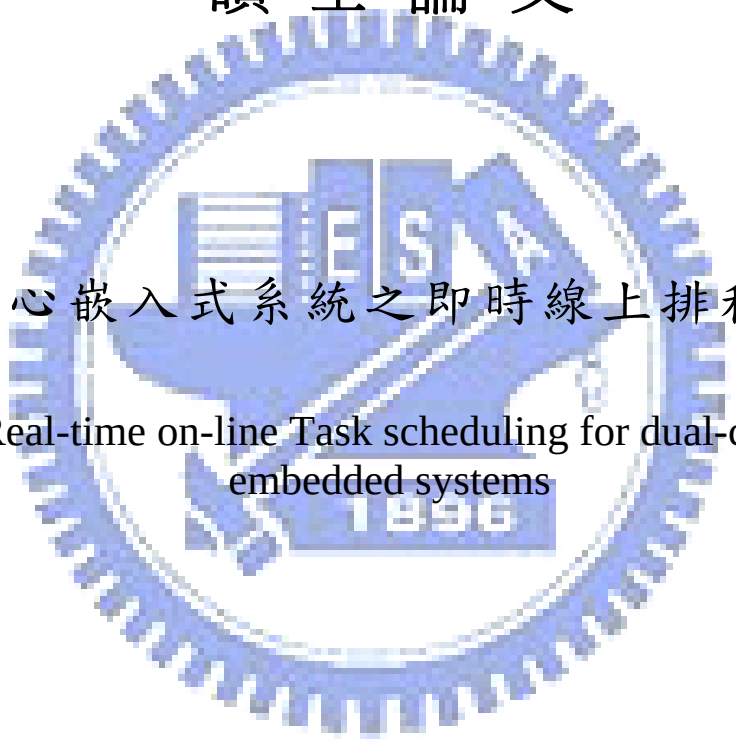
國立交通大學

資訊科學與工程研究所

碩 士 論 文



雙核心嵌入式系統之即時線上排程方法

Real-time on-line Task scheduling for dual-core embedded systems

研 究 生：鄭家明

指導教授：張立平　教授

中 華 民 國　九 十 七　年 八 月

雙核心嵌入式系統之即時線上排程方法

Real-time on-line Task scheduling for dual-core embedded systems

研 究 生：鄭家明　　Student：Chia-Ming Cheng

指導教授：張立平　　Advisor：Li-Pin Chang

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

August 2008

Hsinchu, Taiwan, Republic of China

中華民國九十七年八月

雙核心嵌入式系統之即時線上排程方法

學生：鄭家明　　指導教授：張立平

國立交通大學資訊科學與工程研究所碩士班

摘要

為了解決效能及能耗的問題，很多嵌入式系統產品採用 dual core 或是 multi-core 的架構，在這些架構中，常加入 DSP 來滿足日益漸增的多媒體需求。DSP 因為要處理大量的數學計算，所以 register 數量大，pipeline stage 深，造成 context switch overhead 很大，因此 DSP 在排程行為上屬於 Non-preemptive。此外，一個 DSP 會有多個 Task 在上面執行，容易發生 Resource Contention，加上每一個 Task 需要在不同的 core 之間切換，按照一定的順序在不同的 core 執行不同類型的工作 (precedence constraint)，所以在這樣具有 heterogeneous core 的平台在設計上會有以下的 challenges: (1) 如何在滿足 precedence constraint 的條件下，降低 Task 在 DSP 的 response delay，避免 Task miss deadline (2)如何 on-line 判斷一組 Task set 的可排程性，決定是否 accept 或是 reject 一個 on-line task。在本篇論文，我們的目標是解決上述的兩項問題：我們提出在 DSP 加入 Preemption Point 的方法，降低 Task 的 response delay 以及提昇系統的可排程性。此外，我們也設計在 microprocessor(MPU and DSP)的排程方法，並提出 admission control 方法檢查 Task set 可排程與否。

關鍵字：即時系統(Real-Time)，線上排程(on-line scheduling)，嵌入式系統(Embedded System)

誌　謝

到了寫誌謝這篇內容的時間點，代表兩年的研究生活即將告一段落，人生將要在另一個不同的領域啟航。兩年過去了，回想在碩一時熬夜準備期中、期末考並同時在一個禮拜中還要趕三份 project，當時的情景好像還是昨天的事一般；碩二開始，從訂定論文題目、建立實驗平台、提出理論、設計實驗、產生數據、口試，這段期間遭遇了許多困難與挫折，著實考驗並磨練我面對挫折的勇氣與解決問題的能力。

在這兩年中，要感謝的人實在太多，首先要感謝張立平教授的指導，使我學到不少和嵌入式系統有關的知識以及做研究的態度，沒有張教授的指導，本篇論文將無法完整與嚴謹。還要感謝林松德、黃千庭、許辰暉、許蕙茹同學，可以和你們在 ESSLAB 一起奮鬥努力是我的榮幸。也要感謝學弟妹，黃士庭、蘇宥全、黃明毅、郭郡杰、廖秀芬，實驗室有你們的加入更是增添笑聲與歡欣。

感謝口試委員楊家玲教授、謝仁偉教授、陳雅淑教授不辭辛勞且細心審閱並提供寶貴的建議，使我的論文更臻完善，在此由衷的感謝。

感謝我的父母在背後默默的支持與鼓勵，更要感謝我的未婚妻在這兩年的支持與體諒。

僅以此文獻給我摯愛的雙親及未婚妻。

目　　錄

表目錄

圖目錄

Section 1 Introduction

近年來許多數位多媒體產品，例如攜帶式多媒體裝置(PMP)、智慧手機、電視手機及行動電視相繼問世，手機也加入 MP3、PDA、GPS 導航系統等功能，所以這些產品本身需要更強大的壓縮、解壓縮、或是編碼、解碼的能力來滿足多媒體功能（multi-media）上的需求，DSP(Digital Signal Processing)便扮演重要的角色。DSP 是一種 instruction set 和硬體經過特別設計的 microprocessor，這種特製化的處理器著重於數學運算及訊號處理這類需求的效能表現。DSP 需要滿足兩項基本要求：一、善於處理數學運算，可以在每秒鐘完成數百萬次加法及乘法運算，此外，DSP 必需要能夠處理一些特別的演算法，例如 Convolution、Discrete Fourier Transform、Discrete Cosine Transform。二、DSP 必需要能夠符合即時性（Real-time）的要求，在每個工作的 deadline 到達之前完成該項工作[2]。MPU(micro processor unit)通常不具備浮點運算單元，由單一晶片構成，主要負責運算、感測、監控的工作，例如在遙控器裡頭的 MPU 負責感應使用者的操作，將按鈕訊號轉成紅外線發送給電視或是錄放影機。

由於廢熱以及能源消耗的因素，在 single processor architecture 上，performance 的提昇已經遭遇瓶頸，加上 multi-media application 隸屬於 computation-intensive application，在 single processor 上執行容易遭受 CPU 使用率的瓶頸（utilization bottleneck），使得只有少量的 application 可以同時存在 CPU，等待執行 。 multi-core processor architecture，這種在單一晶片上包含多顆處理器的系統架構，可以將每顆處理器的時脈（clock rate）降低，利用多顆處理器來增加效能並降低能源的消耗[1]，這種架構因此愈來愈受到注意及採用。因此將 MPU 和 DSP 結合，產生的 dual-core architecture，除了可以滿足日益漸增的多媒體運算需求，更可以提昇整體效能。MPU 在這個架構中擔任 Dispatcher 的角色，負責管控系統的運作，DSP 扮演著 worker 的角色，負責做數位訊號處理。

然而，在這個 dual-core（MPU+DSP）架構底下，有以下的問題：一、MPU 和 DSP 之間有一個執行先後的順序限制（precedence constraint），也就是 MPU 和 DSP 排程不是 independent， 例如一個工作需要不斷在 MPU 和 DSP 之間做切換，MPU 操作完才交給 DSP 去處理，MPU 這端還沒處理完，DSP 不能開始處理，只能等待 MPU 完成，這樣會損害 MPU 和 DSP 的可排程性（schedulability），也會降低 MPU 和 DSP 的使用率（utilization）。二、MPU 的排程具有可插斷的性質（preemptible），也就是一個 low priority job 執行到一半可能要讓出 MPU 給 high priority job；但是由於 DSP 附屬的 register 個數較多，pipeline stage 較深，因此 DSP 的 context switch overhead 很大，所以 DSP 通常屬於 non-preemptible，所以 high priority job 不能優先被處理，這會影響到整個系統的即時性（Real time）

和可排程性。三、MPU 和 DSP 所要執行的工作大多屬於 on-line task，無法預先得知和 task 有關的一些屬性，例如 arriving time, computation time 及 deadline，所以需要使用 on-line scheduling algorithm 來處理 MPU 和 DSP 的排程。

針對上述問題，本篇論文提出一個針對 dual core 的 on-line scheduling algorithm 以及一個 admission control policy。在 on-line scheduling algorithm 方面，在 maintain MPU 和 DSP 的 precedence constraint 的條件下，我們希望能解決因爲 DSP non-preemptible 這個屬性所造成的問題，像是 long blocking time，以及 low DSP utilization 這些問題， 所以我們在 DSP non-preemptible portion 加入 preemption point。在 preemption point，low priority job 會將 DSP 使用權交給 high priority job，達到協同式多工（Co-operative），除了可以降低 high priority job 的 response time，增加 DSP 的 utilization，也可以讓 scheduler accept 較多的 task。此外我們在 DSP 上利用 Bandwidth server[3]來確保每個 DSP subtask 的 response，不讓特定的 DSP subtask 一直佔住 DSP，使得其他的 job 一直處於等待 DSP 的情況。admission control policy 主要是用來檢測 MPU 和 DSP 之間工作的切換是否存在一個 feasible schedule，也就是存在一個排程方式使得所有工作都可以在 deadline 之前完成。當一項新進的工作(newly-arrived task)抵達，scheduler 會得知關於這個工作的 computation time，和 deadline，並判斷加入這項工作是否存在 feasible schedule，如果存在就接受，反之，拒絕這項工作，這個判斷的過程是在 on-line 進行，所以判斷需要非常的迅速，因此判斷的機制不能太複雜，我們將於後面的章節討論判斷的方法。feasible schedule 是否存在取決於 deadline，DSP preemption point 的設定，還有 bandwidth server size 的設定，在本篇論文將對這些屬性進行討論，並藉由實驗數據來進行比較。

Section 2 Problem definition

2.1 Architecture and System

在我們的 Dual-core 實驗平台有一顆 ARM MPU（926EJ-S）[4] 以及一顆工研院所研發的 DSP (PAC DSP)，MPU 和 DSP 有一塊 shared memory，可以讓 MPU 及 DSP 這兩顆 processor 的資料共享（sharing）以及同步(simultaneous)。兩顆 processor 之間的溝通是利用 mail box 來進行訊息、命令以及資料的傳遞。Task flow 的流程如下：一個 task 會先進入 MPU 執行一段時間，當 MPU 處理完後，這個 task 接著要進入 DSP 進行操作，於是 MPU 會將 DSP 需要用到的一些參數（i.e. computation time, deadline）放到 mailbox 裡頭，接著發送一個 Interrupt 給 DSP 通知 DSP 去處理，DSP 收到 Interrupt 之後就會開始處理這項工作；DSP 處理完後，會將結果再度放進 mailbox 裡頭，並發送一個 Interrupt 給 MPU 通知

DSP 已經完成所要求的工作。

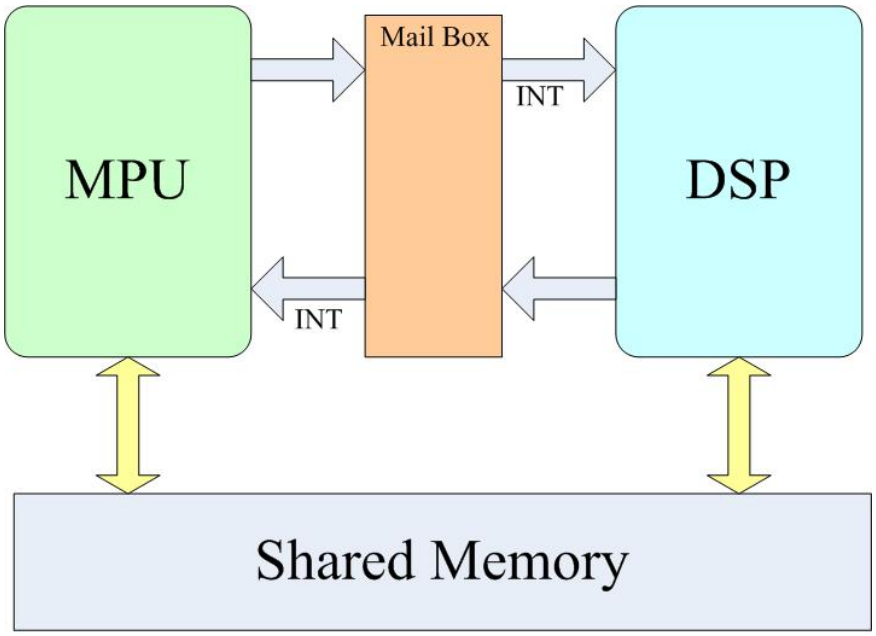


Fig. 1 System Architecture

2.2 MPU/DSP Task characteristics

MPU 通常用來負責 general purpose 工作，例如系統的監控及管理，在我們的實驗平台中，MPU 擔任 Dispatcher 的角色，用來接收工作，並指派工作給 DSP 處理。所以 MPU task 不會涉及複雜的運算，因此執行時間（computation time）不會很長。此外，爲了滿足即時性（Real-Time）的要求，每個工作會根據他的重要性、急迫性給予不同的 Priority。當一個 low priority job 執行到一半，有一個 high priority job 抵達，這個 high priority job 可以將正在執行的 low priority job preempt，讓 priority 較高的 job 優先處理。

DSP 通常用來負責對數位訊號進行處理，像是 Convolution、Discrete Fourier Transform、Discrete Cosine Transform、noise cancellation。這些和訊號處理有關的演算法，在執行過程涉及到大量的加法、乘法運算，所以 DSP task 需要比較長的執行時間（computation demanding）及較多的記憶體空間（High memory bandwidth），例如 FIR filter，在數學上的表示法爲∑xh，x 是 input data vector，h 是 filter coefficient，FIR filter 需要進行大量的加法及乘法，也需要記憶體來儲存輸入資料向量和 filter coefficient。加上 DSP 需要處理 streaming data，必須滿足即時性的要求。因此 DSP 在硬體設計方面，爲了提昇整體效能，附屬較多的 register， pipeline stage 也較深 [4]，造成 DSP 的 context switch overhead 很大，所以 DSP task 通常屬於 non-preemptable。

如同 Fig.2 所示，在我們的實驗平台中，MPU task 和 DSP task 是交錯地執行，也就是 DSP 必須等待 MPU task 完成後才能開始執行， 而 MPU task 完成後，也必須等待 DSP task 執行完成後才能繼續執行。

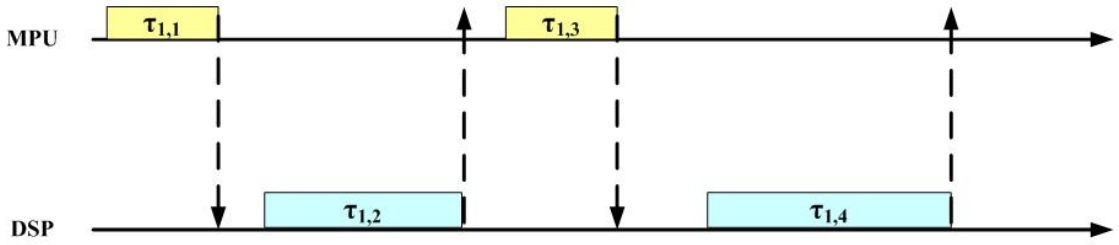


Fig. 2 The flow for MPU and DSP subtasks of a single Task

2.3 Problem Definition

接下來我們要來定義一些在本篇論文會用到的 term，以及要解決的問題。一個 task flow τ 就是一連串需要被完成的工作，這些工作我們稱作 subtask，例如在 Fig.2 這個 task flow 就是 $\tau_{1,1}$->$\tau_{1,2}$->$\tau_{1,3}$->$\tau_{1,4}$，爲了簡要，我們把 task flow 簡稱爲 task。 Task τ_i 的 task release time 就是 τ_i 進入系統的時間，我們把它表示爲 $R(\tau_i)$。 Taskτ_i 的 Starting time 就是該 task 開始執行第一個 subtask 的時間點。Task τ_i 的 Response time，簡稱爲 Resp (τ_i)，就是從 τ_i release 的時間到 τ_i 完成的這一段時間差。Task τ_i 的 Global Deadline，簡稱爲 GD(τ_i)，是 τ_i 在一個週期中，可被允許的最大 response time，在我們的 task model，Global Deadline 就是每一個 period 的結束點。

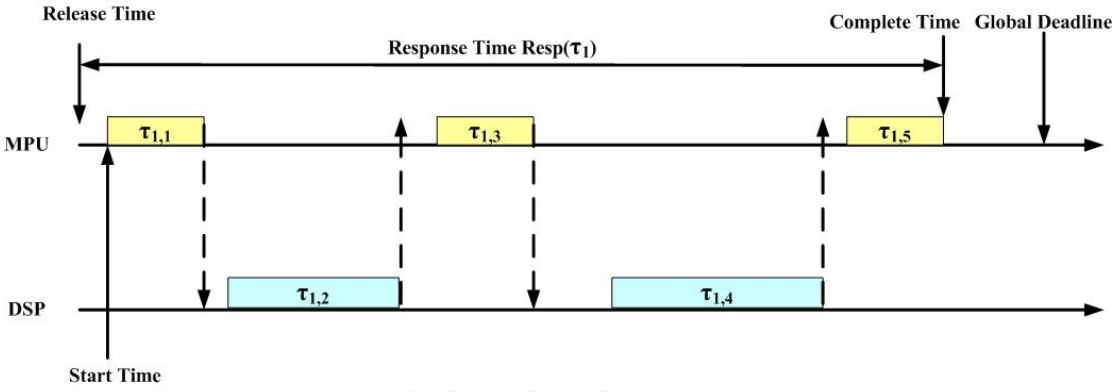


Fig. 3 Terminologies

我們把要執行的工作以 τ_1、τ_2 …來表示，每一個工作 τ_i 都有 n_i 個 subtask，屬於 τ_i 的每一個 subtask，我們以 $\tau_{i,j}$ 來表示，且 $1 \leq j \leq n_i$。每個 subtask 都有它所需要的執行時間，我們以 $e_{i,j}$ 來表示。在本篇論文中，τ_i 所有的 subtask $\tau_{i,j}$ 存在有 Precedence Constraint，也就是任兩個 subtask$\tau_{i,j}$ 和 $\tau_{i,k}$，當 $j < k$，$\tau_{i,k}$ 在 $\tau_{i,j}$ 還沒完

成前是不能開始執行，我們以 $\tau_{i,j} \prec \tau_{i,k}$ 來表示 $\tau_{i,j}$ 和 $\tau_{i,k}$ 存在有 Precedence Constraint。在本篇論文中，一個 task 會有 MPU subtask 及 DSP subtask，分別在 MPU 及 DSP 執行，而且是交錯地執行（如 Fig. 3），所以當 j 是奇數（例如，$\tau_{i,1}$,$\tau_{i,3}$,$\tau_{i,5}$….）$\tau_{i,j}$ 就是 MPU subtask，當 j 是偶數（例如，$\tau_{i,2}$,$\tau_{i,4}$,$\tau_{i,6}$….），$\tau_{i,j}$ 就是 DSP subtask。

一個 task set，簡稱爲 T，是由一些 task 所組成的集合，例如 $T=\{\tau_1, \tau_2, \ldots, \tau_M\}$ 代表一個由 M 個 task 所組合成的 task set。當一個 subtask 要去 request 一個 processing unit(MPU or DSP)，可是這個 processing unit，正被另一個 subtask 所佔住，這種情形就稱爲 Resource Contention，如果兩個 Task 的所有 subtask，都不會有 Resource contention，那這兩個 task 具有 Contention-free 的關係。

如同在 Introduction 所述，我們希望能解決由於 DSP 是 non-preemptible 所造成的問題，例如 low DSP utilization，long response time 。在我們的實驗平台上有以下這些限制：一、所有 task 需要滿足 Precedence Constraint，二、在排程行爲上，MPU 是 fully preemptive 而 DSP 卻是 non-preemptive。在這樣的限制下，我們需要找到一個 feasible schedule，使得所有 task 可以在 deadline 之前完成工作並盡量降低每個 task 的 response time，也希望增加 DSP utilization。這個問題的定義如下：

Definition : 關於一個 Task set $T=\{\tau_1, \tau_2, \ldots, \tau_M\}$，這個問題就是找到一個 feasible schedule，使得 $\forall \tau_i \in T, [R(\tau_i) + Resp(\tau_i)] < GD(\tau i)$。

Section 3 Method

3.1 Over View

在我們的實驗平台 (後面開始簡稱爲：系統) 上有兩個非對稱性的 core，一個是 fully preemptive 的 MPU，另一個是 non-preemptive 的 DSP。 如 Fig. 4 所示，當一個 on-line task 抵達系統，必須先進行 admission control test，根據該 task 的 computation time 以及 deadline 判斷是否存在一個 feasible schedule，使得在系統內部的所有 task 可以在 deadline 之前完成。如果存在有一個 feasible schedule，這個 newly arrived task 將被接受，反之該 task 會被 reject。

在我們的實驗設計上，一個 task 具有多個 subtask，subtask 主要分成兩種，MPU subtask 以及 DSP subtask，MPU subtask 和 DSP subtask 分別會在 MPU 及

DSP 上執行，這兩種 subtask 在排程上具有相依性，也就是 precedence constraint : 在一個 task 的執行順序上，MPU subtask 執行完換 DSP subtask 執行，DSP subtask 執行完，MPU subtask 接著執行，執行的順序是固定的，MPU subtask 尚未執行完 DSP subtask 不能開始執行，反之亦然。

在 MPU 上的 subtask 的排程是採用 preemptable scheduling ，讓 priority 高的 MPU subtask 優先執行。當一個 MPU subtask 完成後，接著就會有一個 DSP subtask 進入 DSP 等待執行，由於系統無法預先得知這個 DSP subtask 何時會進入 DSP，所以我們在 DSP 上使用 Bandwidth server 爲每一個 task 保留一部分的 DSP 執行時間來 guarantee 一個 DSP subtask 最遲可以完成的時間，當 DSP subtask 執行完畢，下一個 MPU subtask 就可以繼續開始執行。

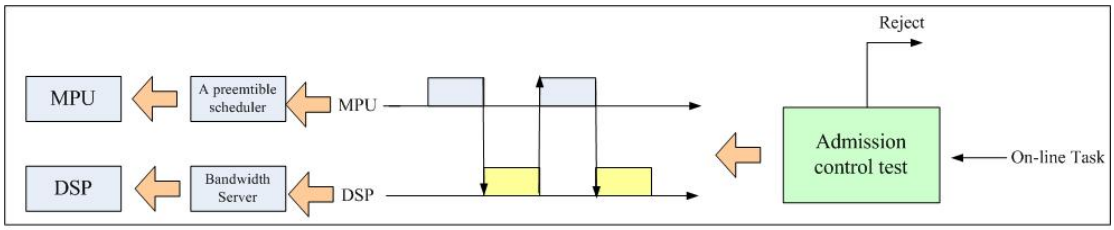


Fig. 4 System overview

3.2 MPU scheduling

在我們的 Task model 底下，task 是屬於 periodic，每一個 Task 都有一個固定的週期大小，在一個週期之中，每個 task 裡頭會有一部份工作在 MPU 上處理，另一部份在 DSP 上執行，而且需要交錯的處理，在 MPU 上處理的工作屬於 Preemptible，我們稱之爲 MPU subtask，而 DSP 上的工作屬於 Non-Preemptible，我們稱之爲 DSP subtask。當一個 Task τ_i 在 MPU 執行完部份工作，會 issue 在 DSP 上面的工作 (DSP subtask)，所以 τ_i 在 MPU 會 suspend，等待 DSP subtask 完成後再 Resume。如 Fig 5.所示，τ_1 在 MPU 完成 $\tau_{1,1}$ 這個工作後，會 Suspend 等待 DSP job 完成後才 Resume，繼續執行在 MPU 上的工作。

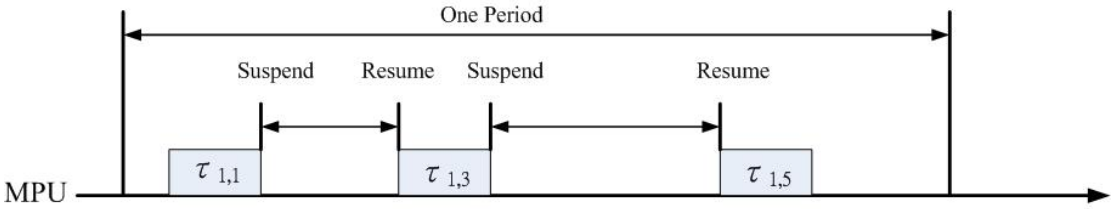


Fig. 5 MPU scheduling

通常 Periodic scheduling algorithm 針對 Task 執行到一半會發生 Suspension 的處理方法是把 Task τ_i 從發生 Suspension 到 Resume 的這一段時間視爲 τ_i 的 Blocking time。不過這種方法不適用於我們的 Task model，有以下兩個原因：一、考慮過於保守，Suspension 到 Resume 的這一段時間對於 τ_i 而言，是不去使用

MPU 而不是 τ_i 不能使用 MPU，如果把這段時間計爲 τ_i 的 Blocking time 將會過於保守。二、Suspension 到 Resume 的這一段時間基本上是決定在於 DSP 的排程，而不是取決於 MPU 上排程，τ_i 什麼時候可以 Resume 我們無法預先得知，我們只能知道最晚什麼時候可以 Resume，這個我們會在後面的章節討論。因此在 MPU 上執行的行爲將會如 Fig 6.所示，每一個 MPU subtask 之間會有一個 Separation， MPU subtask 的樣貌會呈現一段一段，彼此不會交疊在一起。這個 Separation 具有一個最大值，我們將在後面進行討論。

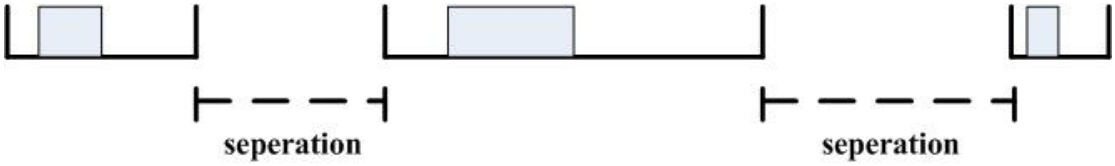


Fig. 6 Executing flow for MPU subtasks of a task

因爲上述的理由，MPU 的排程我們採用 EDF（Earliest Deadline First）來進行排程：

Theorem

A periodic system and a collection of sporadic tasks are schedulable by EDF if the sum of utilization of the former and instantaneous utilization of the latter is no greater than 1 at any time.

這個 theorem apply 的困難在於每個時間點都必須要去做 total CPU utilization 的檢查。但是在 MPU 排程採用 EDF 可行的原因在於，對於所有的 Task，只要我們爲每一個 Task τ_i assign 一個 density (Di)，Di 代表 τ_i 在 MPU 可以被保證使用時間的比例。我們會爲每一個 MPU subtask assign 一個 deadline，deadline assign 的方法基本上會依照該 MPU subtask 所屬 Task 的 density 去訂定，我們在後面的章節會詳細說明如何 assign deadline。如 Fig. 6 所示，因爲每一個 Task 所有的 MPU subtask 都不會交疊，代表在任何時間點，Task τ_i 最多會使用 MPU 時間的比例爲該 task 的 density Di，我們只需要去檢查所有 Task 的 density 總和是否小於或是等於 1，就可以確定是否存在一個 feasible schedule，上述的 theorem 依然可用。因此，MPU Admission Control 的 policy 如下：

Admission Control on MPU
if $\Sigma D_i <= 1$, => exist feasible schedule

3.3 DSP scheduling

當一個 MPU subtask 完成後，接續在這個 subtask 後面的 DSP subtask 會開始在 DSP 等待執行。因爲所有 DSP subtask 必須等待 MPU subtask 完成後才可以開始執行，DSP subtask 進入 DSP 的時間將無法預期，而且這些 DSP subtask 的 inter-arrival time 無法預先得知(non-deterministic)，所以可以將 DSP subtask 視爲 sporadic task。通常在 sporadic task 的排程方法都是採用 Bandwidth server 來保留一段 CPU 執行時間給每一個 task，確保每個 task 可以在一定的時間內完成。因此我們在 DSP 的排程採用 CUS (Constant Utilization Servers) 搭配 EDF 排程方法，在 DSP，每一個 Task 我們會 assign 一個專屬的 CUS server 來服務該 Task。

DSP 的排程屬於 non-preemptive，當一個 Task 進入 DSP 執行，必須等到該 Task 完成之後，其他的 Task 才能開始使用 DSP，加上一般而言，一個 Task 在 DSP 上執行所需要的時間相較在 MPU 上執行所需要的時間，需要比較長的執行時間，所以一旦有一個 Task lock 住 DSP 在 DSP 上執行，其他 Task 必須等待一段相當長的時間，直到該 Task 完成，所以效能比較差。以 Fig. 7 爲例，$\tau_{2,2}$ 在 t1 之前就已經 arrive，不過卻必須等到 $\tau_{1,2}$ 在時間點 t1 執行完畢後，才能開始執行，造成在時間點 t2，$\tau_{2,2}$ 就發生 miss deadline 的情形。

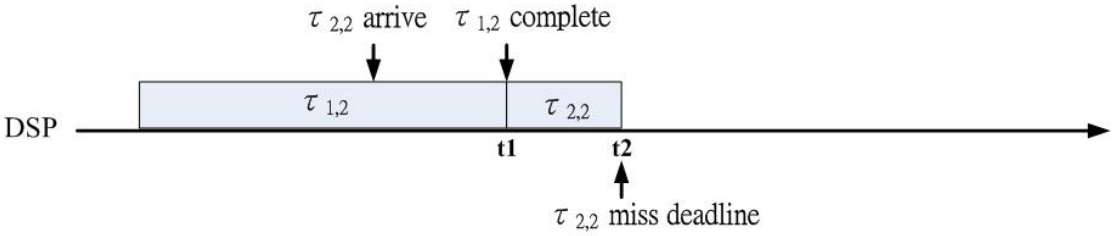


Fig. 7 A task set not with preemption point

爲了解決這個問題，我們在 DSP 排程行爲上加入了 Preemption point，在 Preemption Point 的地方，scheduler 會去檢查是否存在 deadline 更小的 DSP subtask 存在，如果有的話就讓它在 DSP 執行，如果沒有的話，原本在 DSP 上執行的 DSP subtask 就繼續執行，直到遇到下一個 preemption point 或是執行完畢，利用這樣的方法，DSP 就可以達到 co-operative 的效果。在 preemption point 以外的地方，DSP 的排程仍然是屬於 Non-preemptible，所以我們將兩個相鄰的 Preemption point 之間的時間差稱爲 Maximum non-preemptible duration, MNPD，這一段時間是一個 DSP subtask 進入 DSP 最長會被 block 的時間。

以 Fig.8 爲例，在時間點 t_1，DSP subtask $\tau_{1,2}$ 遇到了 preemption point，會去檢查是否有 deadline 更小的 DSP subtask ready，結果是沒有，所以 $\tau_{1,2}$ 繼續執行直到在時間點 t_2 遇到第二個 preemption point，這時候已經有一個 deadline 比 $\tau_{1,2}$ 更小的 DSP subtask $\tau_{2,2}$ ready，所以就讓 $\tau_{2,2}$ 先執行直到時間點 t_3，$\tau_{2,2}$ 完成，scheduler 會從已經 ready 的 DSP subtask 選擇 deadline 最小的 subtask 在 DSP 上執行，在這個例子中，將不會有任何 Task miss deadline，在這個例子中，MNPD 的大小爲

(t_2-t_1)。

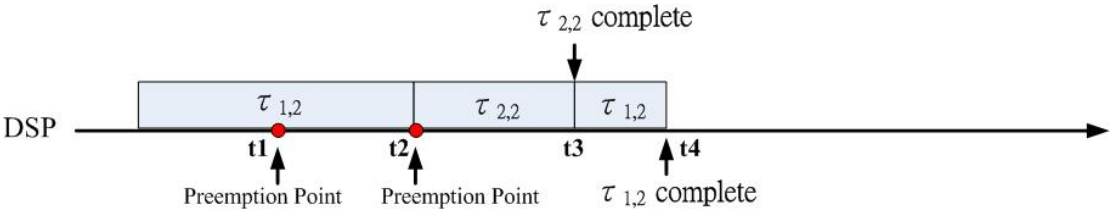


Fig. 8 A task set with preemption points

對於每個 task，在 DSP 都有一個專用的 CUS server 來服務這個 task 的所有 DSP subtask，每一個 CUS server 都有一個 server size，代表該 CUS server 可以被保證分配到 DSP 執行時間的比例。我們將利用每一個 Task 的 CUS server size，求出隸屬該 task 的 DSP subtask 的 deadline 做爲 EDF 排程的依據。接著我們要介紹 CUS 的 replenish rule 並利用下列的方法，在一個 DSP subtask 抵達 DSP，算出該 subtask 的 deadline。其中 Ci 是 CUS server size，e_i 代表 CUS server 的 budget。我們將在後面說明如何設定每一個 CUS server 的 server size 以及這裡求出來的 deadline 所代表的意義：

Replenishment Rules of CUS server with size c_i

R1: Initially, e_i =0, and deadline=0.
R2: when a DSP subtask$\tau_{i,j}$ arrive DSP,
(a) if current time < deadline, do nothing;
(b) if current time >= deadline, deadline =current time + $e_{i,j}$ /C_i, and e_i= $e_{i,j}$;
R3: At the deadline
(a) if the server is backlogged, set deadline = deadline + $e_{i,j}$ /Ci and e_i= $e_{i,j}$;
(b) if the server is idle, do nothing.

利用上述的方法算出的 deadline，就可以使用 EDF 排程方法決定哪一個 DSP subtask 應該先被執行，當一個正在執行 DSP 的 DSP subtask 抵達 preemption point 或是已經完成工作後，scheduler 會去檢查所有 ready DSP subtask 的 deadline，挑選 deadline 最小的 DSP subtask 在 DSP 上執行。此外，爲了確保所有 task 都可以在 deadline 之前完成，所有 CUS server size 外加 Non-Preemption portion 所造成的 DSP utilization 的總和必需要小於 1，這也是我們在 DSP 上所做的 Admission Control 的方法。

Admission Control on DSP

Step1: Find out the Min($e_{i,j}$ /c_i) i=1,2,3…Task#, j=1,2,3…Subtask #
Step2: if ($\sum \mathbf{C_i}$ + MNPD / Min($e_{i,j}$ /C_i)) <=1，=> there exists a feasible

3.4 MPU and DSP precedence constraint

在 Section 3.2 曾經提過，在我們設計的 Task model 底下，所有 Task 都是 Periodic，每一個 Task 在一個 period 的 deadline 就是該 period 的結束點，我們將這個 deadline 稱爲 Global Deadline。當一個 Task 第一次抵達 MPU 或是剛由 DSP 返回時，我們需要知道每一個 MPU subtask 最晚什麼時候會 issue DSP subtask，也就是該 MPU subtask 最晚什麼時候會 suspend，所以我們需要爲每一個 MPU subtask assign 一個 deadline，這個 deadline 就是該 MPU subtask 最晚會 issue 一個 DSP subtask 的時間，我們可以用這個 deadline 做爲該 MPU subtask 在 MPU 上 EDF 排程的依據，而非使用 Global Deadline。以 Fig.9 爲例，Taskτ_1 會在時間點 t1 和 t3 開始在 MPU 上執行 MPU subtask $\tau_{1,1}$ 和 $\tau_{1,3}$，我們分別將 $\tau_{1,1}$ 和 $\tau_{1,2}$ 的 deadline 設定在 t2 和 t4，這就代表 $\tau_{1,1}$ 和 $\tau_{1,3}$ 最遲分別會在 t2 和 t4 完成並 issue DSP subtask$\tau_{1,2}$ 和 $\tau_{1,4}$。

爲了要滿足我們在 Section 3.2 所提出的 Admission Control 方法，加上每一個 Task 都有一個 density 值 Di，因此每一個 Task 所屬的所有 MPU subtask 在 MPU 的執行時間去除以該 MPU subtask 從 arrive 至 deadline 這一段時間所得到的數值必須要和 density, Di 相同，以 Fig. 9 爲例，$e_{1,1}$ / (t2-t1)=D1, $e_{1,3}$ / (t5-t3)=D1。我們將以此觀念爲基礎，在後面詳述如何設定每一個 subtask 的 deadline，以及如何設定每一個 Task 的 density 大小。

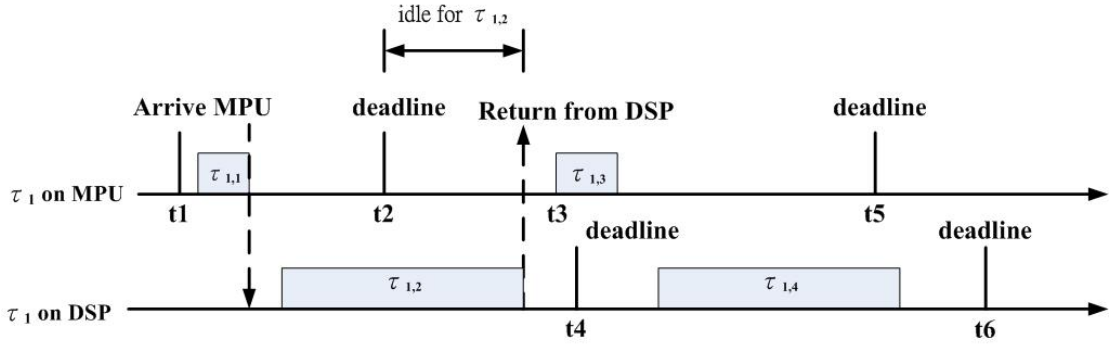


Fig. 9 Local deadlines of MPU and DSP subtasks

在 Section 2 曾經提過，我們的 Task model 必須要滿足 Precedence Constraint 的需求：MPU subtask 和 DSP subtask 會交錯執行而且具有一個先後順序的關係。如同 Fig. 10 所示，Task 1 具有 $\tau_{1,1}$, $\tau_{1,2}$, $\tau_{1,3}$, $\tau_{1,4}$, $\tau_{1,5}$ 這些 subtask，其中 $\tau_{1,1}$, $\tau_{1,3}$, $\tau_{1,5}$ 屬於 MPU subtask，$\tau_{1,2}$, $\tau_{1,4}$,屬於 DSP subtask，$\tau_{1,1}$ 換 $\tau_{1,2}$ 執行，在 $\tau_{1,1}$ 還沒執行完之前 $\tau_{1,2}$ 不能先行執行，$\tau_{1,3}$ 也必須等到完成後才能開始在 MPU 執行，這就是 Precedence Constraint，也是我們的 Task model 所必須要滿足的條件。

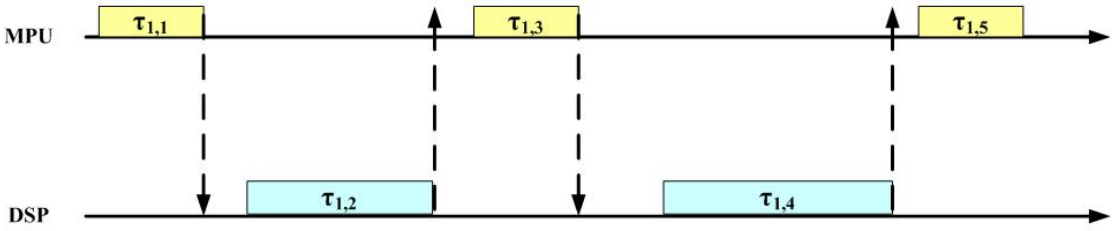


Fig. 10 Precedence Constraint for a single task

當一個 Task 的 MPU subtask ready 之後，就可以利用 EDF 排程方法決定優先在 MPU 執行的 subtask。當這個 subtask 執行完畢後，其所屬的 Task 需要離開 MPU 到 DSP 上開始執行 DSP subtask。我們將 DSP 視爲 Server 服務所有的 Task。如同前面所提，每一個 Task 都有一個 CUS server 來服務它的所有 DSP subtask。當一個 Task 其 DSP subtask 進入 DSP 後，需要有 policy 來知道該 DSP subtask 最晚什麼時候可以完成回到 MPU 繼續處理 MPU subtask。因爲每一個 task 都有一個 CUS server 爲其服務，所以對於每一個 task，我們只要去設定該 task 在 DSP 的 service quality，就可以知道且確定該 task 的 DSP subtask 最遲會完成的時間。關於如何設定每一個 task 在 DSP 上面的 service quality，我們的做法是去設定服務該 task 的 CUS server size，利用 CUS server size 就可以控制一個 Task 在 DSP 的 service quality。

如 Fig. 11 所示，當一個 MPU subtask 完成後，接著會 issue 一個 DSP subtask，在 MPU 上就開始呈現 suspend 的狀態。DSP subtask 進入 DSP 後，什麼時候會開始執行，什麼時候可以完成，無法預先得知。利用先前爲每一個 Task 所設定的 DSP service quality，我們可以明確知道每一個 DSP subtask 最晚可以在什麼時間點完成，也就是每一個 DSP subtask 都具有 bounded response time， response time 存在有最大值。當 DSP subtask 完成後就會離開 DSP，該 Task 在 MPU 就會立刻 Resume，繼續處理 MPU subtask。

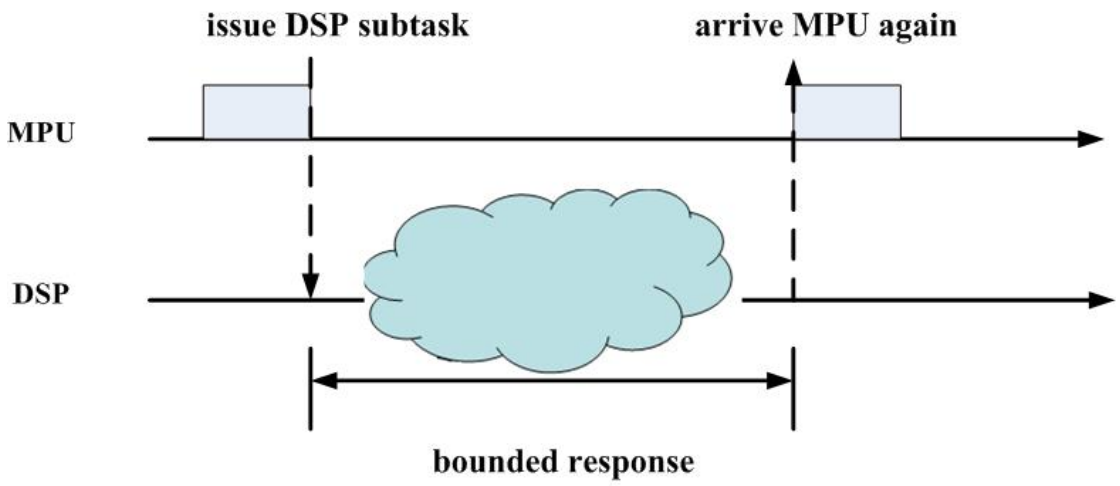


Fig. 11 Bounded response for a DSP subtask

3.5 Task density, deadline and Example

在這個小節，我們將要介紹關於每一個 task 其 Density 的算法以及所有 subtask deadline 設定的方式，並以表 1 的 Task1 做爲例子求出該 Task 的 density 及所有 subtask 的 deadline。

Task 1 with Density=0.25, CUS server size=0.2, Period=145, 2 MPU subtask ($\tau_{1,1}$,$\tau_{1,3}$) and 2 DSP subtask($\tau_{1,2}$,$\tau_{1,4}$)				
Subtask	Computation time	Arrival time	Complete time	Local deadline
$\tau_{1,1}$(MPU)	2	0	5	$d_{1,1}$ = 0+2/0.25=8
$\tau_{1,2}$(DSP)	10	5	42	$d_{1,2}$ = 5+10/0.2=55
$\tau_{1,3}$(MPU)	3	42	52	$d_{1,3}$ = 42+3/0.25=54
$\tau_{1,4}$(DSP)	15	55	75	$d_{1,4}$= 55+15/0.2=130

Table 1: A normal case example

由前面的敘述可以知道，一個 Task 在 DSP 的 response time 和該 Task 的 CUS server size 有關，一旦知道 CUS server size，就可以知道該 CUS server 所 service 的 DSP subtask 最多需要在 DSP 花多少時間，這個時間包括了等待 DSP 的時間及在 DSP 執行的時間。這段時間的算法如下：

for a DSP subtask $\tau_{i,j}$, max response time $s_{i,j} = e_{i,j} / C_i$

以表 1 的 Task 1 爲例， Task 1 的 CUS server size c1=0.2，由上述的算法，$\tau_{1,2}$ 的 max response time $s_{1,2}$ 爲 10/0.2＝50，$\tau_{1,4}$ 的 max response time $s_{1,4}$ 爲 15/0.2＝75。

當一個 Task $\boldsymbol{\tau_i}$ 的所有 DSP subtask 的 max response time 求出來後，我們可以算出 $\boldsymbol{\tau_i}$ 在一個週期內所有 DSP subtask max response time 的總和 S_i。因此在一個週期內，$\boldsymbol{\tau_i}$ 的所有 MPU subtask response time 的總和不能大於 ($P_i - S_i$)，其中 P_i 爲 Task $\boldsymbol{\tau_i}$ 的週期。利用這個關係，我們可以知道如何爲每一個 Task $\boldsymbol{\tau_i}$ 設定 MPU 上的 density D_i。Di 的算法如下式：

D_i = Total MPU subtask execution time in a period / (Period P_i –total max response time S_i)

以表 1 的 Task 1 爲例，$S_1 = s_{1,2} + s_{1,4} = 50 + 75 = 125$，Total MPU subtask execution time = 2 + 3 = 5，週期 P_1 = 145，因此我們可以求出 Task 1 的 density D_1 = 5 / (145-125) = 0.25。

當一個 Task 的 density 大小求出來後，該 Task 的所有 MPU subtask ，當它在 MPU 上 arrive 的時候我們可以由 density 利用下面的方法求出來一個值，我們稱之爲 Local Deadline for MPU 。每一個 MPU subtask 的 Local Deadline 被視爲該 subtask 最遲必須要完成的時間，也就是最遲會進入 DSP 的時間，我們將它用來做爲 MPU 上 EDF 排程方法的依據。

Local Deadline for MPU = arrival time + MPU subtask execution time ($\mathbf{e_{i,j}}$) / Density (D_i)

以表 1 的 Task 1 爲例，MPU subtask $\tau_{1,1}$ 和 $\tau_{1,3}$ 的 arrival time 分別爲 0 和 42，而且 computation time 分別爲 2 和 3，所以 $\tau_{1,1}$ 的 local deadline 就是 0 + 2/0.25＝8，$\tau_{1,2}$ 的 local deadline 就是 42 + 3/0.25＝54。

相同的，對於所有 DSP subtask，當它在 DSP 上 arrive 的時候， 我們一樣可以由 CUS server size 利用下面的方法求出來一個值做爲 Local Deadline for DSP。每一個 DSP subtask 的 Local Deadline 被視爲該 subtask 最遲必須要完成的時間，我們將它用來做爲 DSP 上 EDF 排程方法的依據。

Local Deadline for DSP = arrival time + DSP subtask execution time ($\mathbf{e_{i,j}}$) / CUS server size (C_i)

以表 1 的 Task 1 爲例，DSP subtask $\tau_{1,2}$ 和 $\tau_{1,4}$ 的 arrival time 分別爲 5 和 55，而

且 computation time 分別爲 10 和 15，所以 $\tau_{1,1}$ 的 local deadline 就是 5+ 10/0.2＝55，$\tau_{1,2}$ 的 local deadline 就是 55 + 15/0.2＝130。

總結上述，Task 1 的所有 subtask 的排程，將會如同 Fig. 12 的樣式，其中 $d_{1,1}$ 代表 MPU subtask$\tau_{1,1}$ 的 Local Deadline，$d_{1,2}$ 代表 DSP subtask$\tau_{1,2}$ 的 Local Deadline。Fig.12 有一個地方要注意的就是，當 $\tau_{1,3}$ 完成之後會 issue DSP subtask $\tau_{1,4}$，不過由於我們在 DSP 的排程是採用 CUS 排程方法，$\tau_{1,4}$ 必須在所屬的 CUS server 在時間點 55 的時候才會補充 budget，所以 $\tau_{1,4}$ 必須 backlog 直到 $\tau_{1,2}$ 的 deadline，也就是時間點 $d_{1,2}$，才能進入 DSP 等待開始執行。

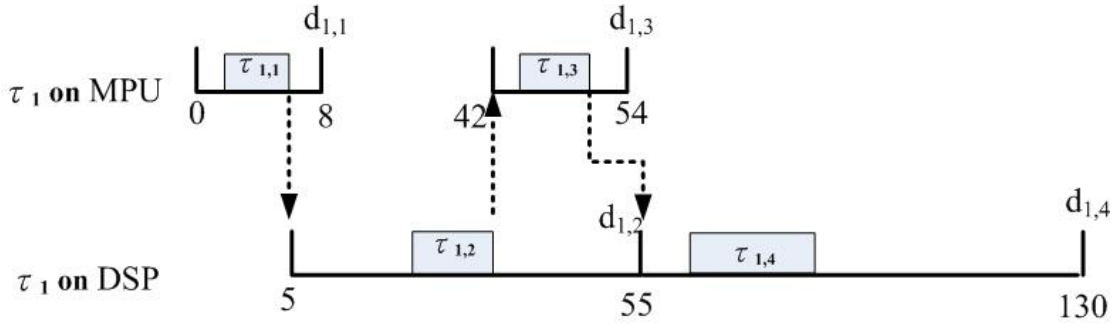


Fig. 12 An example for setting local deadline

Task 1 with Density=0.25, CUS server size=0.2, Period=145, 2 MPU subtask ($\tau_{1,1}$,$\tau_{1,3}$) and 2 DSP subtask($\tau_{1,2}$,$\tau_{1,4}$)				
Subtask	Computation time	Arrival time	Complete time	Local deadline
$\tau_{1,1}$(MPU)	2	0	8	$d_{1,1}$ = 0+2/0.25=8
$\tau_{1,2}$(DSP)	10	8	58	$d_{1,2}$ = 8+10/0.2=58
$\tau_{1,3}$(MPU)	3	58	70	$d_{1,3}$ = 58+3/0.25=70
$\tau_{1,4}$(DSP)	15	**70**	145	$d_{1,4}$= 70+15/0.2=145

Table 2: worst case example

表二所列出的數據是 Task 1 的 worst case，也就是每一個 subtask 都剛好在 deadline 抵達時完成工作。Fig. 13 是 Task 1 在 MPU 以及 DSP 執行的時序圖。

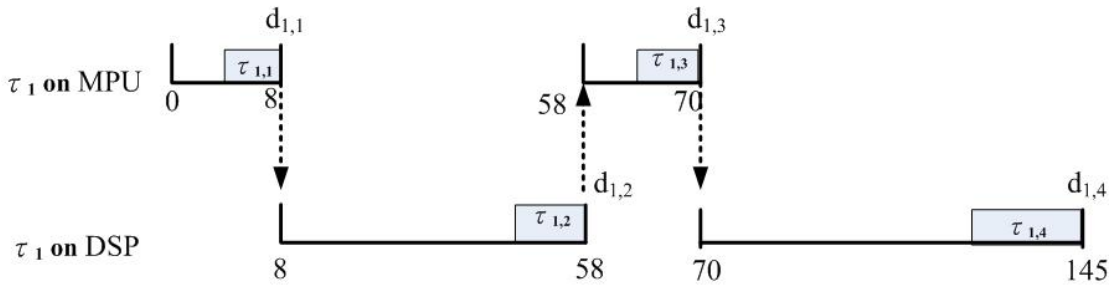


Fig. 13 Worst case of the example

Section 4 Experiment Results

4.1 Experiment Setting and Performance Metrics

如同我們在 section 2 所述，在 Fig. 14 可以看出，我們實驗的系統存在有兩個 Core，其中一個 Core 是 ARM 926EJ-S，我們將這個 Core 用來扮演 MPU 的角色，另一個 Core 是工研院所研發的 DSP (PAC DSP)，用來負責處理每一個 Task 的 DSP subtask。這兩個 Core 之間存在有 shared memory，用來協助這兩個 Core 達到資料共享及同步。此外，關於訊息及命令的傳遞，是採用 Mail Box 的方法，當一個 Core 送出命令或是訊息到 Mail Box 之後，會發送 Interrupt 告知另一個 Core，該 Core 就會知道要去處理這些命令及訊息。在 Operating System 方面，在 MPU 我們採用的 uCOSII 這個 popular 即時作業系統用來管理 resource 以及負責 Task 的排程。

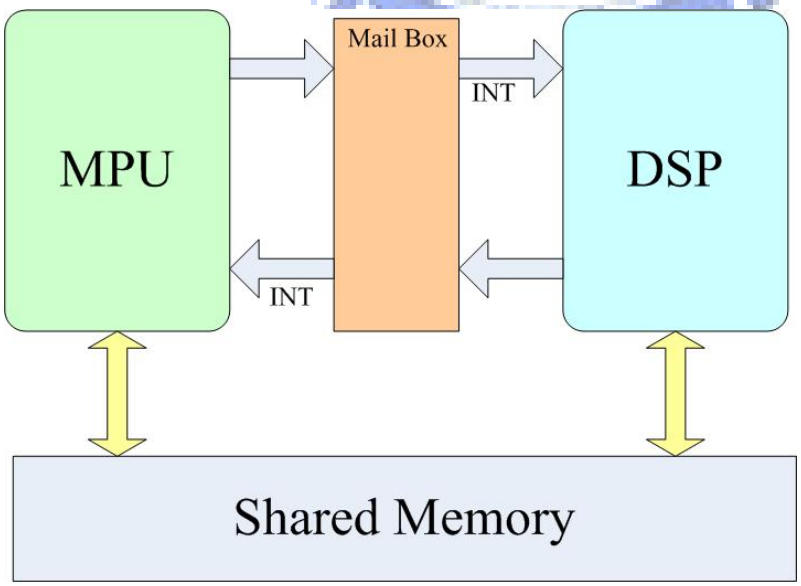


Fig. 14 System Architecture

在實驗的 task set 產生的部份，我們撰寫一個 task set generator 的程式，這個程式產生 task set 的順序如下：

1. 爲每一個 Task τ_i 產生一個 Period P_i，基本上我們讓 Task number 愈大的 Task 擁有較大的週期，也就是 $P_1 < P2 < P_3 < P_4 \ldots$，每個週期會有一個範圍的限制(i.e. $800< P_1 <1500$)，把不同 Task 的週期做分類，會比較容易觀察實驗結果。

2. 利用上一步驟算出的週期，求出這些週期的 Hyper Period。

3. 亂數產生每一個 Task 的 CUS server size，而且 CUS server size 總和加上 MNPD 所造成的 DSP utilization 的總和也不得大於 1。

4. 亂數產生每一個 Task 的 MPU 和 DSP subtask 的 computation time。

5. 利用下列公式求出每一個 task 的 density Di:
Di= (total MPU subtask computation time of Task i in a period) / [Pi-(total DSP subtask computation time in a period / CUS server size Ci)]

6. 檢查所有 Task 的 density 總和是否小於等於 1。不是的話，就捨棄這組 Task set。

在實驗數據的觀察，我們使用的是 RDC (response time divide by computation time)，將一個 subtask 的 Response time 去除以該 subtask 的 computation time。RDC 所代表的意義的就一個單位時間的工作，預期要花多久的時間可以完成，所以 RDC 的值最小爲 1，在最佳的情形下，一個單位時間的工作可以在一個單位時間完成。 RDC 的求法如下：
RDC = (Response time of a subtask) / (Computation time of a subtask)
基本上，一個 Task RDC 的值愈小愈好，代表該 Task 在 Arrive 之後，可以很快的完成工作。在我們的實驗數據當中，我們將一個 Task 其 MPU 和 DSP subtask 的 RDC 值分開計算，分別求出這些數據的平均值及標準差，且製作出對應的圖表。

將 MPU 和 DSP 的 RDC 值分開計算的原因在於 MPU 和 DSP 排程的屬性不同，MPU 是 fully preemptive，DSP 是 co-operative，加上 Task 在 MPU 和 DSP 上面的執行時間差異很大，所以將 MPU 和 DSP 的 RDC 分開計算比較能夠看出差異性。我們的數據紀錄平均值的原因在於，當一個 Task 他的 RDC 平均值低於其他的 Task 時，代表這個 Task 的 Response 效果較好，也就是可以預期當這個 Task arrive 之後，能夠比較快被完成，相反地 RDC 的數值較大，代表著一個 Task arrive 之後，需要經過較長的時間才能 complete。因此利用 RDC 的平均值，我們可以比較出 Task 之間的 Response 效果。另外，藉由觀察 RDC 標準差的數據，我們

可以看出一個 Task 所有 subtask response 的差異性，當一個 Task 其 RDC 的平均值標準差很大時，就代表著這個 Task，在 MPU 或是 DSP 上面執行時，有些 subtask 的 response 很快，有些就很慢，差異很大，這可以用來幫助我們對不同的排程方法進行比較。

4.2 Experiment Result

在這個章節我們將提出實驗數據，首先我們提出的數據是比較有加入 Preemption Point (with preemption point, WP) 和 沒有加入 Preemption Point (not with preemption point, NP)其 RDC 數值的平均值和標準差。基本上每一個實驗，我們會針對三種不同的 CUS server size 的配置方法進行比較，第一種是所有 Task 的 CUS server size 相似(差距不大於 0.05)，第二種是 CUS server size 和 Task 的週期成反比，也就是週期愈大的 Task 其 CUS server size 愈小。第三種是第二種是 CUS server size 和 Task 的週期成正比，也就是週期愈大的 Task 其 CUS server size 愈大。爲了方便觀察出數據的趨勢，我們讓週期隨著 Task 的 number 做遞增，也就是 Period of Task1<= Period of Task2<= Period of Task3<= Period of Task4。

在下列圖表的 X 軸代表不同的 Task，每一個 Task 我們去比較有加入 preemption point (WP) 和沒有加入 preemption point (NP)的差異。Y 軸代表每一個 Task 其實驗結果 RDC 的平均值或是標準差。

由我們的實驗數據可以看到下列的結果：

1. 在 WP 的部份，DSP subtask 的 RDC 的平均值會隨著週期遞增，週期小的 Task(ex：Task1)其 DSP subtask RDC 平均值會低於週期大的 Task(ex：Task4)。
2. 在 NP 的部份，DSP subtask 的 RDC 的平均值會隨著週期遞減，週期小的 Task(ex：Task1)其 DSP subtask RDC 平均值會高於週期大的 Task(ex：Task4)。
3. WP 在 DSP subtask RDC 的標準差會小於 NP 的 DSP subtask RDC 的標準差。
4. 當一個 Task 其 CUS server 上升時，該 Task 的 DSP subtask 的 RDC 平均值將會下降。
5. WP 和 NP 在 MPU subtask 的 RDC 平均值基本上沒有差別。

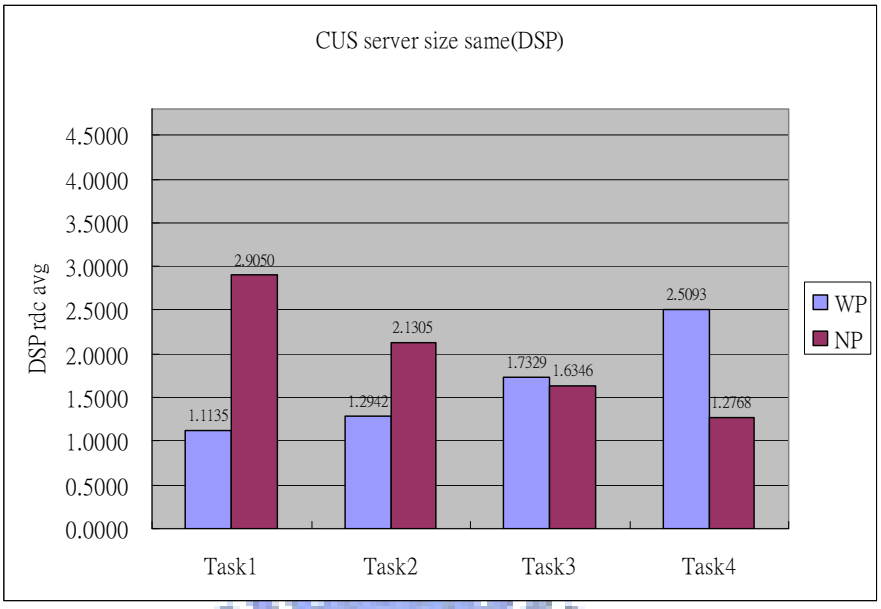


Fig. 15 DSP subtask RDC AVG - same CUS server size

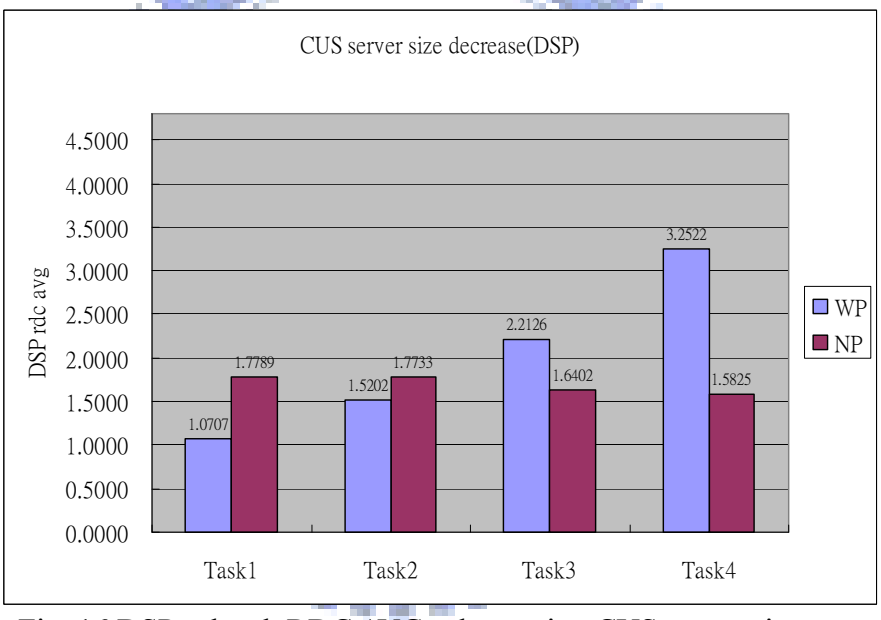


Fig. 16 DSP subtask RDC AVG - decreasing CUS server size

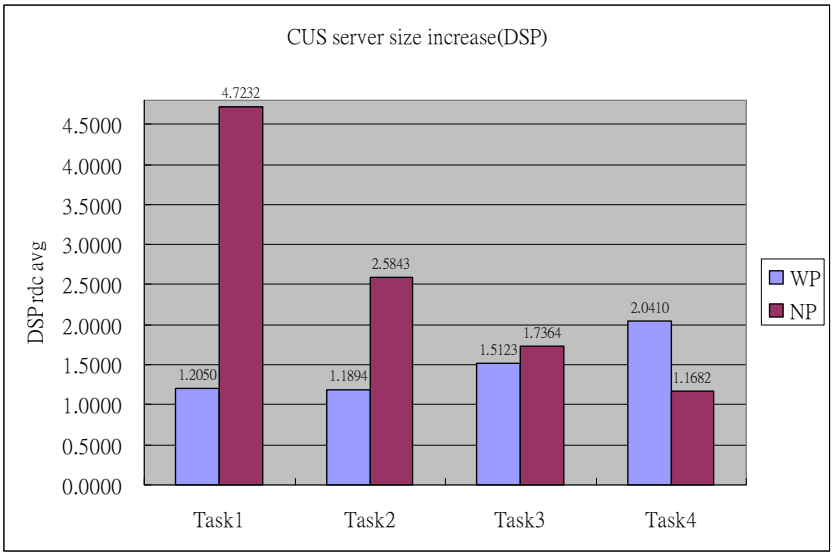


Fig. 17 DSP subtask RDC AVG - increasing CUS server size

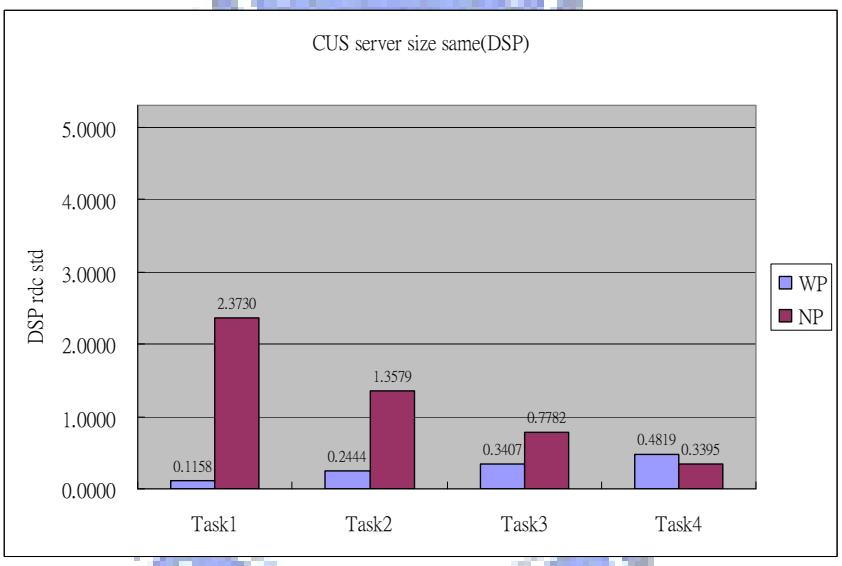


Fig. 18 DSP subtask RDC STD - same CUS server size

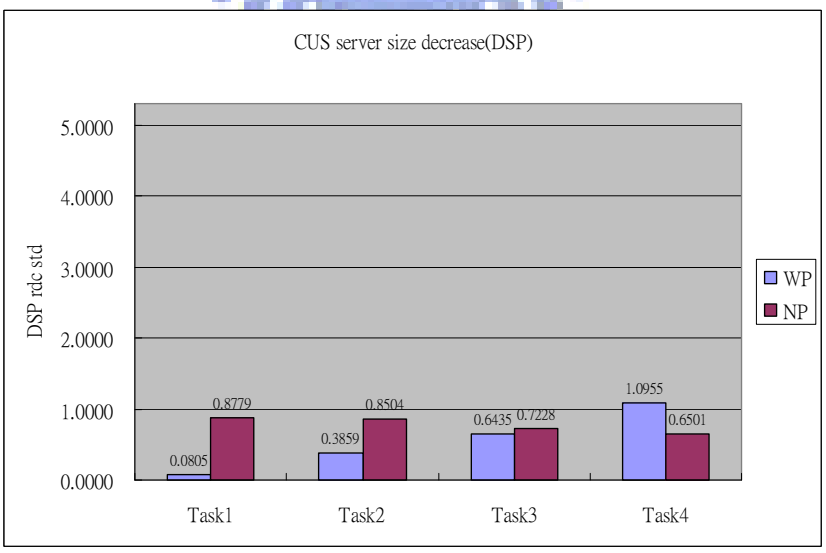


Fig. 19 DSP subtask RDC STD - decreasing CUS server size

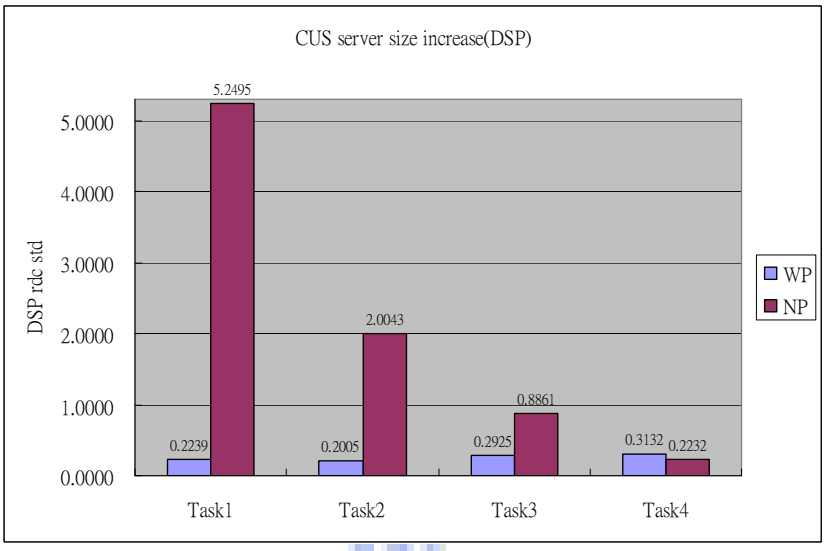


Fig. 20 DSP subtask RDC STD - increasing CUS server size

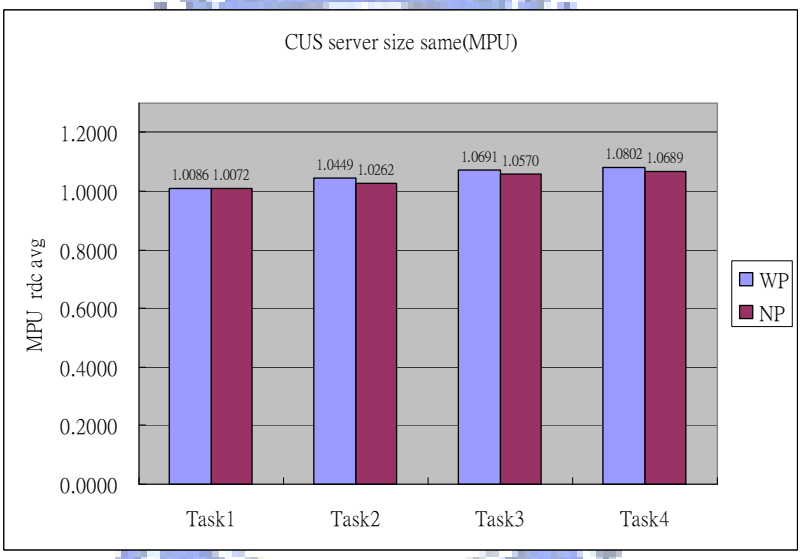


Fig. 21 MPU subtask RDC AVG - same CUS server size

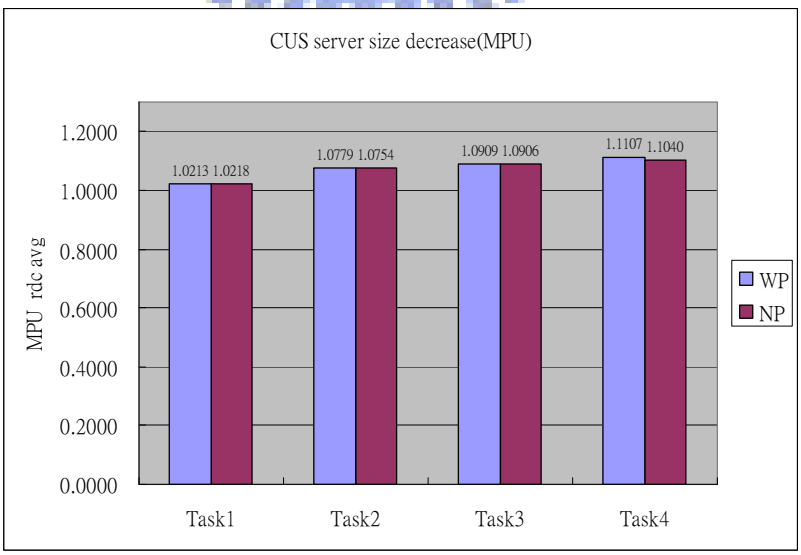


Fig. 22 MPU subtask RDC AVG - decreasing CUS server size

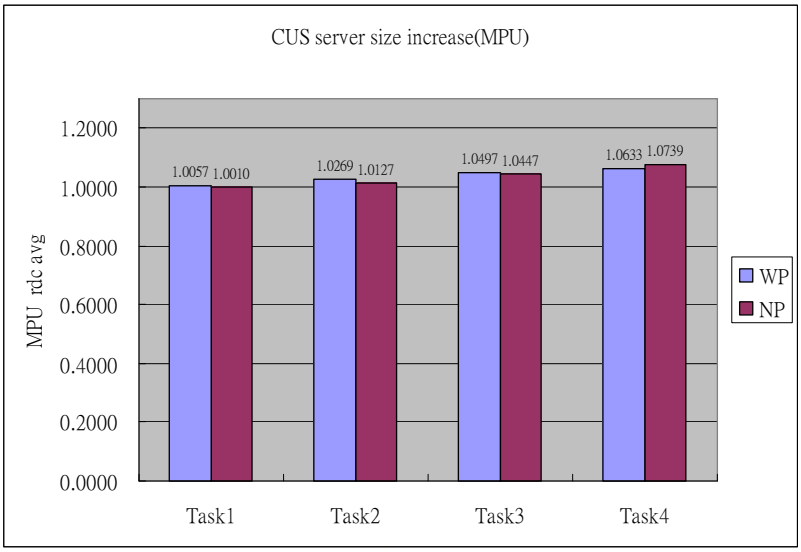


Fig. 23 MPU subtask RDC AVG - increasing CUS server size

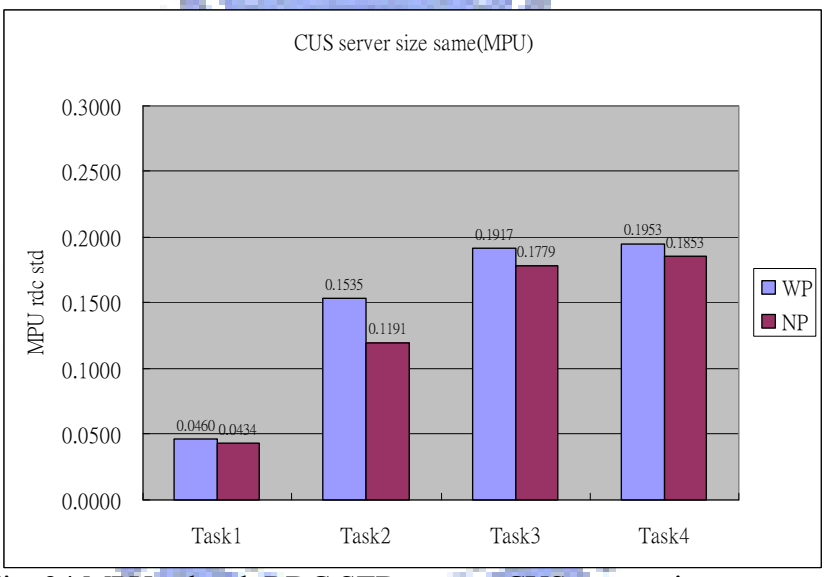


Fig. 24 MPU subtask RDC STD – same CUS server size

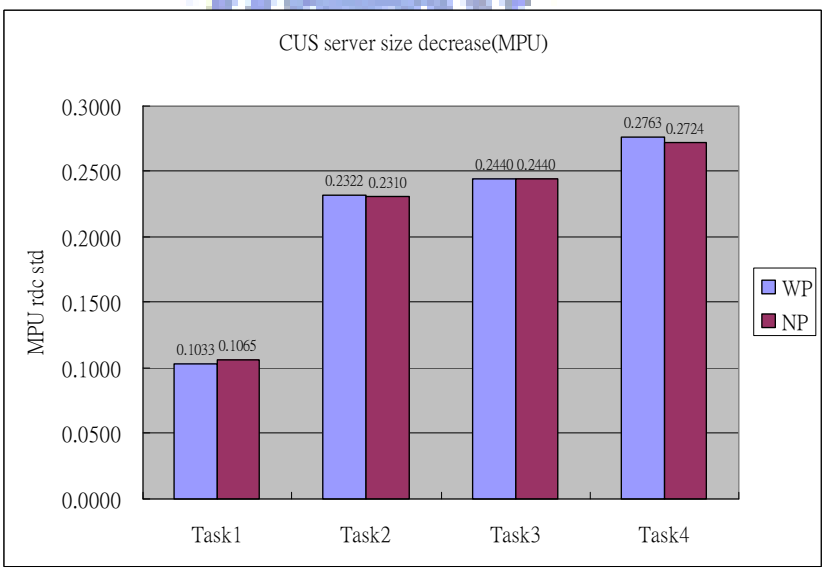


Fig. 25 MPU subtask RDC STD – decreasing CUS server size

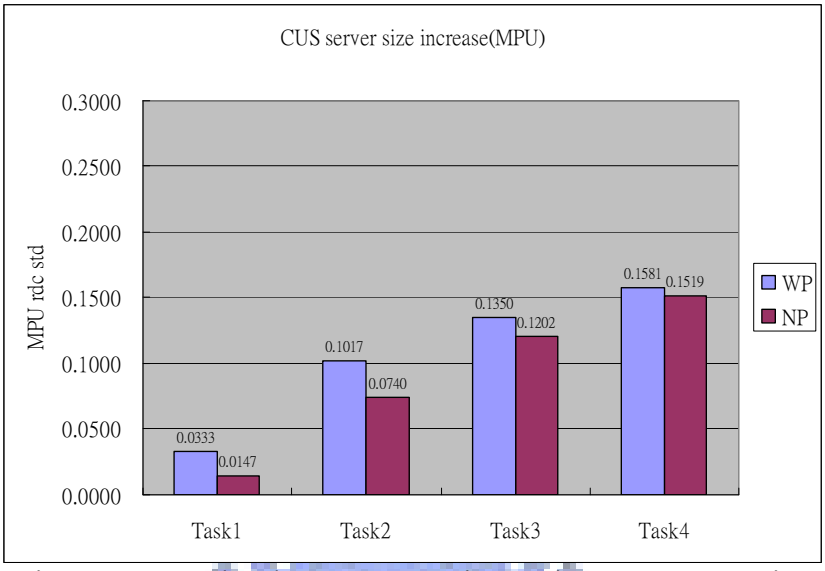


Fig. 26 MPU subtask RDC STD – increasing CUS server size

由 Fig.15,16,17 這三張圖表我們可以看出，WP 的 DSP subtask RDC 平均值會隨著 Task number 呈現遞增的現象，NP 的 DSP subtask RDC 平均值會隨著 Task number 呈現遞減的現象。這個原因在於因爲週期大的 Task 通常具有較大的 computation time，所以在 WP 的情形下，Task number 小的 Task 當他進入 DSP 時，通常可以得到較小的 local deadline，加上有加入 preemption point，所以當時間點抵達 preemption point 的時候，容易就會發生 preemption，使得 Task number 小的 Task 總是可以比較快完成，所以 RDC 值會比較小，因此 RDC 的平均值是隨著 Task 的週期做遞增。在 NP 的情形下，因爲不加入 preemption point，所以當一個 Task τ_1 進入 DSP 時，如果已經有其他的 Task 在使用 DSP，就必須等到該 Task 執行完畢，τ_1 才可以開始在 DSP 執行。對於週期較小的 Task 1 而言，當他進入 DSP 被 block 時，付出的代價會很大，這是因爲正在 DSP 執行的 Task 其執行時間很大，只要 Task 1 被擋住，response time 增加的時間相較於 Task 1 的 computation time 來說相當的大，所以 RDC 的數值就會提高很多。但是對於週期較大的 Task **4** 而言，當他進入 DSP 被 block 時，RDC 付出的代價比較小，因爲正在 DSP 執行的 Task **1** 或是 Task2 其執行時間相較於 Task 4 的 computation time 來得小，所以 RDC 的數值不會增加很多，因此 NP 的 RDC 值會隨著週期呈現遞減的趨勢。

比較 WP 在 Fig. 15,16,17 的 Task 1 可以看出，以 Fig. 15 爲基準，當 Task 1CUS server size 變大，由 Fig. 16 可以看出，Task 1 的 response 變好 (1.135->1.0707)，但是 Task 4 付出了代價（2.5093->3.2522）。當 Task 1CUS server size 變小，由 Fig. 17 可以看出，Task 1 的 response 變差(1.135->1.2050)，，但是 Task 4 response 變好了 （2.5093->2.0410）。由這個結果我們可以看出當週期小的 Task 擁有較大的 CUS server size 時，比較能夠搶到 DSP，response 效果較好，

不過週期大的 Task 必須付出代價，要不是搶不到 DSP，就是在 DSP 執行到一半，就被週期小的 Task preempt，response 效果較差。事實上，Fig. 15,16,17 NP 的部份也有這樣的趨勢。

在 Fig. 18,19,20 可以看到有加入 preemption point 的 DSP subtask RDC 標準差相較於沒有加入 preemption point 的 RDC 標準差會小很多，這代表有加入 preemption point 的 case，每一個 DSP subtask 其 response 的時間差異性不會很大，不會忽快忽慢，response 比較能夠預期，這在排程上是很重要的事情。

在 Fig. 21,22,23 可以看出不管有沒有加入 preemption point，MPU subtask 的 RDC 平均值幾乎都是 1，這是因爲 MPU subtask 的 computation time 較短，在 MPU 較不容易發生 resource contention，每一個 MPU subtask 進入 MPU 幾乎可以立刻開始執行。同時，在 Fig. 24,25,26 可以看出 MPU subtask 的 RDC 標準差很小，Task 的 RDC 變異性不大。

Section 5 Conclusion

這分研究最主要的目的是在滿足 precedence constraint 的條件下，降低每一個 Task 在 DSP 的 response delay；藉由在 DSP 的排程加入 preemption point，使得最 urgent 的 Task (Task with smallest deadline)可以優先被排程，減少在 DSP pending 的時間，避免 miss deadline，同時提高系統的可排程性(schedulability)。藉由我們的數據分析中可以看出，在加入 Preemption Point 的條件下，週期小的 task(always comes with small deadline)的 RDC 的平均值及標準差會比沒有加入 Preemption Point 小很多，代表 response delay 有所下降，而且變異性不大。利用我們設計在 MPU 及 DSP 的排程方法，除了可以滿足 precedence constraint，更可以預測每一個 subtask 的 worst response time。此外，我們所提出在 MPU 及 DSP 的 admission control 方法，可以檢查加入一個 on-line Task 系統是否存在一個 feasible schedule。

References

- [1] R.M. Ramanathan,” Intel Multi-Core Processors Making the Move to Quad-Core and Beyond” , Intel Corporation.
- [2] Robert Oshana, DSP Software Development Techniques for Embedded and Real-Time Systems, Newnes, USA, 2006, p.xii
- [3] S.Baruah, J. Goossens, and G.Lipari. Implementing constant-bandwidth servers upon multiprocessor platforms. In Proc. 8th IEEE Real-Time and Embedded

Technology and applications Symposium, pages 154-163, San Jose, CA, September 2002.

- [4] ARM926EJ-S, http://www.arm.com/products/CPUs/ARM926EJ-S.html
- [5] Jennifer Eyre, Jeff Bier, ” The Evolution of DSP Processors” , Berkeley Design Technology, Inc.
- [6]C. Lee, M. Potkonjak, and W.Wolf. System-level synthesis of application specific systems using a* search and generalized force-directed heuristics. In Proceedings of the 9th Intern. Symposium on System Synthesis, 1996.
- [7] H. Oh and S. Ha. Memory-optimized software synthesis from dataflow program graphs with large data samples.
- EURASIP Journal on Applied Signal Processing, pages 514 – 529, 2003.
- [8]Y. Cho et al. Scheduler implementation in MP SoC design. In Proceedings of the conference on Asia South Pacific Design Automation Conference, 2005.
- [9] L.-F. Leung, C.-Y. Tsui, and W.-H. Ki. “Minimizing energy consumption of multiple-processors-core systems with simultaneous task allocation, scheduling and voltage assignment.” In ASPDAC, 2004.
- [10] M. Ruggieroy et al. “Communication aware allocation and scheduling framework for stream oriented multi-processor systems on chip.” In DATE ,2006.
- [11] Ya-Shu Chen et al. “Dynamic Task Scheduling and Processing Element Allocation for Multi-Function SoCs”

Appendix

1. 針對 JPEG Encoding 的幾個步驟包含 FDCT, Quantization, DPCM, Encode 這四個步驟，我們在 X86 的系統上，利用 RDTSC 這個指令去計算這四個步驟執行時間的比例關係：

	FDCT	Quantization	DPCM	Encode
Clock cycles	7752	7514	153	12818

Table:3 The clock cycles of each step in JPEG encoding

2. 我們的平台上的 D-cache 是 4-way associative，D-cache 大小爲 16KB，type 爲 write back，replacement policy 爲 Round Robin。我們去比較 cache 有經過污染和沒有經過污染在 cache hit/miss 次數以及 CPU 執行 cycles 的差別。

```
for(i=0;i<500;i++)
 a1[i]=i;
for(i=0;i<10000;i++)
 a4[i]=i;
for(i=0;i<10000;i++)
 a6[i]=i+1;
for(i=0;i<10000;i++)
 a7[i]=a6[i];

start=Timer1Value;

for(i=0;i<500;i++)
  a2[i]=a1[i];

for(i=0;i<500;i++)
  a3[i]=a1[i];

for(i=0;i<5000;i++)
  a5[i]=a4[i];


end=Timer1Value;
```

Fig. 27 cache is not polluted

```
for(i=0;i<500;i++)
 a1[i]=i;
for(i=0;i<10000;i++)
 a4[i]=i;
for(i=0;i<10000;i++)
 a6[i]=i+1;
for(i=0;i<10000;i++)
 a7[i]=a6[i];

start=Timer1Value;

for(i=0;i<500;i++)
  a2[i]=a1[i];

for(i=0;i<5000;i++)
  a5[i]=a4[i];

for(i=0;i<500;i++)
  a3[i]=a1[i];


end=Timer1Value;
```

Fig. 28 cache is polluted

在 Fig. 27 的程式碼可以看到當 a2 陣列讀取 a1 陣列後，a3 再次去讀取 a1，此時 a1 的內容仍然存在於 cache 之中，所以 a3 可以直接在 cache 裡頭讀取 a1 陣列的內容。在 Fig. 28 的程式碼可以看到當 a2 陣列讀取 a1 陣列後 a5 去讀取 a4 陣列的內容，因爲 a4 的內容很大，a1 在 cache 之中的內容會被 replace，當 a3 要去讀取 a1 陣列時，會發生 cache miss，需要去 memory 裡頭重新 load data。

	Cache is not polluted	Cache is polluted
Data cache read hit	1704	732
Data cache read miss	2043	2164
Number of core clocks	738964	739688

Table 4: The comparison between cache is polluted or not.