國立交通大學

資訊科學與工程研究所

碩 士 論 文

在異質雙核心處理器平台上研究實作DSP排程器及動態精細分工的H.264編碼器

Design and Analysis of a DSP Scheduler and a Dynamically Partitioned H.264 Encoder on Dual-Core Platforms

研 究 生：蘇郁淵

指導教授：蔡淳仁　教授

中 華 民 國　九 十 五　年 六 月

在異質雙核心處理器平台上研究實作 DSP 排程器及動態精細分工的 H.264 編碼器

Design and Analysis of a DSP Scheduler and a Dynamically Partitioned H.264 Encoder on Dual-core Platforms

研 究 生：蘇郁淵　　Student：Yu-Yuan Su

指導教授：蔡淳仁　　Advisor：Chun-Jen Tsai

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of Master

in

Computer Science

June 2006

Hsinchu, Taiwan, Republic of China

中華民國九十五年六月

摘要

本論文的研究重點，是在異質雙核心平台上以視訊編碼爲例，研究動態精細工作分配法。視訊編碼演算法所需的計算量非常大。過去業界解決異質雙核心的工作切割(partition)問題的方法，是以 profiling 的方式來測量一個應用程式的每一個函式在 RISC 核心和 DSP 核心上運行的效率，然後再決定哪些函式要在 RISC 或 DSP 執行。因爲 DSP 核心是爲訊號處理運算而設計，使用 DSP 核心來運算會比 RISC 核心的效率好，因此往往系統最終的設計會將多媒體或通訊實體層的工作由 DSP 核心負責，而 RISC 核心負責流程控制和資料整合，這樣的做法，是屬於靜態工作分配法。可是未來的多媒體系統會同時執行多項工作，當 DSP 核心上實際運行的工作數量越多，由傳統靜態分工所設計出的應用程式只會一味地增加 DSP 核心負擔。所以要在異質雙核心平台上達到最佳效能，不能只考慮應用程式的計算特性的，還必須考慮到各核心實際運行的負載。因此本論文採用動態精細分工的方式來設計系統，讓兩個核心互相協調溝通，根據實際運行的負載動態地分配工作給各個核心。本論文並在 OMAP 5912 平台上，基於這個架構實作出一個 DSP 排程器及 H.264 視訊編碼器的應用程式以驗證動態精細分工系統的效能。

Abstract

Most modern embedded multimedia devices today are built upon heterogeneous multiprocessor (HMP) platforms. For such systems, a common practice for the industry is to perform static task partition during development time. However, due to the dynamic nature of new generations of multimedia devices, this method can not reach optimal performance when the runtime system state is different from the assumed static state at development time. The research direction of this thesis is to study a new fine-granularity dynamic partitioning/scheduling framework for HMP, where the partitioning/scheduling decision of a computationally intensive task is done at runtime. A DSP scheduler that can support this framework is developed in this thesis. In addition, an H.264 intra-frame encoder is implemented following this dynamic partitioning/scheduling paradigm. Experimental results show that, with minimal programming effort, the proposed system can outperform a pure GPP or DSP implementation when there is only one task, and even a statically-partitioned dual-core solution when there are multiple tasks.

章節目錄

List of Figures

List of Tables

1. 簡介

本論文主要的內容是研究在異質雙核心平台下要如何能有效率地同時運行多個視訊編碼工作。在嵌入式的環境下異質雙核心平台常用來處理視訊編碼這種大量運算的演算法，它是由一個負責介面操作和流程控制的 RISC 和一個適合多媒體運算的 DSP 所組成。但隨著嵌入式的多媒體應用演進，現在同時處理多個多媒體工作的需求，已越來越普遍，如視訊電話，同時會需要處理無線通訊協定、視訊解碼、視訊編碼。以傳統的靜態工作分配的方式設計的系統，在這種狀況下，會把多媒體或通訊相關工作不斷分配給 DSP，常常會使得 DSP 過於忙碌。而 RISC 發展至今，其運算能力也有大幅的提升，也有足夠的能力處理許多訊號處理的演算法。因此如果新的多媒體系統能把 RISC 和 DSP 以動態分工(dynamic task partitioning)的方式設計，讓每一項新加入的資料處理工作，視兩個計算核心的實際負載，分配給適當的核心，將能提昇系統的效能。

本論文實驗所採用的系統平台是 TI 的 OSK5912 發展平台[21]，其核心 OMAP5912 是由 ARM 和 DSP 所組成的異質雙核心架構[20]；而測試用的視訊編碼演算法是 H.264 的 Intra frame 編碼器，這是新一代的視訊壓縮標準，相對於過去的視訊壓縮標準，有著在同資料量下能達到更好的影像水準，也就是有著更好壓縮效率，但運算複雜度也跟著相當的高，要在這環境下要實現動態精細工作分配法，必需考慮到兩個核心之間的溝通協調和資料傳輸所造成的計算成本的增加，另外在 DSP 端需要有一個簡單，但高效率的排程器。而這個排程器有下面這 4 個要求：

1) 低 overhead 的多工排程

2) 快速回應溝通協調的命令

3) 有效率的資料傳輸

4) 能由 RISC 端動態控制改變 DSP 所能提供的服務。

爲了符合第一項要求，本論文是以單純的時間共用（time-sharing）的方式來設計工作排程器，並觀察在 DSP 核心上的 H.264 編碼器的運作方式，設計出 low overhead 的 context switch 機制。而在緊密結合(Tightly-coupled)的模式下執行的視訊編碼程式，必定有許多溝通協調和資料傳輸需求。OMAP5912 提供了 mailbox、MPUI 和 shared memory 三種溝通方式。爲了快速回應溝通協調的要求，本論文針對 mailbox 的 ISR 會使用到的暫存器設計出特別的架構，以改善進出 ISR 前後的 context-saving 和 context-restore 所需的時間。爲了設計出有效率的資料傳輸 API 的架構，本論文也深入測試分析直接存取 shared memory，或用 DMA 來存取的效率上的差異。另外，因爲 DSP 核心的內部記憶體容量並不大，所以同時有多個工作要運行的話必定有其上限，因此就有需要能動態地抽換 DSP 核心支援的視訊編碼服務。爲了設計出一個執行檔格式來同時包裝 RISC 和 DSP 的二元程式碼，我們分析 DSP 核心之 compiler 產生的連結檔和可執行檔，並寫了一個工具來將連結檔中所參考到的程式庫都抽取出來，建立出可執行映像檔的必要資訊，並把映像檔存放在 OMAP5912 的 Flash 記憶體中，讓我們所設計的 OS 核心內的動態服務註冊器能隨著應用程式的需求，去動態抽換 DSP 核心所支援的視訊編碼服務。

本論文下面幾個章節的組織如下。首先，在第二章，我們會先對相關的研究進行討論。接下來，在第三章，我們會對系統的設計概念和背景做一個介紹。第四章則是描述實作的細節，而第五章則是討論實驗的結果。最後，在第六章會對一些未來可能進行的系統改進方式進行討論。

2. 相關研究

隨著各種網路系統的迅速發展，每秒頻寬不斷的提升，網路應用也從早期以文字爲主的應用，進步到包含豐富圖形的應用，現在更已經達到音效視訊內容十分普及的程度。線上收聽音樂，網路電視即時轉播，甚至是視訊電話，已經是嵌入式設備的基本需求，所以多媒體運算的性能需求也越來越高。目前市場上的行動式嵌入系統開發設計，都是以異質雙核心平台爲主流，由一個負責介面流程控制的 RISC 核心，和一顆爲多媒體運算特別設計的 DSP 核心所組成。在本章中，我們會討論幾個異質多核心平台的分工和排程系統的研究。

2.1. Tightly-coupled MPEG-4 Video Encoder Framework

由邱正男等人所提出的一個專爲異質雙核心平台設計的以 Tightly-coupled 的方式運行 MPEG-4 的視訊編碼的架構[1]，相對於傳統都以靜態的 partition 方式，來決定各核心上運行的工作分配，例如：DSP 執行多媒體編解碼或通訊實體層的計算工作，RISC 則是負責控制管理。但隨著 RISC 設計的演進，新一代的 RISC 核心也能夠進行計算量相當大的運算，因此這裡提出了一個將視訊編碼工作以緊密結合(Tightly-coupled)協同運算的方式，同時分配視訊編碼工作給 RISC 和 DSP 核心一起協同運算，如 Figure 1。以 OMAP1510 的硬體平台[2]運行 MPEG-4 Simple Profile[3]的視訊編碼器爲測試基準，以緊密結合的方式其編碼速度比完全交由 DSP 運行還快。

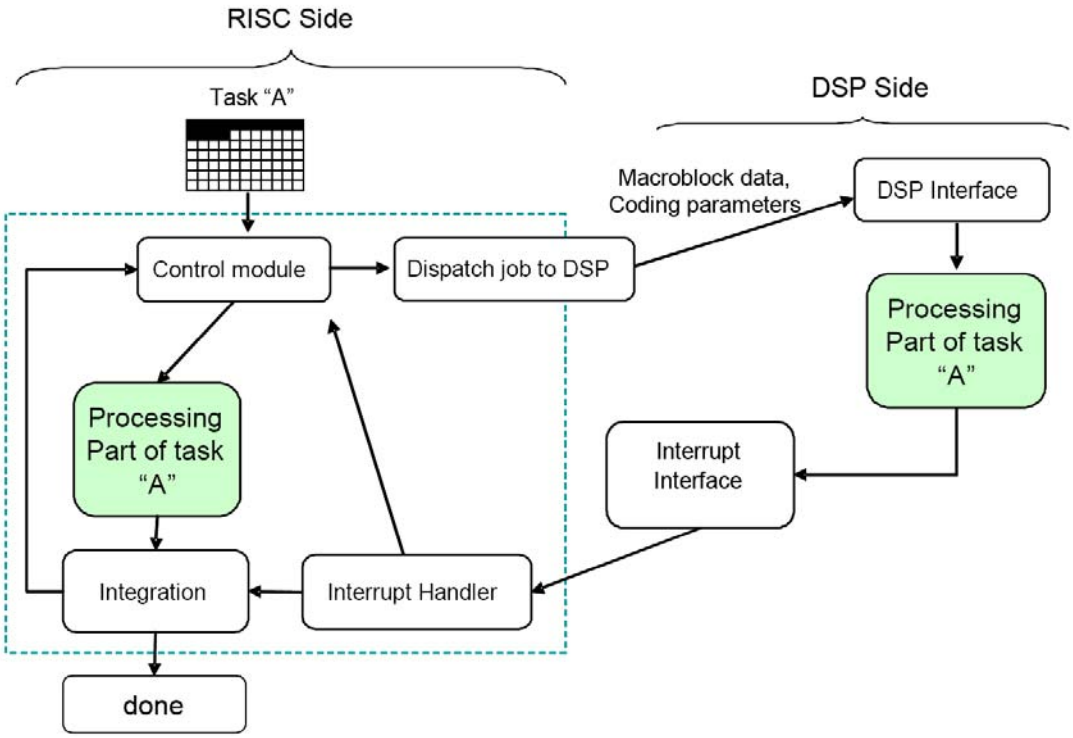


Figure 1. A tightly-coupled dual-core system.

2.2. Dynamic Concurrent Tasks Management

由 Lauwereins 等人所提出的同時運行工作的動態管理[4]，因爲使用者操作的多樣，造成了複雜的功耗和 Timing 的 Constraint，它能達到最低功耗又能找到符合 Timing Constraint 的工作分配方式，主要是由兩個階段的 Schedule 方式，首先從各個工作中找出可能的同步運行的部分產生 MTG model[5]，並再進一步將 MTG model 轉化以降低複雜和提昇平行性，並靜態地測試在各個核心運行的所花費功耗和時間，產生運行時間對比功耗的 Pareto-optimal set，在 Runtime 時再根據靜態分析的結果，找出符合目前功耗和時間的工作分配。主要架構可以參考 Figure 2。

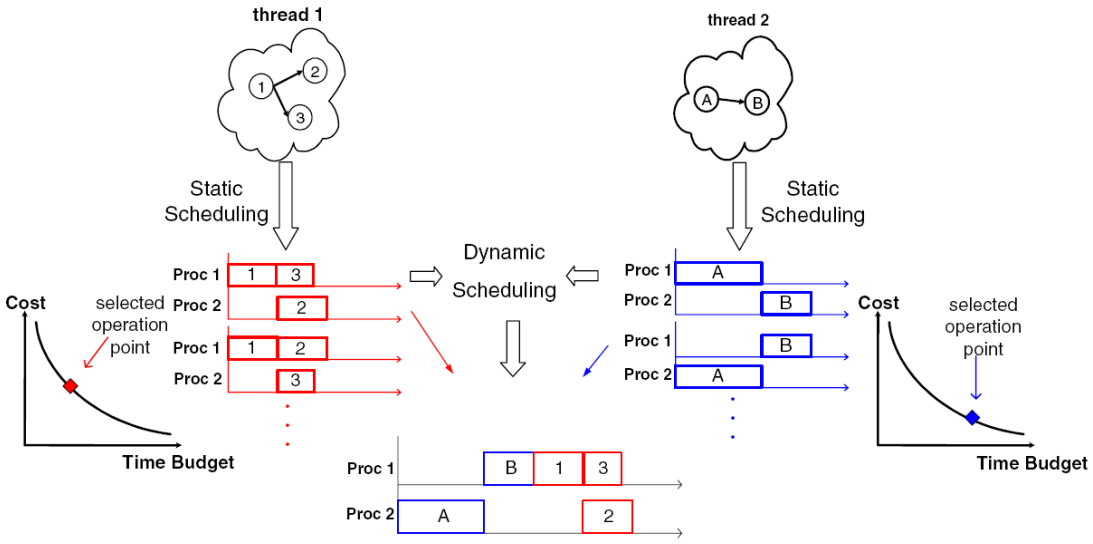


Figure 2. Two Phase Scheduling Method.

2.3. Dataflow Kernel

歐旭江針對異質雙核心平台設計了一個DSP Dataflow Kernel[6]，以資料流的概念，隨著進來的資料順序來對DSP上的Process排程，是一種Non-preemptive的排程方式。而實際的排程是由運行在RISC核心上的DSP Manager所決定，如Figure 3，主要是根據各個Process和Queue的資訊，來決定是否接受RISC核心上的Process之Request。因爲RISC比DSP更適合做這些判斷決策，這樣設計的優點可以簡化DSP Kernel複雜度，以降低提供DSP服務排程的Overhead。

並且 DSP Manager 提供了 DSP Process 對 RISC 發送 Request 的介面，使得 DSP 也可以使用到 RISC 的資源，另外 DSP Manager 還能動態抽換 DSP 服務，適合動態運算多種視訊編碼使用。

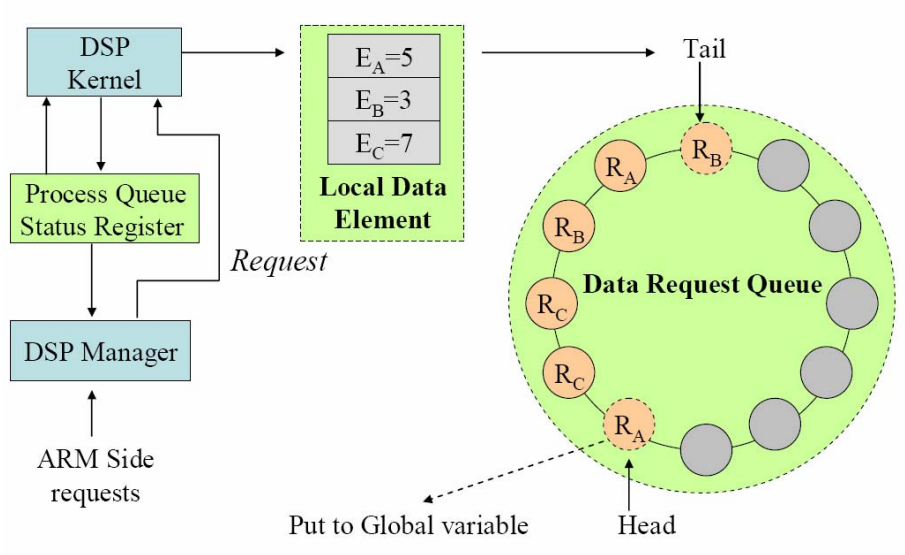


Figure 3. Scheduling Architecture.

2.4. Multi-core DSP Interface

Hung 等人所提出的一個針對異質多核心平台所使用的 compiler directive extension[7]。在嵌入式設備上，爲了運行多媒體應用，有許多設計都會使用 DSP 核心，甚至是異質多核心的架構，來達到高效能、低功耗、高彈性並且價格低的需求。可是隨著這樣的平台的複雜化，其運行的軟體設計就依賴 high-level 的開發工具來加速開發速度，但其慣用的 compiler 時常沒有完全利用到一些先進的硬體設計，因此並完全發揮出這個平台所能達到最佳的效能。

所以爲了在這樣異質多核心的平台下，達到軟體的移值和效能的最佳化，提出了一個可以算是標準 C/C++語言所延伸的註解語言，來指示 compiler 產生出有利用到 DSP 硬體的運算指令，多核心的調度，資料的平行處理，以達到這個平台應有的效能。

3. 系統架構與實作

由於 DSP 的特性並不適合做複雜控制的計算，並爲了使視訊編碼運算能發揮最高效率，所以在本論文中所設計的系統架構如 Figure 4 所示，會把 DSP 上 process 和記憶體空間管理，交由 ARM 這個比較適合控制管理的 RISC 來負責，而 DSP 的系統核心負責提供一個低複雜度的多工排程環境，以及和 RISC 核心間溝通所需要的 Mailbox ISR。

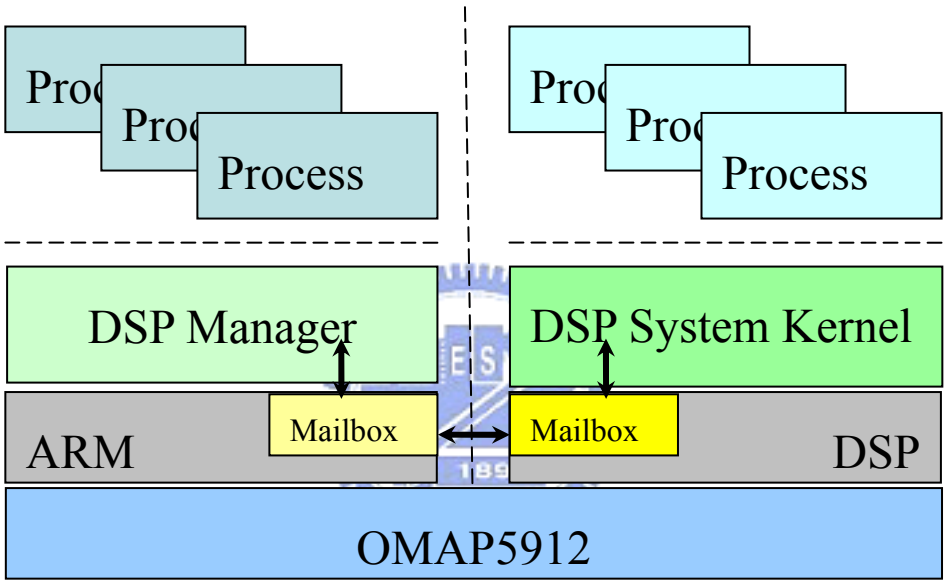


Figure 4. System Arhitecture.

3.1. DSP Manager

DSP Manager 的工作包含下列三個項目，會以 library 的形式，提供給 ARM 端開發者使用。

1) DSP 系統核心的啓動機制

2) DSP 記憶體空間管理

3) 動態服務管理

3.1.1. DSP 系統核心的啓動機制

在 DSP 上執行的系統核心之原始程式碼，經由 TI 的開發工具 CCS 編譯後，會產生一個執行檔。一般在系統開發階段，這個執行檔是由 CCS 經由 JTAG 來載入到 DSP 的程式記憶體空間。但對嵌入式系統而言，開發完後，要銷售的產品，並不會有 JTAG 這種介面來載入程式，大都是將程式的執行檔燒錄到 Flash 上，開機就能獨自運行。因此 DSP 的系統核心也需要提供類似的方式來啓動(boot)。Rishi Bhattacharya 提供一個 OMAP5910 的 DSP 開機範例[10]，將 DSP 的執行檔轉成一個常數資料陣列的宣告，和運行在 ARM 上的主程式一起編譯後，再由 ARM 的主程式經由 MPUI 將這個資料陣列(DSP 的系統核心)載到 DSP 的程式記憶體空間，並加以啓動，詳細步驟如 Figure 5。

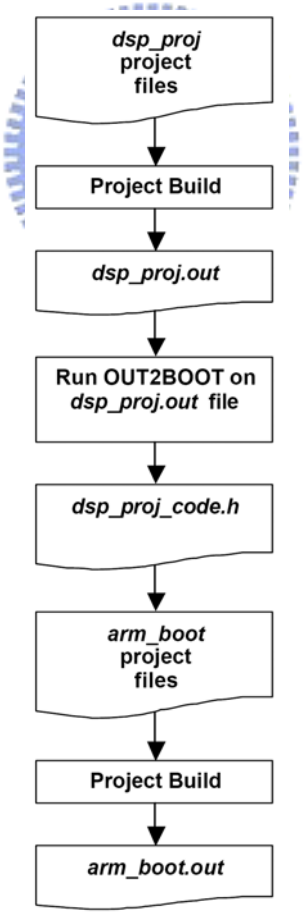


Figure 5. Bootloader Build Flowchart.

其中 OUT2BOOT，是隨[10]附上的程式。它會先用 CCS 裡的工具 Hex Conversion Utility(hex55.exe)， 詳細使用說明可以參照 TI 的文件[11]。將執行檔(.out 檔)轉成 Intel MCS-86 Object 的格式，再轉成 header 檔內的資料陣列宣告，詳細格式如 Figure 6。

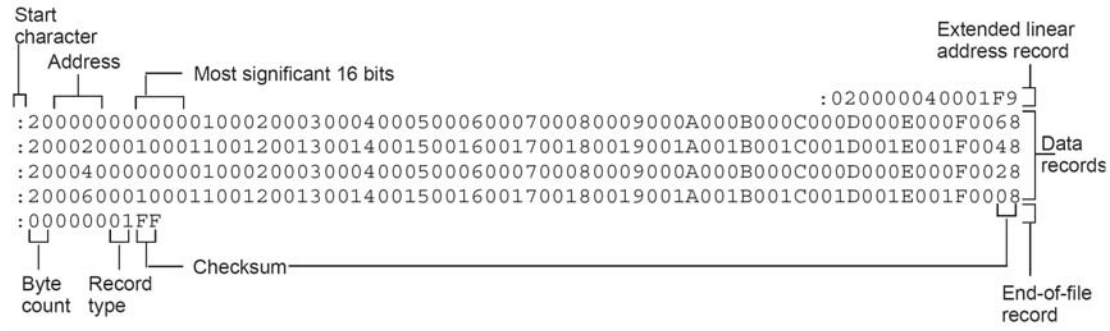


Figure 6. Intel Hex Object Format.

可以看到這種格式可記錄的 address 只有 16 bits，但由 ARM 的 MPUI 載入 DSP 的程式記憶體空間會需要 32 bits。因此被轉出來 header 檔，只固定從 0xE0010000 開始填寫到 0xE001FFFF，這也是對應到 DSP 內部的 SARAM 前 64Kbyte，以 DSP 程式記憶體空間來看是從 0x10000 的位置開始去填寫。會固定從 0x10000 開始，是根據 TI 的文件對 DSP Bootloader 的說明[12]。DSP Bootloader 做好基本的設定後，會根據 Boot 模式，跳到特定的起始位置，0x10000 是其中的 Internal memory boot 模式(5)的起始位置。爲了使編譯完的執行檔，其起始位置也是 0x10000，這需要對 CCS 的 linker 設定檔(.cmd 檔)做些修改，如 Figure 7，其詳細使用說明可以參照 TI 的文件[13]。強制設定負責初始化 C 語言環境的 boot.obj 連結檔，在 static link 時，會 link 成從 0x10000 開始載入，而 boot.obj 執行完 C 語言環境的初始化程序(c_int00 函式)，就會呼叫我們的 main 函式。

```
MEMORY {
            :
            :
      SARAM:    origin = 0x10000, len = 0x18000
            :
            :
}

SECTIONS
{
            :
            :
      .text: { rts55x.lib<boot.obj> (.text) } > SARAM
            :
            :
}
```

Figure 7. Starting Address Configuration.

但這樣的設計就限制了程式只能載入到 SARAM，DSP 內部另外有 DARAM，同時可以存取兩筆資料，如果我們把 DSP 系統核心的資料區域放在 DARAM，應能提升效率。如果執行檔有用到 DARAM，其轉出來的 Intel Object 會有 Recode type 爲 04 的 entry，代表要改變目前 24bit base address 的 high 8bit，但原本 OUT2BOOT 不支援，一處理到這種 entry 就會當掉。我們將 OUT2BOOT 提供的程式碼稍微修改後，並在轉成 header 檔時，加上把 DSP 程式記憶體空間的位置，轉成對應 MPUI 或 shared memory 的位置的功能，就可以解決這個問題，之後要 ARM 設定 DSP 核心系統的記憶體就比較簡單。

但照著[10]的 DSP 開機範例去啓動 DSP 後，DSP 還是不能正常運作，後來在 OSK5912 開發板提供給 CCS 的設定檔 osk5912.gel 裡面發現類似的函式。.gel 檔的作用，是讓 CCS 去對使用的核心初始化，會以類似[10]所描述的方式去啓動 DSP，但會設定更多的暫存器，而填到 SARAM 的 DSP 機械碼，只是不停執行一個無窮迴圈而已。參考.gel 檔來改寫原本 ARM 啓動 DSP 的 dsp_init 函式，這樣就可以很順利讓 DSP 啓動，並順利執行 ARM 根據 header 檔內的資料陣列所設定好在 SARAM 和 DARAM 內的 DSP 核心執行檔之內容。

3.1.2. DSP 記憶體管理

DSP 可以存取的記憶體除了內部的 DARAM 和 SARAM 外，經由 DSP MMU 的設定可啓用 shared memory，這樣 DSP 就可存取共用的 SRAM 和外部的 Flash 和 SDRAM。本論文所設計的系統，針對這些記憶體種類和動態分配的需要，以 linked-list 來實作記憶體管理，並提供了三個函式給系統核心及應用程式使用，如 Figure 8 所示。

```
typedef struct _dspmem
{
        uint32 base;
        uint32 size;
        int used;
        int type;
        struct _dspmem *next;
}dspmem;
```

```
void dspmem_init();
dspmem *dspmem_alloc(uint32 size, int type);
int dspmem_free(dspmem *free_dspmem)
```

Figure 8. DSP Memory Managerment functions.

目前系統規劃了五種不同類別的記憶體（DARAM、SARAM、SRAM、SDRAM、Flash），所以在 dspmem_init 函式裡，會對五個 linked-list 做初始化，設定可以使用的位置和大小。如 Figure 9，當系統或應用程式所註冊的服務要求 DSP 分配大小爲 0x2000 的 SARAM 的記憶體空間以供使用，DSP 記憶體管理程式會從 SARAM Head 開始尋找有足夠空間的節點，找到後修改它所佔用的大小、並標示爲已使用、最後新增一個節點記錄剩餘的空間。從 Figure 9 可以注意到每個節點的 Base address 是用 24 bits 來記錄，這是因爲透過 DSP 的記憶體管理程式分配到的記憶體空間，是要給 DSP 執行的服務所使用的，而 DSP 的資料位址最多有 24-bit 的空間。

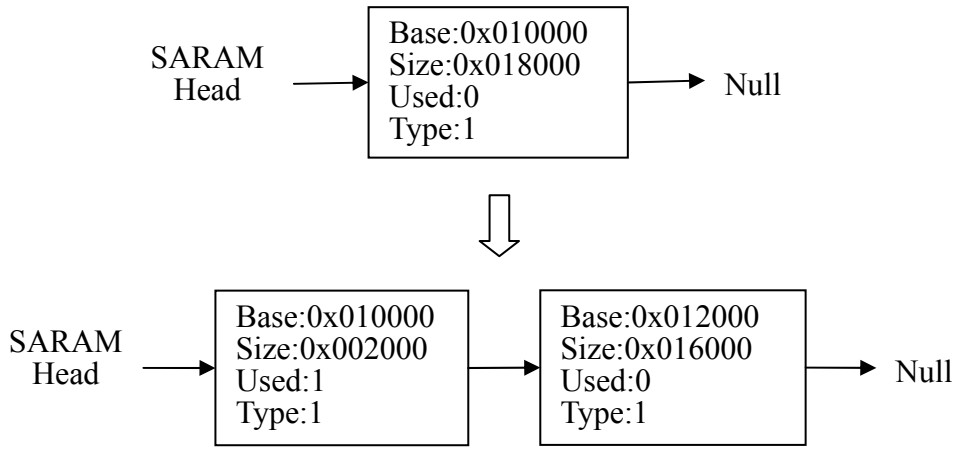


Figure 9. DSP Memory Allocation Example.

3.1.3. 動態服務註冊管理

爲了動態抽換 DSP 上的服務，在 DSP 上執行的每一個服務的執行檔就必需能（在註冊時）獨立載入。而且在載入時才能確定一些位址的值。也就是說，其機械碼內跟記憶體位址有關的值在動態註冊時才可以決定。而原本 CCS 所產生的執行檔是已經靜態指定執行位址的，並不支援動態載入(dynamic loading)，因此本系統的開發就需要研究 CCS 的編譯器產生之連結檔的格式。CCS 使用的連結檔格式是 common object file format，COFF。在系統開發的過程中，我們發展了一個程式，分析連結檔後將所有必要的資訊（如參考到的外部程式等等）連結進來，整合成一個映像檔，寫入到 OSK5912 的 Flash 上，其中，跟最終執行位址相關的資料則仍保留成未定的欄位，等應用程式要註冊這個服務時，再由 ARM 來處理。首先 ARM 會呼叫 DSP 記憶體管理程式，分配一塊 DSP 內部記憶體做爲載入服務的空間，根據分配到的記憶體起始位置做動態載入，經由 MPUI 這個介面載入到 DSP 內部記憶體，最後再向 DSP 系統核心註冊這個服務。

3.1.3.1. COFF file structure

根據 TI 文件對 COFF 檔的描述[14]，每次編譯後產生的連結檔包含了 File header、Section header、Section raw data、Section relocation information、Symbol table、String table。如 Figure 10 所示。

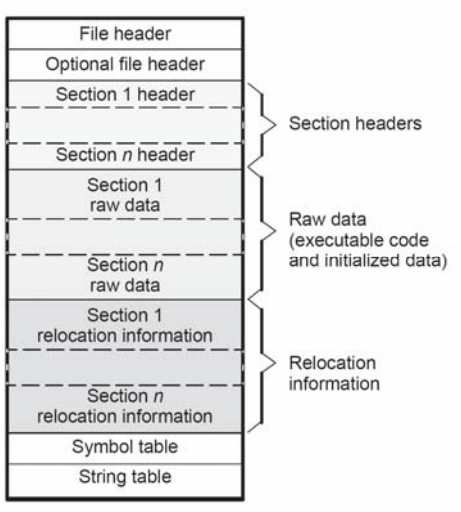


Figure 10. COFF File Structure.

其中 File header 如 Figure 11，可得知 Version ID、Section 個數，Symbol table 的位置和有多少筆 entry；而一般都不會有 Option header。File header 裡面 Version ID 的 2 個 byte 用 little-endian 方式讀取，一般都是 0x00C2，如果是 0xC200，則代表目前的連結檔是以 big-endian 編寫的。

Byte Number	Type	Description
0–1	Unsigned short integer	Version ID; indicates version of COFF file structure
2–3	Unsigned short integer	Number of section headers
4–7	Long integer	Time and date stamp; indicates when the file was created
8–11	Long integer	File pointer; contains the symbol table's starting address
12–15	Long integer	Number of entries in the symbol table
16–17	Unsigned short integer	Number of bytes in the optional header. This field is either 0 or 28; if it is 0, then there is no optional file header
18–19	Unsigned short integer	Flags (see Table A–2)
20–21	Unsigned short integer	Target ID; magic number indicates the file can be executed in a TMS320C55x™ system

Figure 11. File Header Contents.

COFF 檔裡的 Section，代表程式執行會需要的各類記憶體空間，可以分成兩類。

1) Initialized section:

其 raw data 有實際內容，如機械碼、const 變數。

2) Uninitialized section:

保留的記憶體空間，只有 Section 大小，在連結檔裡並沒有實質內容，例如未被初始過的 global 變數等，只是用來告知 linker 要爲其保留記憶體空間。

Section header 的格式如 Figure 12，會需要用到 Section 的名稱、Size、Raw data 位置、Relocation information 位置和個數。名稱的欄位最多只能存 8 個字元，所以字數大於 8 的名稱，會記錄著 String table 裡的 offset，在 String table 裡就會有完整的名稱。之後所有用到名稱的欄位都是以這種方式來記錄。如果是 Initialized section，Size 和 Raw data 位置的欄位都會有非零的值；Uninitialized section 則只有 Size 欄位有非零的值，Raw data 位置和 Relocation information 都會是零。如果 Section 有 Relocation information，則代表其 Raw data 裡有 Virtual address，需要在 link 時被重新定位(relocate)爲 Physical address。

Byte	Type	Description
0–7	Character	This field contains one of the following: 1) An 8-character section name, padded with nulls 2) A pointer into the string table if the section name is longer than 8 characters
8–11	Long integer	Section's physical address
12–15	Long integer	Section's virtual address
16–19	Long integer	Section size in bytes
20–23	Long integer	File pointer to raw data
24–27	Long integer	File pointer to relocation entries
28–31	Long integer	Reserved
32–35	Unsigned long	Number of relocation entries
36–39	Unsigned long	Reserved
40–43	Unsigned long	Flags (see Table A–5)
44–45	Short	Reserved
46–47	Unsigned short	Memory page number

Figure 12. Section Header Contents.

Section 的名稱是很重要的欄位，CCS 爲 DSP 所產生的 COFF 檔會有 Table 1 所列的這幾種 section。

Table 1. COFF File Sections.

名稱	Initialized	存放內容／用途-
.text	是	機械碼
.data	是	assembly 語言的變數初始值
.bss	否	未被初始過的 global 變數
.const	是	const 變數
.switch	是	switch statement label table
.cinit	是	C 語言的 global 變數初始值 table
.pinit	是	C++語言的 object constructor table
.cio	否	C 語言的 IO 暫存空間
.stack	否	主要的系統 stack 記憶體
.sysstack	否	第二個系統 stack 記憶體
..sysmem	否	heap 記憶體

而實際在 release mode 編譯 H.264 Intra frame 編碼器所產生的 COFF 檔，只會用到.text、.bss、.const、.switch、.cinit、.sysmem，這幾個 Section。而其中.cinit section 是在 setup C 語言環境時，根據 Raw data 裡的 Auto initialization table 設定 global 變數的初始值，也就是要填初始值到.bss section 的所保留的空間，因此這兩個 section 是可以整合爲一個 section，另外.sysmem 是只有 C 語言內建的 malloc 和 free 函式會用到，會特別再做處理。所以分析 COFF 檔後，最後只會留下.text、.bss、.const、.switch，這四個 section 在映像檔裡。

Relocation information 是爲了重新定位 Symbol 的 Virtual address 到 physical address 所用的資訊，就是如何做到動態載入最主要的資訊。格式如 Figure 13。會需要用到 Virtual address、Symbol table index 和 Relocation type。這個 Virtual address 是指被重新定位的 Symbol 在 Raw data 裡的 offset，經由這個 offset 可以讀到 Relocatable field，即 Symbol 的 Virtual address。

Byte Number	Type	Description
0–3	Long integer	Virtual address of the reference
4–7	Unsigned long integer	Symbol table index
8–9	Unsigned short integer	Additional byte used for extended address calculations
10–11	Unsigned short integer	Relocation type (see Table A–7)

Figure 13. Relocation Entry Contents

Symbol table 裡的 Symbol 格式如 Figure 14，會需要用到名稱、Value、Section number 和 Storage class。Section number 代表這個 Symbol 屬於哪個 Section，如果 Section number 不是 0，代表這個 Symbol 是 Internal Symbol，再根據 Storage class，分別是 Global 或是 Static，如果是 Global，可以被其他連結檔 Reference 的 Symbol，如果是 Static，則是目前連結檔裡的 Relocation information 會到的 Symbol。

如果 Section number 是 0，代表這個 Symbol 不在目前的連結檔，而是 Reference 其他連結檔的 Symbol，屬於 External Symbol。所以要由連結檔建立映像檔之前，如果連結檔裡要重新定位的 Symbol 屬於 External Symbol，則必需先 Resolve 這些 External Symbol，當發現 External Symbol 在某個連結檔裡，這個連結檔也要一起被建立進映像檔裡。

Byte Number	Type	Description
0–7	Character	This field contains one of the following: 1) An 8-character symbol name, padded with nulls 2) A pointer into the string table if the symbol name is longer than 8 characters
8–11	Long integer	Symbol value; storage class dependent
12–13	Short integer	Section number of the symbol
14–15	Unsigned short integer	Reserved
16	Character	Storage class of the symbol
17	Character	Number of auxiliary entries (always 0 or 1)

Figure 14. Symbol Table Entry Contents

因爲 TI 的文件[14]並沒有詳細說明，每一類 Relocation information，要如何重新定位。爲了能在動態重建執行檔的資訊，因此以反向工程的方式，把原本連結檔裡的 16 進位值和在 CCS 下開啓執行檔相互對照，大略可以把在 Relocation information 所記錄的資訊，分成 Table 2 中四類。

Table 2. Relocation Information

名稱	機械碼內存放 Physical address 的暫存器	Virtual address of	Reference Symbol Name
Internal AR	ARn	global 變數、const 變數	.bss/.const/.sysmem
Internal AC	ACn	global 變數、const 變數、switch statement label table	.text/.bss/ .const/.switch
External AR	ARn	別的 object 檔的 global 變數、const 變數、function	function/variable name
External AC	ACn	別的 object 檔的 global 變數、const 變數	__SYSMEM_SIZE

其中會使用 ARn 或 ACn 哪一種暫存器是根據 physical address 被讀出後的 offset 計算有關，如只是 scalar 變數和一維陣列，通常是會讀到 ARn，而多維的陣列或是陣列的 index 會經由複雜的運算而得出的，則會讀到 ACn；switch statement 如果比較複雜的將會使用 label table，而這 table 的 physical address 都會載入到 ACn，而 external function 的 address 則都是使用 ARn。

ARn 類的 relocation information 會用有一筆 3byte 的 Relocatable field，載入 23bit 的值到 ARn 暫存器。AC 類的 relocation information 則會用兩筆 2byte 的 Relocatable field，以組成 32bit 的值載入到 ACn 暫存器。

3.1.3.2. Virtual Address 的取得

重新定位所需要的 Virtual address 取得的方式可分兩種

1) Internal：

Symbol 位於目前連結檔的某個 Section 裡，經由 Relocation information 可以在 Raw data 裡找到對應的 Relocatable field，Symbol 的 Virtual address 會在 Relocatable field 上。

2) External：

Symbol 位於其他連結檔的某個 Section 裡，這就需要 Resolve 這個 Symbol。到其他連結檔裡的 Symbol table 去找這個 Symbol 名稱，如果找到，則其 Value 欄位代表著 Symbol 在 Section number 對應 Section 裡的 Virtual address。但這會是 byte address，如果 Symbol 不屬於 function，就是不屬於程式記憶體空間的位址，必須再除 2 得到資料記憶體空間用的 word address(16bit address)。

3.1.3.3. Dual-core Service Image File Format

我們從各連結檔中抽取最重要的資料，並考慮 Relocation information 的種類和 Virtual address 取得的方式，設計了如 Figure 15 的映像檔格式。還多保留了 ARM 的執行檔的欄位，是因爲我們的目標是能將同一個函式，編譯成包含兩個核心執行檔的映像檔，這樣就能隨著實際各核心上的負擔，動態載入執行檔到較低負載的核心來運行

ARM executable
DSP object number
Section size header
Section raw data
Sysmem information
Relocation information
Main/Initial entry point

Figure 15. Dual-core Service Image File Format

DSP object number 欄位，標示這個映像檔中 DSP 的執行檔由有多少個連結檔組成。每個連結檔只會留下 4 種 Section，如 Figure 16，首先在 Section size header 標示每個物件各 Section 的大小

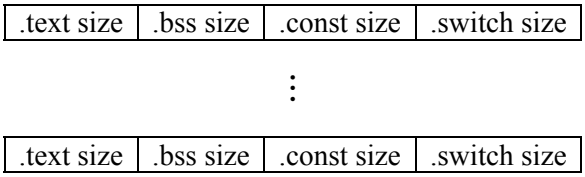


Figure 16. Section Size Header Contents

如 Figure 17，在 Section raw data 裡會有每個物件各 Section 裡的實際內容。並且設計成連續排列的，方便載入到 DSP 程式記憶體空間。

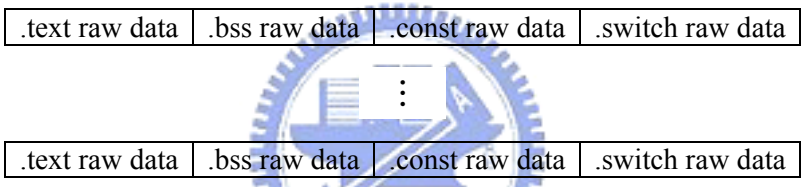


Figure 17. Section Raw Data Contents

Sysmem information 是專爲 malloc 和 free 這類記憶體管理的函式特別設計的。如 Figure 18，它們會用到 memory.obj 和_lock.obj 這兩個連結檔，memory.obj 有一個特別的 Relocation information，以 External AC 的方式來設定 Sysmem 的大小；而_lock.obj 裡另外有一個特別的 Relocation information，會在.bss 重新定位一個 macro function 的位置。因爲這兩種 Relocation information，只有這邊用到，所以特別處理。

Figure 18. Sysmem Information Contents

之前提到的四類 Relocation information，扣掉設定 Sysmem 大小的 External AC，就只留下這三類，External AR、Internal AR、Internal AC。如 Figure 19，在映像檔的 Relocation information 欄位，每個物件都會有它自己的這三類 Relocation information，全部都是用來修改.text 的 Raw data。每一類會先標示其數目，再接著每筆重新定位所需的資訊。

External AR number	External AR information	Internal AR Number	Internal AR information	Internal AC number	Internal AC information
⋮					
External AR number	External AR information	Internal AR Number	Internal AR information	Internal AC number	Internal AC information

Figure 19. Relocation Informations Contents

如 Figure 20，External AR 會需要其 Relocatable field 在 Raw data 裡的 offset，和標示屬於哪一個 Section 的 Type 欄位，並且在建立映像檔時，已經把這個 External Symbol 會使用哪個外部物件找出，第三個欄位就是其 ID，最後是在外部連結檔的 Symbol table 裡讀到的 Virtual address。

Relocatable field offset	Type	External object ID	Virtual address
⋮			
Relocatable field offset	Type	External object ID	Virtual address

Figure 20. External AR Information Contents

如 Figure 21，Internal AR 類似 External AR，只是 Type 所標示的 Section 是屬於目前物件的 Section，所以不需要 External object ID。

Relocatable field offset	Type	Virtual address
⋮		
Relocatable field offset	Type	Virtual address

Figure 21. Internal AR Information Contents

如 Figure 22，Internal AC 跟 Internal AR 最大差別，在於 Relocatable field 有兩筆，分別代表 Physical address 高低各 16bit。

High relocatable field offset	Low relocatable field offset	Type	Virtual address
⋮			
High relocatable field offset	Low relocatable field offset	Type	Virtual address

Figure 22. Internal AC Information Contents

最後 Entry point 有兩個。如 Figure 23，除了一定要提供的 Main function 的 Entry point 外，可以多提供 Initial function 的 Entry point，讓服務在第一次運行時，可以先做初始化。

Main function entry point	Initial function entry point

Figure 23. Main/Initial Entry Point Contents

3.1.3.4. 動態連結的細節

由 ARM 來執行動態連結，主要有四個工作。

首先會分配一塊 DSP 內部記憶體做爲服務的載入空間。接下來系統會從 Flash 讀取映像檔，先計算總共需要的記憶體大小，包含各物件的各個 Section 大小和 Sysmem 大小。用 DSP 記憶體管理函式，分配一塊 SARAM 的記憶體，並由分配的記憶體 Base address，計算各物件的各個 Section 之 Base address，並且將 Flash 裡映像檔中的各物件 Raw data 讀取到 SDRAM，做爲之後做動態連結的暫存。

根據被分配到的記憶體起始位置做動態連結。會以映像檔上面物件的順序，以三種 Relocation information 做 Virtual address 的重新定位。以 External AR 爲例子，根據 External object ID 和 Type 欄位，取得 External Symbol 所在的 Section 之 Base address，加上 Virtual address 後，就可以得到 Physical address，填到 SDRAM

上目前物件的.text section 的 Raw data 之 Base address 加上 Relocatable field offset 的位置。

經由 MPUI 介面載入到 DSP 內部記憶體。當所有 Relocation information 都被處理完後，暫存在 SDRAM 上的 Raw data，已經跟可執行檔上的內容是一樣的了，但要搬到 DSP 的程式記憶體空間，DSP 才能夠執行。由 ARM 或是 System DMA，經由 MPUI 來做搬到 DSP 內部的 SARAM。如果要讓 DSP 或是 DSP DMA 經由 Shared memory 來搬移，則 Endianism conversion 要設定成 byte 的 order 是一樣的，即 Byte and word swap 模式。

最後再向 DSP 系統核心註冊。經由 Mailbox，發送 REGISTER_SERIVCE 的命令，因爲 Init function 和 Main function 的 entry point，各是 24bit，一個 Mailbox 無法傳送 48bit 的訊息，所以會放在 Shared memory 上，DSP 系統核心會分配一個 Process，並從 Shared memory 上讀取 Entry point，設定爲 Process 要運行的演算法的進入點，最後用 Mailbox 回傳 Process ID 給 ARM。

3.1.3.5. Invoke DSP service

要呼叫 DSP 上的服務，會用 Mailbox 傳送 INVOKE_SERVICE 命令和註冊時分配的 PID。當服務的工作運行完畢，DSP 會回傳 RETURN_INVOKE 訊息和運行完的服務之 PID。假如一個服務，同時需要被呼叫二次以上，有兩種方式可以達成。首先一個比較簡單的方式，傳送 DUPLICATE_SERVICE 和被註冊過的服務 PID，來再要求需要再被多註冊一次，DSP 系統核心會再分配一個 Process，並且 Entry point 會設定成與被傳來的 PID 所對應的服務的 Entry point。但這個方式就必需在服務不會去修改到.bss 的空間，也是 global 變數不會被修改，這樣的條件下，同一份執行碼才能同時被執行。另一種方式，一般的流程，用動態連結產生另一份功能一樣的執行碼來註冊，缺點是需要多一塊記憶體空間，但除了空間的限制外，就沒有其他限制。被多次註冊得來 Process，一樣也是用 INVOKE_SERIVCE 命令，來要求運行服務。當服務不再需要呼叫時，可以把它

反註冊，傳送 UNREGISTER_SERVICE 和要被反註冊服務 PID，將會釋放 Process 回到初始狀態。如果在這時，服務的執行碼只有被目前的 Process 所使用，反註冊後就可以把它分配到的 DSP 記憶體釋放，以達到動態抽換；如還有別的 Process，因爲多次註冊而使用同一塊服務執行碼的話，必需等到最後一個 Process 反註冊時，才能釋放 DSP 記憶體。

3.2. DSP System Kernel

本論文所開發的是在 DSP 方面的動態分工系統核心，在這邊的一個基本假設是整個異質多核心系統是會由適合控制工作的 ARM 來管理整個系統的工作分配，包含 DSP 上 Processes 的運行和 DSP 記憶體的分配。而 DSP 上的系統核心，主要是爲了達成多工環境，提供最基本的 Scheduler 和接受 ARM 方面傳來的動態服務命令的 Mailbox ISR 所組成，並且提供兩個核心間大筆資料傳輸的 DMA 介面。

3.2.1. Process

在 DSP 執行的服務會以 Process 爲單位在運行。一個 Process 在執行時，會需要 Process control block 來記錄其目前的資訊。如 Figure 24。有兩種 Stack 指標，分別代表主要的 Stack 和 System stack，因爲 System stack 指標和 Stack 指標共用高的 7bit，所以只需記錄低的 16bit。Process 的狀態，有 Unregistered、Ready、Busy、IDLE，其中 IDLE 是專門給代表目前沒有服務在運行的 Process 使用。Process 的 ID，目前設計最多有 32 個 Process。以 function 這種型態來宣告 Entry point，可以很容易去用 C 語言呼叫 Initial function 和 Main function 的。Initialized 是記錄 Initial function 是否被呼叫過的 flag。剩下的 pad 是爲了使 PCB 大小是 16 個 16bit，讓之後要用組語去撰寫存取各個 PCB 的 base address 計算，會比較簡單而有效率。

```
typedef void(*function)();

typedef struct _PCB{
        uint32 pc;
        uint32 sp;
        uint16 ssp;
        enum STATE state;
        int id;
        function init_entry_point;
        function entry_point;
        int initialized;
        uint16 pad[4];
}PCB;
```

Figure 24. Process Control Block

爲了滿足動態服務的設計，Process 在運行服務時，其 State 切換會如 Figure 25，一開始建立新的 Process 後，會是在 Unregistered state。註冊一個服務後，被分配的 Process，會改成 Ready。 當 ARM 有需要 DSP 去運行服務時，會用 Invoke 命令，使得 Process 變成 Busy。運行完的 Process 會回到 Ready，可以再次被 Invoke。當不再需要 Invoke，就可以 Unregister 掉，回到初始的 Unregistered state。

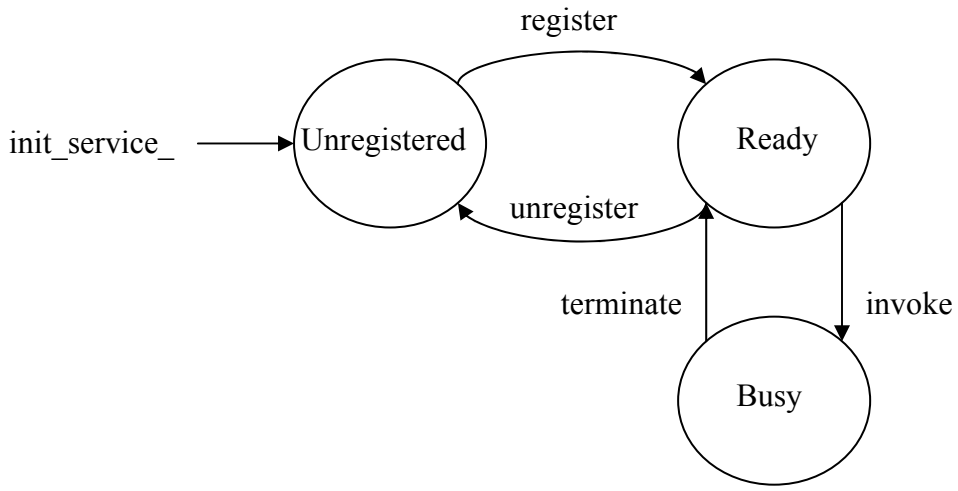


Figure 25. Process State Transitions

如 Figure 26，這是在 DSP 系統核心上運行的 Process 其建立過程的大略程式碼，首先初始化 IDLE Process，在 DSP 沒有任何服務在運行時，DSP 則會運行這個 Process，目前並沒針對功耗去做最佳化，所以 IDLE Process 其運行目的，只是不停地從工作 Queue 中，找工作來執行。之後會用 init_service_slot 函式，產生要運行實際服務的 Process，然後啓動 Timer，代表著 DSP 系統核心開始運行。而 process 和 IDLE_process 這兩個函式裡都是用無窮迴圈不停的執行其工作。

```
void main() {
        /* System Initialization */
                :
                :
        /* System Initialization */

        idle_pcb = &pcb[MAX_PROCESS];
        idle_pcb->id = MAX_PROCESS;
        idle_pcb->state = IDLE;
        current_pcb = idle_pcb;

        for(int i=0; i<MAX_PROCESS; i++)
                if(init_service_slot() != idle_pcb->id)
                        process();

        DSP_CNTL_TIMER3 |= 1; // Start the timer
        IDLE_process();
}
```

Figure 26. Initialization of Unregistered Process

實際運行服務的 Process 是經由 Figure 27 的 init_service_slot 函式所建立的，在設定完新 Process 的 PCB 的初始資訊後，會繼承原本正在執行中的 idle Process 的暫存器內容，並分裂出新的 Process，return 的 Process id 是爲了給呼叫 init_service_slot 函式的呼叫者分辨目前是哪一個 Process 所使用的。

```
int init_service_slot()
{
        static int pid = 0;
        uint16 *stack = (uint16 *)0x7C00 - 0x380*pid;

        select_pcb = &pcb[pid];

        select_pcb->sp = (uint32)stack;
        select_pcb->ssp = ((uint32)stack - 0x300) & 0xFFFF;
        select_pcb->state = UNREGISTERED;
        select_pcb->id = pid++;

        push_and_start();

        return current_pcb->id;
}
```

Figure 27. Funtion init_service_slot()

Stack 指標的分配是從 0x7C00 到 0x0C00，屬於 DARAM 記憶體空間，會把 Stack 存放在 DARAM，是因爲 C55x 的 DSP 指令集中有能夠一次進行兩筆 16bit 資料的 stack operation，而 DARAM 能同時處理兩筆存取，所以把 stack 設定在這邊可以加快 context switch 的速度。因爲 Stack 的使用是往低的 address 成長，在 PCB 中 System stack 的指標是由 Stack 指標減去 0x300 得來的，就代表每個 Process 被分配的 Stack 大小爲 0x300 個 16bit，而 System stack 的大小，因爲每個 Process 的 Stack 指標相差 0x380，扣去 Stack 大小的 0x300，System stack 大小則是 0x80 個 16bit。

在 push_and_start 函式中，會備份 IDLE Process 的暫存器內容，並切換目前運行中的 PCB 爲新的 Process 的 PCB，這時從 push_and_start 函式離開後，邏輯上就可以算可以分裂成了兩個 Process，一個是原本在執行的 IDLE Process，一個是新建的 Process，但實際上 DSP 核心正在執行的會是新建立的 Process，在系統正式運行前，新建立的 Process 會先放棄 DSP 使用權，切換回 IDLE Process 的 PCB，就可再繼續呼叫 init_service_slot 函式，建立新的 Process。

DSP 系統核心在運行時，會有兩種 Process，一個是 IDLE Process，一個是服務在運行的 Process，各自會分別執行 IDLE_process 和 process 函式，會以 Figure 28 的方式來做切換，在 IDLE_process 函式會呼叫 yield 函式，會使目前的 Process 自願放棄 DSP 使用權，並從工作 Queue 選取下一個工作，切換 PCB 並做 context switch，IDLE Process 會以這樣的方式不停的從工作 Queue，找工作來運行，如果有工作可以運行，就會切換到 process 函式來運行服務的工作實際內容(1 號箭頭)。服務的工作運行結束，dequeue 函式會將目前工作，從工作 Queue 裡移除。之後呼叫 yield 函式，再選取工作 Queue 裡下一個工作來執行(2 號箭頭)，如果工作 Queue 裡沒有工作可執行，就會回到 IDLE process(1 號箭頭)。

其中對工作 Queue 進行修改的 yield 函式和 dequeue 函式，必需要有中斷的 mask 保護，確保修改工作 Queue 時，不會有中斷發生，修改後的工作運行的順序邏輯上才會是正確。

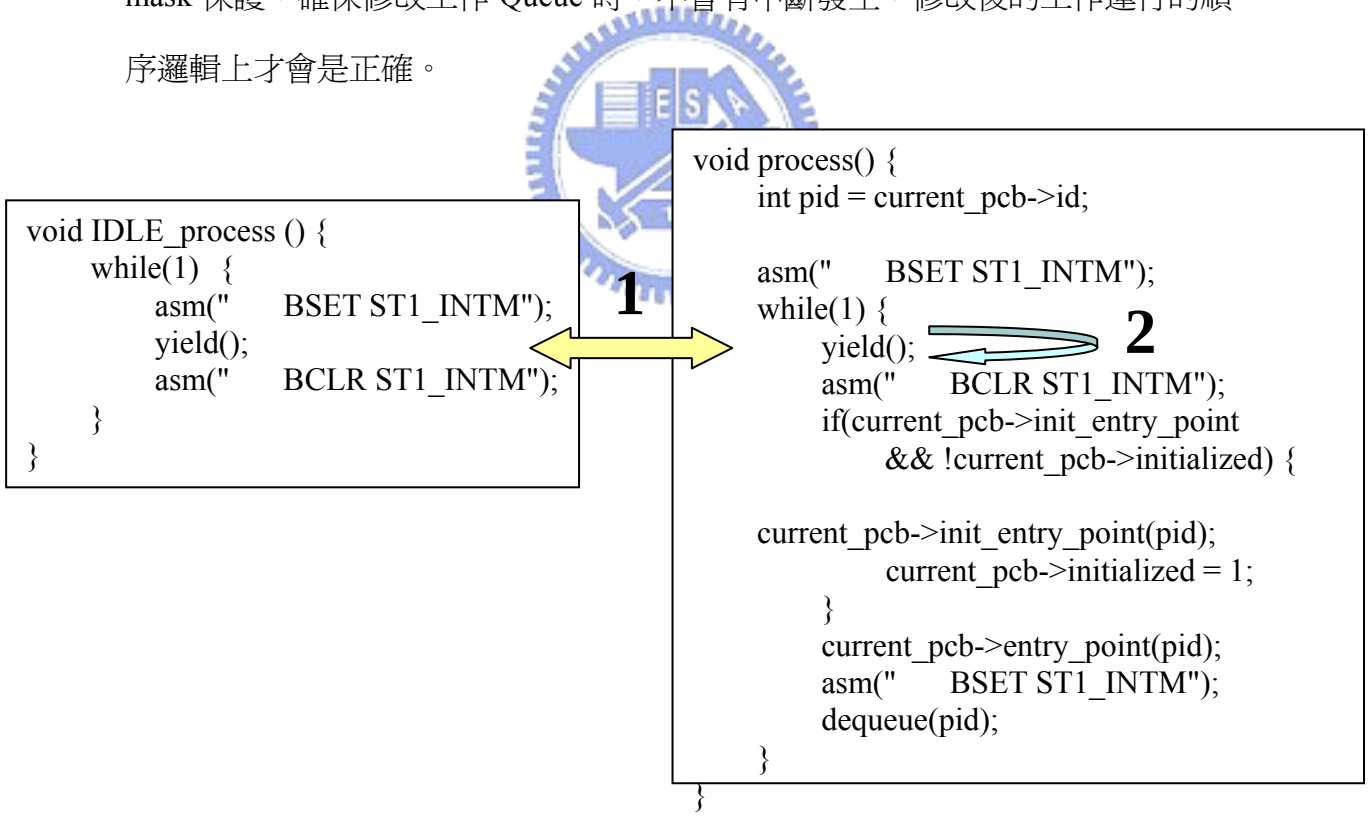


Figure 28. Process Running Flowchart

服務實際會執行的內容，可分成兩個部分，如果有提供初始化函式，第一次被執行的服務，會先執行初始化函式，另一部分則是服務要進行的演算法。因爲這兩種 entry point 都是以 function 的型態來宣告，所以可以直接呼叫。

服務所運行的工作函式，都會傳入一個 pid，這是用來使 ARM 傳遞參數所會用到的，隨著演算法不同，有些可能不需要參數，有些可能需要多個參數，甚至有需要陣列當參數，這些參數就可以放在 shared memory 上參數表，經由 pid 來讀取各個服務所需的參數。爲了讀取這些參數，原本演算法的函式就需要一個 wrap 函式，用來讀取參數並傳給原本的函式。

如 Figure 29，原本的演算法 dequant 函式，需要四個參數，而且前兩個還是陣列。在 ARM 下 Mailbox 命令，把這項工作加到工作 Queue 之前，要先把需要的參數根據服務的 pid，填到 shared memory 上的參數表，每個參數都以 32bit 的長度寫入，實際傳入演算法的函式前，會藉由轉型而傳入正確的長度，而陣列的參數，其起始位置，是由 ARM 動態決定，而其內容會根據陣列長度，看是由 ARM 或是 DMA 來搬。經由 wrap 的方式，dequant 函式就能正確執行。但因爲每個演算法需要參數的個數和型態，都不是固定的，像這樣的 wrap 函式，目前還是用人工撰寫的，未來如果有特製的 compiler 配合，就可以自動化這個步驟。

```
#define PARAMETER_TABLE 0x600000
#define PARAMETER_SIZE 4
void dequant_wrap(int16 id) {
        uint32 *ptr = (uint32*)PARAMETER_TABLE +
                        PARAMETER_SIZE * id;
        uint32 p0,p1,p2,p3;

        p0 = *ptr++;
        p1 = *ptr++;
        p2 = *ptr++;
        p3 = *ptr++;

        dequant((int16 *)p0,(int16 *)p1, (uint32)p2, (uint32)p3);
}
```

Figure 29. Wrapper Function Example

3.2.2. Scheduler

爲了讓 DSP 上運行的各個服務，能以較精細的多工方式(fine-grain multitasking)同時運行，就必需讓各個服務反應時間儘量快速，不會因爲某個服務佔用 CPU 時間過久，而造成只需要少許 CPU 時間的服務等待過久，因此採用了 CPU 時間分配最公平的 Timer-sharing 排程方式，以 Timer 倒數完後的 Interrupt service routine 來實作 scheduler。

因爲這個 Scheduler 爲了使 DSP 能多工運行視訊碼編工作，排程的效率也是很重要的，目標是能達到低 overhead。並且隨著運行服務增加，排程也還是要很有效率，也要達到時間複雜度是 O(1)，因此採用了 Round-robin 的方式，只需要一個工作 Queue，Scheduler 就可以很容易選取下一個工作。

工作 Queue 是以雙向的 linked-list 實作，因爲只有單向的 linked-list，要把新增的工作加到 Queue 的最後，就必需有一個 tail 的指標，指向著 Queue，但這樣在 Timer ISR 裡面就除了改 head 外，還要再多改 tail，一般來說服務在運行是有可能會經過多次 Timer ISR，但新增只有一次，沒必要在時常被呼叫的 Timer ISR 增加 overhead。如果不用 tail 的指標，那每次新增工作就只能用迴圈從 head->next，測試達到最後一個節點後，再加入，但如果 Queue 裡面的工作越多，新增工作所花費的時間會跟著增加。因此最後決定以雙向的 linked-list 來實作，其優點是不管從 Queue 裡新增或移除工作都是 O(1)的時間。

Figure 30 是當發生 context switch 前後的工作 Queue，實線指向下一個工作，虛線指向前一個工作，原本執行的順序是 T1、T0、T3，context switch 時，只要更新 Head 爲原本 Head 指向的下一個工作，所以新的執行順序是 T0、T3、T1。

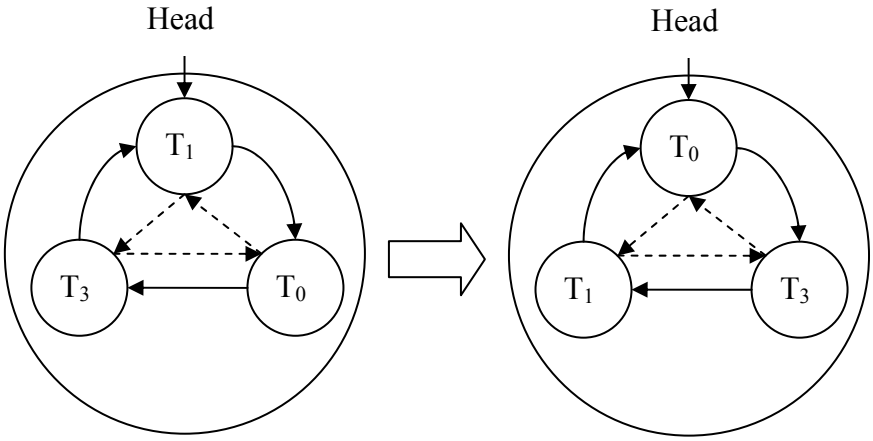


Figure 30. Task Queue Operation for Context Switching

Figure 31 是新增工作到工作 Queue，原本的執行順序是 T1、T0、T3，新增的 T4 則插入 Head 指向的前一個位置，需修改 T1 的前一個，T4 的下一個，和 T4 的前後，共有 4 個指標 assign 運算。之後新的執行順序是 T1、T0、T3、T4。

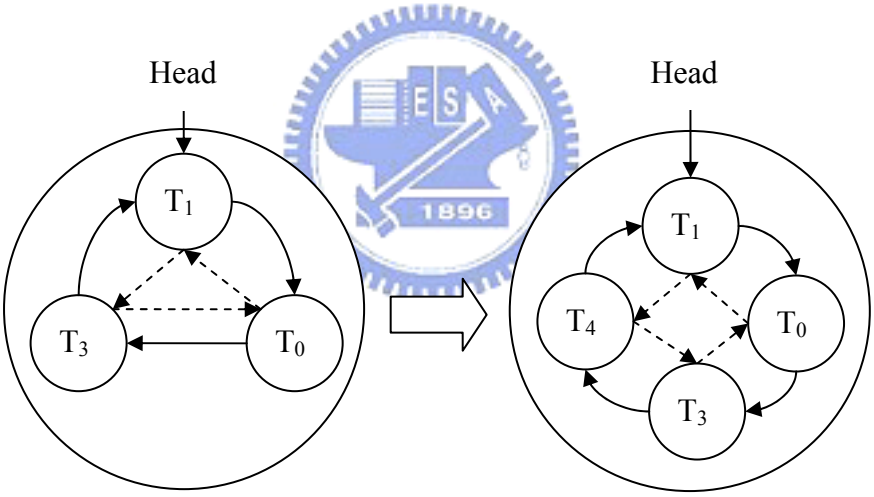


Figure 31. Adding Task to Task Queue

Figure 32 是工作 T1 執行完後，要從工作 Queue 裡移除，首先把 T1 的下一個工作 T0 和 T1 的前一個工作 T4，相互連結，只要兩個指標 assign 運算，但 Head 並不在這時更改爲 T1 的下一個工作 T0，因爲根據之前的 Framework，從執行 dequeue 函式後，T1 就會自願放棄 DSP 使用權，就會選取下一個工作來做 context switch，這時 Head 就會更改爲 T1 的下一個工作 T0。而工作 Queue 裡沒有一個工作會再指向 T0，所以 T0 就完全被移除掉，新的工作順序是 T0、T3、T4。

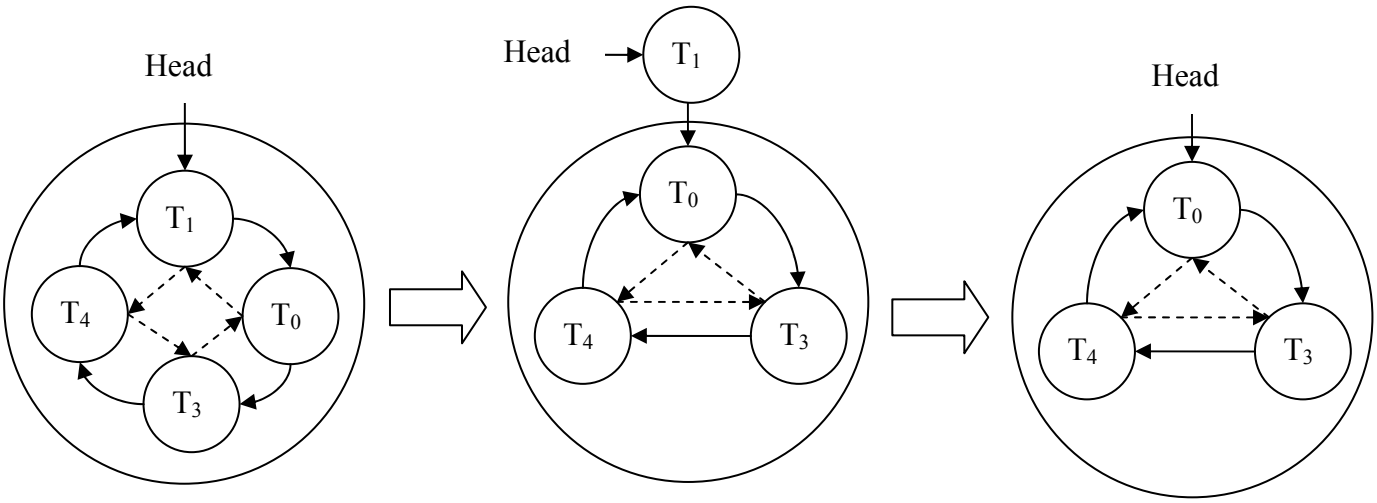


Figure 32. Task Queue Operation for Task Termination

因爲在 Timer 的中斷發生時，被中斷的工作正運行到哪個指令，可以說是隨機的，context switch 最直覺的做法就是把所有暫存器都 push 到 stack，等到選出下個工作並切換 stack 位置後，再 pop 回之前暫存的值。但越多暫存器被儲存，就代表要花越多的時間來做 context switch，這樣爲了達到多工所帶來的 overhead 就會越高。DSP 的暫存器，如 Table 3 有這 13 類暫存器，總共有 55 個暫存器。

Table 3. DSP Registers

Register	Description
AC0-AC3	Accumulator
XAR0-XAR7/AR0-AR7	Auxiliary Register
BK03, BK47, BKC BSA01, BSA23, BSA45, BSA67, BSAC	Circular Buffer Register
XCDP / CDP	Coefficient Data Pointer
XDP / DP	Data Page Register
DBIER0, DBIER1 IER0, IER1 IFR0, IFR1 IVPD, IVPH	Interrupt Register
PDP	Peripheral Data Page Register
PC, RETA, CFCT	Program Flow Registers
BRC0-1, BRS1, RSA0-1, REA0-1 RPTC, CSR	Repeat Registers
XSP / SP, XSSP / SSP	Stack Pointers
ST0_55-ST3_55	Status Registers
T0-T3	Temporary Registers
TRN0, TRN1	Transition Registers

根據DSP的文件對Stack操作的描述[15]，當中斷發生時，在DSP開始執行我們的ISR之前，會把必要的暫存器PUSH進Stack，離開ISR時，會從Stack上POP這些暫存器備份的值。這樣的自動Context switching，輔助我們去重建被中斷的工作之context，如Figure 33，會先把ST0_55、ST2_55、DBSTAT、ST1_55、CFCT和RETA，push到stack裡，再把PC複製到RETA，而跟Repeat指令有關的資訊則備份在CFCT。因此這幾個暫存器，並不需要由scheduler來備份。

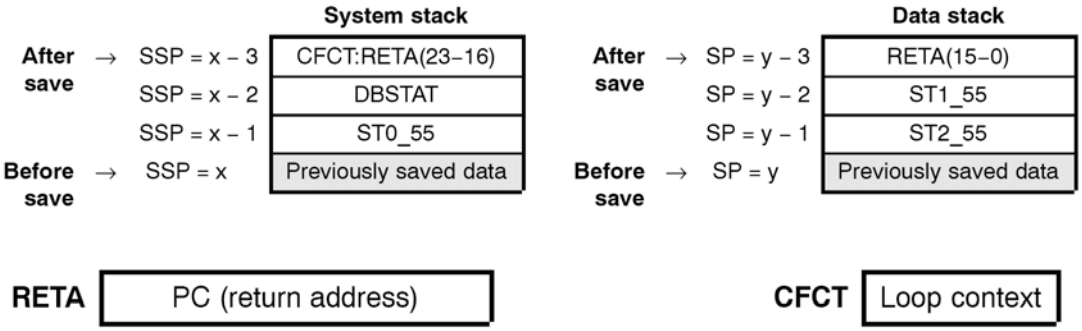


Figure 33. Auto Context Switch

如果我們設計的 DSP Scheduler 在每次進行 context switching 時都要把所有暫存器存到堆疊上，顯然 overhead 會太高。因此，在本論文實作的系統中，我們針對 DSP 可能會運行的服務(i.e. H.264 的 Intra frame 編碼器)所會用到的暫存器進行分析，得知 Circular Buffer Register、Interrupt Register，Data Page Register 和 Peripheral Data Page Register 等等在這些服務中並不會用到。所以剩下總共 30 個暫存器需要備份。

Figure 34 是 Timer ISR 裡主要的程式碼，可分爲五個部分：

1) 把正在運行工作的資訊備份在 stack。

2) 由工作 Queue 裡取得選取下一個工作，如果 Queue 裡沒有等待的工作，就選取 Idle process。

3) 切換正在目前在運行的 PCB，爲被選取的 Process 的 PCB。

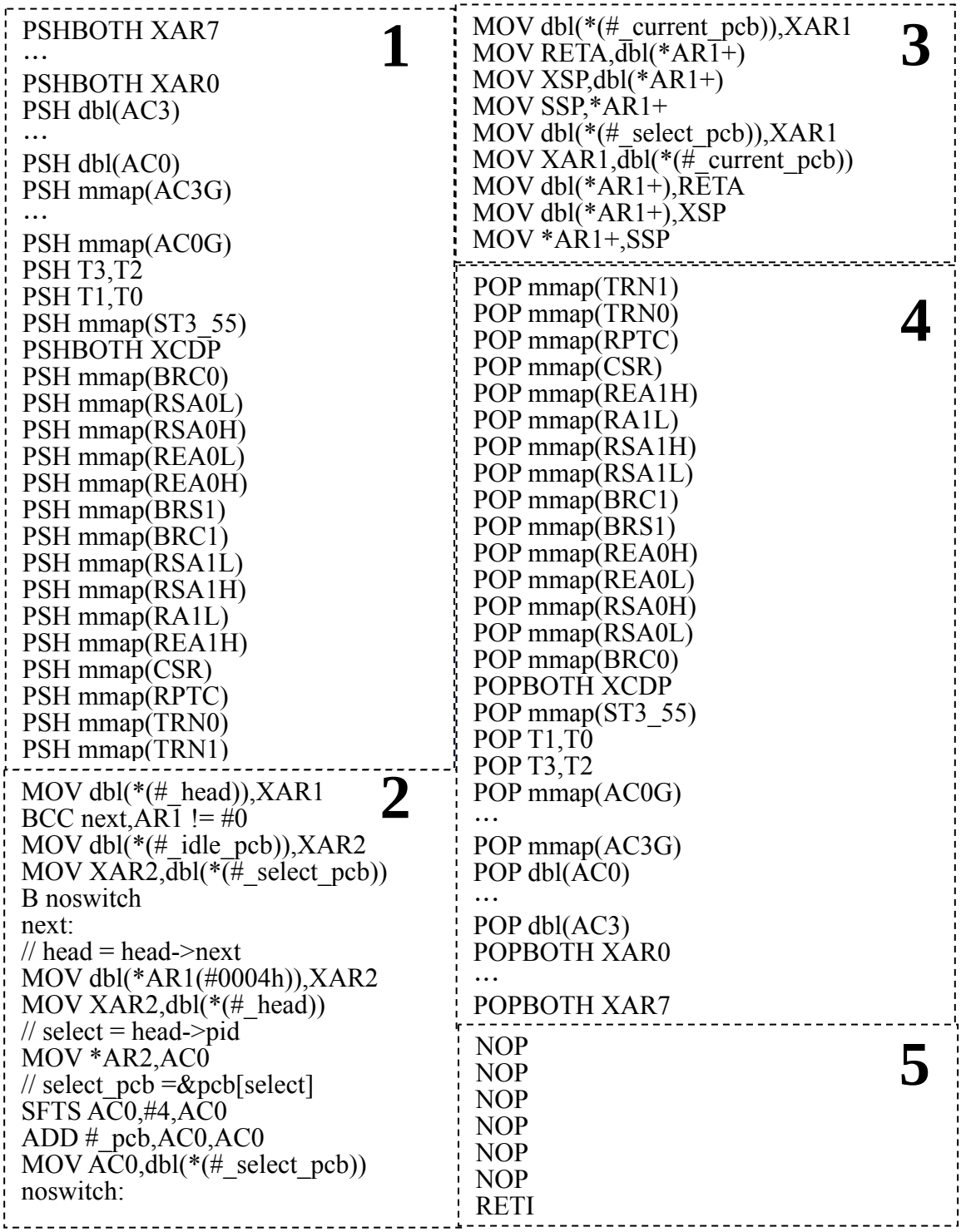


Figure 34. Context Switch

4) 把被選取工作的資訊從 stack 裡取出。

5) 返回中斷，前 6 個 NOP 指令是根據 TI 文件裡的 Advisory 要求所增加的[16]。

3.2.3. Mailbox ISR

DSP 的系統核心，爲了提供動態服務的功能，所以會用到 Mailbox 來傳送跟動態服務有關的命令和資訊，如 Table 4，各自的功能在動態服務的章節，有仔細詳述。

Table 4. Dynamic Service Command

命令	參數
A2D_REGISTER_SERVICE	None
A2D_INVOKE_SERVICE	Process ID
A2D_DUPLICATE_SERVICE	Process ID
A2D_UNREGISTER_SERVICE	Process ID

ARM 有兩組 Mailbox 可以傳送訊息給 DSP，而運行在 DSP 的系統核心，已經使用掉了一組，做爲註冊服務和增加工作到工作 Queue 使用。剩下的一組如果有多個工作也需要用到 Mailbox 來做溝通協調的話，但 Mailbox 的 ISR 也只能有一個，所以剩下一組的也不敷使用多個工作設置各自的 ISR。因此有必要設計一套工作用的 Mailbox ISR 的機制。但如同 Timer 的中斷，Mailbox 中斷發生時，工作正執行到哪個指令也是隨機的，所以需要對被中斷的工作做 context save/restore，最簡單的做法是把所有暫存器都備份起來，但爲了能儘快回應 Mailbox 的命令，針對 Mailbox ISR 的內容，只備份會改變其值的暫存器。DSP 系統核心所使用的 Mailbox ISR 已經最佳化，只需要備份 XAR1-XAR4、AC0、AC1、T0、T1 這 8 個暫存器，但是工作的 Mailbox ISR 會使用到暫存器，並不能事先確定，這要有 compiler 的支援才能幫我們自動去決定要備份哪些暫存器，所以目前暫時使用工作命令表的方式，將 ARM 要下達給工作的命令，經由 MPUI 寫到 DSP 內部記憶體的工作命令表，讓各個工作去讀取。

工作可以用下面的這兩種 Framework 來處理命令。

第一個 Framework，如 Figure 35，工作是被動的由 ARM 下達命令，這時

ARM 傳來的資料是命令的代號。

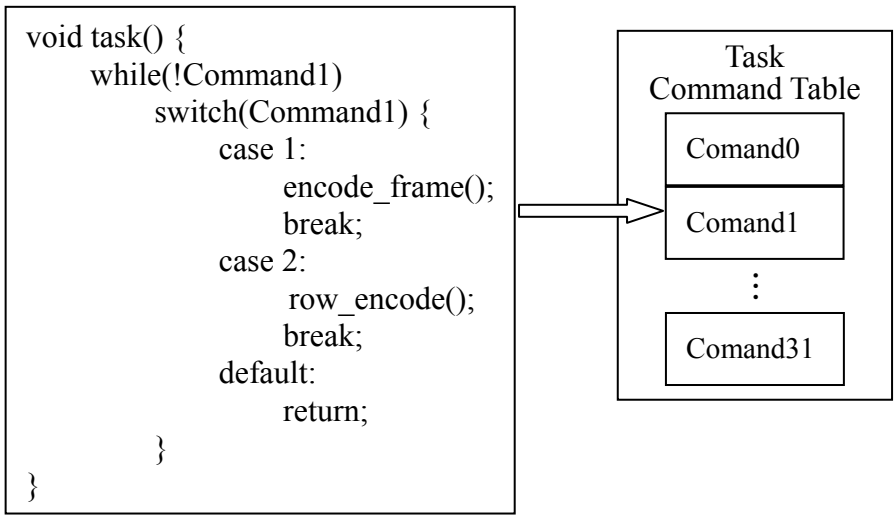


Figure 35. Passive Task Framework

第二個 Framework，如 Figure 36，則是工作主動向 ARM 要求資料並處理完後，再要求 ARM 整合，這時 ARM 傳來的資料是爲了讓工作離開 Busy wait 用的。

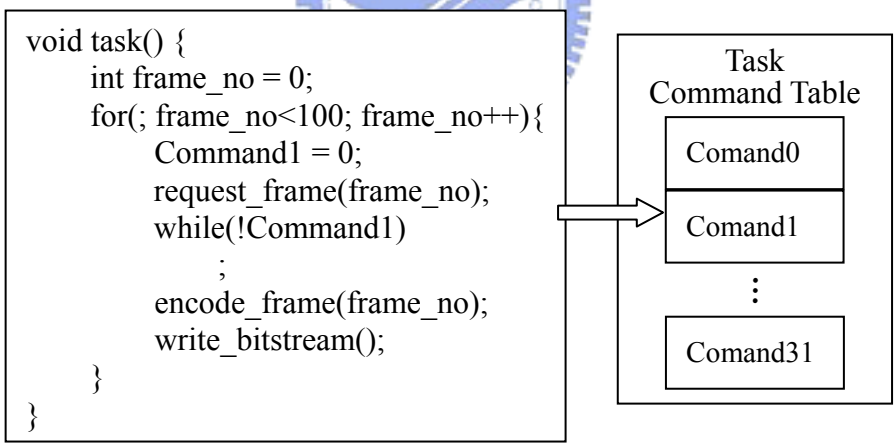


Figure 36. Active Task Framework

3.2.4. DMA 設計

OMAP5912 有提供兩個 DMA，一個是 ARM 所控制的 System DMA，另一個則是 DSP 所控制的 DSP DMA。最大的差異是其運作的記憶體空間不同，System DMA 所運作的記憶體空間，包含 Flash、SDRAM、SRAM、MPUI，而 DSP DMA 所運作的記憶體空間，包含 DARAM、SARAM、Shared memory。

這兩種 DMA 的運作方式，基本上都一樣，每傳送完一個區塊後，DMA 控制器就會 disable 這個 DMA channel，直到 ARM/DSP 再去啓動它(REENABLE)，所以再啓動之前，可以先對 channel 的設定做更改。

另外還有 Auto-initialization 的模式，這時每傳完一個區塊，DMA 會自動複製 channel 的 configuration 暫存器到它內部的 working 暫存器。但 Auto-initialization 的模式有兩種運行方式，一種是重覆(REPEAT)的傳送同一個區塊，這適合讓資料隨時保持同步，另一種是每次傳送前會等 ARM/DSP 去設定完(ENDPROG)才運作。乍看之下，ENDPROG 和 REENABLE 都是要等 ARM/DSP 再對 DMA 下命令，但實際上是有些許差異，在 ENDPROG 方式下如果能在區塊傳送完之前就設定好，DMA 在傳送完後可以立刻進行下一筆傳送，但 REENABLE 方式則是要在 DMA 傳完後再去啓動，而不管啓動前有沒有更改設定。跟 ENPROG 比較起來，REENABLE 會需要額外的等待。至於 REPEAT 的方式，雖然完全不會有機會讓 DMA 去等待，如果有要使用 DMA 的 ISR，這樣就會產生很多次不必要的 ISR 呼叫，除非是發生 ISR 後，就把這個 DMA channel 的中斷 disable，但這需要 disable DMA channel 才能去設定，直到下次需要呼叫 ISR 時，還要再去設定，這樣使用上反而更繁雜沒效率。由此分析下來決定使用 ENDPROG 的方式，來實作 DMA 的使用。

因爲 DSP 的資料記憶體空間是 16bit-addrssing，而 ARM 的記憶體空間是 byte-addressing，所以如果有一個 8bit 的陣列從 ARM 的空間用 DMA 來搬移到

DSP 的空間，直接搬移的結果，DSP 讀起來將會是兩個 8bit 資料 compact 成一個 16bit 資料，除非 DSP 上存取的演算法有特別設計，不然直接將 ARM 上運行的程式碼 port 到 DSP 是不能正確讀取的。要能正確讀取，則需要 DMA 從 ARM 空間讀一個 8bit 資料就轉型成 16bit 資料寫到 DSP 空間，這樣 DMA 搬移是屬於來源和目標的資料單位不一樣大小，可是不管是 System DMA 或 DSP DMA 都只支援來源和目標的資料單位都一樣。但 DMA 有支援目標和來源 index 的大小不同的模式，以這個模式，來源的 index 設成 1、目標的 index 設成 2，這樣每次讀取出 8bit 資料在寫入端會以間隔 8bit 來寫入。但這樣做只寫了 16bit 資料的低 8bit，而如果原本 DSP 空間上高 8bit 的位置有 0 以外的值，這樣 DMA 搬移的結果還是不正確。因此需要在 DMA 搬移之前，將 DSP 空間上的值全都設爲 0，這樣 DMA 搬移的結果才會是正確。而填 0 的動作也可以交由 DMA 來執行，配合之前提到的 ENDPROG 的運行方式，我們可以先下填 0 的 DMA 命令，然後立刻再下 index 大小不一樣的 DMA 搬移，兩個 DMA 搬移就能完成，而且兩個 DMA 命令之間也不需要等待時間。反過如果是從 DSP 空間搬到 ARM 空間，因爲目標是 8bit，沒有高 8bit 的值要填 0，就只需要一次 DMA 搬移。

3.3. 適合異質多核心平台的應用程式移植步驟

一般來說要把一個普通單核心的應用程式移植到異質多核心的平台上，首先要把原本的程式進行分割，看哪部分工作適合這個核心執行，哪部分工作適合另一個核心執行，但除了這種爲了平行化的效能而修改演算法的工作分配之外，底層還是需要有爲了多個核心上資料的溝通，使資料共用，而對原本的程式碼做些微修改，在這方面提出一種 porting 樣式，這種 porting style 主要目的是爲了簡化這種修改，使得每次單核心版本的程式碼有更新時，要再 porting 到多核心時，可減少不必要的額外修改。

主要分爲三個步驟，(1)變更變數的資料型態，(2)初始化時變數在記憶體上的存放位址，(3)在資料需要同步時對這類變數去做搬移。

3.3.1. 變更資料型態

一般常見的 CPU 都是以 byte (8bit)爲記憶體 addressing 單位，而有些 DSP 依設計的不同可能會有以 word (16bit)爲記憶體 addressing 單位，爲了適應不同核心的 addressing 單位，多核心要共用資料就要以 addressing 單位最大的爲基準。以下的例子就分別以 CPU 用 8bit 而 DSP 用 16bit 爲 addressing 單位來說明，所以 CPU 和 DSP 有共用資料的話，其 addressing 單位以 16bit 爲基準。假如 CPU 和 DSP 現在要共用一個 8bit 陣列，其原本的型態假設是 uint8，這時就另外定義 share_uint8 爲 uint16，並把原本的型態改成 share_uint8，這樣做的優點是，可以很明確的知道這個陣列是要來共用的，也知道原本的型態是 uint8，而實際要存取任一筆資料時，不管是用陣列或是指標的方式，compiler 就依照其實際的型態 uint16 來計算位址，會把 offset 的單位從 1(byte)產生成 2(halfword)，所以對程式碼來說，就只有型態有關的部分需要修改，如函數的參數傳遞、資料轉型和 sizeof 函數等，存取的部分不需要在修改；唯一的缺點是記憶體 addressing 單位小的核心，共用資料就需要倍數的空間來存放。而程式開發者只要不在裡面存放超過

uint8 可表示範圍內的值就不會有錯誤。

3.3.2. 初始化記憶體位址

一般的程式寫作大都不會強制設定變數存放的記憶體位址，而是用 malloc 這類的函式來幫我們分配其位置，但在這種異質多核心的硬體平台設計，大多會設計讓各個核心都能存取的記憶體區塊來做爲共用記憶體，如 SRAM 記憶體，甚至有些核心會有自己專用的 scratchpad 記憶體，這類記憶體的特點就是存取速度比一般的 SDRAM 記憶體還快，所以資料如果能存放在上面運算的存取速度也就能提升。因此就有需要手動設定共用或是常存取的變數在記憶體上存放的位址。我們可以新增一個 shared_mem.c 程式檔給多個核心共用，這樣的優點是對各個核心的記憶體位址和大小的配置可以很容易同步，而爲了適應各個核心上的差異、如共用記憶體區塊的起始位址和 addressing 單位，可以用#if 這種方式來判別現在是要 compile 給哪個核心，然後各自定義應給的值；原本用 malloc 函式得來的存放的位址，就改成剛設定的記憶體位址，以 Figure 37 的程式碼來當例子。假如在共用記憶體上有兩個 128 byte 的陣列，在之前改變資料型態裡面所提到的，要以 addressing 單位比較大的爲基準，所以陣列的每個 8bit 是儲存在 DSP 的 addressing 單位的 16bit(2byte)上，所以每個陣列實際會佔用 256 byte，而 DSP 因爲其 addressing 單位 16bit 是 ARM addressing 單位 8bit 的兩倍，所以要把記憶體位址再除以 MEM_UINT(2)，才是 128 個 16bit。

share_mem.c

```
#ifdef DSP
#define SHARE_MEM 0x600000
#define MEM_UNIT 2
#else
#define SHARE_MEM 0x20000000
#define MEM_UNIT 1
#endif

share_uint8 *array1_ptr = (share_uint8 *)(SHARE_MEM+0x00000)/MEM_UNIT);
share_uint8 *array2_ptr = (share_uint8 *)(SHARE_MEM+0x00100)/MEM_UNIT);
```

init.c

```
#include <share_mem.h>

void init()
{     array1 = (share_uint8 *) array1_ptr;//(uint8 *)malloc(128*sizeof(uint8));
      array2 = (share_uint8 *) array2_ptr;//(uint8 *)malloc(128*sizeof(uint8));   }
```

Figure 37. Shared Memory Initializatin Example

3.3.3. 資料搬移

基本上把資料放在共用記憶體上運算，各個核心存取到的就已經當前的值，除非是因爲有核心使用了 Data cache 就有可能造成 coherency 的問題，所以在需要同步前要能 flush 掉 cache，或是事先要設定共用記憶體的這段區塊是不會被 cache 的。而之前有提到過異質多核心平台可能有會 scratchpad 記憶體的設計，所以如果共用的資料是經常會存取的話，就在 scratchpad 記憶體多分配一份空間，然後在同步完後搬進 scratchpad 記憶體，對其運算時的存取將會加快，缺點就是要多一次搬移的 overhead。當這類資料量太大，可以改用直接記憶體存取(DMA)的方式，直接搬移兩個核心的記憶體，而不經由共用記憶體，可加速搬移的時間。

正如之前在變更資料型態裡有提到的多核心共用資料要以記憶體 addressing 單位最大的爲基準，如果要運算的資料單位小於最大的記憶體 addressing 單位，每單位資料存放在共用記憶體要用到的空間大小，也就是最大的記憶體 addressing 單位。因此資料搬移的方式，就不能使用目標和來源單位大小一致的 memcpy 函數。

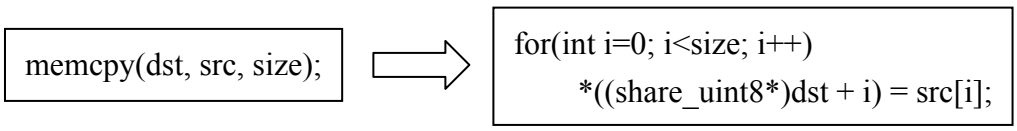


Figure 38. Memcpy to Looply Copy

如 Figure 38，有一個 uint8 的陣列要搬到共用記憶體上，原本的目標會和來源一樣都是 uint8*，但最大的 addressing 單位是 16bit，因此目標就要改成 share_uint8，也就是 uint16，因此目標和來源的大小就不一致，memcpy 就要改成用 for 迴圈的方式，搬移就變比較沒有效率，但這樣搬移的結果才會是正確的；當然最後運算完，資料整合也需要類似的方式來存取。

所以如果 compiler 能針對這類的 pattern，產生出效率跟 memcpy 函式接近的指令，或是 DMA 能夠支援來源和目標的資料單位大小可以不一致，將可以減少不同核心間資料搬移的 overhead。

3.4. Porting Issues for TI OMAP OSK5912

因爲 DSP 是個 16bit 的處理器，在 porting 到 OSK5912 的 DSP 時，常會有計算結果跟預期的有出入的問題。並且在記憶體的使用上有著特別的限制

運算最常出現的錯誤是跟單位資料長度有關，DSP 的 int 變數長度是 16bit，而一般個人電腦的 x86 CPU 和 OMAP5912 的 ARM 都是 32bit，所以有時候不小心把某個變數設成 int 型態，並有可能會在上存放超 16bit 的值的，單純把程式碼移值到 DSP，就會出錯。

另一個常見的錯誤是跟 compiler 有關，如 Figure 39，以 C 語言和其產生的 DSP 指令來說明，把一個 uint16 的變數左移，因爲有可能會超過原本 16bit 可容納的長度，所以他從 stack 讀出來所存放的暫存器是用 ACn 這種有 40bit 的 Accumulator，可是運算完後卻又跟 0xFFFF 做 AND 運算，最後還是只剩 16bit 的內容。會產生這樣的錯誤，最主要是 compiler 對運算式的拆解後，以 button up 的方式去計算各子運算式，所以子運算式並不知道之後會被用在資料長度較長的運算，只單純用目前的子運算式裡的變數來決定運算完後的暫存資料長度。像這個例子中，左移這個子運算裡 a 是 16bit，所以運算完的暫存資料長度也跟著是 16bit，compiler 知道左移有可能會超 16bit，所以還再做 AND 來維持 16bit，因此最後計算結果反而就錯了。

```
b = a << 10; // uint16 a; uint32 b;
      MOV uns(@#00h),AC0
      SFTS AC0,#10,AC0
      AND #65535,AC0,AC0
      MOV AC0,dbl(@#02h)
```

Figure 39. 16bit Left Shift

所以如果原先預期的運算結果有可能有超過 16bit 的運算式，爲了避免被 compile 產生出錯誤的指令，就必需明確的做轉型，如 Figure 40。藉著運算前先轉型成長度較長的資料型態，所以左移的子運算裡變數長度最大的就是 32bit，因此左移後的暫存資料型態也就是 32bit，這樣就不需要再做 AND 來維持 16bit 的長度了，最後計算結果也就跟著正確了。

```
b = (uint32)a << 10; // uint16 a; uint32 b;
      MOV uns(@#00h),AC0
      SFTS AC0,#10,AC0
      MOV AC0,dbl(@#02h)
```

Figure 40.　32bit Left Shift

在 TI 的文件對於各種 Data Addressing Mode，都有著類似下面這段文字，"All additions to and subtractions from the pointers are done modulo 64K. You cannot address data across main data pages without changing the value in the extended auxiliary register (XARn)."[17]，指出其指標的運算都會 modulo 64K 這個特性，這是來於 DSP 有著 64K 的 page boundary，這造成了三個限制，(1)DSP 的 stack 和 system stack 必需在同一個 page 上，(2)malloc 函式所分配的計憶體區塊不能跨過 page boundary，也因此最大只能要求到 64K 個 16bit，(3)指標的 address 計算是用 16bit 的運算指令。

除了第一個是因爲 DSP 的暫存器設計讓 SP(stack)和(SSP(system stack)共用高位元的 8bit，另外兩個應該可以說是 compiler 的問題。如 Figure 41，DSP 的 compiler 對於 address 計算所產生的指令，因爲 DSP 記憶體空間總共需要 23bit 的 address，所以指標變數存在 stack 上會用到兩個 16bit，才足夠容納 23bit 的 address，而指標存取的 indirect 搬移指令要用 23bit 的 Auxiliary register XARn，首先從 stack 上讀兩個 16bit 暫存到 XARn，但然後只用 16bit 的指令來對 XARn 的低位元的 16bit 做運算，因此如果剛好在計算發生 overflow，高位元的 7bit 卻沒有被進位，產生出來的 address 就是錯的，這就是 page boundary 的由來。

```
y++; // uint8 *y;
      MOV     dbl(@#08h), XAR3
      AMAR    *AR3+
      MOV     XAR3,dbl(@#08h)
```

Figure 41. Addition to Pointer

起初想到的解決方式就是在計算前先轉型成 32bit，如 Figure 42。但這樣會多一個暫存器搬移的指令，這樣效率就會比較差。

```
y = (uint8*)((uint32)y + 1); // uint8 *y;
      MOV     dbl(@#08h),XAR3
      MOV     XAR3,AC0
      ADD     #1,AC0
      MOV     AC0,dbl(@#08h)
```

Figure 42. Addition to Pointer after Casting

但單純的 32bit 的計算， compiler 是能產生出比 address 計算還有效率的指令，如 Figure 43，可是如果之後要用指標去存取值時產生的 indirect 搬移指令，就還是要再搬移到 XARn，所以效率是跟第二個例子一樣。或許 compiler 的開發者因爲是考慮到效率的問題，所以才加上不能跨過 page boundary 的限制。我們這樣結論，指標運算不能跨過 page boundary，而分配 DSP 記憶體，並不建議使用 CCS 所提供的 malloc 函式，改成自己手動分配，如果需要的記憶體小於 64K 個 16bit，只需注意別跨過 page boundary，指標運算就能正常並有效率的運作，而如果需要大於 64K 個 16bit，指標運算就必需先轉型成 uint32，雖然效率差了點，但維持了運算的正確性。

```
y++; // uint32 y;
      MOV    dbl(@#08h),AC0
      ADD    #1,AC0
      MOV    AC1,dbl(@#08h)
```

Figure 43. Addition to 32bit Value

4. 動態分工 H.264 編碼器設計

在本論文中，實驗所用的視訊編碼演算法，是以邱正男所提出的 Tightly-Coupled H.264 Video Encoder[8]，來驗證系統的效能。不過在[8]的論文中，DSP 的部份並沒有排程器，換句話說，DSP 端只能一次執行一項工作，完成後才能接受 GPP 所指定的下一份工作。在這種 single task 的應用環境下，並不能看出動態分工架構的好處。而本論文因爲實作了 DSP 的排程器，配合了[8]所提出的架構，所以可以在複雜的多工環境下，展現出動態分工排程的好處。本章會先介紹所謂 tightly-coupled 和 loosely-coupled 的系統差異所在，接下來會介紹本論文開發的系統的主要應用－H.264 Intra Encoder 的設計。

4.1. Loosely-coupled VS Tightly-coupled System

在[8]所定義的 loosely-coupled system，如 Figure 44，在兩個核心間，只有在視訊編碼的開頭傳輸要被處理的資料，和結尾的資料整合的傳輸，才會需要溝通，這樣的優點是工作切割的非常乾淨，設計上很有模組化的特性，易於快速的開發，以搶攻市場佔有率。但會這樣切割工作，因爲是根據 Offline 進行單一應用的 Profiling 來決定該項應用程式的工作分配方式。但隨著多媒體應用的演進和使用者操作的多樣化，造成了 DSP 可能會同時需要進行多個多媒體或通訊相關的運算，這時 DSP 已經有很大的負載了，而傳統的靜態工作分配法，只會一味丟給 DSP 更多的工作，而讓系統效能更加惡化。

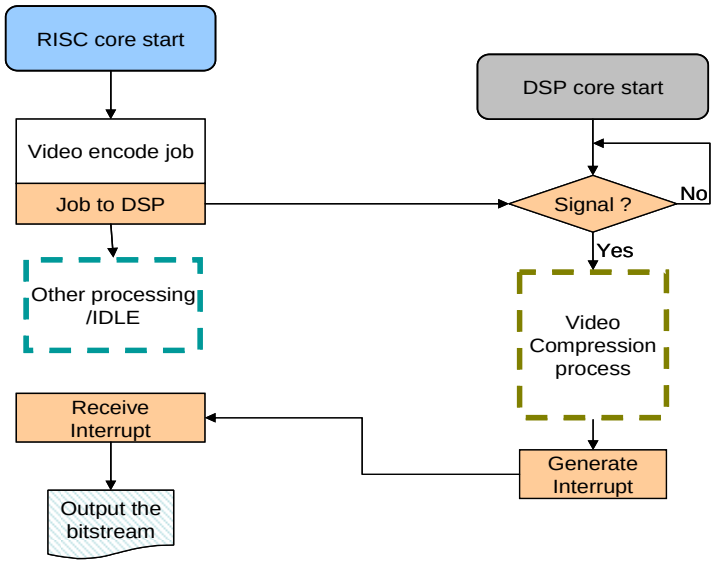


Figure 44. Loosely-Coupled Video Encoder

這時[8]所定義的 Tightly-coupled 的方式，如 Figure 45，可以在不改變硬體架構下，運用 RISC 的計算資源輔助 DSP，將可減輕 DSP 的負擔。兩個核心各自運行部份的工作，最後再整合，而工作的分配，則是根據動態兩個核心間的負載所決定的。

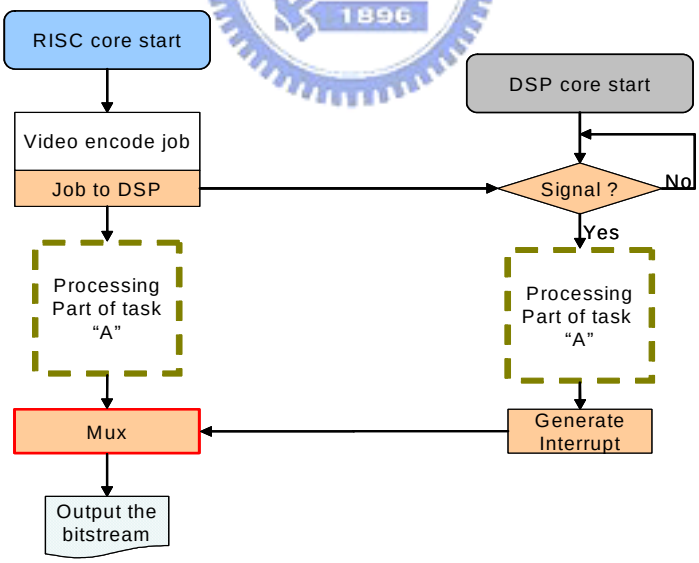


Figure 45. Tightly-Coupled Video Encoder

4.2. H.264 Intra frame 編碼器

4.2.1. 數位視訊編碼概論

一般我們可以取得的多媒體資料，都是經過編碼處理過，直到需要觀看，才即時解碼回原本的資料格式。如果不經過編碼，多媒體的資料量是非常的大，需要龐大的儲存空間、或是傳輸頻寬，以常見的 DVD 規格來當例子，目前主要都是以 720x480、每秒 30 張的視訊影片爲主流，再乘上每一個點以 RGB 表示需要 3byte，每秒總共會有將近 30Mbyte 的資料量，以 4.7Gbyte 的 DVD 的容量，也只能儲存大約 150 秒的畫面。因此對於視訊資料進行編碼是其必要。

一般的視訊系統大概如 Figure 46 所示，先將 RGB 顏色空間資料轉換到 YCbCr 顏色空間資料，做視訊編碼的來源，經由編碼產生資料量比較小的 Bitstream(實線箭頭)，方便儲存或經由網路傳輸；之後需要觀看再以反向的順序解碼回 YCbCr 的格式(虛線箭頭)。

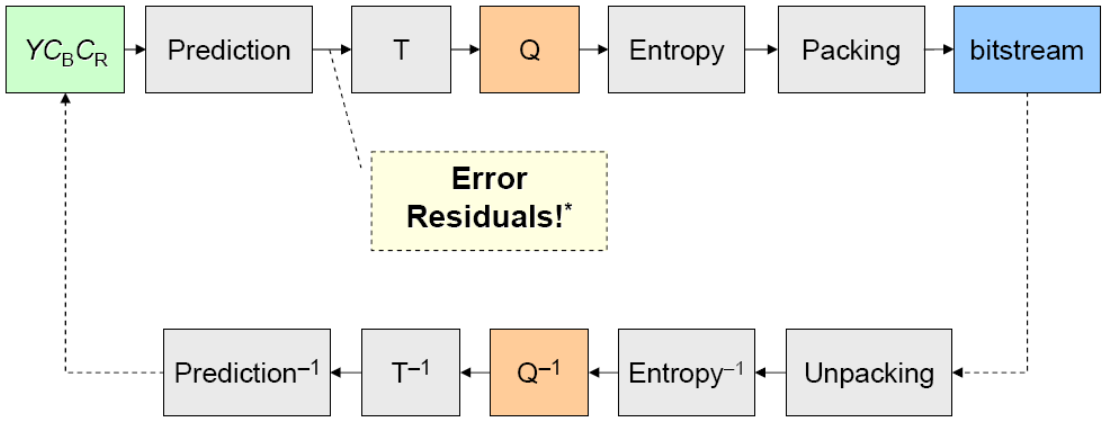


Figure 46. Simplified Video Coding Model

首先會進行預測(Prediction)處理，將對要編碼的 YCbCr 進行某種預測，以消去重覆的資料。常見的有空間上的預測，如畫面的顏色大都會有連續性，所以可以從周圍的顏色來預測目前的顏色，或是用時間上的預測，這也是視訊資料和純影像資料最大的不同，因爲視訊畫面上的物件會隨著時間有著方向性的移動，所以如預測出其移動方向，將可以不需要在每張視訊資料記錄物件的資料，只要從上一張的物件經由移動方向修正，就可以還原出原本的視訊畫面。

將預測處理得到的預測結果，和原先畫面相減去之後，將產生的資料量比較小的差值(Residual)，再從時間域(Time domain)轉換(Transform)到頻率域(Frequency domain)，因爲人眼對高頻的改變比較不易察覺，所以我們可以藉由轉成頻率域，將資料分成高低頻，提供消除不容易看出的高頻資料的可能。

再來會以量化(Quantization)處理，是再進一步的去除細微的數值差異，如0,1,2,3,4,5,6,7,…，在量化單位 3 的處理下，可以產出 0,0,01,1,1,2,2,…，主要會以不影響人眼視覺品質的前提下，將數值限制在某些易於編碼的數值集合中，並且還能將高頻資料消去。

之後的熵編碼(Entropy coding)，會將量化的結果，以各個數值出現的機率決定編碼後所佔的資料量，以達到以較少的位元數表示較常出現的數值，反過來，不常出現的數值則會以較多的位元數表示，例如有名的 Huffman 演算法就是在進來這樣的編碼。

最後再把熵編碼的資料包裝起來，就成易於儲存或網路傳輸的 Bitstream，如有需要還原爲 YCbCr 格式，就依照編碼的反向步驟，一步一步解碼回接近當初的 YCbCr 資料。

4.2.2. H.264 Intra prediction

H.264[9]是目前當紅的視訊編碼，有著在相同資料量下，能達到更好的人眼視覺品質的優點，但是需要相當複雜的編碼運算。

其中爲了減少空間上的重覆性，會以 Intra Prediction 來減少，每個編碼處理單位 MB(MacroBlock)的資料量。MB 可以用 16x16 或是 4x4 的方式來做空間上的預測，16x16 支援四個模式：Vertical、Horizontal、DC 和 Plane；而 4x4 支援 9 種模式，如 Figure 47，有八個方向的預測再加上 DC mode。

X	A	B	C	D	E	F	G	H
I	a	b	c	d				
J	e	f	g	h				
K	i	j	k	l				
L	m	n	o	p				

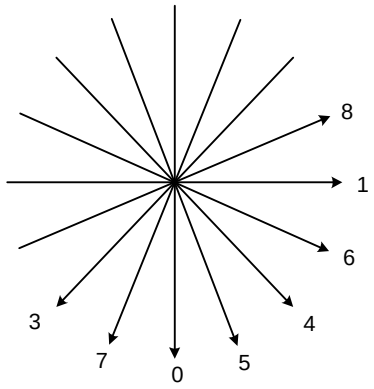


Figure 47. 4x4 Intra Prediction Mode

4.3. Tightly-Coupled Video Encoder

[8]所提出的 Tightly-Coupled Video Encoder，在兩個核心間工作的分配主是因爲 H.264 Intra Prediction 的限制，如 Figure 47，要進行目前 MB 的編碼，需要左邊、上面和右上的 MB 編碼後所重建回的 YCbCr 格式的各個點，才能去運行 Intra Prediction。

所以[8]設計的視訊編碼方式，可以參考 Figure 48，各核心要編碼的 MB 要考慮到左邊和上面 MB，這樣的相依性。首先由 DSP 對第一列的 MB 進行編碼，因爲 DSP 專爲多媒體處理而最設計，而 ARM 接著準備第二列的 MB 編碼，但要其上方的 MB 編碼完並重建回 YCbCr 格式各個點。ARM 要處理的 MB 之 Intra Prediction 才能正確運作，所以上方 MB 的由 DSP 處理完畢，ARM 才會開始進

行編碼。

等到 DSP 處理完每一列，就會開始接手 ARM 還未處理的 MB。這時左邊 MB 因爲是由 ARM 所編碼的，所以等到 ARM 處理完目前的 MB，DSP 就可以開把這一列剩下的 MB 進行編碼。ARM 這時則再去處理下一列。

以這樣 DSP 處理上排 MB，而 ARM 處下 MB 的方式進行，兩個核心就會因爲動態的負載，達到動態的分配工作。

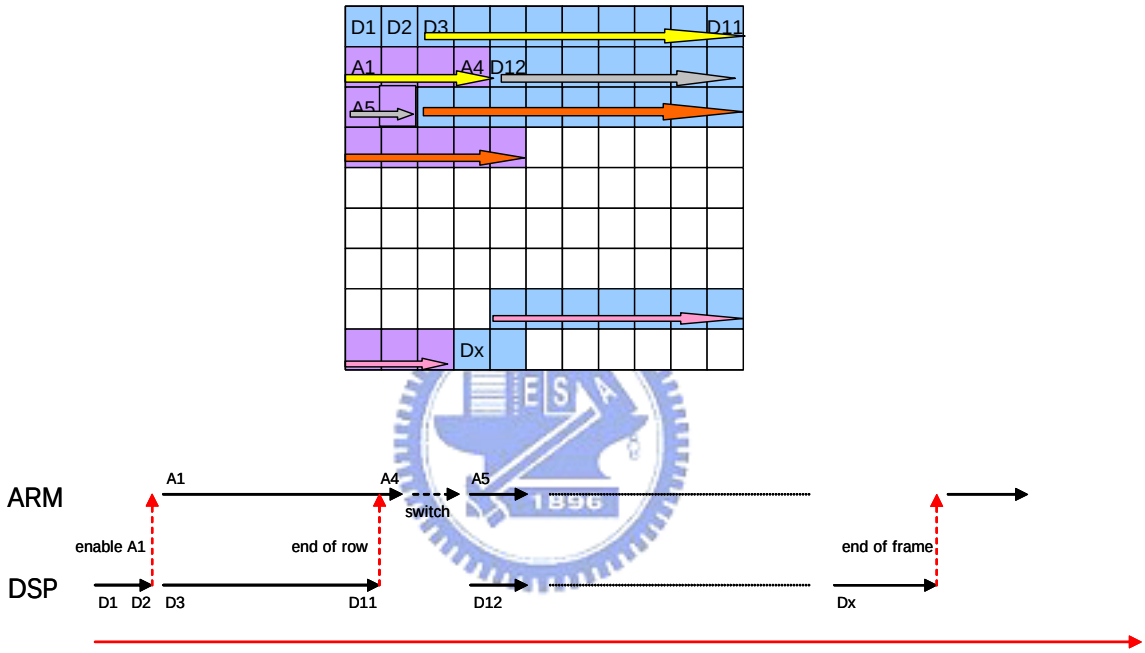


Figure 48. Tightly-coupled Video Encoding Process

5. 實驗結果

在本章中，我們進行一些實驗來驗證我們所開發的系統效能，首先我們列出實驗的執行環境如下：

一、實驗平台設定如下：

OMAP5912 內部的三個主要元件，ARM、DSP、Traffic Controller(TC)，都以 96Mhz 的頻率在運作。測試結果以 ARM 端平均的 Timer count 值爲單位。

二、H.264 編碼參數設定如下：

以 H.264 Intra frame 編碼，來源是 FOREMAN 的 300 張 yuv 檔，大小 176x144(qcif)，QP 設定爲 26，Slice 大小爲整張 Frame，

5.1. 編碼速度測試

首先，我們以 H.264 Intra frame 編碼器，來測試在純 ARM 和純 DSP 和第三章所提到的以兩個核心一起協同運算(Tightly-coupled)，這三種方式的視訊編碼效率。

首先測試兩個核心基本的運算能力，因爲 ARM 有 Cache，而 DSP 有內部記憶體，都可以用來提升編碼效率，所以在不使用 Cache 和內部記憶體下的測試結果如 Table 5。我們可以看出基本上 DSP 的運算能力大約比 ARM 快了 1.4 倍。

Table 5. Encoding performance without Cache

Encoding process	Timer count	%
Pure ARM	464,756,901.91	139.10
Pure DSP	334,118,977.80	100.00

再來測試打開 I-Cache 後，雙方編碼速度各自能提升多少，雖然 DSP 也有 I-Cache，這邊是直接把程式(.text)載入到 DSP 內部記憶體的 SARAM，而沒有使用 DSP 的 I-Cache，測試結果如 Table 6。可以看出 ARM 的 I-Cache 效率不錯，速度提升了 4.4 倍，而 DSP 將運行的程式放在 SARAM，只提升了 2.4 倍，反而讓 ARM 的效率贏過了 DSP

Table 6. Encoding performance with I-Cache

Encoding process	Timer count	%(ARM)	%(DSP)
Pure ARM	464,756,901.91	437.32	334.56
Pure ARM w/ I-Cache	106,274,430.96	100.00	76.50
Pure DSP	334,118,977.80	314.39	240.52
Pure DSP w/ SARAM	138,917,244.36	130.72	100.00

再加上 D-Cache 來測試，但 DSP 並沒有 D-Cache，而且 DSP 內部記憶體大小不夠把要編碼的 Frame 和之後重建出來的 Frame 都塞進去，所除了這兩個之外的資料都是放在 DSP 內部記憶體的 DARAM，測試結果如 Table 7。可以看到 ARM 再加上 D-Cache 後，速度提升了 3.1 倍，而 DSP 雖然無法把全部的資料都放在 DARAM，但也提升了 3.3 倍速度，可是還是比 ARM 慢了 1.2 倍。

Table 7. Encoding Performance with Cache

Encoding process	Timer count	%(ARM)	%(DSP)
Pure ARM	464,756,901.91	1,362.36	1,117.27
Pure ARM w/ I-Cache	106,274,430.96	311.53	255.48
Pure ARM w/ Cache	34,114,137.91	100.00	82.01
Pure DSP	334,118,977.80	979.41	803.21
Pure DSP w/ SARAM	138,917,244.36	407.21	333.95
Pure DSP w/ On-chip RAM	41,597,727.35	121.94	100.00

以 Table 7 的數據來看，Tightly-coupled 的方式將採用兩個核心最快的 Cache 設定來測試，配合之前總合的結果如 Table 8。可以看到 Tightly-coupled 的方式比純 ARM 最快的速度還快了 1.2 倍，比純 DSP 最快的速度快了 1.5 倍。

Table 8. Overall Encoding Performance

Encoding process	I-Cache	D-Cache	.text	Data	Shared data	Timer count	%
Pure ARM	Off	Off	SDRAM	SDRAM	SDRAM	464,756,901.91	1,696.75
	On	Off	SDRAM	SDRAM	SDRAM	106,274,430.96	387.99
	On	On	SDRAM	SDRAM	SDRAM	34,114,137.91	124.55
Pure DSP	Off		SDRAM	SDRAM	SDRAM	334,118,977.80	1,219.82
	Off		SARAM	SDRAM	SDRAM	138,917,244.36	507.17
	Off		SARAM	DARAM	SDRAM	41,597,727.35	151.87
Tight-coupled	A:On D:Off	A:ON D:	A:SDRAM D:SARAM	A:SDRAM D:DARAM	SDRAM	27,390,931.79	100.00

5.2. Scheduler 效率

爲了提供多工環境的而在 DSP 上增加了 Scheduler，其運行所帶來 overhead 也是必須值得參考。將分別以三種 Time quantum，10ms、1ms、0.1ms，和以動態服務的方式載入多個工作，在多 Process 的環境下，讓 DSP 運行 H.264 Intra frame 編碼器來測試對編碼效率的影響，將會測得 ARM 端的 Timer count 和 Scheduler 進行 Context switch 的次數。

測試結果如 Table 9，其中各項結果的百分比，是以前一節的純 DSP 運行 H.264 Intra frame 編碼器的 ARM 端 Timer count，41,597,727.35 爲基準。而 Process 數大於 1 的項目，除了運行 H.264 Intra frame 編碼器之外，多餘的 Process 都運行了單純地在無窮迴圈中對一個變數做 increment 運算。

Table 9. Scheduler Performance

Process # / Time Quantum	1	2	3	4
10ms	41,607,094.34 100.02% 43.39	82,929,499.07 199.36% 86.03	124,071,769.4 298.27% 128.81	165,291,004.58 397.36% 171.75
1ms	41,674,910.05 100.19% 434.35	83,354,386.41 200.38% 867.77	125,058,149.79 300.64% 1,302.45	166,656,587.36 400.64% 1,735.49
0.1ms	42,376,089.25 101.87% 4,417.15	84,774,984.39 203.80% 8,830.27	127,159,841.64 305.69% 13,245.34	169,544,797.57 407.58% 17,660.40

取一個 Process 和 Timer quantum 爲 0.1ms 的項目，即運行 H.264 Intra frame 編碼器每 0.1ms 就會被強迫進行 Context switch，跟沒有 Scheduler 的條件下的 ARM 端 Timer count，來計算平均每次 Context switch 所需時間，ARM timer 是以 96Mhz 在運行，所 timer count 再除以 96Mhz，可得到時間：

(42,376,089.25 – 41,597,727.35) / 4,417.15 ≈ 176.21 (ARM timer count)

≈ 0.001836 ms

另外再由 DSP 來測量 Context switch 的時間，如 Figure 49，以離開 Timer ISR 前的 DSP Timer count 來計算實際花費時間，DSP timer 是以 12Mhz 在運行，所 timer count 再除以 12Mhz，可得到時間

(0x0000EA5F – 0x0000EA54)*2 = 22 (DSP timer count)

≈ 0.001833 ms

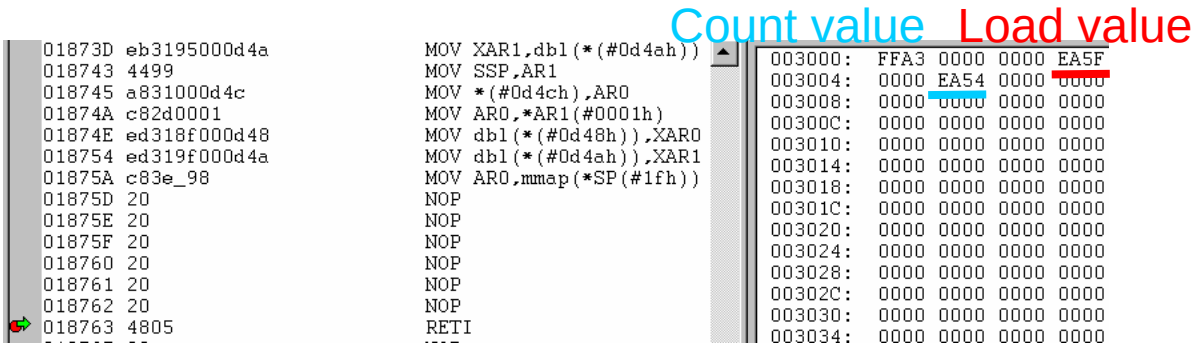
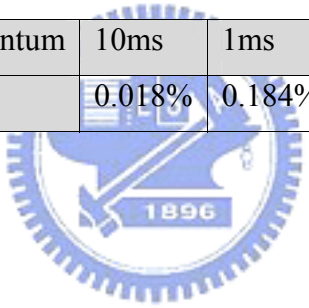


Figure 49. Context Switch Time Measured by DSP Timer

分別由 ARM 和 DSP 所測量的時間都蠻接近的。根據對 DSP 上運行服務的反應速率的需求，可以選擇不同的 Time quantum。而 Scheduler 所帶來的 overhead 如 Table 10。

Table 10. Scheduler Overhead vs Time Quantum

Time Quantum	10ms	1ms	0.1ms
Overhead	0.018%	0.184%	1.836%



5.3. DMA 效率

由過去學長的論文[8]，可得知 OMAP5912 的 System DMA 用來在兩個核心的資料搬移，效率大不如 DSP DMA。而且並沒有以 Burst 模式來測試，所以將會測試 DSP DMA 在 Burst 模式的效率。並且還測試在 3.2.4 小節所提到的，以 ENDPROG 的方式連續下兩個 DMA 命令，來搬移來源和目標的資料單位大小不同之大筆資料。

測試方式是由 Shared memory 上，搬移 32Kbyte 到 DSP 內部記憶體。測試結果如下，其中參考了跟 OMAP5912 類似架構的數據，是由 TI 官方測試的 OMAP5910 平台上的 DMA 數據[18]，單位是每 16bit 的 Traffic Control(TC) Cycle 數，Table 11 的數據是以 OMAP5912 的 TC 的 96Mhz 運行速度，計算得來的。

Table 11. DSP DMA 16to16 Performance with Burst Mode

MB/s	W/o Burst	W/ Burst	5910 Spec w/o Burst	5910 Spec w/ Burst
SDRAM->SARAM	15.858	55.473	16.422	62.923
SDRAM->DARAM	15.855	55.506	16.422	62.923
SRAM->SARAM	21.254	60.995	36.621	104.632
SRAM->DARAM	21.254	60.995	36.621	104.632

如 Table 12，來源和目標都是以 16bit 爲單位搬移 32Kbyte，跟來源是 8bit 而目標 16bit 的搬移，以比較出兩次 DMA 搬移和單純用 DSP 搬移的效率，並以 DSP DMA 的數據一起對比。可以看出，單位大小一樣時，DSP 用 load-store 的方式搬移，和沒有開 Burst 模式的 DSP DMA 效率是類似的，而單位大小不一樣時，用 DSP DMA 並開啓 Burst 模式來搬移明顯比 DSP 有效率。

Table 12. 16to16 and 8to16 Performance

MB/s	16to16 DSP	16to16 DSP DMA w/o Burst	8to16 DSP	8to16 DSP DMA w/ Burst
SDRAM->SARAM	15.000	15.858	6.015	34.722
SDRAM->DARAM	15.007	15.855	6.015	34.735
SRAM->SARAM	22.888	21.254	7.630	37.722
SRAM->DARAM	22.888	21.254	7.630	34.735

6. 結論與展望

6.1. 研究結果討論

在 5.1 的實驗中，我們分別測試出純 ARM、純 DSP 和動態精細分工法三種不同的編碼法各自最快的編碼速度。我們可以結論出，隨著 RISC 處理器的演進，過去靜態分析所設計的分工編碼方式，跟配合 RISC 一起協同運算的動態分工法效率比較起來，對於某些多媒體的工作，即使只有一個工作需要執行，也有可能比較慢。如果在 DSP 有多個工作要運行的環境下，動態精細分工的方式更能夠分配較多的工作給 RISC，以減輕 DSP 上的負擔。

雖然我們的實驗結果有可能是因爲對 DSP 程式並沒有用組語做到最佳化，導致純 ARM 的編碼效率比純 DSP 還快，但本論文研究的重點之一，也正是要能讓程式設計師幾乎不額外花工夫進行異質雙核心平台的最佳化，就能自動獲得效能的提昇。而且新一代 RISC 的架構及 Cache 設計所帶來的效率提升，也是不可以忽略的，在 TI 的新一代異質雙核心平台 DM64x 系列，也以 Data Cache 來取代了 DSP 內部記憶體[19]，比起要以特別的存取資料方式來配合內部記憶體的使用，Data Cache 可以很容易提升資料存取效率。

在 5.2 的實驗，測試出我們設計出的 DSP Scheduler 的效率是不錯，在 10ms 的 Time quantum 設定，對整體視訊編碼影響的 overhead 不到 0.02%。並且支援動態服務的抽換，隨著嵌入式應用的普遍化，這樣支援動態服務的多工環境，可以很容易隨使用者的需求，同時提供多個服務的運行。

6.2. 未來工作及展望

未來主要有兩個方向是需要繼續努力的。

- 異質雙核心的 Compiler

  如 3.2.1 最後所提到的，目前爲了達成傳遞參數給在 DSP 執行服務的工作，是以人工撰寫參數傳遞的函式，如有 compiler 能自動對主要工作函式所需的參數，產生有效率的組語，將可以簡化移植的作業。

  並且在 3.1.3.3 所提到的 Dual-core service image 如能由專爲異質雙核心所設計的 compiler 所產生，將能簡化在這樣平台上的程式開發和分工。

- 針對視訊編碼所需資料的設計配合內部記憶體的 DMA 存取演算法

  ARM 只有 8Kbyte 的 D-Cache，就能動態隨著資料存取，將整個要編碼的和重建的 Frame 的都快取在 D-Cache 裡。DSP 雖然有總共 80K 個 16bit 的內部記憶體，無法同時將兩個 Frame 載入到內部記憶體，使得要以比較慢的方式存取位於 Shared memory 上的兩個 Frame。如能針對目前視訊編碼所需的資料，以簡單的 DMA 命令，動態地將資料搬到內部記憶體再來存取運算，達到使用 D-Cache 的效果，一定能提升 DSP 視訊編碼的效率。

- 硬體平台的改進

  根據本論文的研究，現有的異質雙核心平台架構都是針對傳統的靜態工作切割的運作模式設計的，因此並不適合執行動態精細分工的系統。從我們系統實作的經驗，可以發現，如果雙核心之間能透過較有彈性的匯流排架構和分散式的記憶體模組來設計溝通介面，將可大幅提昇動態精細分工系統的效能。

7. 參考文獻

[1] Cheng-Nan Chiu, Chien-Tang Tseng, and Chun-Jen Tsai, “Tightly-coupled MPEG-4 video encoder framework on asymmetric dual-core platforms,” *Circuits and Systems, 2005. ISCAS 2005. Vol. 3, Pages: 2132-2135*, May 2005.

[2] J. Chaoui et al., *OMAP: Enabling Multimedia Applications in Third Generation (3G) Wireless Terminals, Texas Instrument Technical White Paper SWPA001*, Dec. 2000.

[3] *ISO/IEC 14496-2 Info. Tech. -Coding of audio-visual objects- Part 2: Visual*, 3nd edition, April, 2003.

[4] Rudy Lauwereins, Chun Wong, Paul Marchal, Johan Vounckx, Patrick David, Stefaan Himpe, Francky Catthoor, Peng Yang, "Managing Dynamic Concurrent Tasks in Embedded Real-Time Multimedia Systems," *isss*, pp. 112-119, Proceedings of the 15th international symposium on System Synthesis (ISSS '02), 2002.

[5] F. Thoen and F. Catthoor, *Modeling Verification and Exploration of Task-level Concurrency in Real-Time Embedded System*, Kluwer Academic Publishers, 1999

[6] 歐旭江, ”The Design and Implementation of a Dataflow Kernel in Heterogeneous Multiprocessor Enviroment,” *Master Thesis*, National Chung Kung University, 2005.

[7] Ming-Yu Hung, Bo-Syun. Hsu, Yung-Chia Lin and Jenq-Kuen Lee, “MDI: An Annotation Language for Heterogeneous DSP Processors,” National Tsing Hua University.

[8] Cheng-Nan Chiu, “H.264 Video Encoding Optimization on Dual-Core Platform,” National Chiao Tung University, June 2005.

[9] Joint Video Term of ITU-T and ISO/IEC JTC 1, “Drift ITU-T Recommendation and Final Draft International Standard of Joint Video Specification (ITU-T Rec. H.264 | ISO/IEC 144496-10 AVC),” May 2003.

[10] Rishi Bhattacharya, “DSP Booting Example,” *System Initialization for the OMAP5912 Device, SPRA828A,* Texas Instruments, Dallas, Texas, August 2002.

[11] Texas Instruments, “Hex Conversion Utility Description,” *TMS320C55x Assembly Language Tools User's Guide, SPRU280H*, Texas Instruments, Dallas, Texas, July 2004.

[12] Texas Instruments, “DSP Bootloader,” *OMAP5912 Multimedia Processor DSP Subsystem Reference Guide, SPRU890A*, Texas Instruments, Dallas, Texas, May 2005.

[13] Texas Instruments, “Linker Description,” *TMS320C55x Assembly Language Tools User's Guide, SPRU280H*, Texas Instruments, Dallas, Texas, July 2004.

[14] Texas Instruments, “Common Object File Format,” *TMS320C55x Assembly Language Tools User's Guide, SPRU280H*, Texas Instruments, Dallas, Texas, July 2004.

[15] Texas Instruments, “Stack Operation,” *TMS320C55x DSP CPU Reference Guide, SPRU371F*, Texas Instruments, Dallas, Texas, February 2004.

[16] Texas Instruments, *TMS320VC5509 Digital Signal Processor Silicon Errata, SPRZ006E*, Texas Instruments, Dallas, Texas, August 2004.

[17] Texas Instruments, *TMS320C55x DSP Mnemonic Instruction Set Reference Guide, SPRU374G*, Texas Instruments, Dallas, Texas, October 2002.

[18] Texas Instruments, *OMAP System DMA Throughput Analysis, SPRA883*, Texas Instruments, Dallas, Texas, December 2002.

[19] Texas Instruments, *TMS320DM6443 Digital Media System-on-Chip, SPRU282C*, Texas Instruments, Dallas, Texas, June 2006.

[20] Texas Instruments, *OMAP5912 Applications Processor Data Manual, SPRS231E*, Texas Instruments, Dallas, Texas, December 2005.

[21] Spectrum Digitail, *OMAP Starter Kit(OSK) OMAP5912 Target Module Hardware Design Specification,* Spectrum Digitail, Stafford, Texas, July 2006.