，若是執行的路徑與預先參考的執行路徑不同時，則工作電壓和執行頻率將會被調整。所以如果新執行路徑的執行時間多過於參考路徑，則工作電壓和執行頻率將會上升以保持工作的即時性。 換句話說，如果新的路徑比參考路徑早完成的話，那麼CPU的工作電壓和執行頻率要下降以減少能源的消耗。

2. 推測法

推測法為一開始執行就用低速然後視需要的時候再加速，比一開始用高速當slack發生時再減速來得好。 一開始用低速執行，如果工作結束得比WCET早，如此將不需要用高速去執行。 理論上，如果能夠推測Task的工作時間那麼就能夠計算出最佳化的排程。此外，運用推測法CPU的速度將在既定的時間點提高工作電壓和執行頻率。因此推測法有可能無法完全運用所有的slack，而路徑法則可以完全運用slack。

1.3.2 Inter-Task DVS

Inter-DVS是利用TASK與TASK之間的時間作工作電壓和執行頻率的調節，步驟為:(1)執行。(2)工作完成並計算下一個工作的執行時間。(3)指定下一個工作的供應電壓。(4)執行下一個工作。大部份的Inter-DVS演算法在計算下一個工作的執行時間上有所不同

一般Inter-DVS包括二個方面：slack的評估和slack的分配。

Slack評估的目的是盡可能的將slack time找出來，而slack分配目的是將slack分散使得工作的條件儘可能平均。Slack的來源有兩個：靜態的slack是指在WCET到時間限制間那些額外的時間。而動態的slack將視工作在執行時所產生的空閒時間。

1.3.2 Slack評估的方法

(1)靜態 靜態常數的評估法是用來計算最穩定工作速度的方法，保證工作排程的最小執行速度。

(2)動態 三個常用的動態評估技術：

- **擴張法** 即使所有的工作用最大常速度排程，因為真正的工作的執行時間常常少於它們的WCET所以會有動態slack times。一個簡單的方法用來估計動態slack time，就是用下一個工作的啟始時間。

- **優先權利用法** 這個方法的基本想法是，當高優先權工作在它的deadline前完成那麼接下來低優先權的工作，能夠使用高優先權剩下來的slack time。相反的，也有可能是高優先權的工作去利用從低優先權工作的slack time。然而後者要精確地實作，需要大量的計算。

- **經驗法** TASK的實際執行時間往往都不會到達最長執行時間。為提高預測的準確率，可以使用過去執行過的經驗來提高slack的預測準確率。

1.4 相關研究

在過去的研究中電壓調節點方面的研究大約分為兩類:一種為Intra-Task，另一種為Interval Scheduling的方式。 Intra-Task的方法主要是利用編譯程式(Compiler)的幫助，自動[10]或手動[11](由使用者指定)的方式加入電壓的調節點，主要是希望實際電壓設定的曲線能夠符合應用軟體的確時需要，這種方法的好處是:電壓需求的曲線確時能夠較接近最佳的狀況，因為，設計者可以依其特別的運作方式在程式內部較敏感的地方加入調節點以修正電壓的曲線。 其缺點是，程式碼的增加。因為，每個點都要加入相關的特徵值以修正用電的情況，那麼相關的資料和軟體都會增加程式碼。 其次，程式內部執行的路徑(又可當作所需執行的程式碼)因為判斷式而有所不同，若在每一個Base-Block都加入調節點。那麼程式的執行效率將會大大的降低。

Interval Scheduling方面，是介於Task Level和Intra-Task的方式，主要是OS透過一個固定的Interval，搜集和設定目前的電源需求。 這個方法的優點是，它能夠提供比Task Level更多的電壓調整點，因此能夠更加的接近實際電源曲線，且程式方面不需插入額外的程式碼。因此易於使用。 其缺點是，此方法較適用於電壓與頻率的變化為線性的改變。但實際上目前的CPU都是非連續的電壓階層和頻率。因此，即使計算出結果，但仍然不適用。此外，Interval scheduling的方式，其Task內相依的狀況無法確實的被反應出來。

此外，上述兩種方法並未針對電壓調節點做深入的探討。 電壓調整點本身就會耗費能源還有時間的延遲。 若太多的電壓調整點則會造成系統效率的降低與能量的耗損甚至即時性的破壞。若是電壓調整點太少，則容易造成能量的浪費。

為此，本研究採用Inter-Task的電壓調整策略，並針對非連續電壓和頻率的CPU為平台，在硬即時系統的架構下，求取電壓調整點與電源調整的平衡策略。以降低電壓調整點所造成的能量和時間的耗損並且保持其電壓管理的特性與即時性質。

1.5 問題描述

若有一組Task，$\tau_i \in Task\{1..N\}, 1 \le i \le N$ ，C_i為 τ_i需要執行的週期，S_i為 τ_i執行的速度單位為Hz，S_{max}和S_{min}分別是最快和最慢的執行速度，T_{wcet_i} 為最長的執行時間，T_{ac_i} 是實際的執行時間，C_o是電壓調節點的耗損:

那麼當無電壓調節耗損的狀況時，CPU的利用率U為:

$$U = \sum_{i=1}^{N} \frac{(C_i / S_i)}{T_{wcet_i}}$$

從上式來看，U<1時，系統是可排程的且具電壓調整的可能，以節省耗電. U=1時，系統的執行速度必需為S_{max}，無法省電. 當U>1時，Task將無法排程。

因此，當計入電壓調節耗損時仍然必需滿足以上條件。

$$1-U \geq \sum_{i=1}^{N} \frac{(C_o / S_i)}{T_{wcet_i}}$$

因此，將產生兩個問題:

- 若把C_o的執行耗損併入WCET內，可解決無法排程的問題，但是如此一來容易造成CPU的閒置與浪費。因此，需要一個方法幫助設計者決定CPU的運算能力。
- 因此，可否將電壓調節點從N個改為M個，$0 \leq M \leq N$。 (N-M)則是可減少轉換的耗損，但不能破壞原來的即時性與省電的特性。

上述的問題可以經由排程的策略和耗電分析方面著手以符合實際應用上的需要。

1.6 研究目的與方法

研究目的

瞭解動態電壓排程的電源及工作頻率改變時的時間延遲與能量耗損

從系統觀點提出改善的措施

研究方法

針對硬即時系統的電壓排程做為平台，採取電腦軟體模擬的方式進行。

模擬的重點在於即時性的比較與能量消耗的比較。

詳細的內容請參考第四章。

1.7 報告內容概述

本章為緒論，主要提出研究的動機與初步的背景資料做一簡要的說明。 接著為討論調節時期時間的延遲與能量的耗損；其原因皆為DVS而生，所以在第二章中則討論DVS相關原因與過去的研究並區分為動態排程與靜態排程兩方面。 DVS與即時系統間的關係則在

第三章中敘述。 第四章則是先將整個DVS排程時間與能量的關係建立其數學模型，然後提出轉換與耗損之間平衡的公式與演算法以增加省電的機會且不會造成即時性的消失。第五章主要是呈現實驗相關的工作，從實驗環境的建立，實驗參數，測試樣本，數據分析與製圖。 第六章將討論本研究的重點與成果和未來的工作。接著，參考資料與附錄將列印於後。

第二章 關於DVS (Dynamic Voltage Scaling)

2.1 DVS電壓排程

DVS的目地是將task在執行時的速度降低以至於CPU主電壓的降低而達到節省耗電量。在即時系統方面目前在DVS中被提出的方法是針對TASK在執行時間的限制下，尋求最小的電源消耗。 目前有許多即時系統方面相關的排程理論[12，13]都是將CPU的執行速度當作為常數， 因此，並不符合現在實際上的應用。

要將CPU執行的速度降低得要多方面的考慮。首先是靜態的DVS:是將一堆TASK以最長的執行時間計算其所需最低的執行速度。

關於DVS排程方面， 其中EDF (Earliest Deadline First)算是一種蠻簡單且排程效果不錯的一種方法，有些較為複雜的方法是採用經驗法則，也就是說系統利用過去執行的經驗做為下次決定執行參數的依據，這些經驗的取得可以由設計者介入調整或動態的經由一套回授的機制將相關資料予以保留以供未來的執行週期運用。在相關的應用或研究上，各家的結果若跟不使用DVS的排程系統比較下，大約能夠節省1.4%到90%的能量[14，15，16，17，18，19，20]。

這一方面過去的研究中 Yao 等人[21]曾提出了靜態的離線(static off-line)的演算法，其假設在獨立非週期的情況下，TASK滿足所有的時間限制，能夠獲得最低的電源消耗量。 這演算法在n個TASK的時間複雜度為:$O(n\log^2 n)$。

與此相關的研究在[22]中提出了一種非週期性的在線(on line)排程方法，這種演算法是計算全部TASK所需執行效率的總和，然後設定固定的執行速度給所有的TASK。然而，此方法將不會影響到離線(off-line)排程的結果。 在論文[23]中則對非先佔式多工(Non-preemptive)的能源排程做了討論。 論文[24]則討論有一堆週期性的工作並具有相同的工作週期則CPU的電壓變化率則有其上限。因為作者提到，這類問題即使是線性變化的還是難以控制，因此提出了方法來解決這個問題。關於上述靜態排程方法在一般週期性的工作模式中其電緣消耗的特性有潛在的不同[25]， Aydin等人[26]最近則說明了靜態的動態電壓排程與回餽式的排程有著相同的問題。

2.2 即時系統的排程設計

設計即時系統時，基本上都考慮系統在最差的狀況下運作，但是系統實際運作時其運作應處於較佳的狀況也就是說時間上或資源上都比最差期況下要寬裕。 因此，大多數的研究專門使用最差狀況的執行時間(WCET， Worst-Case Execution Time)去保證系統執行時的時間限制，所以可以從那些執行後所剩下時間或CPU的運算能力中獲得好處。事實上，程式執行時它真實的執行時間有很大的變化;例如:在參考文獻[28]中提到，一般的應用程式在最佳狀況和最差狀況下程式執行的效率可能差上10倍。 動態的監視和搜集那些沒被用到的運算能力或執行效率就是在電源排程的研究中如何在有效的執行情況下精確的獲得執行所需的時間，據此與WCET的狀況相比較以節省那些多餘的電源。在這方面的研究中，主要針對CPU的執行速度做動態的調整，調整的依據是參考過去執行的情況，因此，系統得不斷的記錄運行的狀態。 但是這會有一個問題，因為它是參考過去的執行狀況，也許用平均值的方法或其它估計的方式並無法依據真正的時間限制，使得在某些情況下TASK所被負與的執行速度會過慢將導致即時性的破壞。

2.3 靜態電壓排程

靜態的電壓排程(Static voltage scheduling)又稱為離線電壓排程(Off-line Voltage Scheduling)，是指系統在離線的狀況下將利用Task τ_i在時間的限制下需執行最多的週期藉以計算出CPU執行的速度，若所需週期為C_i則在S_{max}的執行速度下若是在時間限制內就能夠執行完畢那麼τ_i就有可能因為降低執行的速度而節省電源並且能夠符合時間的限制。

上述方法稱為”靜態電壓排程”並且強調它的”靜態”屬性，因為這種方法只有討論最差情況(worst-case)的task參數。往往最差情況的參數是設計人員在設計時就予以規劃，但缺乏實際執行時的參考資料可供電壓調節之用。

以下還是針對Task Level及System Level這兩種層次討論靜態電壓排程:

針對Task level的電源管理， Task τ_i它的時間限制(deadline)為D_i，則τ_i在執行速度設定為 $S_i = \frac{C_i}{D_i}$ 時是能夠安全的被執行完畢，當然這裡必需考慮的是:若$S_i > S_{max}$那麼Task τ_i將不可能在時間的限制內被執行完畢也就是無法排程。此外，若$S_i < S_{min}$那麼CPU的速度

還是要被設定為$S_i=S_{min}$因為，CPU的執行速度有其上下限，S_{min}為下限，因此必須額外考慮臨界點的問題。

基於以上說明，單一的Task Level之電壓調整最好在程式開啟始的位置就利用相關的參數將速度計算出來，使得往後在執行時就能節省電源。

System Level方面， 若有N個週期性的Task使用EDF的方式排程，若是其執行速度的設定落在g(S)的函數圖型上，那麼所有task的時間限制都能滿足而且耗電量的總合也會是最小的。如此一來，一堆週期性的task在CPU使用率U的狀況下，U必須小於等於1(大於1將無法排程)，則CPU的耗電量將是$\max\{S_{min}, US_{max}\}$。

為了適當的表示出靜態電源管理的成果;在此,將g_{idle}假設為0且$g(S)=\alpha S^3$(因為電壓與頻率間存在對應的關係，為簡化計算過程所以假設$V \approx S$)。 若$T_1 \cdots T_N$工作週期的最小公倍數為T_{lcm}則當執行速度參數$S_{max}=1$，T_{lcm}這段週期的能量消耗為

$$\sum_{i=1}^{N} g_i(S_{\max}) \frac{C_i}{S_{\max}} \frac{T_{lcm}}{T_i} = \alpha U T_{lcm}$$

若執行速度設為US_{max}那麼，在T_{lcm}週期的能量消耗將為$\sum_{i=1}^{N} g_i(U) \frac{C_i}{US_{\max}} \frac{T_{lcm}}{T_i} = \alpha U^3 T_{lcm}$將低於$S_{max}=1$的時候，因為$U^2$小於$\alpha U T_{lcm}$。

舉例來說，若U為0.5則靜態排程出來的結果將只有原消耗能量的25%。

這種最佳的狀況是因為我們可以給予一個固定的實際利用率給函數$g_i(S)$,但實際上不同的Task被賦予不同的工作，因此有可能會使用不同的硬體或大小不同的記憶體甚至是不同的存取模式。 因此，對每個不同的Task都會有不同的耗電率，所以$g_i()$也會因應τ_i而有所不同。這不同的耗電率可以對應到不同的Task或在一Task內不同的區段(Intra-Task)。

所以這裡就是要找到省電的最佳化:

$$\min\{\sum_{i=1}^{N} g_i(S_i) \frac{C_i}{T_i S_i}\}$$

如此:

$$\sum_{i=1}^{N} \frac{C_i}{T_i S_i} \le 1$$

$$S_{\min} \le S_i \le S_{\max}$$

上述的式子是說明τ_i在執行時的速度為S_i，那麼τ_i的執行時間為$\frac{C_i}{S_i}$，所以Task執行的時間將從$\frac{C_i}{T_i}$到$\frac{C_i}{T_i S_i}$。EDF排程只要是將每個Task的利用率累加，其結果小於等於1則不會產生miss-deadline的情況。

在解決以上最佳化的問題之後(詳細的推導過程可以參考[29])，所得的相關參數，例如:速度的參數。可以保存在Task的參數表內，當Task執行時再透過系統去設定相對的速度。所以OS也要做點小改變，在做Context Switch時也要跟著改CPU的執行速度。

針對靜態電壓排程做點小的總結:在靜態排程中從系統的角度或單一Task的角度來看排程方法和目標是相同的;都是只要考慮實際利用率和WCET的關係。若是考慮單一的Task那麼問題是相當的單純，若是考慮若干Task那麼必需所有的Task一起考慮，否則有可能會發生不能排程的現像，此時，以單一的Task而言未必是最佳的狀況，但為維持所有Task的工作能夠準時完成所有的利用率必須合併計算。

2.4 動態電壓排程

動態電壓的排程主要是跟據Task實際執行的狀況加以加以統計，精準的計算出與WCET之間的最大範圍，此範圍將是實際能夠加以省電的區域，因此，又比靜態的電壓排程更貼近於實際耗電的曲線。為了簡化討論的過程，因此假設所有的Task耗電的因素都是相同的，S_s是經過靜態電壓排程所給定的執行速度的參數。

當Task實際執行時間若在WCET的指定時間前完成，那麼這段剩餘的時間就稱為slack(沒被用到的可執行時間)，slack是會變動的，因為Task內部被執行的cycle數目有多有少，因此，充份的利用這些slack再加以省電，更可彌補靜態排程的盲點。

2.4.1 空閒時間(slacks)

有一種估計slack的方法是利用過去同一Task被執行後的情況下加以統計，換句話說，就是從歷史中找答案。

動態的調整電壓和執行時的頻率都會造成系統在時間上的延遲，所以若WCET的時間為t_{wc}，經過調整後實際所花費的時間為t_{ac}，那麼$t_{ac} \le t_{wc}$才行，否則會因調整而miss-deadline，然而$slack_{early}=t_{wc}-t_{ac}$，這$slack_{early}$就是實際上可以節省能量的時間長度。此外，有另一種計算的方式是以執行的cycle數來做計算，假設對最多需要執行的cycle為C_{wc}，實際執行cycle數的平均值為C_{av}。 S_s是由靜態排程所決定的執行速度。因此，t_{wc}將會等於$\frac{C_{wc}}{S_s}$，也就是說t_{wc}是Π_{wc}個cycles在S_s的速度下執行所需的時間。換句話說，slack的時間為$D-t_{ac}-\frac{C_{wc}}{S_s}=D-t_{ac}-(D-t_{wc})=t_{wc}-t_{ac}$就是指從目前的時間到deadline以$S_s$的速度執行完$C_{wc}$個cycles後的slack(空閒時間)。

在指定的工作下，讓Task執行到WCET的狀況其實相當的罕見，有可能是不正常的情況發生，一般而言，設計者在發展時已將各種可能都考慮進去了，因此，時間都會有所餘裕。因此，也給了能夠充份利用這些slack的機會。 此外，程式執行的cycles不一定會每次相同，所以透過執行時期的統計，就能夠依其平均值予以預測slack，做為指定下次速度的依據。要預測下一次的執行過後的$slack_{speculate}$為:

$$slack_{speculate}=slack_{early}+\frac{C_{wc}-C_{av}}{S_s}$$

這個預測成立的假設在於Task的執行cycles是較為平均的，因此，以實際執行的平均時間來說，比利用WCET要精準得多。

經過Scheduler計算過後獲得一個預測的slack，就可以將slack加入可執行的時間裡面。所以，若C為所需執行的cycle數，S_s是靜態排程所決定的速度，從上面所述已知執行的時間為$\frac{C}{S_s}$，加上預測出來的slack，那麼下一次執行的速度為:

$$S_{next}=\frac{C}{\frac{C}{S_s}+slack}$$

S_{next}被計算出來的結果，可以說是完成工作的最低要求的速度。在實際執行的情況下所花的時間為$\frac{C}{S_{next}}$，萬一最壞的狀況產生時，若真的使用S_{next}的話，那將會造成miss deadline，因此，在速度的計算上要做一些補救的措施;以執行的cycle而言為C，C_{wc}為最多需執行的cycle數，那麼我起碼要有$\frac{C_{wc}-C}{S_{\max}}$的時間用最高執行速度來做補救。因此，

$$D - t_{ac} - \frac{C}{S_{next}} \geq \frac{C_{wc} - C}{S_{\max}}$$

所以總執行時間必需保留補救的時間:

$$S_{next} \geq \frac{C}{D - t_{ac} - \frac{C_{wc} - C}{S_{\max}}} = S_{feasible}$$

將上述兩個式子加以整理，獲得最大的slack為:

$$slack \leq slack_{\max} = (D - t_{ac} - \frac{C_{wc} - C}{S_{\max}}) - \frac{C}{S_s}$$

調整執行速度的耗損

在上述排程的模型當中都忽略了調整執行速度的損耗，這損耗包含兩個部份:一是時間的損耗，二是能量的損耗。 在這一個章節當中將研究耗損會花費多長的時間和多少的能量，跟上述的模型間又有多少的影響。

2.5 電壓調節點

“電壓調節點”顧名思義就是指系統進行電源管理的時候，為改變CPU執行速度與轉換電壓時的執行點，因此，它對電源管理會產生影響，所以是我們需要討論的對象之一。“電壓調節點”基本上是由若干程式碼與各程式或Task執行時管理CPU速度或相關參數的資料所構成。 “電壓調節點”的程式有部份存在於一般的應用程式中，一些存在系統核心當中，無論調節的程式存在於系統的那部份，其功能主要是決定出CPU新的執行速度後產生系統的呼叫，然後由核心內部執行CPU速度與電壓的變更工作。因此，從系統層次規劃上”電源調節”的架構規化可以分為兩個層次:

Task-Level: 其範圍僅是利用Task本身的資訊，例如:時間上的限制和所需執行的週期。針對這些訊息決定CPU的執行速度。 在Task-Level內電壓調節的位置可以由設計者在Task內部插入相關的函式或由編譯程式做相關的處理。 所以，設計者或編譯程式可以在程式啟始的位置就執行電壓的調節。

System-Level: 其決定CPU執行速度的範圍是針對整個系統，系統將參考所有執行中Task的相關參數以決定CPU的執行速度。 例如:系統可以在下一個Task開始執行前就預先決定其執行的速度。

為了決定執行的速度，電壓調節的程式必需瞭解Task的相關參數，在許多的研究中都將此參數稱為task profile information，例如: 在時間和效率的限制下最差的執行情況與平均的執行情況以及執行時期的資料，例如:使用CPU的時間或者程式提早被結束。task profile information可以先被計算並加以儲存，但是執行時期的資料得在真正被執行的時候才有辦法搜集得到而且也許需要OS或硬體上的幫忙，才能順利的搜集到。不同執行速度管理的方法主要基於以下兩個主要觀念:第一，找出系統執行時有空閒的地方，第二，如何充份運用這些空閒的地方來達成電源管理的目地。

第三章 即時系統的工作模式

3.1 討論即時性模式

大部份的即時系統討論時都用 τ_i 做表示，i代表有i個Task，這些Task有時執行時間的限制 D_i，這是即時系統中最基本也是最重要的元素之一。若假設有一個Task在時間0的時候開始執行，則 D_i 可以被視為這個Task， τ_i 在時間上的工作區間。

C_i 是指 τ_i 需要多少的運算週期才能完成。為了便於數學模型的推導 C_i 是不會跟隨執行速度或CPU架構，例如：超純量或管線設計而改變。此外，也不討論記憶體存取時因CPU不同的執行速度而產生的變化，在此要特別提出這一點，因為記憶體的存取會比在CPU內部做存取更花費執行週期，在某些的CPU架構下，記憶體的存取的週期會因CPU速度的不同而有不同，有時CPU的速度較快，所以必需耗費較多的週期等待記憶體的資料傳輸，CPU較慢時等待的週期可能變得少些，在這裡將這個問題都假設為最長週期，來保持 C_i 的穩定性。

在此，我們將運用亂數產生若干 C_i 的值來做實驗。C_i 的單位是週期若在單位時間內則為赫茲(Hz)頻率. 目前的CPU執行的速度大都以百萬赫茲(MHz)為單位，為簡化計算與模擬，時間將以微秒為對應(micro-second， μs) 。這樣使得計算的單位趨於一致。

在Task內部，會影響執行週期的長短主要在於控制流程方面的問題，控制流程則包含了:迴圈(loop)，判斷式(IF-THEN-ELSE)或副程式等等。以迴圈來說，又區分為固定迴圈(For)和變動迴圈(條件迴圈)，迴圈的次數會影響 C_i 被執行的總數。判斷式則要視其條件和執行的分支來決定 C_i 的數量。副程式方面則是一種遞迴的想法，重複上述的說明.此外，就是一些固定會被執行到的工作。以下將就控制流程和週期性的工作予以探討。

3.2 控制流程

將Task P_i 切為 n_i 段， $\tau_{i,(j)}$，$1<=j<=n_i$ 若某一段是迴圈，呼叫副程式或其它的分支，則 P_i 的流程圖在圖(3.1)，假設每一斷 $\tau_{i,(j)}$ 都要執行 $C_{i,(j)}$ 週期。 每一段都以一個方塊做表示，每一段若是一個迴圈的話則用一個圓形和曲線表示，曲線上的數字為迴圈執行的

次數。 區段流程圖是一個完整程式的小分支，可以用個端點記號來表示。程式執行的方式必需遵循指定的方式由起點向終點執行。

在流程圖中的某一點j，$C_{WC_{i,(j)}}$ 是指j這點所需執行的週期，那麼 $C_{WC_{i,(j)}}, 1 \le j \le n_i$ ，能夠被遞迴計算為

$$C_{WC_{i,(j)}} = C_{i,(j)} + \max_{k \in B(j)} \{C_{WC_{i,(k)}}\}$$

$B_{(j)}$是指j段內的子區段。如果方塊1為程式P_i中第一個被執行的區段，則$C_i=C_{i,(1)}$帶表這一段程式最大的執行週期。

雖然已知最差的執行情況下要在時間限制內完成是即時系統必需確保的問題，P_i的執行還是比C_i小很多[28]，這是因為所輸入的資料和系統架構(例如:快取記憶體的大小)所決定。 實際的執行週期往往小於最差情況下的估計和程式內部執行路徑的不同所導致。如果 $C_{avg_{i,(j)}}$ 是指這段執行週期的平均值則它的數學式如下:

$$Avg(C_{i,(j)}) = C_{avg_{i,(j)}} + \sum_{k \in B(j)} Pb_k \cdot C_{i,(k)}$$

其中Pb_k是執行時從j段到k段的機率，所以若j段內只有一條路到B(j)那麼它的機率為:

$$\sum_{k \in B(j)} Pb_k = 1$$

3.3 週期性工作模式

即時系統大都是做一些週期性的工作，因此，假設是N個週期性的Task被EDF方式排程，若有一個工作我們定為τ_i則它的工作週期的時間我們用T_i表示，在此將對執行時的週期做一個假設，假設τ_i開始執行的時間就落在週期開始的地方， τ_i結束的地方就是週期結束的地方。在此先提到一種frame-based system，這是一種特殊週期的即時系統，就是以frame為單位，一直重覆的執行著;Frame是由一堆Task所組成，因此，在同一個Frame內的Task都具有相同的工作週期和相同的啟動時間，我們用T表示。這樣的系統安排通常是運用在研究方面，因為如此的安排比一般的即時系統更為方便進行改善和證明。

有一組Task，$\{\tau_1,...,\tau_N\}$，令 $U = \sum_{i=1}^{N} \frac{C_i}{T_i}$ 表示這組Task在最高的執行速度下則CPU使用率的總合為1，所以U必需被考慮的就是在系統中一定要保留下來的CPU使用率，那麼就能

夠得知若$U \le 1, T_i = D_i$且使用EDF方式排程，則在任何情況下，每一個Task都能被滿足其時間的限制[29]。

3.4 耗電模型

具電壓調節功能的CPU其電源消量與執行速度之間的關係為平方比。電壓為平方項，執行速度則為線性變化[14]。所以DVS之所以能夠省電就是因為CPU或其它主要的元件能夠改變工作電壓和操作頻率。所以在執行速度為S，其電源消耗的函數為g(S)。 這個g(S)的函數圖型為二次多項式的曲線[14]。若τ_i在t_1和t_2的時間內由CPU執行，那麼在CPU速度S維持不變也就是把S當常數的情形下，t_1與t_2間所消耗的能量為

$$\int_{t_1}^{t_2} g(S(t))dt \text{ 也等於} g(S)(t_2-t_1).$$

另一方面，若CPU的速度可以從S_{max}:最快的執行速度，最耗電的狀況;S_{min}最慢的執行速度與最不耗電之間做線性的變化，還有S_{idle}指CPU處於idle的狀態下其耗電會比S_{min}小，但會比g(0)大， $g(0) < S_{idel} < S_{\min}$ 。速度的管理最主要的是從導出耗電量的曲線函數g(S)中找出結果，在此特別假設τ_i被分配到t_i的時間執行，因為τ_i需要C_i個執行週期，這些週期又使用了t_i的時間，所以CPU的速度為$S_i = \frac{C_i}{t_i}$ 。

因為，耗電函數是曲線函數g則為:

$$g(S_i)t_i \le s(S')x + g(S'')(t_i - x)$$

X代表CPU速度不同時執行時間， S'與S''是兩種不同的速度$(S' \neq S'')$，所以$S'x + S''(t_i - x) = C_i$代表在t_i的時間內使用不同的執行速度，比上固定執行速度下所能節省的電源。值得注意的是:前面的狀況是單一Task且執行時間與時間限制相同($t_i=D_i$)，若是不只一個Task的情況下速度分配的情況可能無法符合每一個Task的時間限制。

第四章 電壓排程的耗損

“電壓排程”望文生義就是以使用電壓的高低做為排程的依據，然而電壓轉換時會產生能量的耗損與時間的延遲。

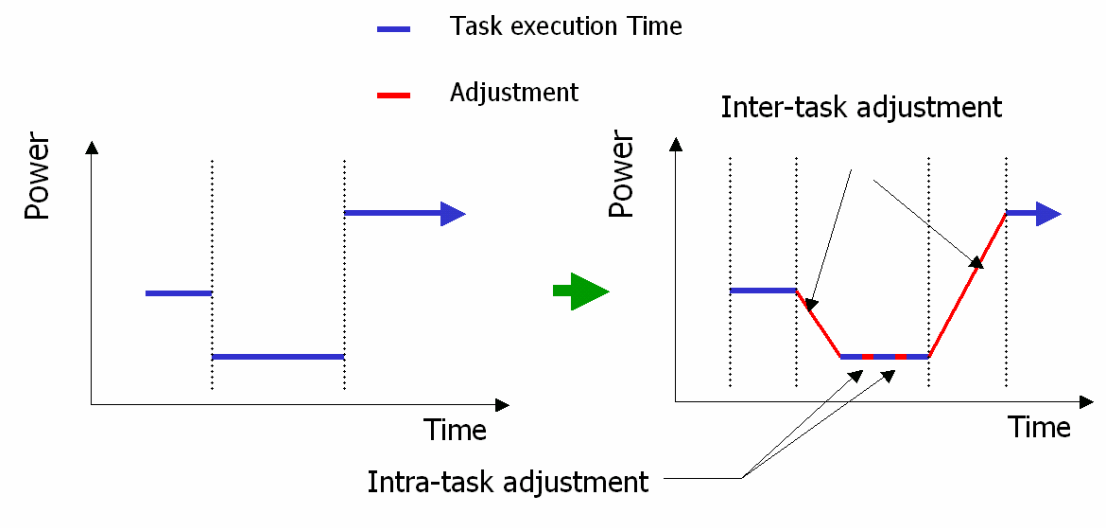


圖 4.1 Task 時序圖

如圖4.1所示，過去的討論大都基於左邊的方法，雖然合理但卻不切實際，上圖右邊是較趨向實際的情況。 在Task轉換的過程中若是需要調整電壓或頻率，那麼在時間軸上兩Task間是不連續的，這就是本研究中所指出的耗損。

那麼，既然耗損存在，是否有方法能夠減少這方面的耗損或是降低這耗損對系統的影響? 答案是肯定的。因此，本研究將解決以下的問題:

在能量的耗損上:減少電壓調整所發生的次數，但是不能損及原來省電方面的架構。在時間的延遲上不能損及即時系統的時間限制。

調整執行速度的耗損，在上述排程的模型當中都忽略了調整執行速度的損耗，這損耗包含兩個部份:一是時間的損耗，二是能量的損耗。 在這一個章節當中將研究耗損會花費多長的時間和多少的能量，跟上述的模型間又有多少的影響。

4.1 時間的耗損

直覺上時間的耗損主要分為兩個方面:

□ 軟體方面 耗於計算出新的合理速度

□ 硬體方面 耗於電路本身的延遲，DC-to-DC的轉換延遲與時脈產生器所產生的延遲

因為在即時系統中，保持工作的即時性是最重要的。因此，任何會影響即時性的因素都必須被謹慎的考慮。

當考慮改變CPU的電壓或執行速度時，C_i和C_{avg_i}，都必需將這些耗損的時間加進來，以Intel SA1110的CPU[30]改變執行的速度將耗費200us[31]，在59Mhz的工作頻率下，大約需要11000個週期，在最高速221.2Mhz時約需45400個週期。 此外，改變電壓則需更長的時間，因為可變式的電壓源大都由DC-to-DC的轉換器供應，這轉換器無法立刻的針對所需的電壓做出立即的反應，通常需要一小段的時間，對於人類而言，轉換的時間是微不足道的，但是在即時系統中，這轉換的時間是無法被忽略的。而且，轉換過程中CPU通常無法執行指令，怕會有不穩態產生。 因此，有時得視需要插入一些延遲或無動作的指令。 以Intel SA1110為例，改變電壓將耗費250us的時間， 1pARM CPU[32]（低電壓ARM8架構的CPU)從10Mhz的工作頻率升至100Mhz需15us。 另外的例子則為:全美達(Transmeta) TM5400是特別依DVS[33]所設計的CPU，它是以每階33MHZ的頻率調整，每一階的時間是相同的，因此，調整時需視情況累加其轉換時間，另外有些CPU在改變操作電壓和操作頻率的同時能夠執行指令[32，14]，但是在轉換的時間內頻率是變動的在這轉換的週期內需要特別注意，以免有錯誤發生。 為使模型趨於單純，因此，本文將以轉換週期時無法執行指令為討論的前題。

如此，以SA1110為例:改變頻率的損耗時間為200us，改變頻率為250us，這些時間與執行速度無關是固定的。所以以F_o代表改變頻率的耗損，V_o代表改變電壓的耗損，因此，以時間而言slack必需大於(F_o+V_o)才值得轉換，此外，必需具備下列條件並令C_o為軟體的耗損才能符合即時系統的需要:

$$\frac{C + C_o}{(D - (F_o + V_o)) \cdot S_{current}} \le 1$$

4.2 能量耗損

藉由上述的模型推導，以下將開始推導能對應的能量耗損:

由VLSI相關的文獻中指出，降低電壓後意味著能量的下降，因此運作的頻率因應能量的改變，因此必需與以降低。 可由下列式子表示:

$$g_d = C_L \frac{V}{(V - V_t)^2}$$

上述式子為單一個gate的延遲性與電壓的關係。C_L為負載電容(load capacitance)，V為工作電壓，V_t為工作電壓的下限。

另外，由於操作頻率與電壓基本上也呈一個對應的關係[34]，所以能量也與頻率成平方比($E \propto f^2$)，因此跟據g_d的公式，頻率應該如下面式子:

$$f = \frac{k(V - V_t)^2}{V}$$

k為電路常數[35]。 能量與頻率的關係則為:

$$E \propto (V_t + \frac{f}{2k} + \sqrt{\frac{V_t f}{k} + (\frac{f}{2k})^2})^2$$

因此，若是較複雜的電路由n個gate所構成，那麼在V_w的電壓下的延遲則為:

$$D_g = \sum_{i=1}^{n} g_{d_i} = \frac{V_w}{(V_w - V_t)^2} \sum_{i=1}^{n} C_{L_i} = \frac{C_{L_{SUM}} \cdot V_w}{(V_w - V_t)^2}$$

$C_{L_{SUM}}$為負載電容的總合。

此時它的功率消耗為:

$$P_w = C_{L_{SUM}} \cdot f \cdot V_w^2$$

f為工作頻率。由此式可看出，功率的耗損是與電壓呈二次比與頻率為一次比，因此，降低電壓比降低頻率要能夠節省更多的能源。若此，CPU每次執行一個指令就需要經過n個gate，那麼所花費的功率也就為P_w。

因此就產生了一個功率的函數P，由功率函數計算消耗能量有以下的關係:

$$E = \int_{T_1}^{T_2} P(t)\, dt$$

T_1，T_2為時間區間。 $t \in \{T_1 ... T_2\}, 0 < T_1 < T_2$則上述則可計算出$T_1$到$T_2$間的能量。

所以將上述時間與頻率的參數代入如下:

$$E_i = \int_{T_1}^{T_2} P(\frac{C_i}{S})dt = \int_{a_i}^{d_i} P(\frac{C_i}{S})dt$$

則為一個Task執行指定工作時所需的能量，a_i是Task的arrived time， d_i為deadline，S為操作頻率。

軟體耗損的能量則為:若處理DVS程式為C_{so}個cycles，那麼軟體的耗損為:

$$E_{so} = \int_0^{\frac{C_{so}}{S}} P(\frac{C_{so}}{S})dt$$

硬體的耗損可分為調整頻率和調整電壓，時間分別為F_{to}及V_{to}:

頻率調整的能量為:

$$E_F = \int_0^{F_{to}} P(F_{to})dt$$

電壓調整的能量為:

$$E_V = \int_0^{V_{to}} P(V_{to})dt$$

那麼，完整的Task所需的能量如下:

$$E_{T_i} = E_i + E_{so} + E_F + E_V$$

執行的時間為:

$$T_{total_i} = \frac{C_i}{S_i} + \frac{C_{so}}{S_i} + F_{to} + V_{to}$$

且必需

$$T_{total_i} \le T_{deadline_i} - T_{current}$$

否則會無法排程

此時，$Task_i$的$slack_i$為:

$$slack_i = T_{deadline_i} - (T_{current} + T_{total_i})$$

$slack_i$在目前的工作情況下的消耗能量為

$$E_{slack_i} = \int_0^{T_{slack_i}} P(T_{slack_i})dt$$

，

4.3 電壓調整策略

過去的電壓調整策略，大都是依CPU和Workload的使用率直接來做調整並沒有考慮耗損時，那麼很容易造成無法排程或miss-deadline。

使用即時系統的目的還是在利用和確保”即時”的工作性質，因此任何排程策略的產生首先必需確保Miss-deadline不會產生。因此，本研究提出結合電壓調整之耗損的排程策略將從即時性質開始討論。

4.3.1 Miss-deadline

Miss-deadline的原因主要是沒有足夠的時間處理調整時的耗損，因此Task在面臨調整時需要將調整的時間計算進去，但是如此一來，似乎只要將WCET的時間加上轉換的時間就萬無一失了?但是，增加WCET將意味著需要更高階的CPU才能保證這所有的工作都能準時完成，然而CPU效能的提升意味著CPU實際的利用率可能會降低，如此的考慮方法太過於消極。因此，在本研究中提出有效的策略來保證即時性質，並且比將調整的延遲加入WCET內要能夠減少為做電壓調整所發生的耗損。

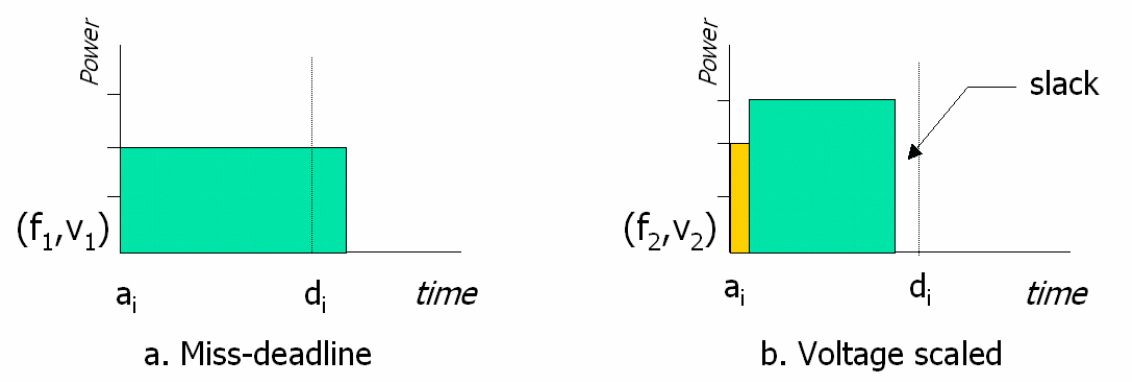


圖 4.2 Voltage scaling up

計算 $\alpha_i = \frac{C_i \cdot S_{current}}{D_i - t_{current}}$ ， $a_i \le t_{current}$

若 α_i>1，則需做升頻與升壓的動作。因此需重新計算升頻與升壓的時間延遲的結果是否仍然滿足工作的執行所需:

$$\alpha_i = \frac{(C_i + C_o)\cdot S_{new}}{D_i - (F_o + V_o) - t_{current}} \quad , \quad S_{new} \le S_{\max}$$

如果α_i>1，則無法排序，若是α_i ≤1則可以繼續執行，基本上scheduler必須提供足夠的運算能力以確保即時系統的運行。

相反的，若是現在的運算能力足以應付現行的工作所需。那麼在降壓降頻後新的操作頻率仍然需要足夠應付工作所需否則將得不償失。因此，下列式子必須成立，才需做實際調降的工作:

$slack_i \ge Overhead_i$ ，又 $Overhead_i = \frac{C_o}{S_{current}} + F_o + V_o$ ， $slack_i = D_i - \frac{C_i}{S_{new}} - t_{current}$

所以 $D_i - \frac{C_i}{S_{new}} - t_{current} \ge \frac{C_o}{S_{current}} + F_o + V_o$ ， 此情況如下圖4.3所示，圖a.說明了workload和slack已經超過需要太多，圖b.則表示調降的結果。

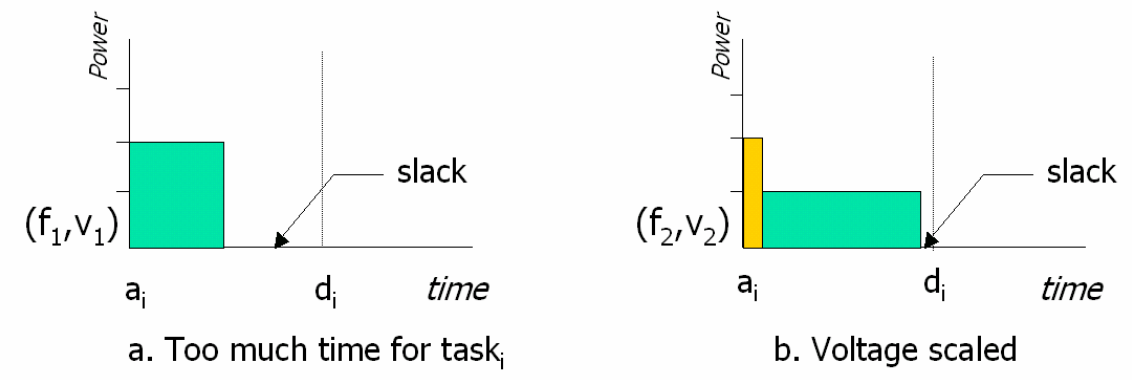


圖 4.3 Voltage Scaling down

此外，還有若干的狀況需要進一步考慮調降的工作。 若是調降後會產生miss-deadline則無法執行scaling down的動作，圖4.4說明了這樣的情況。

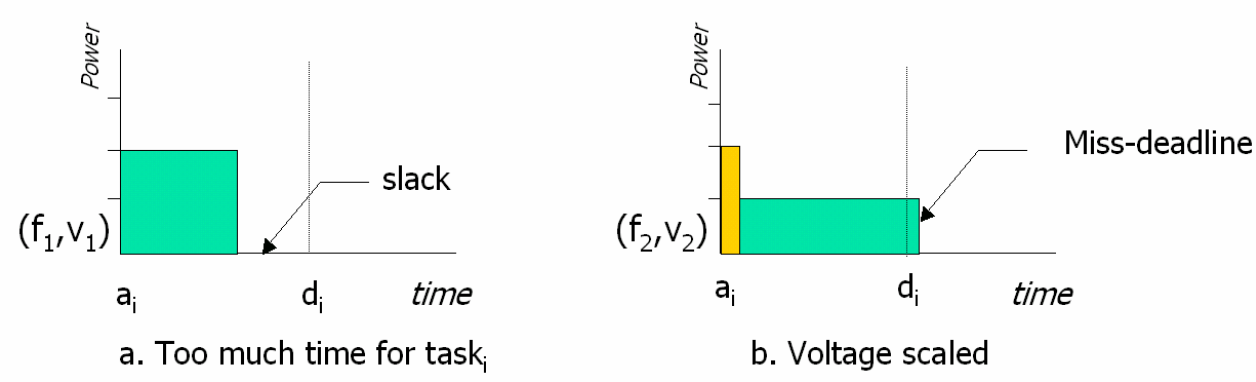


圖 4.4 Voltage scaling down but miss deadline.

上述說明了slack必需足夠才能做電壓的調整。此外，考慮一種排程的狀況，若Task的workload小於轉換的耗損，那麼不斷的做轉換將導致更多的損耗。

證明如下:

假設有N個Task{1..N}，Task{1，3，5，…，N}的WCET使用的工作電壓為v_1， Task{2，4，6，8…，N-1}的WCET使用v_2且 $D_{i-1} = a_i$ ，暫不考慮idle的情況。

令 $T_{ac_i} < Overhead_i$，$V_2 > V_1, S_{v_1} = S_{v_2} \cdot 80\%$ 則使用本研究所提的演算法和直接使用DVS的差別如下:

$$E_{DVS} = \frac{N}{2} E_1 + \frac{N}{2} E_2 + N \cdot E_{overhead}$$

$$E = \sum_{i=1}^{N/2} (\int_{a_i}^{D_i} P(\frac{C_i}{S_{v_1}}) dt + \int_{a_{i+1}}^{D_{i+1}} P(\frac{C_{i+1}}{S_{v_2}}) dt) + \sum_{i=1}^{N} P(Overhead_i)$$

$$\because V_2 : V_1 = 1 : 0.8, \therefore E_2 : E_1 = 1 : 0.64$$

$$E_{DVS} = \frac{82 \cdot N}{100} E_2 + N \cdot E_{overhead}$$

$E_{DVS\_O} = N \cdot E_2$ ， 根據上述的轉換策略所決定的演算法此類Job將不會做電壓調整。

因此，當原式所定義:

$E_{overhead} > E_{V_2}$

所以， $E_{dvs} > E_{dvs\_o}$

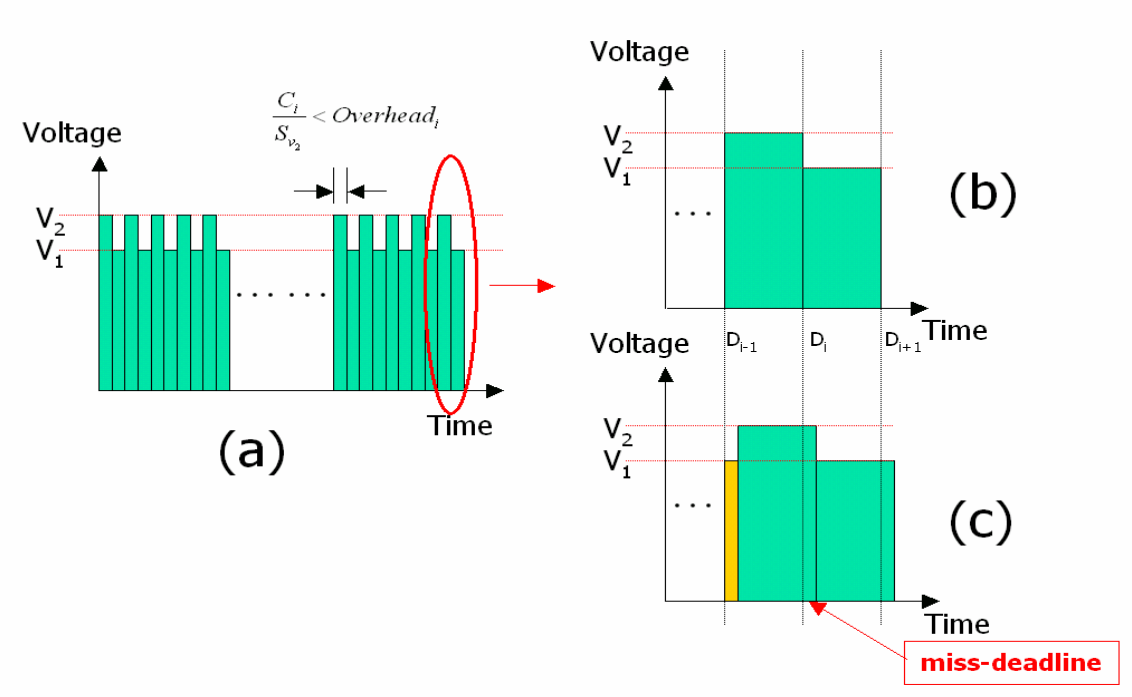


圖 4.5 工作量小於電壓調節的耗損.

藉由圖4.5 (a)說明了，電壓的調節必須視其工作量的大小做判斷，也就是投資報酬率的觀念。在這樣的定義當中暫時忽略miss-deadline的情況發生，單就能量的損耗而言做計算，換言之;若是調降後所節省下來的能量小於電壓調節時的能量消耗，尤其工作量又小於電壓調節時的損耗，那麼反而會導致額外的浪費。 此外，在圖4.ZZ的(b)及(c)又導出miss-deadline的狀況其中若 $V_2 = V_{\max}, S_{v_2} = S_{\max}$ 則會發生miss-deadline，當然這樣的假設過於極端，然而確可借此突顯出電壓調節造成延遲的重要性。誠如前文所撰，將電壓調節的延遲時間加入WCET內就不會產生miss-deadline了，當然，這是解決的方法之一，可是WCET的增加勢必造成為處理所有極端的情況下必須採用更高階的CPU，如此一來對系統設計是不利的且因CPU運算能力的升級，有可能會更耗電。 因此，為deadline的控制並充份利用slack和idle的時間做電壓調整的動作。將擬定另外的策略。基本上，Scheduler若計算出現行的運算量是CPU目前的運算能力無法負擔的，那麼將無條件加以調整提昇電壓和頻率。 以符合需要且確保工作的即時性。但是，操作電壓和頻率的下降則有較多的考慮，例如:若是工作量在較低的頻率下雖可運算結束，但是加上電壓調節的時間延遲，則有可能會超過其限制的時間這麼做將得不償失，沒有必要。此時，slack將會產生，可以將slack搜集起來並視情況再利用 。

圖 4.6 說明著這樣的關係:

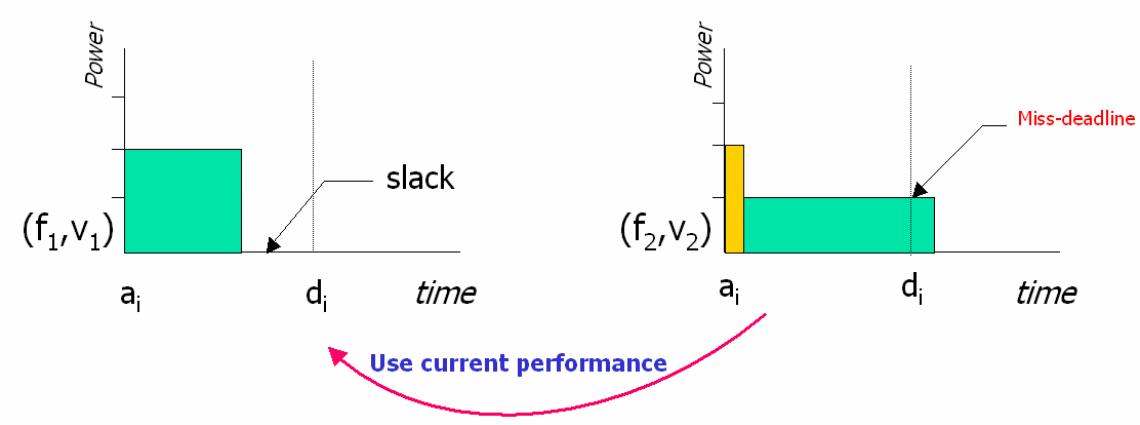


圖 4.6 保持原來的操作參數

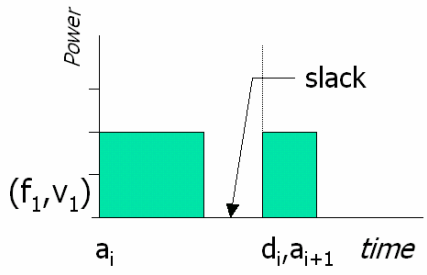
因此，下面將針對一些task排列的狀況，來增加slack和idle的重新利用。

4.3.2 slacks和idle的利用與搜集

首先對slack的定義再說明:slack是指task實際完成的時間到WCET之間的時間。

而idle則是兩task之間的空閒時間。 在某些情況下slack也可以視作idle。

下面以數學式與圖型說明:



$$slack_i = d_i - t_{ac} - a_i - overhead_i$$

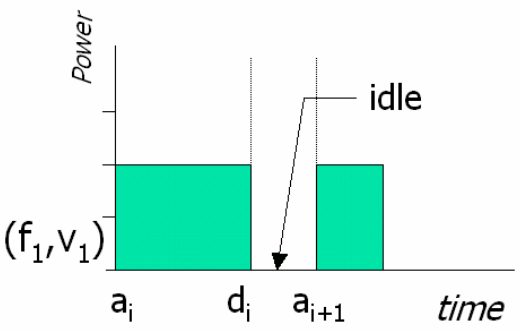
$$= d_i - \frac{C_i}{S_i} - a_i - (\frac{C_o}{S_{i-1}} + V_o + F_o)$$

圖 4.7 slack

上圖為突顯slack與idle的時間在觀念上的不同，因此task$_i$與task$_{i+1}$之間是緊緊相連的，假設task$_i$的deadline與task$_{i+1}$的arrive time是相同的。 然而slack會依task執行的狀況而有所不同，在第三章中已討論過它的機率模型，在此將不再贅述。

此外在Inter-Task的演算法中，slack的再利用基本上是給下個task使用的，因此，意味著若要執行下一個Task將會有調節的機會。所以，需要調節與否將視所剩餘的時間是否足夠轉換的耗損所需。

至於idle方面，可由圖 4.8做清楚的認知與定義。 Idle是指OS目前並沒有任何task在執行，CPU是處於閒置的狀態。

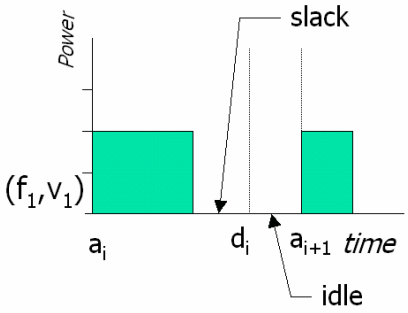


$$idle = a_{i+1} - d_i, a_{i+1} > d_i$$

圖 4.8 idle

圖 4.8 的idle為兩個task在時間上並沒有任何的重疊，這段沒有工作的時間就是idle time，目前較常見的CPU幾乎都有專門提供idle的工作模式，idle的工作模式下，CPU是處於較省電的狀態。 因此，能夠獲得較長的idle模式也是電源管理的重點之一。而Idle與slack不同，idle發生時當然可以利用idle的時間將電壓和操作頻率降到最低。但是在下一個task來到之後將再需要一次的調節，所以在idle時間做調整將需要兩次調節的時間。

其次，slack和idle也有混合的情況出現。



$$slack_i = d_i - t_{ac} - a_i - overhead_i$$
$$= d_i - \frac{C_i}{S_i} - a_i - (\frac{C_o}{S_{i-1}} + V_o + F_o)$$
$$idle_{i,i+1} = a_{i+1} - d_i, a_{i+1} > d_i$$
$$idle_{total} = slack_i + idle_{i,i+1}$$

圖 4.9 Idle 與 slack 的混合狀態

圖4.9 是一種混合的狀態，結合著slack與真正的idle時間而成。 有了上述這三種空閒的情況。因此，電壓調整的策略可以充份的利用這些空閒的時間做調整且降低或不會對工作造成即時性的危害。

首先，只有slack的情況，由於slack的發生和時間的長度是具機率性質。因此，運用方面先視它的長度及下一task的工作需要做調整。所以，有下面幾種情況:

slack>overhead，且下一工作的需求是升壓與升頻。那麼scheduler可以將CPU進入idle模式，直到a_{i+1}-overhead的時間後開始升頻，如圖4.10(a)所示。

slack>overhead，且下一工作的需求是降壓與降頻。那麼scheduler可以將先降壓降頻然後再進入idle模式，如圖4.10(b)所示。

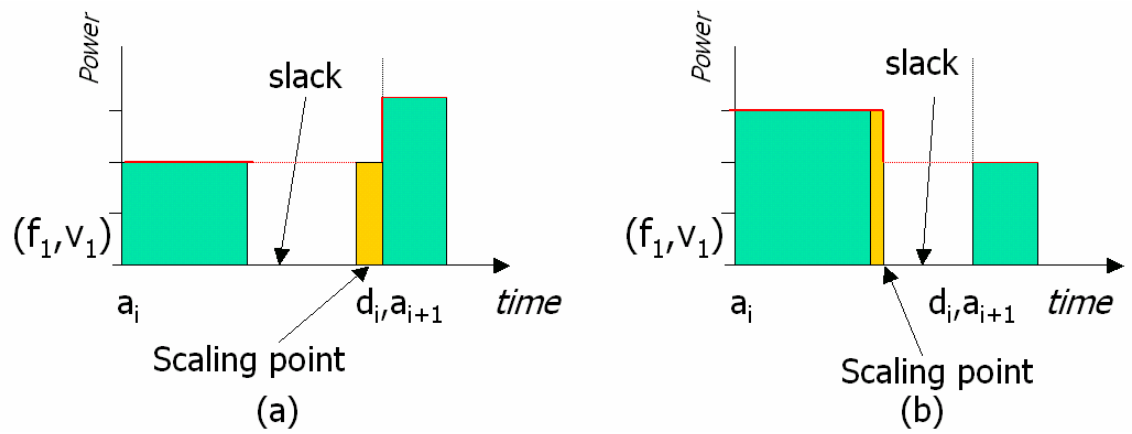


圖 4.10 利用 slack 做升降壓

slack<overhead， 且下一工作為升壓與升頻。 那麼scheduler將立刻升壓升頻。

slack<overhead，且下一工作為降壓與降頻。那麼還要計算調節佔用多少的工作時間，並且如果仍需降頻則立刻降頻，若是無法降頻則使用原來的狀態進入idle模式。此外，若下一個task的WCET小於overhead那麼也將保持原來的狀態。以免得不嘗失。

Idle時間方面，若task queue內已有task待處理，基本上考慮的方式與slack是一樣的考慮方式。 若是task queue內沒有task在等待處理那麼只能直接進入idle模式，然後視需要再加以調整。

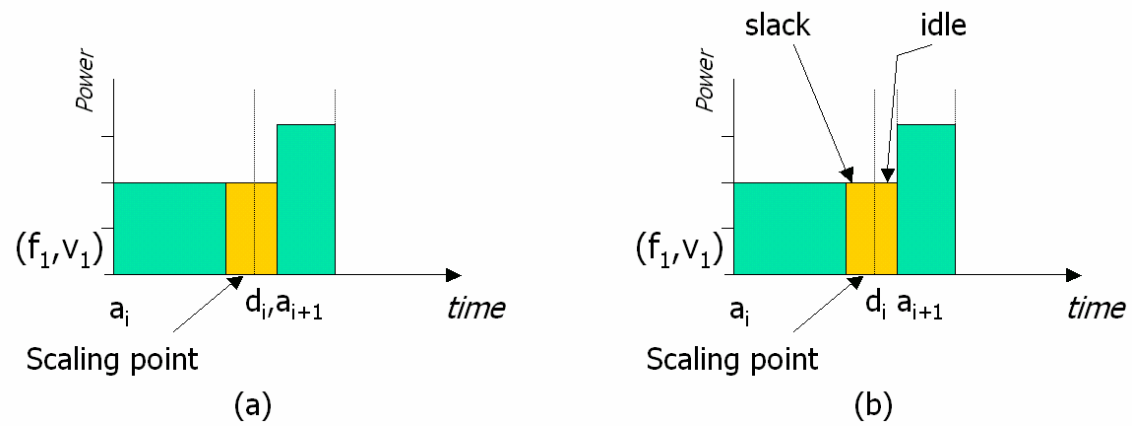


圖 4.11 運用 idle 與 slack 做調整

圖4.11 (a)說明了idle或 slack時間不足時會佔用到task的執行時間。 圖4.11(b)則說明了slack和idle的運用。

基於上述的情況可知，動態排程和靜態排程最大的不同就是對予slack的預測，在本報告中提出一個方法，其方法的基本概念是從「應用導向」來增加slack預測的正確性。「應用導向」是指應用程式在執行時，基本上都有一定的脈絡可循。例如:一個MP3的撥放程式其動作大體上為讀取MP3檔案內容、解碼、撥放等一直重覆的動作。
因此，在動態排程時最主要的一個限制就是下一個工作不知何時會產生。因此，如果能夠知道下一個工作或完整的工作串列，那麼處理動態排程將跟靜態排程一樣的準確。
在本報告中以雙向指標的資料結構利用context switch時將指標分別指向其上下TASK，以建立上下游的關係，如此一來當下一回合又進入此TASK時，將能夠知道下一個工作的參數，以增加slack預測的準確性。

4.3.3 演算法

```
PowerTransitionAwareScheduler(TaskQueue T)
α=workload by current operate speed
if α > 1 then
  idle_time=arrive_time-current_time
  if idle_time ≥ overhead then   // use idle_time for power transition
    idle_time=idle_time-overhead
    α=re-calculate workload with higher operate speed
    if α > 1 then fail(unable to schedule)
    set_idle(idle_time) // enter idle state and wakeup when idle_time reached
  else
    α=re-calculate work with new speed and partial or full ovrhead
    if α > 1 then fail(unable to schedule)
    voltage_scale_up(α)
    freq_scale_up(α)
  endif
else if alpha <1 then
   idle_time=arrive_time-current_time
   if (deadline-arrive time) ≤ overhead then
     if idle_time>0 then set_idle(idle_time) and keep current operate speed
   endif
   if idle_time ≥ overhead then
      α=re-calcuate workload with lower operate speed
      if α >1 then keep current operate speed
      else freq_scale_down(α)
           voltage_scale_down(α)
           idle_time=idle_time-overhead
      endif
      set_idle(idle_time)
   else
      α=re-calcuate workload with lower operate speed and partial or full overhead
      if α > 1 then keep current operate speed
      else freq_scale_down(α)
           voltage_scale_down(α)
      endif
   endif
endif

   perform(task)
end.
```

第五章 實驗

本研究使用軟體模擬驗證，軟體模擬的環境參數主要分為兩方面，第一，為平台的參數。第二則為測試樣本。 因為軟體模擬就是為了縮短裡論與實際方面的距離，若沒有採用較實際的工作參數。那麼，所獲得的資料將降低其參考的價值。 同樣的，測試樣本的涵蓋面需要儘可能的廣泛，才能得到較為完整的結果。

5.1 模擬平台的參數

為驗證本報告在第四章所提的方法將採用軟體模擬。模擬參數的樣本來自於Intel公司所出品的SA1110的CPU的32位元整合型CPU，本程式所取的參數表如表5.1，其中CPU的核心電壓為1.47V至2.10V，外部I/O的電壓為3.3V，操作頻率分為12階，範圍由59MHZ~221.2MHZ。

Parameter	AC, AD (133 MHz)	BC, BD (206 MHz)
Maximum Run Mode Power	500 mW	1000mW
Typical Run Mode Power[1]	200 mW	350 mW
Maximum Idle Mode Power[2]	100 mW	200 mW
Typical Idle Mode Power[2]	75 mW	100 mW
Maximum Sleep Mode Current[2]	75 u A	75 u A
Typical Sleep Mode Current[2]	50 u A	50 uA
Vddi Max	1.63 V	2.10 V
Vddi Typ	1.55 V	1.75 V
Vddi Min	1.47 V	1.65 V
Vddx Max	3.60 V	3.60 V
Vddx Typ	3.30 V	3.30 V
Vddx Min	3.00 V	3.00 V

NOTES:
1. Typical operation defined using the following parameters:
   320x240 LCD operating at 70 fps (passive color LCD,
   8-bit color depth, single panel (1 DMA unit); and
   UART3 transmitting and receiving 115.2 kbps (using 2 DMA units).
2. Room Temperature

CCF[4:0]	Core Clock Frequency in MHz	
	3.6864-MHz Crystal Oscillator	3.5795-MHz Crystal Oscillator
00000	59.0	57.3
00001	73.7	71.6
00010	88.5	85.9
00011	103.2	100.2
00100	118.0	114.5
00101	132.7	128.9
00110	147.5	143.2
00111	162.2	157.5
01000	176.9	171.8
01001	191.7	186.1
01010	206.4	200.5
01011	221.2	214.8
01100– 11111	Not supported.	—

* 擷自 SA1110 Developer's Manual， Oct 2001

表5.1 SA1110參數表

為方便計算，模擬程式內的時間刻度為1μs，如此一來可以將頻率與時間的關係簡化。

5.2 測試樣本

驗證本研究的Task set的特徵參數，包含了Tasks的數量，執行時間的分配和WCPU(Worst Case Processor Utilization)。資料結構如下:

```
struct TASK {
        pID         : unsigned long; // Task ID
        aTIME       : unsigned long; // arrived TIME
        dTIME       : unsigned long; // deadline
        real_cycles : unsigned long; // Workload by cycles
        max_cycles  : unsigned long; // Worst Case Execution cycles
        min_cycles  : unsigned long; // Best Case Execution cycles
        uplink[5]   : struct TASK;   // Tasks UpLink
        downlink[5] : struct TASK;   // Tasks downLink
}
```

此外，為了自動產生一些特別規格的Task Set，所以程式內部也具備了產生Task Set的功能，以減少輸入的時間提升模擬的效率。 以下，是相關輸入的參數:

```
struct _TASK_SET_{
        N      : unsigned long; // Number of Tasks
        Period : unsigned long; // Range of period
        aTIME  : unsigned long; // arrive time
        dTIME  : unsigned long; // deadline
        cycles : unsigned long; // Workload by cycles
}
```

這Task Set的產生程式只是產生Task，並不會依不同特性的排程演算法做排列，因此，若選用EDF排程，則需另外依deadline做排序，若RMS則需要利用period做排序。

因此，只需要將Task的規格寫成一文字檔， 輸入之後將自動產生具相關規格和特性的Task Set。

下面的列表是本研究的工作樣本。所產生不同的CPU Utilization的資料將置於附錄A。

Utilization	WCET	Execution Time	Cycles	Period
0.25	450	45	9954	450
	2250	168.75	62213	2250
	3375	253.125	93319	3375
0.50	2250	281.25	62213	2250
	3375	421.875	93319	3375

	900	225	49770	900
0.75	450	168.75	37328	450
	2250	168.75	62213	2250
	3375	1012.5	93319	3375
1.00	450	112.5	24885	450
	2250	1125	62213	2250
	3375	843.75	93319	3375

表5.2 Task Sets parameters

Utilization的值為，所有Task Set的執行時間與WCET比率的總合，雖然在設計樣本的同時就需要考慮，但在程式內部仍然有檢查的機置來避免這種沒有辦法排程的情況產生。

5.3 模擬程式

模擬程式主要分成三個部份:

Slack Estimation：slack預測的機置，此機置可以與各種不同的排程演算法互相搭配。以取得slack的長短做有效的運用。

Return of Investment Control ： 過濾掉投資報酬率較差的轉換。 例如:slack不夠長的，Task執行時間太短的Task。都不應轉換以免因轉換而造成更大的耗損。

Power Control ： 電壓和頻率控制的機制，此為配合第四章的轉換策略。

執行部份 ： 因為所需執行的週期是虛擬的，所以主要是用來計算消耗的能量和時間，此外，利用參數設定程式週期的Low Bound，以亂數產生出介於Low Bound到WCET間需要執行的週期數，以產生不同的slack。

能量與時間週期控制單元 ： 用以記錄耗費的能量與經過時間，並加以記錄且將結果轉換為Excel檔案格式，以方便分析與統計。

架構圖:

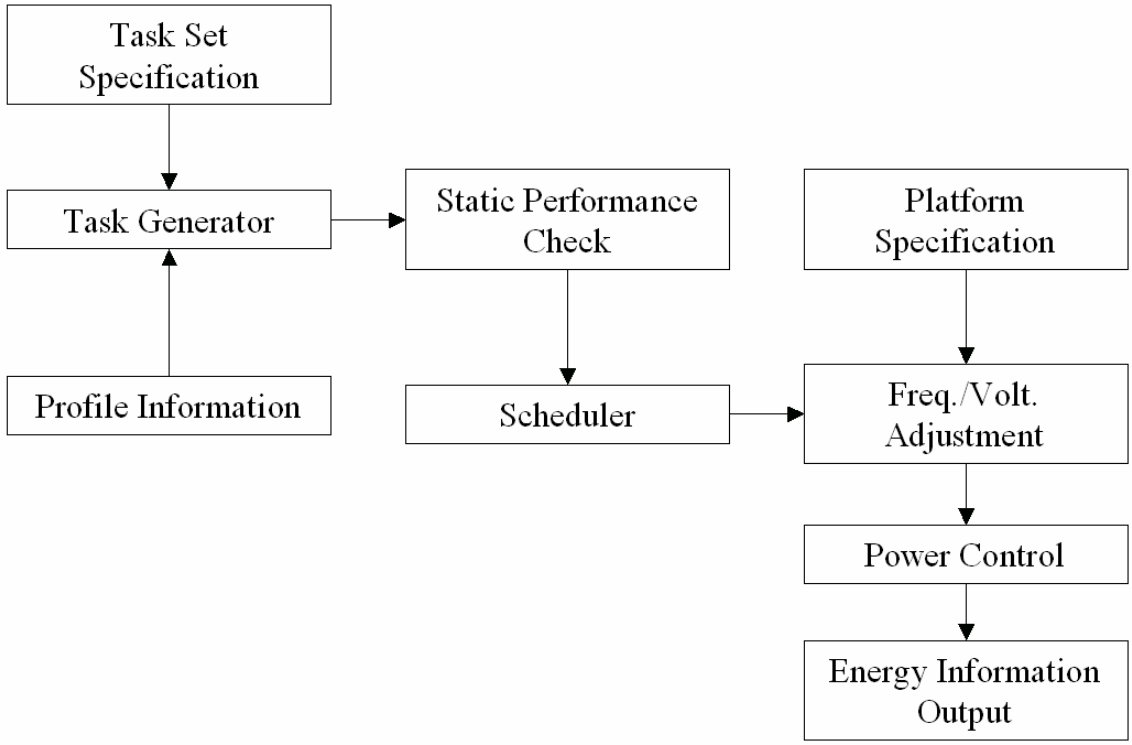

Task Set Specification
Task Generator
Profile Information
Static Performance Check
Scheduler
Platform Specification
Freq./Volt. Adjustment
Power Control
Energy Information Output

圖 5.1 模擬程式架構圖

5.4 模擬計劃

模擬的情況和條件以off-line為主，首先必須設定對照組，本研究的對照組如下:

1. No DVS
2. Statically-Scaled EDF
3. Look-Ahead DVS
4. Statically-Scaled EDF + Our proposed scheme
5. Look-Ahead DVS + Our proposed scheme

模擬的目地:

1. 比較各種演算法在相同的Task Set的Power Transition個數
2. 比較能量消耗的情形
3. 比較即時性質

環境設定:

1. Power Transition個數

   輸入一數量的Task，在不同的CPU Utilization下，範圍從10%至100%間取得轉換個數的數據。 如此，可以計算出能量與時間耗損的純量也可看出轉換耗損佔用所用資源的比率

2. 能量消耗

   給予四組不同的slack比率(0%,25%,50%,75%)並在不同的CPU利用率下10%~100%藉以模擬各種不同情況的能量消耗。

3. 比較即時性

   當Power transition對於Task的比率增加時，系統的效率會變差，一但系統的效率不好將損及即時性質。因此將Task所對Power transition的比率從0.25一直提到30倍，也就是說Task所佔的時間是Power Transition的30倍至0.25倍，如此可以看出系統的耗電情況和所佔用的時間。

第六章 分 析與討論

依據模擬計劃的說明，針對Power Transition個數的情況如下:

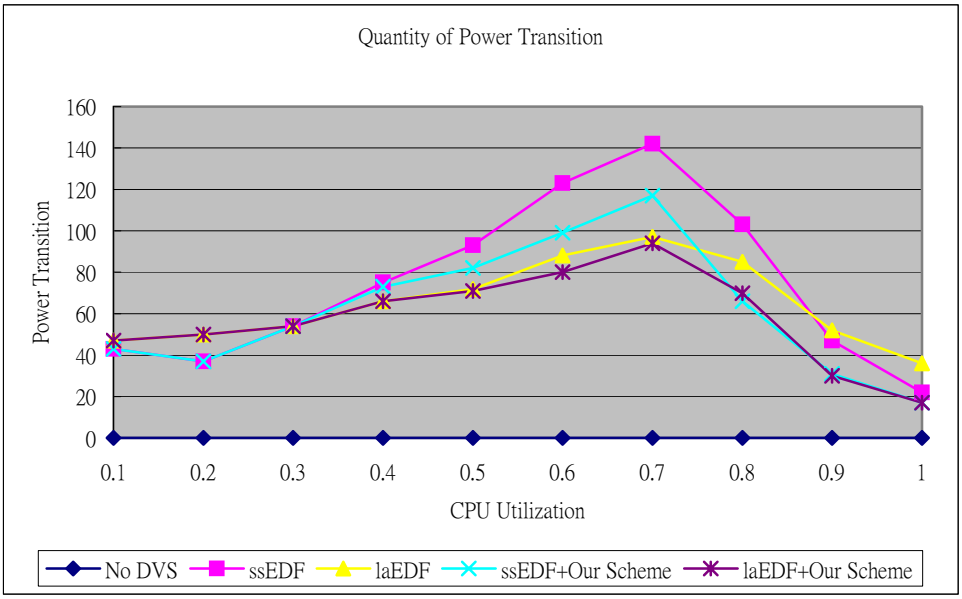


圖 6.1 Quantity of Power Transition

圖6.1呈現了Power Transition分佈的狀況，其高峰部份大約是在CPU利用率的60%~80%之間，這樣的趨勢說明了Power Transition的轉換耗損在使用率大約70%左右達到最高峰，在圖的兩側CPU是處於較低工作量或高工作量反而是比較穩定的。 在本圖所示在一般的狀態下本研究所提出的方法可以降低轉換次數約30%左右。 因此，意味著在轉換上的耗損也能夠減少30%。

除此之外，實際工作的過程請參考下面圖6.2工作圖，這是未經過本報告所提方法調整的，在80%的CPU使用率下，40個Task的執行情況共發生11次的調整。 圖6.BB為經過本研究調整的情況，只發生兩次的調整。 但是消耗能量的總合並沒有增加。雖然有時需要使用較高的電壓來唯持即時性，但是因減少花費於轉換的耗損，所以整體的能量消耗反而比較低。

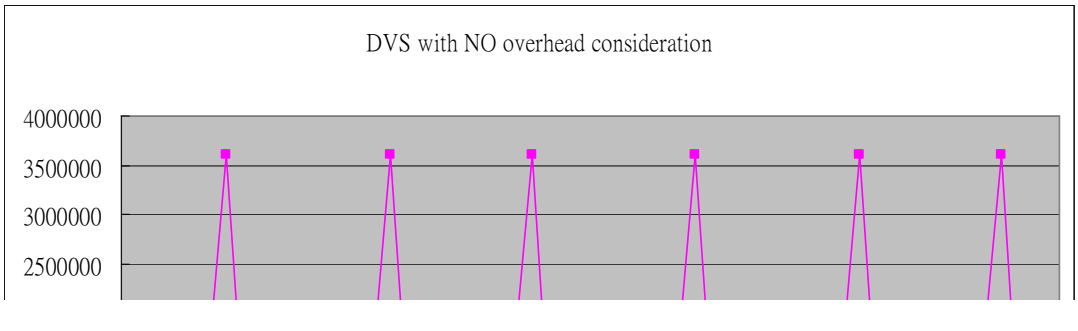


圖 6.2 ssEDF

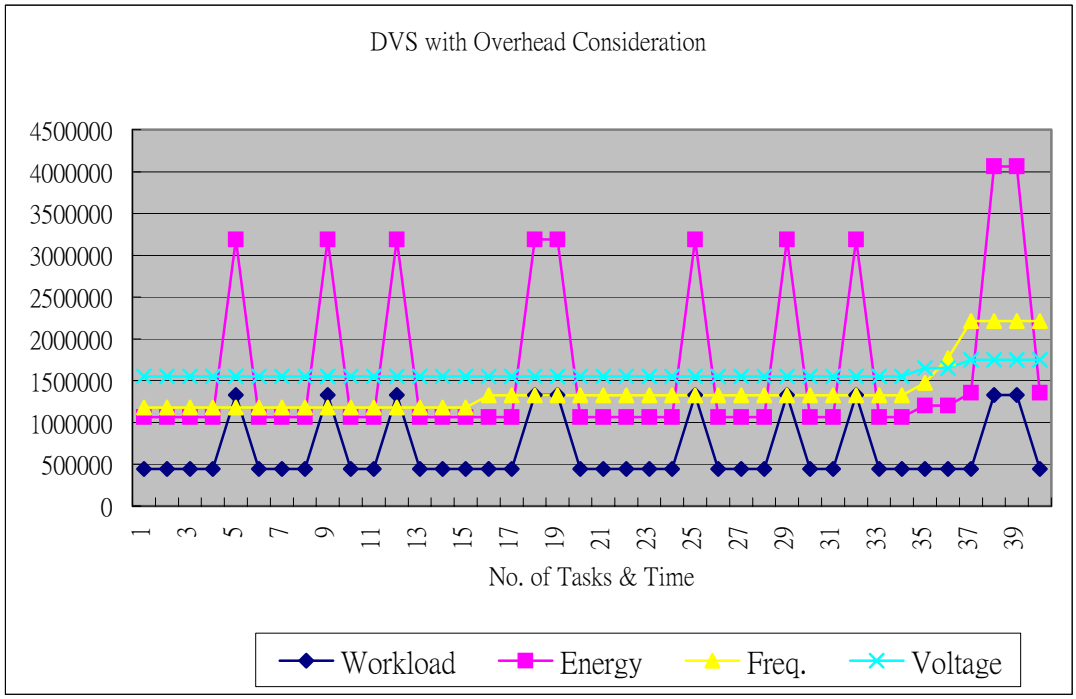


圖 6.3 ssEDF + Our Scheme

在能量比較方面，底下圖6.4到圖6.7為不同的工作量下的能源消耗的比較，CPU Utilization是指每個工作量的upper bound，也就是說每個Task只執行CPU Utilization的所對應的週期數，其餘的就是slack。例如：某個TASK的執行時間為10秒，若其Deadline為100秒，則CPU例用率為0.1，若是TASK實際執行時間為2.5秒，則其佔原WCET的25%。其它的以此類推。

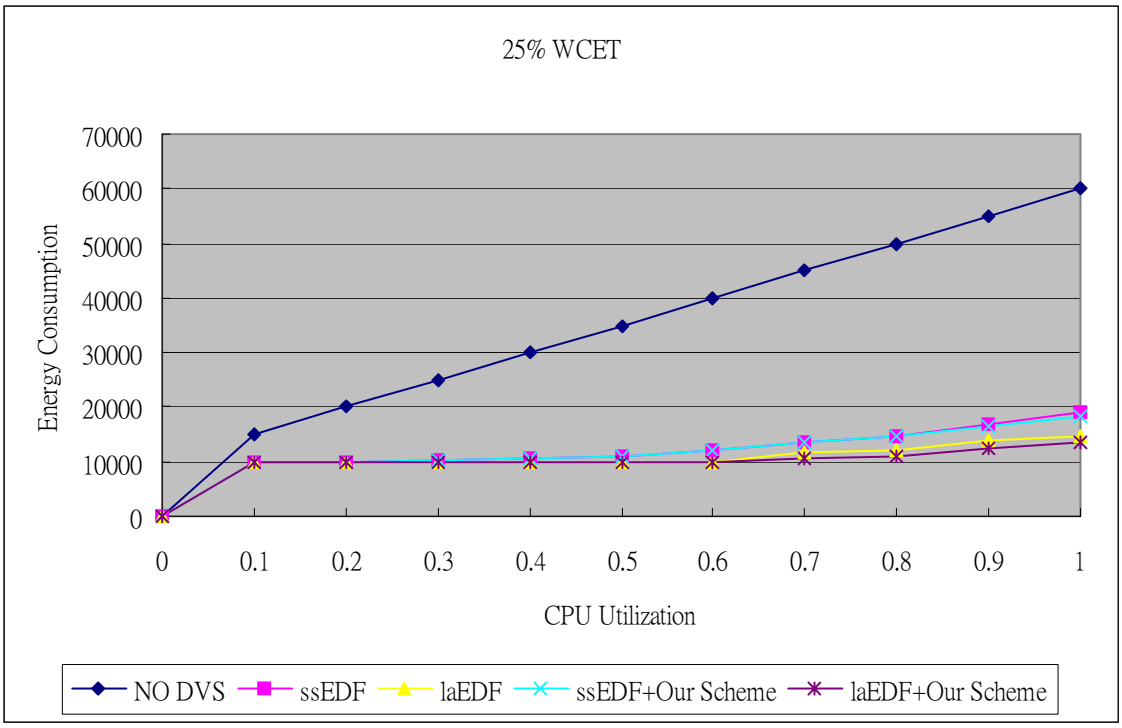


圖 6.4 Energy Consumption of 25% WCET

以圖6.4的情況來分析，實際的CPU利用率還不是很高。因此，CPU一直處於較低速的狀態就運行就足夠應付工作所需。所以轉換的機會較少;CPU一直是處於較穩定的情況所以與原來的的演算法差距不會太大。大約是2%~至5%。

圖6.5的情況來說，轉換的機會較多，因此具有比較多的能量差別，尤其是ssEDF演算法大約有17%的差別且總耗損能量有2%左又的差距。

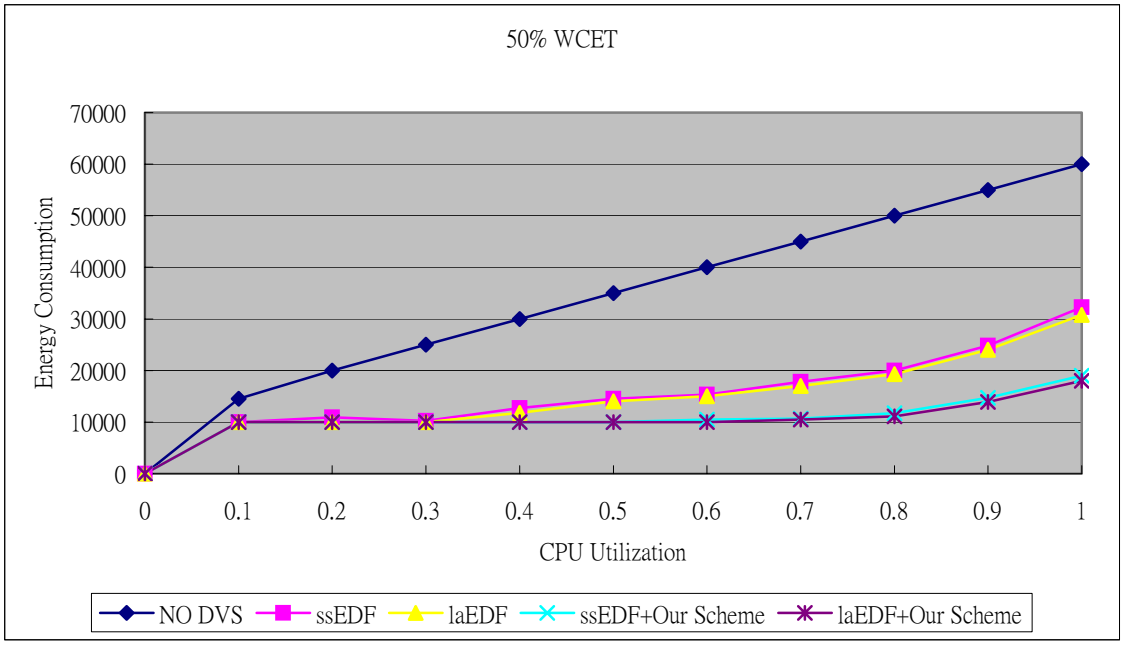


圖 6.5 Energy Consumption of 50% WCET

圖6.6 是75% WCET的情況，這種情況下調整的機會最頻繁。因此，在這種狀況來說總耗電量與調節上的耗損差距最大。

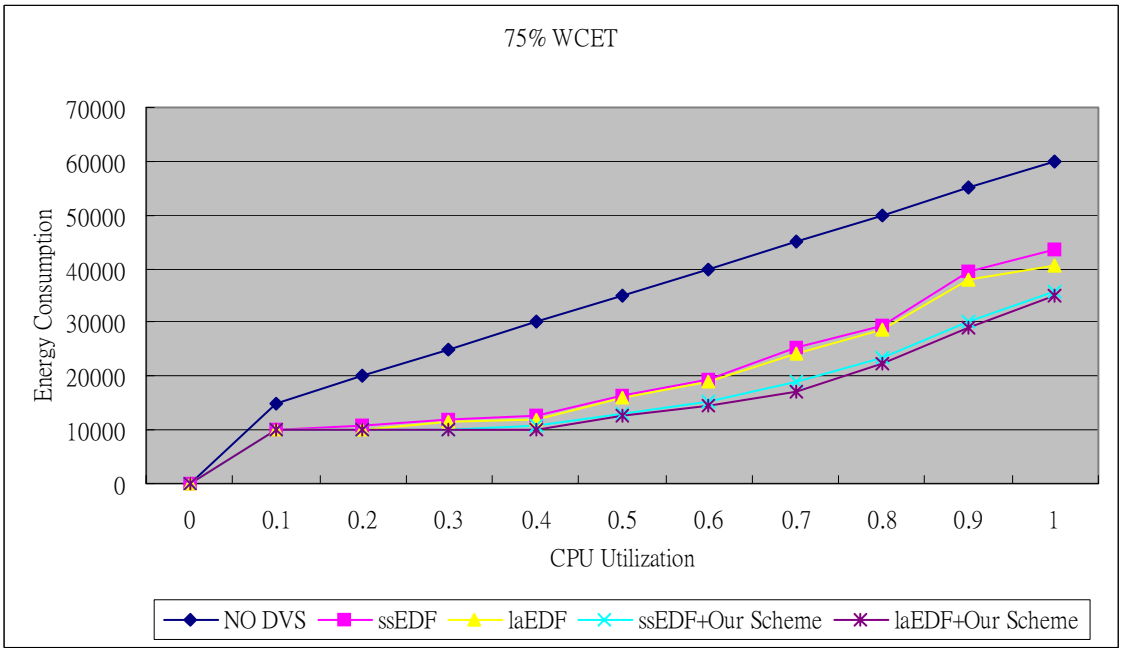


圖 6.6 Energy Consumption of 75% WCET

圖6.7 所表示的情況則本方法就比較糟了，其原因為:雖然轉換調整的機會變少了，但是為保持即時性，所以CPU將被要求較高的運算能力。因此，整體的能量消耗就稍稍多了些。

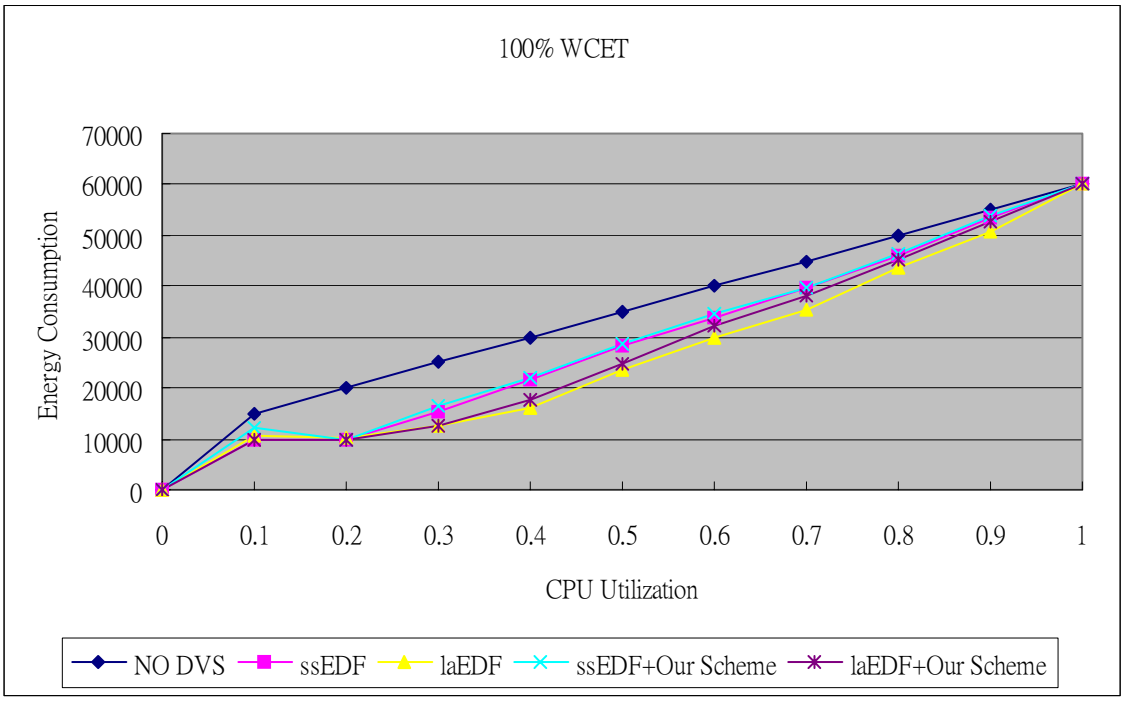


圖 6.7 Energy Consumption of 100% WCET

最後在即時性值方面，若CPU利用率接近1，且Overhead的時間大過於Task的可執行時間時Miss-Deadline的情況會越嚴重，且造成OS的效率不佳。 圖6.8為執行200個Task，且CPU利用率為75%，可以看出Overhead比例對Task的影響。

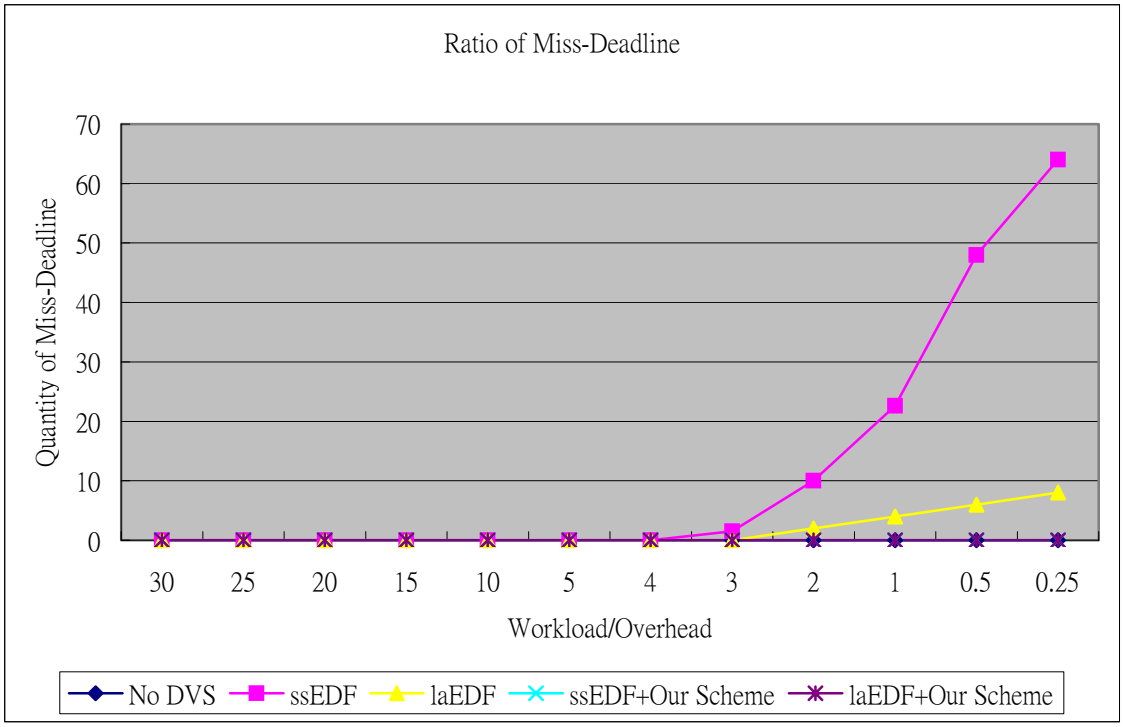


圖 6.8 Ratio of Miss-Deadline

第七章 結論

在本研究中專注於一個一直被忽略的問題，而Power Transition Overhead就是一個典型的例子。

在本研究中使用SA1110的電源參數，探討在轉換的過程中所造成時間的延遲和能量的耗損，透過數學模型與電子電路的物理性質已證明了這種現像的存在，在文中提出了一個演算法主要包含三個部份:Slack Estimation，Return of Investment與Power Control，能夠有效的降低許多不必要的轉換，在大多數的場合裡以整體的耗電量而言也有節省的趨勢。 此外，對即時性也有較佳的保障。 針對減少的耗損而言，以一般的Statically-Schedule EDF的演算法做比較有30%的減少，與較佳且先進的Look-Ahead EDF演算法做比較，使用本文的前後也有2%到5%的降低。 雖然在CPU接近滿載(超過90%)時的耗電量會稍微高一點，但在即時系統中，應該還是符合絕大多數的工作範圍。

因此，這項議題將來深入研究將有一個主要的方面，就是slack estimation，如果能夠增進slack預測的可靠度，那麼將可以加以利用，使得能夠更貼近系統實際的能量需要以求取最佳的利用率，降低耗電量。

參考文獻

[1] A. Vahdat, A. R. Lebeck and C. S. Ellis. “Every Joule is Precious: A Case for Revisiting Operating System Design for Energy Efficiency.” In *the 9th ACM SIGOPS European Workshop*, September 2000.

[2] P. J. M. Havinga and G. J. M. Smith. “Design Techniques for Low-power Systems.” Journal of Systems Architecture. Vol. 46:1, 2000

[3] M. Pedram. “Power Minimization in IC Design: Principles and Applications.” *ACM Transactions on Design Automation of Electronics Systems*. 1:1 - pp. 3-56, January 1996.

[4] Intel, Microsoft, and Toshiba. “Advanced Configuration and Power Management Interface (ACPI) Specification”, 1999. www.intel.com/ial/WfM/design/pmdt/acpidesc.htm.

[5] M. Srivastava, A. P. Chandrakasan and R. W. Brodersen. “Predictive System Shutdown and other Architectural Techniques for Energy Efficient Programmable Computation.” *IEEE Trans. on VLSI Systems*, 4(1): 42-55, 1996.

[6] Intel, Mobile Intel Pentium III Processor-M http://www.intel.com/products/notebook/processors/pentiumiii-m/index.htm?iid=sr+iii&

[7] M. Weiser, B. Welch, A. Demers, and S. Shenker. “Scheduling for Reduced CPU Energy.” *In Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, 1994.

[8] E. Chan, K. Govil, and H. Wasserman. “Comparing Algorithms for Dynamic Speed-setting of a Low-Power CPU.” *In Proceedings of the First ACM International Conference on Mobile Computing and Networking (MOBICOM 95)*, November 1995.

[9] J. Pouwelse, K. Langendoen and H. Sips. Dynamic Voltage Scaling on a Low-Power Microprocessor. 7th International Conference on Mobile Computing and Networking (MOBICOM), Rome, Italy, July 2001.

[10] D. Shin, J. Kim and S. Lee, "Intra-Task Voltage Scheduling for Low-Energy Hard Real-Time Applications," *IEEE Design and Test of Computers,* March 2001

[11]N. AbouGhazaleh, D. Mossé, B. Childers and R. Melhem "Toward The Placement of Power Management Points in Real Time Applications", *COLP'01 (Workshop on Compilers and Operating Systems for Low Power),* Barcelona, Spain, 2001

[12]L.H. Chandrasena and M.J. Liebelt. "A Rate Selection Algorithm for Quantized Undithered Dynamic Supply Voltage Scaling," *ISLPED'00: International Symposium on Low Power Electronics and Design,* pp. 213-215, July 2000.

[13]J.Chang and M. Pedram. "Energy Minimization Using Multiple Supply Voltages," *IEEE Transactions on Very Large Scale Integration Systems,* Vol.5, No.4, pp. 436-443, December 1997.

[14]I. Hong, et al. "Power Optimization of Variable-Voltage Core-Based Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* Vol.18, No.12, pp. 1702-1714, December 1999.

[15]T. Ishihara and H. Yasuura. "Voltage Scheduling Problem for Dynamically Variable Voltage Processors," *International Symposium on Low Power Electronics and Design,* pp. 197-202, August 1998.

[16]J. W. S. Liu. *Real-Time Systems*. Prebtice Hall, 2000.

[17]G. Buttazzo. *Hard Real-Time Computing Systems.* Kluwer Academic Publishers, 1997.

[18]G. Qu and M. Potkonjak. "Techniques for Energy Minimization of Communication Pipelines," *IEEE/ACM International Conference on Computer Aided Design,* pp. 597-600, November 1998.

[19]G. Quan and X. Hu. "Energy Efficient Fixed-Priority Scheduling for Real-Time Systems on Variable Voltage Processors," *DAC'01: IEEE/ACM Design Automation Conference,* pp. 828-833, June 2001.

[20]S. Raje and M. Sarrafzadeh. "Variable voltage scheduling," *ISLPED'95:*

International Symposium on Low Power Electronics and Design, pp. 9-14, April 1995.

[21]F. Yao, A. Demers and S. Shenker. A Scheduling Model for Reduced CPU Energy. *IEEE Annual Foundations of Computer Science*, pp. 374 -382, 1995.

[22]I. Hong, M. Potkonjak and M. B. Srivastava. On-line Scheduling of Hard Real-Time Tasks on Variable Voltage Processor. In *Computer-Aided Design (ICCAD)' 98*. pp. 653-656.

[23]I. Hong, D. Kirovski, G. Qu, M. Potkonjak and M. Srivastava. Power optimization of variable voltage core-based systems. In *Proceedings of the 35th Design Automation Conference, DAC' 98*

[24]I. Hong, G. Qu, M. Potkonjak and M. Srivastava. "Synthesis Techniques for Low-Power Hard Real-Time Systems on Variable Voltage Processors." In *Proceedings of 19th IEEE Real-Time Systems Symposium (RTSS' 98)*, Madrid, December 1998.

[25]H. Aydin, R. Melhem, D. Moss´e and P.M. Alvarez."Determining Optimal Processor Speeds for Periodic Real-Time Tasks with Different Power Characteristics." In *Proceedings of the 13th EuroMicro Conference on Real-Time Systems (ECRTS' 01)*, Delft, Netherlands, June 2001.

[26]H. Aydin, R. Melhem, D. Moss´e and P.M. Alvarez. "Dynamic and Aggressive Scheduling Techniques for Power-Aware Real-Time Systems." In *Proceedings of 22nd Real-Time Systems Symposium*, December 2001.

[27]R. Ernst and W. Ye. Embedded Program Timing Analysis based on Path Clustering and Architecture Classification. In *Computer-Aided Design (ICCAD)'* 97. pp. 598-604.

[28]H. Aydin, R. Melhem, D. Moss´e and P.M. Alvarez. Determining Optimal Processor Speeds for Periodic Real-Time Tasks with Different Power Characteristics. In *Proceedings of the 13th EuroMicro Conference on Real-Time Systems (ECRTS' 01)*, Delft, Netherlands, June 2001.

[29]C.L. Liu and J.W.Layland. "Scheduling Algorithms for Multiprogramming in Hard Real-time Environment." J. of *ACM 20(1)*: pp.46-61, 1973.

[30]Intel, Intel StrongARM SA1110 CPU http://www.intel.com/design/pca/applicationsprocessors/datashts/index.htm

[31]D. Grunwald, P. Levis, C. Morrey III, M. Neufeld, and K. Farkas. "*Policies for dynamic clock scheduling*." In Symp. on Operating Systems Design and

Implementation, Oct 2000.

[32]T. Pering, T. Burd, and R. Brodersen. “Voltage Scheduling in the lpARM Microprocessor System.” *International Symposium on Low Power Electronics and Design 2000*, pp.96-101, 2000.

[33]http://www.transmeta.com

[34]M. Weiser, B. Welch, A. J. Demers, and S. Shenker. “Scheduling for reduced CPU energy.” In *Operating Systems Design and Implementation (OSDI ’94)*, pages 13－23, 1994.

[35]A. Chandrakasan, S. Sheng, and R.W.Brodersen. “Low-power CMOS digitaldesign.” *IEEE J. Solid-State Circuits*, 27(4):473-484, 1992.

附錄 A 測試參數表

	WCET	Execution Time	Cycles	Period
0.25	112.5	11.25	2488.5	112.5
	562.5	42.1875	62213	562.5
	843.75	63.28125	93319	843.75
0.5	225	22.5	4977	225
	1125	84.375	62213	1125
	1687.5	126.5625	93319	1687.5
1	450	45	9954	450
	2250	168.75	62213	2250
	3375	253.125	93319	3375
2	900	90	19908	900
	4500	337.5	62213	4500
	6750	506.25	93319	6750
3	1350	135	29862	1350
	6750	506.25	62213	6750
	10125	759.375	93319	10125
4	1800	180	39816	1800
	9000	675	62213	9000
	13500	1012.5	93319	13500
5	2250	225	49770	2250
	11250	843.75	62213	11250
	16875	1265.625	93319	16875
10	4500	450	99540	4500
	22500	1687.5	62213	22500
	33750	2531.25	93319	33750
15	6750	675	149310	6750
	33750	2531.25	62213	33750
	50625	3796.875	93319	50625
20	9000	900	199080	9000
	45000	3375	62213	45000
	67500	5062.5	93319	67500
25	11250	1125	248850	11250
	56250	4218.75	62213	56250
	84375	6328.125	93319	84375
30	13500	1350	298620	13500
	67500	5062.5	62213	67500
	101250	7593.75	93319	101250

	WCET	Execution Time	Cycles	Period
0.25	112.5	28.125	6221	112.5
	562.5	70.3125	62213	562.5
	843.75	105.46875	93319	843.75

1.5	225	56.25	12442.5	225
	1125	140.625	62213	1125
	1687.5	210.9375	93319	1687.5
1	450	112.5	24885	450
	2250	281.25	62213	2250
	3375	421.875	93319	3375
2	900	225	49770	900
	4500	562.5	62213	4500
	6750	843.75	93319	6750
3	1350	337.5	74655	1350
	6750	843.75	62213	6750
	10125	1265.625	93319	10125
4	1800	450	99540	1800
	9000	1125	62213	9000
	13500	1687.5	93319	13500
5	2250	562.5	124425	2250
	11250	1406.25	62213	11250
	16875	2109.375	93319	16875
10	4500	1125	248850	4500
	22500	2812.5	62213	22500
	33750	4218.75	93319	33750
15	6750	1687.5	373275	6750
	33750	4218.75	62213	33750
	50625	6328.125	93319	50625
20	9000	2250	497700	9000
	45000	5625	62213	45000
	67500	8437.5	93319	67500
25	11250	2812.5	622125	11250
	56250	7031.25	62213	56250
	84375	10546.875	93319	84375
30	13500	3375	746550	13500
	67500	8437.5	62213	67500
	101250	12656.25	93319	101250
	WCET	Execution Time	Cycles	Period
0.25	112.5	28.125	6221	112.5
	562.5	70.3125	62213	562.5
	843.75	105.46875	93319	843.75
2.5	225	56.25	12442.5	225
	1125	140.625	62213	1125
	1687.5	210.9375	93319	1687.5
1	450	112.5	24885	450

	2250	281.25	62213	2250
	3375	421.875	93319	3375
2	900	225	49770	900
	4500	562.5	62213	4500
	6750	843.75	93319	6750
3	1350	337.5	74655	1350
	6750	843.75	62213	6750
	10125	1265.625	93319	10125
4	1800	450	99540	1800
	9000	1125	62213	9000
	13500	1687.5	93319	13500
5	2250	562.5	124425	2250
	11250	1406.25	62213	11250
	16875	2109.375	93319	16875
10	4500	1125	248850	4500
	22500	2812.5	62213	22500
	33750	4218.75	93319	33750
15	6750	1687.5	373275	6750
	33750	4218.75	62213	33750
	50625	6328.125	93319	50625
20	9000	2250	497700	9000
	45000	5625	62213	45000
	67500	8437.5	93319	67500
25	11250	2812.5	622125	11250
	56250	7031.25	62213	56250
	84375	10546.875	93319	84375
30	13500	3375	746550	13500
	67500	8437.5	62213	67500
	101250	12656.25	93319	101250
	WCET	Execution Time	Cycles	Period
0.25	112.5	42.1875	9331.875	112.5
	562.5	42.1875	62213	562.5
	843.75	253.125	93319	843.75
0.5	225	84.375	18663.75	225
	1125	84.375	62213	1125
	1687.5	506.25	93319	1687.5
1	450	168.75	37327.5	450
	2250	168.75	62213	2250
	3375	1012.5	93319	3375
2	900	337.5	74655	900
	4500	337.5	62213	4500

	6750	2025	93319	6750
3	1350	506.25	111982.5	1350
	6750	506.25	62213	6750
	10125	3037.5	93319	10125
4	1800	675	149310	1800
	9000	675	62213	9000
	13500	4050	93319	13500
5	2250	843.75	186637.5	2250
	11250	843.75	62213	11250
	16875	5062.5	93319	16875
10	4500	1687.5	373275	4500
	22500	1687.5	62213	22500
	33750	10125	93319	33750
15	6750	2531.25	559912.5	6750
	33750	2531.25	62213	33750
	50625	15187.5	93319	50625
20	9000	3375	746550	9000
	45000	3375	62213	45000
	67500	20250	93319	67500
25	11250	4218.75	933187.5	11250
	56250	4218.75	62213	56250
	84375	25312.5	93319	84375
30	13500	5062.5	1119825	13500
	67500	5062.5	62213	67500
	101250	30375	93319	101250