行政院國家科學委員會專題研究計畫　成果報告

子計畫二：多媒體通訊數位基頻 SoC 加速架構及嵌入式作業系統界面的研究(3/3)

計畫類別： 整合型計畫
計畫編號： NSC94-2220-E-009-008-
執行期間： 94 年 08 月 01 日至 95 年 07 月 31 日
執行單位： 國立交通大學資訊工程學系(所)

計畫主持人： 蔡淳仁

計畫參與人員： 王岳宜、蕭哲民、蘇郁淵、李國丞、林君玲

報告類型： 完整報告

處理方式： 本計畫可公開查詢

中　華　民　國 95 年 10 月 31 日

行政院國家科學委員會專題研究計畫成果報告

MPEG-4/21 SOC 設計及新世代行動通訊之研究-

子計畫二：多媒體通訊數位基頻 SoC 加速架構及嵌入式作業系統界面的研究(3/3)

計畫編號：NSC 94-2220-E-009-008

執行期限：94 年 8 月 1 日至 95 年 7 月 31 日

主持人：蔡淳仁 國立交通大學資訊工程系

參與人員：王岳宜、蕭哲民、蘇郁淵、李國丞、林君玲 國立交通大學資訊工程系

中文摘要

本子計畫的主要目的是在研究多媒體數位通訊基頻 SoC 的應用程式加速架構，以及異質多核心的作業系統分工排程器的設計。傳統的通訊基頻晶片，只提供 layer 2 以下的運算功能，以及語音壓縮解壓縮功能。但為了有效支援新一代多媒體通訊應用，許多基頻晶片大廠如 TI、Freescale、及 Qualcomm 都已經推出了整合多媒體甚至 Java 加速功能的單一基頻晶片。在三年的整合計畫中，本計畫主要的完成項目有下面幾項。首先是設計了一個多標準視訊編碼硬體加速 SoC 平台，以利軟硬體協同設計。這個平台有別於以往針對單一多媒體壓縮標準而做的純硬體佈線的設計。本平台的架構設計是以能直接支援 MPEG 正在發展中的 Reconfigurable Video Coding (RVC) 的技術為目標。其次本計畫在異質多核心的作業系統分工排程器的設計方面，完整地在 TI OMAP OSK5912 平台上發展完成了一套 Heterogeneous Multi-Processor (HMP) 的動態工作切割排程作業系統核心，以善用 arm 以及 dsp 雙核心效能，並證明在複雜的多媒體應用上，其效能會比業界慣用的動態工作切割排程雙核心系統好。最後，本計畫也研究了適用手機的 Java VM 環境的加速功能。本計畫以一套公開 RTL 程式碼的、功能類似 KVM 的 Java Processor －JOP 為出發點，設計了一套創新的 Dynamic Code Optimization 的加速機制，把 JOP 在 Spartan III FPGA 上的效能提昇超過 10%。整體而言，本子計畫大體完成了最初計畫提案中每一項提到的可以整合到基頻晶片的應用程式加速功能。

關鍵詞：多媒體通訊、嵌入式作業系統、數位基頻晶片、可動態調整視訊編碼器、Java 處理器

Abstract

The goal of this project is to design an application acceleration architecture that can be integrated into a multimedia communication baseband SoC, and a OS kernel scheduler for dynamic partitioning of tasks for heterogeneous multi-core systems. Conventional baseband processor only provides computational acceleration for speech codecs and network protocol stacks at layer-2 and below. However, in order to support new multimedia communication applications efficiently, new chip venders such as TI, Freescale, and Qualcomm have all announced baseband chipset with multimedia acceleration capabilities. For the past three years, our project team has completed the following major tasks. First of all, we have designed a multi-format video codec acceleration SoC platform. The platform is different from the conventional codec SoC that is hard-wired for a particular codec. Instead, we have followed the latest Reconfigurable Video Codec Framework Standard that is being developed within MPEG. Secondly, we have designed a dynamic task partitioning OS kernel scheduler for heterogeneous multi-processor (HMP) platforms. The design is completely implemented in a embedded prototyping board based on TI-OMAP 5912. We have also shown that for complex multimedia applications, this dynamic partitioning approach can outperform the traditional static partitioning approach. Finally, we have also investigated techniques to accelerate hardware-based Java Runtime environment. The project team developed an innovative Dynamic Code Optimization for Java processors. The proposed technique is implemented for an open source Java processor, JOP, on Spartan III FPGA and obtains over 10% performance gain. In summary, we have accomplished all the goals listed in the original project proposal.

Keywords: multimedia communication, embedded OS, digital baseband processor, reconfigurable video coding, Java processors

目錄

一、 前言 2
二、 研究目的 2
三、 文獻探討 4
四、 結果與討論 4
五、 計劃成果自評 4
附錄一、 可重組的視訊加速 SoC 平台 6
附錄二、 異質多核心作業系統動態分工排程器 12
1. 簡介 12
2. 相關研究 13
2.1 多核心平台排程演算法 13
2.2 對稱式多核心平台與非對稱式多核心平台 14
2.3 同質多核心平台系統下的非對稱式排程 15
2.4 動態式分工及排程 16
2.5 共享資源控制 17
2.6 靜態式分工 18
3. 理論與實作背景 18
3.1 OMAP 5912 Application Processor 19
3.2 OMAP 5912 Starter Kit：OSK 5912 (OMAP 5912 OSK) 19
3.3 eCos 20
3.4 eCos Overview 20
3.5 Configure Tool 20
3.6 Component 20
i. HAL 21
ii. The Kernel 23
iii. The Scheduler 24
a. Multilevel Queue Scheduler 24
b. Bitmap Scheduler 24
iv. Synchronization Mechanisms 24
v. Threads and Interrupt Handling 25
vi. RedBoot 25
3.7 移植 eCos 到 OMAP 5912 26
4. 異質多核動態分工排程器設計 27
4.1 Scheduler API 27
4.2 Service Registrar 和 Core Service Table 28
4.3 Dispatcher 28
4.4 Task Dispatcher 29
4.5 Task Terminator 30
4.6 Loading Tables 30
4.7 雙核心溝通方法 31
4.8 DSP API 32
5. 實驗結果 32
5.1 實驗環境 32
5.2 動態排程實驗 33
5.3 相異處理器實驗 34
5.4 相異 bit rate 實驗 36
5.5 DSP delay 實驗 37

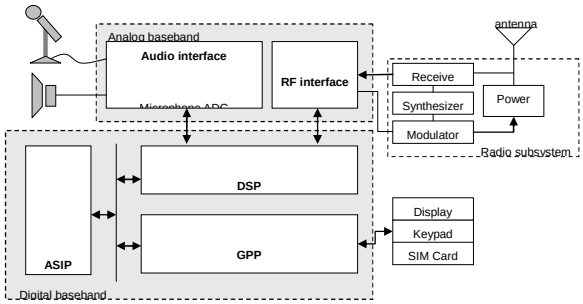
6. 結論與展望 38
7. 參考文獻 39
附錄三、 Java 處理器加速機制的設計 41
1. Introduction 41
1.1 Why Dynamic Code Optimization (DCO) 41
1.2 Dynamically-Typed OO Languages 41
1.3 Dynamic Message Sending 41
1.4 DCO for Java VM Using HW/SW Co-design Approach 43
2. Related Work 43
2.1 Previous DCO Mechanisms 43
2.2 Lookup Cache Mechanism in Smalltalk-80 43
2.3 Inline Cache Mechanism in Smalltalk-80 44
2.4 Polymorphic Inline Cache in SELF System 45
2.5 Java Virtual Machine Reference Implementation 46
2.6 Sun's K Virtual Machine Reference Implementation 49
3. Proposed Dynamic Code Optimization System 50
3.1 Data Structure Using in Our Dynamic Code Optimization 50
i. Data Arrangement in the External Memory 50
ii. Method Cache 51
iii. Runtime Data Structure 52
a. Stack Frame 52
b. Data layout 53
c. Runtime Class Structure 53
3.2 The Proposed Dynamic Code Optimization Scheme 53
i. Analysis of Bytecode Execution Frequency 54
ii. Access Time of External Memory & Internal Memory 54
iii. Architecture Overview 55
3.3 Implementation Details 56
i. Hardware Modules 56
ii. Software Modules 57
4. Performance Study 58
4.1 Xilinx Spartan-3 Development Board 58
4.2 Java Benchmark Programs 58
i. Sieve of Eratosthenes 58
ii. Kfl 58
iii. UDP/IP 59
4.3 Experiment Results 59
i. Execution Time 59
ii. Power consumption 59
iii. Microcode Execution Cycles 60
iv. External Memory Access Times 61
5. Conclusion and Future Work 61
6. REFERENCES 62

一、前言

本報告是過去三年的整合計畫的完整報告。在三年的整合計畫中，本子計畫大體完成了最初計畫提案中每一項提到的可以整合到基頻晶片的應用程式加速功能。首先，我們設計了一個多標準視訊編碼硬體加速 SoC 平台。並在 ARM Integrator 的 SoC Emulation Platform 上驗證了這個平台。這個平台有別於以往業界針對單一多媒體壓縮標準而做的純硬體佈線的設計。本平台的架構設計是以能直接支援 MPEG 正在發展中的 Reconfigurable Video Coding (RVC) 的技術為目標。其次本計畫在異質多核心的作業系統分工排程器的設計方面，完整地在 TI OMAP OSK5912 平台上發展完成了一套 Heterogeneous Multi-Processor (HMP) 的動態工作切割排程作業系統核心，以善用 arm 以及 dsp 雙核心效能，並證明在複雜的多媒體應用上，其效能會比業界慣用的動態工作切割排程雙核心系統好。最後，本計畫也研究了適用手機的 Java VM 環境的加速功能。本計畫以一套公開 RTL 程式碼的、功能類似 KVM 的 Java Processor－JOP 為出發點，設計了一套創新的 Dynamic Code Optimization 的加速機制，把 JOP 在 Spartan III FPGA 上的效能提昇超過 10%。

二、研究目的

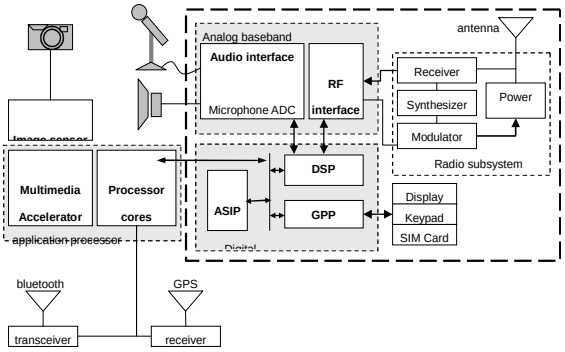
未來行動通訊網路及相關應用一定會成為後 PC 時代的主要科技產業．雖然各種寬頻行動網路（CDMA-2000, WLAN, UMTS/WCDMA）的架設已慢慢成熟，電信業者也推出各種行動數據服務，但行動寬頻網的主要訴求：多媒體通訊應用,卻遲遲不能起飛。其中最主要的理由是多媒體手機的設計一直無法達到理想的境界。由於多媒體資料的傳輸及處理具有高運算量的特性，手機的體積又要越做越小，耗電量又要低，因此一個專為多媒體演算法及傳輸協定設計的低耗電整合式系統晶片（System-on- Chip, SoC）是手機及行動通訊設備的關鍵元件。而想要把複雜的數顆晶片整合到一顆系統晶片，最大的挑戰，除了硬體線路的整合，還包含了內嵌韌體的整合。以國外一流大廠的手機為例，最早的多媒體手機大概要用到十多顆晶片，而最新的手機大概只用到五、六顆晶片，最主要的關鍵，就是把不同網路的基頻晶片和應用處理器整合在一起。過去台灣工業界在 IC 元件設計方面算是具世界水準，但是對於高度整合系統晶片所需的複雜韌體的設計則較缺乏經驗。因此在系統設計上，有時甚至於會“棄軟從硬”，一些用韌體可達到的功能，反而用硬體線路來取代，這樣的做法，也許可以減少需要同時維護軟硬體的麻煩，但是卻阻礙了系統晶片的彈性和成長空間。



圖一、二代手機架構

傳統的二代手機包含了射頻模組、類比基頻模組、數位基頻模組、和耗電管控模組等等（圖一）。其中，數位基頻帶模組是負責語音編碼、容錯編碼、通訊協定處理、短訊服務處理、及使用者界面等工作。在架構上，一般是採用了双處理器核心的設計。也就是包含一個低效能嵌入式通用微處理器（GPP）核心加上一個訊號處理器（DSP）核心。其中 GPP 所執行的任務包含了 layer 2/3 通訊協定的處理、簡訊服務處理、人機界面、及簡單的手機作業系統等，而 DSP 所處理的工作包含 layer 1 通訊協定、語音壓縮解壓縮等等。這樣的架構應付傳統的語音通訊需求已經足夠，但卻不足以滿足多媒體通訊在效能上的要求。特別是如果把低耗電的要求再加上去，更是一個複雜的問題。為了在傳統手機架構上很快加入多媒體的功能，國外手機晶片大廠的做法是另外在系統中加入一顆應用程式處理器（Application Processor）（如圖二）。在這邊要特別強調的是，對多

媒體手機而言，應用程式處理器的存在只是一個過度時期的需求。如果我們仔細分析一下應用程式處理器的架構,其實跟(數位）基頻處理核心是十分接近的，除了有一個通用微處理器外，也常常包含一個 DSP 核心及一些多媒體加速邏輯。那麼，為什麼不乾脆擴充基頻處理核心的能力來取代掉應用程式處理器的功能呢？一開始的多媒體手機不這樣做，最主要的理由並不是硬體架構上難以達成這樣的目標，而是在韌體的整合上有其困難度。



圖三、初期多媒體手機架構

以現在的行動多媒體應用硬體架構而言，有以下的特點是過去嵌入式作業系統設計時所沒有考量到的。第一、現今的 GPP 核心（通常為 RISC 架構）效能比起早期的雙核基頻晶片的 GPP 核心要強多了，而且也常常內建一些訊號處理的加速指令，第二、新的嵌入式多媒體應用，常常要同時執行好幾個弱即時(soft real-time)的多媒體工作（包含視訊、音效、繪圖等等），因此，傳統上，針對單一時刻只執行單一耗計算量的工作的異質核心軟體分割方法(software partition，如圖二)，常常在動態執行的狀況下不能達到最佳效能及工作分配(load balancing)，第三、多媒體應用的瓶頸往往是卡在記憶頻寬上，因此，新一代的嵌入式系統記憶體架構往往採用了異質分散式記憶體架構(heterogeneous distributed memory blocks)，配合較有彈性的晶片內嵌元件的連結通道(interconnect)，這也是在過去的作業系統的記憶體管控模組所沒有的設計。

總結一下前面的討論，當嵌入式系統的設計越趨複雜，系統晶片的整合度越高，有效率的嵌入式系統和系統晶片的軟韌體開發將會是未來這個領域的技術重心所在。然而嵌入式軟體的開發並不能直接根據 PC 經驗而有樣學樣，必須根據新應用來思考新方向，特別是在嵌入式作業系統的設計上，不僅是要重新思考排程器(scheduler)和記憶體管控模組(memory manager)的架構，甚至於應該重新設計程式設計模式，才能達到最佳效果。

另外，在多媒體硬體加速架構方面，過去的業界習慣針對一個多媒體標準進行純硬體最佳化的設計。但是由於現在的數位多媒體的標準一直在演進中（如 video codec 從早期的 MPEG-1/2 到現在的 MPEG-4, H.264, 及 WMV 9，傳輸協定也會從早期的 MPEG-2 transport 慢慢往 IP network 的方向走，而 Java profiles 及 presentation scripting language 如 SMIL 等標準也一直在增訂中)。在過去採用純硬體佈線（hardwired）的方式設計系統的年代，升級到支援新標準的平台往往只能透過購買全新的硬體設備來達成。在未來服務導向的時代，這樣的設計只會增加使用者的負擔而阻礙新服務的推出，因此現在一個設計完善的多媒體平台一定要具有高度使用者擴充性。換句話說，當新服務推出時，使用者不需要購買一個新的設備，只要升級現有平台的韌體或可程式化邏輯元件就可達到享用新服務的目的。

基於以上的理念，配合 MPEG 正在制訂中中的 Reconfigurable Video Coding (RVC) 標準，在 FPGA 的發展板上設計一個具擴充性的多媒體加速平台。這個平台的開發採用先進的軟硬體協同設計（hardware/software co-design）概念，以最少的硬體設備達到最大的應用軟體效能。並維持軟硬體的可擴充性。

另外，Java 幾乎已經是嵌入式系統的應用程式的標準環境了。在手機上，Java 應用程式必須符合 CLDC/MIDP/KVM 的規範。由於早期手機的 Java 環境都是用內嵌式處理器來執行軟體 VM 模擬器，所以效能不彰，而使得應用程式的開發受限。雖然目前有一些 Java 加速器的設計，但在手機上除了 ARM 的 Java 副處理器之外，

都不算成功。我們在此計畫中，也花時間研究一個接近手機用 KVM 的公開硬體程式碼的 Java 處理器，並研究手機 Java 環境的效能加速法。

三、 文獻探討

本計畫的總合成果主要有三大方向，首先是以可擴充的視訊加速平台、在異質雙核心平台上設計的動態分工排程作業系統、以及手機用的 Java Processor 的加速。關於這三個研究方向的文獻探討，請參見綜合報告之後的成果報告一、二、和三。

四、 結果與討論

在可擴充的視訊加速平台的架構設計方面，我們是以能直接支援 MPEG 正在發展中的 RVC Framework 的技術為目標。在 RVC 的架構中，視訊編碼的工具（如 IDCT、VLC）是獨立掛在（軟體或硬體）平台上的一些 Functional Units。而這些 Functional Units 可以透過一個 table-driven 的 Finite State Machine (FSM) 來控制組成一個特定的 data path 來處理某一個視訊標準。如果這個 FSM 是以軟體實作，那個這個加速平台就可以做到動態重組成 H.264 或 MPEG-4 等不同視訊標準的功能。至於我們在這部份研究的詳細報告，請參見附錄一。

另外，在異質雙核心平台上設計的動態分工排程作業系統的設計方面。我們特別設計了一個程序排程器，可以動態根據 RISC core 和 DSP core 的負荷，來決定要叫用那一個版本的執行碼（RISC 或 DSP）來完成工作。另外，在動態分工的應用上，我們除了完成了 MPEG-4 的 encoder 和 decoder，另外也完成了 H.264 的 Intra encoder。關於這個異質雙核心平台的動態分工排程器的詳細報告，請參見附錄二。

最後，在 Java 處理器的加速研究方面，我們設計利用硬體記錄了每一個 byte code 的執行次數，對重覆執行，而且需要做動態 resolution 的指令，進行 dynamic code optimization 的動作。根據實驗，這樣的系統設計，可以得到相當不錯的加速。這部份的詳細報告，請參見附錄三。

五、 計劃成果自評

總合三年的成果，和原計畫提出的目標相當吻合。在達成預期目標情況方面有以下數點：

1. 多媒體雙核心系統中，在 TI OMAP OSK5912 平台上發展一套 Heterogeneous Multi-Processor (HMP) 的動態工作切割排程作業系統核心。在開發這個技術的過程中，我們有以下成果：
   I. 發展自己的 DSP scheduler 來幫助作 task 的排程。
   II. Porting eCos 到 TI omap 平台上，並開發雙核心有效的溝通協定。
   III. 設計一個可以配合 eCos MLQ scheduler 的動態分工模組。
   IV. 設計新的異質多核心的新 Programming model。
   V. 實作出支援動態分工應用程式的開發工具。
2. 視訊編碼硬體加速平台上，實作出以下 IPs：
   I. Motion-estimation：計算到達到 1/4 pel，參考多個 reference frames 以及所有 sub block 模式
   II. H.264 deblocking filter
   III. MPEG4 IDCT
   IV. H.264 transform/inverse transform unit
   V. H.264 quantizer and de-quantizer
   VI. H.264 intra predictor

   本平台的設計是以 MPEG 正在

發展中的 Reconfigurable Video Coding (RVC) 的架構為主要的設計目標，以期能支援不同視訊壓縮法的解碼器的動態產生。由於 RVC 為目前 MPEG 工作中的項目，所以目前的設計都是以軟體（C model 或其它 behavioral model 的模擬平台，如 Moses for CAL 來進行研究）。本團隊因為積極參與 MPEG 標準的制訂，所以能隨時根據最新的結果來修正設計這個平台。

3. 實作出 Java Dynamic Code Optimization for Java Processor 的軟硬體系統。

附錄一、可重組的視訊加速 SoC 平台

I. INTRODUCTION

Most multimedia devices today have to support multiple codec standards. Take video codecs for example, a portable multimedia player usually supports the playback of the MPEG-1/2, MPEG-4 SP, WMV, and H.264/MPEG-4 Part 10 video contents. In order to reduce system cost, a single-chip SoC solution that supports all these standards is a sensible approach. From IC designers' point of view this is not a serious problem since most (if not all) popular video codecs share the same block-based motion compensated transform coding data flow. In addition, many coding tools have similar architecture. However, there are some application issues that makes traditional codec design approaches unsatisfactory [1].

A major problem with existing approach of defining a codec standard is the lack of flexibility when new applications emerge. A video codec is composed of several coding tools (e.g. DCT/IDCT, MC, VLC/VLD, etc.). However, for a codec standard, the conformance point is defined at codec-level, instead of tool-level. Different profiles/levels are created for each codec to address the need of different classes of applications. This approach works fine in the past since the application scenarios were quite simple (e.g. DVD, DTV). However, with the exponential growth of new multimedia applications, the old approach of defining conformance point at codec-level becomes awkward. Quite often, a new application designer finds it impossible to find a reasonable codec profile@level to fit the target application well. For example, the FMO tool of H.264 is useless for many applications but a decoder may still need to support it simply because it is included in AVC baseline profile. In general, application environment is changing faster than an international standard can catch up that there should be a more efficient way of allowing a codec to adapt to new applications while maintaining interoperability among different solutions.

MPEG has recognized this issue and started a new work item called Video Coding Tools Repository (VCTR) in 2004. After some investigations, the direction and benefit of VCTR is becoming clear [2]. Later, this effort becomes the Reconfigurable Video Coding (RVC) framework in 2006 [3]. This new framework defines the conformance point at tool-level. Therefore, in principle, an RVC-enabled codec can negotiate on-the-fly with the video bitstream encoder/sender about which coding tools is required and how the data path can be wired among these coding tools in order to decode the video bitstream. After the setup stage, the decoder can decode the bitstream correctly. With this approach, an SoC can support multiple codec standards as well as creating customized codecs in real time as long as it contains all the standard-conforming tools that is necessary to decode bitstreams from different encoders.

So far, the RVC framework is still in development. Most of the investigations are done using C models and behavioral model

simulators such as Moses [4]. In this report, SoC architecture that can be used to implement the RVC framework is proposed. The report is organized as follows. The RVC framework is introduced in section II. The SoC architecture for direct support of RVC is presented in section III. Some comparisons of the RVC architecture to a common hard-wired solution is also given in this section. Section IV studies an implementation to get an idea on the cost for such flexibility. Finally, some discussions are given in section V.

II. MPEG RVC FRAMEWORK

The concept of MPEG RVC framework can be illustrated in Fig. 1. The key difference between RVC and the old MPEG codec standards is that the interface of each coding tools is defined precisely so that they can be used (like LEGO blocks) to build various codecs. The decoder configuration describes how input bitstream can be parsed so that the raw input data to each coding tools can be extracted. A decoder description language is under development so that the configuration of a specific codec (such as H.264) can be described using a (small) configuration bitstream. The decoder configuration bitstream will be processed by an RVC decoder before decoding of a video bitstream conforming to the described standard. Note that after processing a configuration bitstream, the RVC decoder will generate a Global Control Unit (GCU) that governs the operation of the coding tools.

In principle, the configuration description tells the RVC decoder how to wire the coding tools to form a data path. In the RVC framework, each coding tools is called a functional unit (FU) and is specified in Fig. 2 [1]. In Fig. 2, a control signal is a signal embedded in the video bitstream (for example, the width and height of the video frame). A context signal is a signal generated from the processing of bitstream data (for example, the AC prediction direction in the MPEG-4 Part 2 video standard). The context-control unit reads in the context and control signals generated by previous FU's and generates (or passes on) some context and control signals to the next FU's based on the result of the processing unit.

A partial example of a configured RVC codec that behaves like an MPEG-4 Simple Profile video decoder is shown in Fig. 3. In Fig. 3, VLD is the FU for variable length decoding, RLD is the FU for run-length decoding, and MBG is the 8x8 block coefficients composition FU.

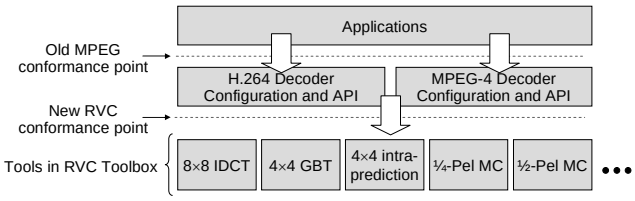


Fig. 1. Concept of MPEG RVC framework

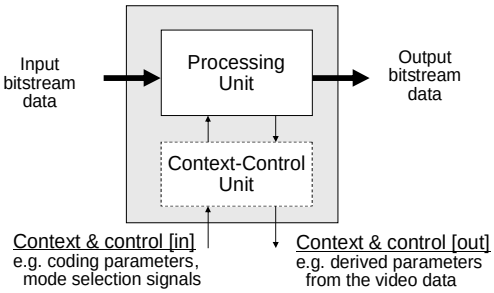


Fig. 2. Definition of an FU in RVC

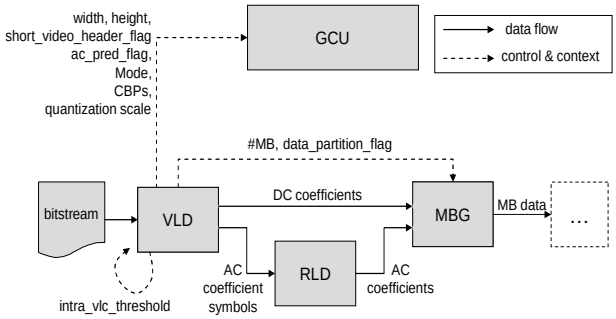


Fig. 3. Example of RVC configuration

III. SOC ARCHITECTURE FOR RVC

Since the specification of the video decoder configuration language and the actual mechanism of a GCU are still under development at MPEG, this report proposes a potential VLSI architecture that is suitable for supporting the RVC framework and perform some early analysis on such architecture. The RVC framework actually fits the platform-based design principle of SoC quite well. For maximal flexibility, the GCU will be implemented in software and running on the processor core of an SoC. Each coding tool can be implemented as an IP on the bus with limited configurability via a private register file. The proposed architecture is show in Fig. 4.

In Fig. 4, the coding tools are not attached to the main system bus (AMBA AHB) directly. A local bus, MMB, is used to off-load the bandwidth from the main system bus. Here, MMB stands for Multi-Media Bus. In our implementation, the bus protocol of MMB is a simplified version of AHB. A two-way DMA is used to transfer data between external SDRAM and internal SRAM banks. The DMA can be invoked from either the ARM core or the coding tool IPs (as long as the tool is implemented as an MMB master). The reason for multiple SRAM's on the MMB is to reduce the memory bandwidth requirement for parallel operations of the coding tools.

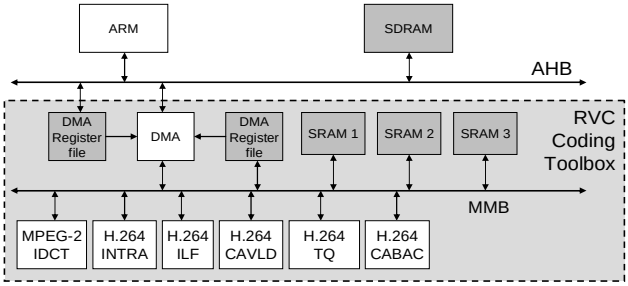


Fig. 4. SoC architecture for RVC framework

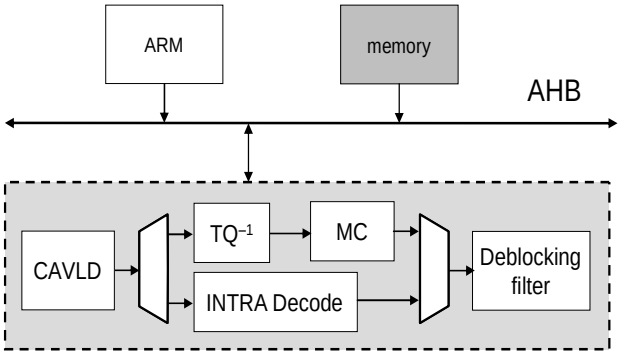


Fig. 5. Hard-wired decoder example

Although local bus and multiple SRAM banks are used to alleviate the bandwidth issue, the performance of this architecture still cannot match that of a hard-wired architecture. For example, a hard-wired H.264 baseline decoder may have a tighter MB decoding pipeline as shown in Fig. 5. There are two main advantages of the architecture in Fig. 5. First of all, the decoding pipeline is controlled by a hard-wired FSM with cycle-based synchronization. On the other hand, for the RVC framework, the controller will be implemented in software, and

hence, cannot guarantee cycle-based operation of the pipeline. Another advantage of the hard-wired approach is that it does not require excessive accesses to external memory.

It is important to point out that the purpose of the RVC framework is not to obtain the most efficient design of a single codec, but to allow a flexible and extensible design of codec systems. Multi-standard codec support (or even generate customized codec on-the-fly) can be achieved by configuring a new GCU via decoder description bitstreams. In the next section, we will study an actual implementation of the proposed architecture in Fig. 4 to get an idea about the cost one has to pay for such flexibility.

IV. IMPLEMENTATION STUDY OF THE PROPOSED SYSTEM

In this section, an implementation of the proposed system architecture (Fig. 4) is investigated. The implementation is based on an SoC emulation platform, the ARM Integrator [6]. The platform is composed of a main board, an ARM 9 processor core module, and a Xilinx VirtexE XCV2000E FPGA logic module. The platform adopts the AMBA bus protocol. The RVC coding toolbox logic of the proposed system is implemented in the FPGA. The local bus protocol, MMB, of the toolbox logic is a reduced version of AHB with much less wires and a minimal implementation of bus arbiter and decoder.

In the proposed system architecture, the finite state machine (FSM) that drives the operation of the coding tool FUs is implemented in software. As a result, the codec pipeline is not executed in a lock step fashion but instead driven by the software FSM via control signals. Each coding tool FU (please refer to Fig. 2) is implemented so that the input bitstream data is coming from a SRAM bank on the MMB and the output bitstream data will be stored in another SRAM bank on the MMB. Block RAMs of the Virtex II FPGA and the ZBT SRAM of the ARM Integrator are used for this purpose. Table I and Table II list the required memory for the input data and output data. It is obvious that such implementation is not as efficient as a tightly-coupled pipeline [5] where different pipeline stages are connected via registers or FIFO.

On the other hand, since the system control FSM is implemented in software, the Global Control Unit of the MPEG RVC framework can be dynamically implemented using this FSM. Therefore, any video decoders can be emulated on-the-fly by the proposed architecture as long as all the coding tools required by the target codec are supported by the architecture. Therefore, the proposed architecture is very flexible and scalable. It is important to point out that in order to support dynamic reconfiguration of the RVC decoder, the software-based system FSM shall not be a hard-coded FSM. Instead, it should be implemented as a table-driven FSM where the table content can be modified by the RVC decoder configuration bitstream.

The implementation of the processing unit and context-control unit of a coding tool FU follows traditional hard-wired IP design methodology where the processing unit is

implemented as a data path and the context-control unit is a hard-wired FSM with register files for memory-mapped I/O configuration and signaling. Currently, most of the FUs supported in the proposed platforms are for H.264. The synthesis report of some of the implemented FUs is shown in TABLE III.

V. CONCLUSIONS

This report introduces the MPEG RVC framework and proposes an SoC architecture to support the framework. Since the RVC framework is still under development at MPEG. There is not much research on how the framework can be efficiently supported using an SoC platform design paradigm. The table-driven software FSM for dynamic generation of a GCU and the decoder configuration language is still yet to be defined by MPEG. However, based on our study, the proposed architecture is very feasible for practical SoC implementation of the RVC framework. Although a reconfigurable video codec cannot compete with a hard-wired codec for performance given current VLSI implementation technology, it is much more scalable in the sense that any new codecs (coding tools) can be added into the platform with minimal effort.

TABLE I. The data size from external of the FUs in the proposed RVC architecture

Data form External Memory					
Intra predictor(input)	Luma 256 bytes		Chroma 128 bytes	Total 384 bytes	Cycles 96
Deblocking filter(input)	Luma 384 bytes	Chroma 256 bytes	Block info 120 bytes	Total 760 bytes	Cycles 190

TABLE II. The data size from internal memory of the FUs in the proposed RVC architecture

Data from Internal Memory				
Intra predictor(output)	Residual Luma 256 bytes	Residual Chroma 128 bytes	Total 384 bytes	Cycles 96
TQ/TQ^{-1}(output)	Trans.&quant. Luma 512 bytes	Trans.&quant Chroma 256 bytes	Total 768 bytes	Cycles 192
TQ/TQ^{-1}(input)	Luma 256 bytes	Chroma 128 bytes	Total 384	Cycles 96

			bytes	
Deblocking filter(output)	Deblocked Luma 256 bytes	Deblocked Chroma 128 bytes	Total 384 bytes	Cycles 96
CAVLC(input)	residual data 768 bytes	MVD 64 bytes	Total 832 bytes	Cycles 208

TABLE III. The Synthesis Report of some Logics

Module name	H.264 Transform	Quantizer	Intra predictor 1 (other modes)	Intra predictor 2 (DC mode)	Inloop Filter	CAVLC*	MPEG-2 IDCT
Clock rate	72MHZ	NA	198 MHZ	158 MHZ	60MHZ	50MHZ	77MHZ
Logic size	252 LUTS	197 LUTS with MULT18X18 LE	879 LUTS	188 LUTS	3105 LUTS	3125 LUTS	3232 LUTS
Bandwidth	16/18 (output/clk)	1/1 (output/clk)	4/1 output/clk	1/1 output/clk (I4MB) 1/5 output/clk (I16MB)	2/1 (output/clk)	depend on content	64/158 (output/clk)
Memory usage	1 (16x16 bit)	96 words by 14 bits 52 words by 5 bits 52 words by 3 bits	NA	NA	16x384 bits	128x22-bit 16x16-bit	64x16 bit

*CAVLC is based on a Spartan II FPGA device, and the others are based on a VirtexE FPGA device

REFERENCES

[1] E. S. Jang, K. Asai, and C.-J. Tsai, *Study of Video Coding Tool Repository v5.0, MPEG Meeting Document N7329*, Poznan, July 2005.

[2] *C.-J. Tsai, Suggestions on the Direction of VCTR, MPEG Input Document M12074*, Busan, April, 2005.

[3] ISO/IEC MPEG Video Group, *Final Call for Proposals on Reconfigurable Video Coding, MPEG Meeting Document N8070*, Montreux, April 2006.

[4] J. Janneck et al., Moses Tool Suite, https://sourceforge.net/projects/mosestoolsuite/.

[5] T.-C Chen, Y.-W. Huang, and L.-G. Chen, "Analysis and design of macroblock pipelining for H.264/AVC VLSI architecture," *Proc. of IEEE ISCAS 2004*, Kobe, 2004.

[6] http://www.arm.com/products/DevTools/IntegratorAP.htm

附錄二、異質多核心作業系統動態分工排程器

1. 簡介

在街頭上，隨處可見用手機在聊天談事情的人們；或是掛著耳機，利用 mp3 播放器在聆聽音樂的青少年；上班族也幾乎用 PDA 取代了以往紙本記事的習慣。如此廣大流行的手持式裝置，如今越來越擴展它的應用層面，例如手機支援百萬像素的拍照功能，使手機也有了數位相機的能力；3G 的影像電話功能，不只是對話，也能同時看到對方的表情，讓遠距溝通變得更生動；mp3 播放器的文字瀏覽，秀圖系統甚至影片播放功能，令單純聽音樂的 mp3 播放器提高其附加價值，搖身一變成為微形的數位娛樂中心；另外 PDA 的衛星定位導航功能，打破了我們一向認為 PDA 只不過是個可以帶著走的超小型桌上電腦的既有想法，發揮了在移動力上的特性。相信未來必定會推出更強大更高品質的應用，使得嵌入式系統的複雜度迅速地提升，相對的嵌入式系統的工作效能也必需提高。

為了諸如此類眾多新的功能，以多媒體應用來說，嵌入式系統必需完成極大量的多媒體資料處理工作；換句話說，嵌入式系統要在相同甚至更短的時間內，處理更大量的資料，做更多的運算工作。提高嵌入式系統的能力是必要的。就過去電腦系統的發展史來看，提高系統的能力不外乎是提高處理器的能力為主，而處理器的能力就直接關係到它每秒可以運算的次數，每秒可以執行的計算量，亦即處理器的頻率。然而目前利用此一概念發展的單一核心嵌入式平台已不敷使用。

考慮手持裝置的特性：輕巧以及移動力佳。嵌入式平台便有了體積上的限制，其中便影響到一個重要的耗電量的問題。體積上的考量，手持裝置無法配置大容量大體積的電池，同時顧及其移動的特性，也無法接受一再需要補充電力的要求。提高核心頻率會消耗大量電力，這一點會成為嵌入式平台的致命傷，再者，高核心頻率相伴而來的是產生許多的熱量，散熱方面也是一個難題。在現今市場上，整體行動裝置效能的提升不是利用提高核心頻率的方法，而是以增加核心數(處理器數量)來平衡高核心頻率需求及大耗電量和高熱能產生的缺點。這種多個處理器的架構，我們稱之為多核心架構(multiprocessor architecture)。

事實上，異質多核心架構在嵌入式系統的發展已被業界廣泛地使用，例如德州儀器公司的 OMAP(Open Multimedia Application Platform)，以及 Freescale 的 MXC。在非對稱式多核心系統晶片(system-on-chip: SoC)架構裡，會有顆一般功能的處理器(general purpose processor: GPP)核心，做為嵌入式系統作業系統的控制核心，配上一顆數位訊號處理器(digital signal processor: DSP)核心。DSP 可以大量即時處理多媒體資料，如 MPEG 1、MPEG 2、MPEG 4 或是音訊資料等等。以德州儀器公司的 OMAP 5912 OSK (OMAP Starter Kit)為例 [1]，其 GPP 採用 ARM 公司 ARM926EJS，DSP 則是德儀自行研發的 TMS320C55X。研發人員可以依照資料處理的性質，將工作分配給 OMAP 架構微處理器中的 ARM 微處理器或者是 DSP 微處理器去處理。非對稱式多核心架構可有效率利的處理嵌入式系統上的工作(task)，發揮系統的最大效能，特別是對於多媒體的應用程式有令人亮眼的表現。

現存的即時作業系統(real-time operating system)對於非對稱式多核心架構大部份是採用靜態式分工(statically partitioned)的方法。所謂靜態式分工方法是系統設計時研發人員就做好工作的分配，屬於控制流程的工作就交由 GPP 執行，屬於多媒體運算處理的工作多交由 DSP 執行。在這種分工架構之下，有兩個不同的 schedulers 為兩顆核心獨立運行已分配好的工作。換句話說，兩顆核心各自處理已分配好份內的工作，完成之後，在下一個工作來臨之前是閒置的狀態，因為

在獨立的視野裡，已是最好的效能發揮。這類型的分工方式在傳統行動通訊平台及應用程式環境下是相當有效的法。過去常用的 GPP 核心在特殊工作處理的功能性和速度都有所不足，意即 GPP 沒辦法勝任 DSP 的工作。並且過去的嵌入式應用程式環境通常是單純的前景/背景(foreground/background)工作模式，所以不需用到複雜的動態排程。

但是新一代多媒體應用會拓展到更寬廣的層面，再加上硬體裝置上有了新的提升。首先，多媒體應用程式已經複雜到一個境界，為了提升系統效能和減低能量的消耗，必需用動態調整兩顆核心的工作量取代系統設計時做好的工作分配。其次是 GPP 的能力已被大幅提高，可以幾乎和 DSP 等速地處理某些多媒體資料，換句話說， 在這些情況下，GPP 可以用來分擔 DSP 的工作負載。接著是多媒體應用程式在記憶體和計算量的需求已經大大超越過往，多媒體資料經常會被包裝成運輸串流(transport stream)，往返於兩顆核心之間，但是在執行時核心之間溝通的成本並非固定，譬如傳輸時電力的消耗與總電量的關係、工作有沒有完成時間的限制(deadline)等……，有太多因素要考量，是不可能在系統設計時就預測到並且做好資料傳輸的設定。著眼於系統效能，靜態式分工系統設計不再合適，即時作業系統排程器在設計上要有新的突破。

考慮以上種種原因，我們便提出一種新的動態精細分工式(tightly-coupled)作業系統排程器[21]，[22]，這種新的排程法會由單一排程器監控各顆異質核心的工作狀態，並能動態地分配工作給當下最合適的處理器核心。排程視野的廣度上，由系統設計時就定好的靜態工作分配延伸到執行時的動態分配；而深度上，考量整個系統即時的狀況，做出最適當的工作分配並減少微處理器的閒置浪費，取得比靜態式分工系統更大的效能發揮。

2. 相關研究

這一章將會介紹此領域的相關研究。依順會介紹多核心平台著名的排程演算法，對稱多核心平台及非對稱式多核心平台，非對稱式多核心平台系統的排程，動態排程，共享資源的控制，非對稱式系統晶片 SoC，和靜態式分工系統。

2.1 多核心平台排程演算法

多核心處理器架構排程器的研究越來越受到重視。過去十多年來，在實用上，多核心排程演算法的發展重點是放在對稱式多核心系統(symmetric multiprocessor system)上。多核心排程技巧在同質多核心平台部份可以被分成兩類，partition scheduling 和 global scheduling[23]。Partition scheduling 是指每一個核心有自己的工作駐列(task queue)，包括 ready 駐列和 wait 駐列。工作排程的考慮會以各自區域的 priority 為主，與其他處理器獨立。每一個工作一旦被分配到一個處理器，在其生命週期內都不會移到別的處理器。Global scheduling 則是將所有準備完成的工作放在一個共同的 priority 駐列。最高 priority 的工作會被挑選放到一個工作量較低的處理器執行。這種 scheduling 模式在同質多核心的系統上表現較前者佳。

以下簡單列出一些常用的多核心排程演算法[3]:

- Rate monotonic： 每一個週期性的工作有固定的 priority，priority 的順序是根據該工作的執行頻率高低而定，例如要等待 interrupt 的工作，其 priority 相對較低。在 1973 年，Liu 和 Laylan 證明這個演算法是固定 priority 演算法中最理想的一種。
- Earliest Deadline First：這種演算法可將週期性和非週期性的工作一起排程，主要概念是越早結束的工作越先執行。M. L. Dertouzos 在 1974 年證實當瞬間有許多工作等待執行時，此演算法是最有效率的。
- Deadline Monotonic： D.M.結合上述兩種演算-- priority 的給定除了根據該工作的執行頻率外，另外會再考慮 deadline 越早，priority 越高。
- Background Scheduling： 此種演算法同時處理 soft real-time aperiodic task

和 hard real-time periodic task。兩種型式的工作分別置入兩個不同的駐列。此演算法實用上雖沒有很高的利用性，但其優點在於實作很簡單。

- Pooling Server： P.S.可處理非週期性的工作。每個時間區塊一過，server便服務下一個時間區塊可以執行的工作。若沒有工作在等待被執行，則會閒置 server，等到下一個時間間隔再甦醒。
- Deferrable Server： 此演算法類似上一個，但是若下一個時間區塊沒有等待被執行的工作，則 server 服務可能被服務的工作，而不是閒置 sever。
- Sporadic Server： S.S.使用於非週期性工作，可以增進其反應時間，使得非週性工作的效能追上週期性工作的效能。
- Dynamic Sporadic Server： 這個演算法利用 deadline 調整 priority，增進 Sporadic Server 的效能。
- Robust Earliest Deadline： 這是 1995 年 Buttazzo 和 Stankovic 發展的演算法，作用於 over loading 環境中的非週期性工作。此演算法不只可以減少 deadline 預測錯誤，也可降低系統 over loading 的程度。
- Constant Bandwidth Server： 這是在 1998 年 Buttazzo 和 Abeni 發展的演算法，用來解決即時多媒體應用的問題。例如在串流影音的系統中，對串流資料的傳輸和處理的 delay 和 jitter，必須要控制在一定的範圍內。
- Adaptative Bandwidth Reservation： Abeni 和 Buttazzo 在 1999 年提出對 constant bandwidth server 的改良。對於執行時間未知的工作所能分配到的處理器的頻寬可以經由 Adaptative Bandwidth Reservation 來控制。在這裡，頻寬(bandwidth)一詞指的是處理器分配給工作的時間或是工作被執行的週期。

2.2 對稱式多核心平台與非對稱式多核心平台

前面提到目前實用上多核心作業系統的排程演算法大部份都是以對稱式的多核心平台為目標，比方說，Satoshi Kaneko et al 在 2004 提出的一個多核心平台[4]。這個 600MHz 單晶片多核心平台包括兩個 M32R 32-bit CPU 核心，一個 512-KB 共用的 SRAM，和一個內部分享的 pipeline bus。

這個平台是由 0.15um CMOS 製程製造，適用於嵌入式系統。此多核心平台是對稱式的多程序處理平台，並且支援 modified-exclusive-shared-invalid (MESI) 的快取統一協定。該系統繼承了先前單晶片多核心平台的諸項優點，並針對嵌入式處理器做了最佳化，以使得系統效能增加的同時也能減低電力的消耗。為了增加核心的效能，他們在平台內部置入一個共享的 pipeline bus。此 bus 的特性是低延遲和每秒 4.8 G-bit 的大頻寬。此外也用多個低耗電技術，例如擁有不同使用電力的模型選擇: 睡眠模式、工作模式、和等待模式。不同系統情況下，不同核心甚至週邊有不同的模式選擇，以達到最高的省電約 18.4%。使得此多核心平台在 600MHz 1.5V 之下功作僅消耗 800 mW，待機時更只耗 1.5mW。

有些應用，如 3G 通訊和嵌入式多媒體應用，會同時執行控制的工作和大量資料處理的工作。一般實作上，為了達到最佳的性能／耗電量比值，異質多核心 (Heterogeneous Multi-Processor) 的架構是一般業界常用的設計方法[1], [13]。例如飛利浦半導體部門發展了一套 Silicon System Platform (SSP)。SSP 是零件的工具箱，是一種一般性、開放性和可程式化的架構。主要用來產生有軟體和硬體 IP blocks 的特定應用產品領域。過往研發新產品，可能必需打造整個新平台架構，付多相當的成本花費。利用 SSP 概念，為新應用產品而修改的架構會比試著去產生整個新架構更有實作的效率。使用 SSP 設計產品的速度很快而且技術風險低，因為架構中軟體硬體的功能性已經驗證過，而且還可以結合其他工作元件更容易達到設計的目標。同時其中有很大的空間讓設計團隊創造不同市場需求的產品；一系列的產品由入門到進階的產品，只需在平台上增

減功能區塊，就可有效地減少開發時間及成本花費。日後使用者甚至可以隨著更新軟體的版本來增強或增加產品的功能性。飛利浦的 Nexperia 平台是一個單晶片系統的 SSP，用來開發數位視訊產品。Nexperia 平台上主要包括 MIPS 處理器和飛利浦的 TriMedia VLIW 媒體處理器，及其他 IP 元件。結合 MIPS 及 TriMedia 兩種不同的計算核心，整合成單晶片系統。飛利浦利用此平台創造出多功能的機上盒，它可以即時解碼多個視訊串流、執行數位錄製、壓製訊號用於視訊電話、瀏覽網站和收發電子郵件等多項功能。其他如德州儀器(TI), 飛思卡爾(Freescale)和 Toshiba 等知名大廠都有自己的異質多核心平台，本論文在 TI OMAP 上實作，稍後章節將會詳細介紹 OMAP 平台。

在軟體的開發過程中，軟體測試是很重要而且很昂貴的一部份。有一種軟體測試的方法稱之為資料流測試(Data Flow Testing)，使用資料流測試可以決定一個軟體的測試是否充份且完整。Harrold 提出一個新的方法把整個資料流測試工作量切割成適當的大小[5]。這些測試的工作量可以靜態地也可以動態地接受排程。也可以改變成適合共用式記憶體或分散式記憶體的環境。在[5]中把資料流測試演算法實作出單一核心平台的版本和多核心平台的版本，並根據大量的軟體實驗來驗證資料流演算法的正確性。另外，這些實驗也可證實多核心平台的效能優於單核心平台。平均效能上多核心平台比單核心平台加速 1.7 倍。

Annavaram 等人討論過非對稱式多核心系統的效能優於對稱式多核心系統的看法[6]。激發此篇論文的研究動機有下例三點：首先，單晶片多核心平台上 CPU 核心的數目增加，同時間可以執行的運算量上升。第二，可以利用單晶片多核心架構優點的多執行緒軟體變得更流行。因為演算法的性質，這些多執行緒程式被分階段連續的執行。然而 Amdahl's law 指出平行化程式的加速將會被計算的連續部份限制。第三，不斷增加的晶片整合層級和逐步降低使用的電壓結合使得如何減少電量的耗損成為首要注重的設計限制。此論文的目標是最小化多執行緒程式的執行時間。該執行緒包含平行處理和連續處理的階段，同時也保要有多核心單晶片的電力消耗限制。為了減少 Amdahl's law 影響，在論文中對於電量花費的計算是根據可獲得的平行度來決定處理的指令數，並以這些指令花費的電量為準。使用該等式，電力 = 每個指令的能量(Energy per Instruction: EPI) * 每秒指令數(Instructions per second: IPS)。假設電力固定的情況下，因此限制平行量的多核心單晶片是低 IPS，會花較多的 EPI。相反地，高平行量時，會花較少的 EPI。根據[6]的實驗，在相同的耗電量前題下，一個複雜的系統在使用非對稱式多核心、多執行緒執行時，會比對稱式多核心系統增加百分之三十八的效能。

隨著近年來多媒體裝置的流行，多執行緒平台研究關注的焦點已由對稱式多核心系統轉移到非對稱式多核心系統。非對稱式多核心系統比對稱式多核心系統有更佳的效能/時脈比，因此在多不同工作執行時非對稱式多核心系統更適合嵌入式裝置。

2.3 同質多核心平台系統下的非對稱式排程

前面提過，異質多核心平台在處理通訊及多媒體相關工作時可以得到最佳的效能/時脈比，但目前並沒有論文是針對異質多核心平台探討動態自動排程的設計。不過倒是有不少論文是針對同質多核心平台研究非對稱式動態自動排程的可行性。Wendorf 等人 提出多個工作分配和排程方法[7]，範圍由非對稱 master/slave 排程到對稱式排程。他們在許多情況下測試這些分配和排程方法。對於非對稱系統，結果顯示 OS Preempt 策略幾乎在所有的清況下都有最高的效能。作業系統的工作的 priority 相對高於一般應用程式，而在兩者有相同 priority 時，作業系統的工作可以較優先得到處理器的使用權，稱之為 OS Preempt。相對於其他策略 OS Preempt 可以減少時間耗損百分之三十到百分之六十。在許多測試情況下非對稱的系統和對

稱式的系統幾乎有一樣的效能，甚至前者有優於後者的情況。重要的是，在對稱式系統中，作業系統工作因 functionality partition 仍需在全部可以使用的處理器中選擇執行者，相較之下非對稱式系統指定單一處理器微理作業系統工作，更容易實作。結果也指出，在不同工作分配和排程演算法下，process switch overhead 和多處理器之間的對於分享資源的競爭是決定系統效能的因素中相對較不重要的。

Greenberg 提出了一個簡單的 master-slave 架構[8]。在一些電腦作業系統下，一個程序(process)可以在 user mode 或是 system mode 的模式下執行。一個 user mode 的程序可以在執行中進行一個系統呼叫(system call)變成 system mode。這個程序在結束這些呼叫後便回到 user mode。在 master-slave 的多核心架構下，系統呼叫如 kernel call 只可以在 master 核心執行，剩下的呼叫就被視為如 user call，和其它工作一樣可以在 master 核心或 slave 核心執行。當 slave 核心上的 user mode 程序欲使用 kernel call，slave 核心會將該程序交給 master 核心處理，而非由 slave 核心處理。在 Greenberg 提出的設計中，工作會先在兩個駐列等待。一個駐列稱為 master 駐列，另一個則稱為 slave 駐列。Master 駐列的工作都是在系統模式，而 salve 駐列上的工作都是在使用者模式。

如前所述 master 駐列是只在 master 核心執行的駐列，slave 駐列卻是可在 master 核心或 salve 核心執行。此論文利用兩種簡單又實作的排程演算法來平衡排程的彈性和 queue-switching 的成本花費。最後並提出一個分析公式，用來測量硬體和 work load 參數，同時考量 master-slave 系統的電力和限制，並進而尋找到非對稱式多核心系統中最佳 slave 核心的數量。

2.4 動態式分工及排程

許多對時間有嚴格要求的應用都需要動態排程方能達到預定的效能。Manimaran 等人把一個系統的效能定義為該系統能在 deadline 之前完成的工作所佔的百分比[9]。在這篇論文中，他們提出在多核心系統上使用的一種演算法，可以動態地對可執行的即時工作進行排程，並具有容錯的功能。系統的運作是基於以下兩個限制條件: 一、每一個工作一旦分配給處理器以後，是不會被打斷的(non-preemptive)。二、每個工作有兩種版本，這個假設是用來改善處理器錯誤的問題並可以得到較高的效能。

系統的內有 N 個處理器和 N+1 個駐列，其中包含了 N 個 local 駐列和一個 global 駐列。每一個處理器和一個 local 駐列為一個組合。排程器自 global 駐列中取得最高 priority 的工作，動態地依系統狀態和各個處理器的狀態，決定將置入哪一個 local 駐列。提出的演算法有下列三個技巧:

1)距離概念：決定 task 駐列中兩個工作版本的相對位置。

2)彈性的系統復原：在效能和容錯等級的取捨。

3) 資源的回收：回收被判定為 deadlock 的工作和已完成的工作所分配到的資源。

利用動態排程方法和上述技巧系統的效能和容錯性達成應用上時間的限制。

Avritzer 等人發展出一個效能分析模組[10]。該模組對使用 load sharing 演算法的高度非對稱系統做效能的評估。load sharing 演算法是基於全系統的狀態進行排程工作。load sharing 演算法有兩種實作的層級，第一種層級在作業系統內部，稱為 kernel 層級或 shell 層級。第二種層級為使用者層級，在 shell 的前端。前者雖有效率上的優點，但是異質機器之間的相容性使得實作上十分困難。雖然後者的 overhead 比前者大，但有三個理由令此論文決定使用後者實作。第一、不必考慮異質機器的相容性，容易實作。第二、對機器和使用者 load sharing 會透明化。第三、使用者利用 shell 前端控制可以決定要不要加入負載分享的機制當中。

Load sharing 的主心概念是要儘可能

縮短整個系統的反應時間，執行方法是把工作分配給利用率低的機器。動態的 load sharing 可分成由傳送者初始化的型態和由接收者初始化的型態兩種。傳送者初始化的型態使用時機是系統負載不高時，接收者初始化的型態是系統呈現高負載時使用。此論文提出了一個分界型(threshold type)的 load sharing 演算法，此演算法會隨著某些分界值的變動而調整最適當的工作參數，例如每個機器上的工作數量。實作上該演算法的模型是建立以全系統為視野的全系統狀態馬克夫鏈和並計算出能在最差狀況下達到最小 latency 的系統。此論文的結論指出在非對稱式的環境下，小心地動態調整 load sharing 的演算法，會比靜態設定 load sharing 的演算法的效能有大幅增進。

2.5 共享資源控制

Majumdar 提到多核心系統上程序之間會有競爭分享的資源[11]，例如變數就會儲存在分享記憶體上。保持資料一致性的機制不可缺少，如此才能確保系統的正確性。可是這種機制又通常會降低系統效能。這篇論文研究以多核心平台為基礎的應用程式為對象，如電話交換器和即時資料庫，控制分享資源的競爭以達到高度 throughput 及高度 scalability。將已存在的程式改成 re-entrance 或是將程序做適當的排程是兩種可實行的控制記憶體競爭方法。此論文著重於第二種方法。對數種控制資源競爭的排程演算法量化其結果，可以了解系統內部的行為和每一種演算法最重要的特性。結合數種排程演算法的特性的優點，衍生出混合式的控制資源競爭排程演算法： Hybrid-K。Hybrid-K 可以把所有程序執行時間縮短為依序執行每個程序所花費執行時間的 1/K。參數 K 代表系統增加的處理器數目。因此增進的效能會依 K 的增加而上升。然而需要注意的一點是實驗使用的處理器數目最大只到 10，因此 K 大於 10 的情況尚待驗證。

Saewong 等人指出如何安排同時存取多個資源[12]，這是眾所皆知的一個 NP complete 的問題。在分散式即時系統之中，通常都是用 Decoupling 的方法來管理點對點延遲的系統。不幸地，當利用單獨的核心來管理多個資源時，Decoupling 的方法就會失敗。利用單獨核心管理資源方式的優點是可以減少衝突以全系統的觀點來分配資源的使用。例如控制核心可以利用裝置驅動程式、檔案系統或協定服務(protocol service)來控制相關的資源。控制核心我們稱之為 host 核心。Host 核心具有兩個角色：其一，host 核心如一般的核心可執行應用程式。其二，host 核心可以控制和管理其他 time-shared 的資源。此論文研究協同排程的控制和受控制資源的問題，提出合作排程伺服器(Cooperative Scheduling Server :CSS)。

CSS 是一個專用的伺服器，利用固定的一個處理器來控制眾多可以分享的資源，例如： 磁碟機和程序之間的溝通。下列兩個概念是 CSS 的目標基礎。首先，在一個控制器上(如 CPU)先執行一個非週期性的伺服器，該伺服器可以處理所有局部資源的使用要求。這表示 conjunctive admission control 是在控制端和受控制端一起實行的。接著，在應用程式層級的時間限制被分割進入多個階段，每一個階段都會被保證在一個特定的資源上完成。Real time file system (RTFS)是一個即時的檔案系統，它可以提供在 CPU 低負載時，對磁碟頻寬的保證。有了 file system CSS (FSCSS)，磁碟頻寬的保證也可以在高 CPU 負載和高磁碟工作量下達成。以下列出協同排程演算法設計需要考慮到的因素。第一、資源異質性產生的排程失誤問題。依受控資源的觀點，host 核心必須確保這些資源相對活動不能被其他更高 priority 的活動過份地延遲。依 CPU 的觀點，native CPU application 又必須保有完成的時間限制。因此會有 confliction 和 scheduling miss。第二、 conjunctive admission control；每一個受控制資源的 admission control 必須不只是考慮自己擁有資源的存取，還有 host 核心的可獲得性。因此，要保證即時的服務，協同排程的允許控策略需要搭配資源存取資料的反應時間和處理器排程器去分配 CSS 程序的反應時間。第三、分享資源的同步問題。資源的存取可以平行化處理。資源和 host

核心做好同步，可以允許每個資源達到最大平行化。 第四、有效的資源利用。即時排程的主要目標是達到高利用率和對於應用程式的 deadline 保證。因此，除了保證多資源存取的 deadline 之外，系統應該提供整個系統資源的高利用率。

2.6 靜態式分工

一般的異質雙核心系統架構(比如由 Ferrari 等人提出的 The Janus system[14]，是由一個一般功能處理器和一個特殊功能處理器所組成。這兩個運算單元共同使用一個公用匯流排(bus)，而且可以自由地使用 RAM 和 ROM 等記憶體。而其他週邊輸出輸入設備則由一般功能處理器控制。通常這兩顆處理器是建構在單一晶片上，可以完全分享整個架構上的記憶體空間，也可以將處理器之間的溝通所需的成本忽略成極小。如此的設計通常會將一般功能處理器視為 master 處理器，而特殊功能處理器便視為 DSP。

然而這些系統大多是設計成靜態式分工的方法。Gai 等人曾討論由 GPP 和 DSP 非對稱架構多核心排程的問題[15]。在這篇論文中，DSP 被當成是類似有計算能力的資源，在 DSP 上執行的工作，都是由 GPP 一次一個分配過去。等到 DSP 完成工作，再回到 GPP 繼續下一個工作。如此設計是因為 DSP 對某些工作的能力比 GPP 有效率很多，DSP 在這些工作上所省下的時間和單獨由 GPP 執行整個工作所花的時間相較，會大於 GPP 的閒置和兩個核心的溝通所需的時間。這種方式的實作方法是由兩個 task 駐列來完成。一個是 GPP 駐列，存放一般的工作，並由 GPP 負責執行。另一個則是 DSP 駐列，存放給 DSP 執行的工作。當 DSP 閒置時即是可以接受新工作，排程器選擇在這兩個駐列的頂端有最高 priority 的工作。若是選擇到 GPP 駐列，就由 GPP 來執行工作，反之 GPP 便將 DSP 駐列上的工作傳給 DSP 執行。而當 DSP 正在工作，排程器只選擇 GPP 駐列上最高 priority 的工作交由 GPP 執行。

由過去的研究顯示，非對稱式多核心平台的優點及可行性十分明顯，而且同一個工作如果能動態根據不同核心來排程，也會大大提昇效能。下一章，我們將提出精細分工的工作模型和相關背景。

3. 理論與實作背景

我們以動態精細分工工作模型為概念，實作出非對稱式異質多核心平台排程器。所謂動態精細分工系統和目前廣為使用的靜態式分工(statically partitioned)系統是相對的。在靜態式分工系統中，一項工作會分配到哪一個處理器是在系統設計時就決定好的。為了提高整體系統的效能，我們提出了動態精細分工工作模式。假設在系統平台上有兩個處理器核心，分別是 GPP 核心以及 DSP 核心。新的工作被執行前，在 GPP 上的排程器將監看每個處理器核心的執行時期狀態，和決定哪一個核心較適合執行該工作，再動態地分配給 GPP 或 DSP 執行，減少處理器核心閒置的時間，提高處理器核心利用率，進而縮短全部工作執行時間，增加整個系統平台的效能，這種工作模式我們稱之為精細分工工作模型。

本篇論文提出的排程器是實作於 OMAP5912 OSK 平台上，使用的作業系統在 ARM 處理器核心部分是以 eCos 2.0 版本為基礎進行修改，在 DSP 處理器核心的排程核心是由我們自行設計的。在本章中，我們會介紹 OMAP 5912 應用處理器 (OMAP 5912 Application Processor)和 OMAP 59120 發展板(OMAP 5912 Starter Kit: OSK 5912)，以及嵌入式作業系統 eCos 2.0 版本。過去，本驗室也曾開發過在 Linux 下利用 DSP Gateway 及 TI 發展的 DSP/BIOS 排程器[21]。根據過去的實驗結果，利用 DSP Gateway 的溝通機制成本太高，每秒傳輸只有 3 MBytes，不合乎精細分工系統的需求，因此我們在本論文中改用較為精簡的 eCos 作業系統，並提出有效率的 mailbox 和 shared memory 的溝通機制，以證實精細分工系統可以得較高的效能。所有 eCos 移植到 5912 OSK 的過程將在下一章說明。本論文研究實作的細節會在第五章詳細介紹。

3.1 OMAP 5912 Application Processor

OMAP5912 應用處理器是一塊高度整合的 SoC，包括的重要元件有：GPP-ARM 核心、DSP 核心、和 Traffic controller 等等。OSK 5912 為使用 OMAP 5912 應用處理器的發展平台。

OMAP5912 應用處器整合 ARM 926 EJ-S RISC 核心和 TI TMS320C55x DSP 核心。ARM9 RISC 核心在嵌入式系統被廣為使用，C55x DSP 核心對於數位訊號處理展現高效能和低耗電的特性。因此 OMAP5912 應用處理器適合多媒體嵌入式裝置，經由切割每個應用程式為眾多工作和適切地分配工作給兩個處理器核心執行可以有優秀的效能表現。

Fig. 6 為 OMAP 5912 功能區塊圖[1]。MPU(ARM9)、MPU peripheral bridge、Memory traffic controller 以及 system DMA 四者透過 MPU BUS 溝通。MPU 由 MPU bridge 透過 public/ private peripheral bus 和其週邊溝通。DSP 透過 public/ private peripheral bus 和其 peripheral 溝通。此外 DSP 可藉由 DSP MMU 或是 MPU Interface 和系統其他部份做溝通。

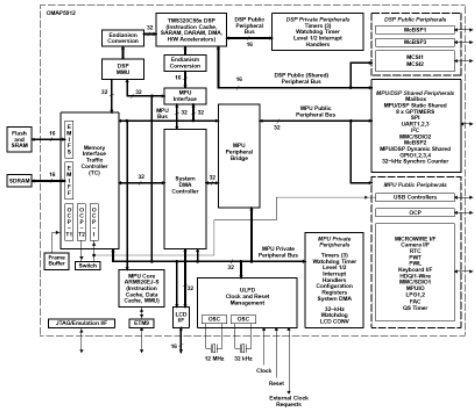


Fig. 6. OMAP 5912 功能區塊圖

OMAP5912 應用處理器 DSP 的記憶體包括內部記憶體 DARAM 和 SARAM。ARM 定義一個 word 等於 4 個 byte，採 byte addressing，所有週邊和擴充的 memory 以及 control register 都由 32 位元來定位。DSP 定義一個 word 等於 2 個 byte，是採 word addressing。

當 ARM 對應一塊實體記憶體到 DSP 的記憶體空間，DSP 可以透過 DSP MMU 來存取該塊記憶體，同時在 ARM 的虛擬記憶中有一塊配置為 DSP 記憶體空間，也會被對應到該塊實體記憶體。

在 OMAP5912 應用處理器上 Memory traffic controller 是一個很重要的內外部記憶體存取元件。Memory traffic controller 可以令 DSP 和 ARM 利用 TI OCP (Open Core Protocol)存取內部共用記憶體或週邊裝置，存取外部記憶體可利用兩種高速記憶介面來完成，分別為 External Memory Interface Fast(EMIFF)和 External Memory Interface Slow(EMIFS)。

EMIFF 相較 EMIFS 是較快速的記憶體裝置，在 OSK 5912 發展板上對應 EMIFF 配置的記憶體是 SDRAM，最大可支援到 64 M Bytes。存取資料的寬度和位址的寬度都是 16 bits，也提供了兩個 bank 選擇位元，亦即可以將 SDRAM 分成四個區域來使用。使用者的應用程式預設是諸存到此 SDRAM。

EMIFS 所連接的外部裝置記憶是 NOR FLASH。透過介面可以 8 bits / 16 bits / 32bits 的寬度在每個 NOR FLASH 晶片上存取資料，其使用的位址寬度為 25 bits。OSK 5912 發展板上共有四塊外部 FLASH 晶片，每塊晶片最大容量為 64 M Bytes，所以可使用的總記憶體容量為 128 M Bytes。此四塊 NOR FLASH 分別為 CS0, CS1, CS2, 和 CS3。Boot ROM 位於 CS0，系統開發者設計的 boot-loader 或作業系統則是存放在 CS3。經過設定，啟動的模式可以利用 CS0 的 boot ROM 或是 CS3 的 boot-loader 開機。

3.2 OMAP 5912 Starter Kit：OSK 5912 (OMAP 5912 OSK)

OSK 5912 是對軟體和硬體做高度整合的平台，主要可做為視訊 圖片訊號處理裝置和行動溝通裝置。可以使用一般的嵌入式作業系統做為 OSK 5912 上 ARM 處理器的作業系統，而 TI 提供 DSP/BIOS 做為 DSP 處理器的即時核心 (real-time kernel)。Fig. 7 為 OSK 5912 正視圖[19]：

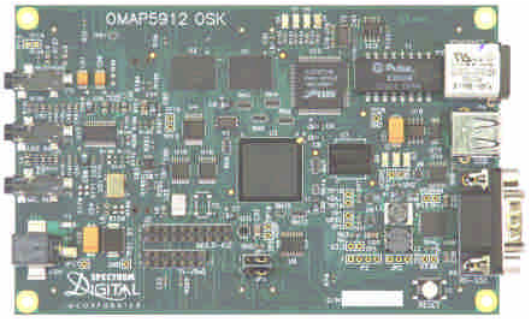


Fig. 7. OSK 5912 正視圖

Hardware Features 如下：

- ARM 926EJS 處理器核心運行於頻率 192 MHz。
- Texas Instruments TMS320C55x 運行於頻率 192 MHz。
- 內建音訊編碼解碼器 TLV320AIC23 codec
- 64 Mega Bytes DDR RAM
- 256 Mega Bytes on board Flash ROM
- 10 MBPS Ethernet port
- On board IEEE 1149.1 JTAG connector for optional emulation
- Software Features 如下：
- Compatible with MontaVista's Linux for OSK5912
- Compatible with OMAP Code Composer Studio from Texas Instruments

3.3 eCos

OSK 5912 所採用的原始作業系統是 MontaVista Linux，但是根據我們去年的經驗，Linux 配合 DSP Gateway 的效能表現無法達到精細分工系統所需的要求，故本論文沒有採用原始發展系統所採用的整合軟體。接下來介紹 ARM 端採用的 eCos 作業系統。在下一章我們會討論如何將 eCos 移植到 OSK5912 的平台下。

3.4 eCos Overview

eCos 是一個開放程式碼，可設定(configurable)，可移植和免費的嵌入式即時作業系統。eCos 的一項重大的技術革新是設定系統(configuration system)。設定系統允許應用程式設計者對 run time 元件加入或調整所需的功能和實作方式。傳統上，作業系統會限制實作的方法，無法選擇。設定系統使得 eCos 開發者創造符合特定應用程式的特定作業系統，也使得 eCos 適合更大範圍的嵌入式應用。設定系統的使用可以保證資源的最小化，和其他不需要的功能和特徵都可以被移除。如此便利性的因素是 eCos 它是一個元件架構的系統。eCos 被設計為可以移植到許多目標架構和目標平台，包括 16 32 64 位元架構和 MPU, MCU, DSP。eCos 支援許多不同平台架構，如 ARM、Intel StrongARM 及 XScale、Fujitsu FR-V、Hitachi SH2/3/4、Hitachi H8/300H、Intel x86、MIPS、Matsushita AM3x、Motorola PowerPC、Motorola 68k/Coldfire、NEC V850 和 Sun SPARC，其他尚包括許多流行的架構和發展板。

3.5 Configure Tool

嵌入式系統正被推動朝著更小更快更便宜更精緻，所以更需方便地控制系統內所有的軟體。有不同的方法可以控制應用程式內元件的特性。eCos 元件控制的哲學是為了減少系統大小，對資源最自由的配置。持著此設計哲學，最小化的系統不必支援某些複雜系統上才有的強大功能。有一種在 run time 控制軟體元件的方法，例如動態連結程式庫 (Dynamic Link Libraries)，不必預先對元計做設定，但是這個方法會會導致程式大小增加。另一種方法是在 link time 時，當需要某個特殊功能元件就會被包入，反之則除去，例如 GNU linker。這方法的特性是擁有某元件的全部功能或都不擁有。Compile time 的元件控制，使得系統開發者可以建立特定應用程式需要的元件，可以保持所有的程式碼都是系統所需要的。對於嵌入式系統來說，這是解決程式碼多寡的好方法。eCos 有一套十分方便的 configure tool，讓系統開發者在 compile time 決定所需的元件和元件的能力，而不必動手修改元件的程式本體。

3.6 Component

要了解 eCos，則了解元件的基礎架構非常重要。元件基礎架構專為滿足嵌入式系統和嵌入式設計的相關需求而存在。設計的 eCos 元件基礎架構可以控制元件達

到最小記憶體使用、允許使用者控制時間行為以符合即時性、和使用一般的程式語言，如 HAL 的實作就包括 C、C++、和組語。大部份嵌入式系統本身所支援的功能比特定應用程式需要的功能更多。通常系統內多餘的程式碼支援的功能，卻是嵌入式程式開發者所不關心也不需要的，而且更多的程式碼使產生錯誤的機會更大。舉 Hello world 程式為例。很多即時作業系統支援了 mutexes, task switch，卻是在此簡單程式所不需要的。eCos 給開發者最終 run time 元件的控制權，沒必要的功能可以輕易地被移除。eCos 系統的大小可由幾百 Byte 到幾百 K-Byte 彈性地增減。

即時嵌入式系統的標準功能包括包括插斷處理，例外處理，錯誤處理，執行緒同步，排程，計時器，和裝置驅動程式。這些標準的功能在 eCos 中由各個元件負責，稱之為標準元件。這些標準元件由即時 kernel 為核心組成。列出元件如下：

- Hardware Abstraction Layer (HAL)—硬體抽象層隱藏每一個支援的 CPU 和平台的特別特徵，以至於 kernel 和其他 run time 元件都是在一個容易移植的形式。
- Kernel—Kernel，包括記憶體、快取、插斷處理、例外處理，執行緒、同步機制，排程器，計時器，計數器和鬧鐘。
- ISO C and math libraries—常用標準程式庫。
- u-ITRON and POSIC —μITRON and POSIX 應用程式者介面
- Device drivers—裝置驅動程式，支援廣泛的裝置標準序列埠控制器，Ethernet 網路控制器，PCMCIA 控制器，USB 控制器，PCI 控制器，和 Flash Rom 等等。
- RebBoot ROM monitor — RedBoot ROM monitor 是一個 Bootstrap 應用程式，為了方便移植它使用 eCos HAL，而且他可以透過序例埠和網路兩種 booting 方式。
- GNU debugger (GDB) support—GNU 除錯器，提供目標平台軟體可以和 GDB host 溝通，對應用程式除錯。
- 其它軟體元件 — SNMP, HTTP, TFTP, 和 FTP。

eCos kernel 或是應用程式都是在 supervisor 模式下執行，所以在 eCos 系統中，user 模式和 kernel 模式並沒有區分。

Fig. 8為 eCos 系統區塊圖[18]：

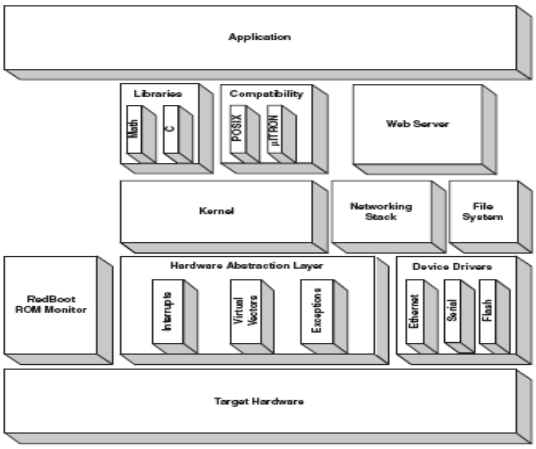


Fig. 8. eCos 系統區塊圖

下面段落將介紹三個重要的 eCos 元件，分別是 HAL、kernel、及 RedBoot。

i. HAL

HAL 將處理器架構的底層硬體和平台的底層硬體抽象化可以有效率地移植 eCos kernel 到別的平台。為了移植 eCos 到新的平台，主要的工作是修改 HAL 以適應支援新的硬體平台，因此了解 HAL 這個軟體元件的架構相當重要。HAL 有一般化的 API，把特殊的硬體行為封裝起來，由硬體來完成設計的功能，並且允許應用層直接存取硬體和任何架構特徵。例如有同樣意義的 interrupt，實際執行插斷的程序依不同架構而相異。為了使 HAL 的可用性達到最廣的範圍，HAL 是用 C 語言和組合語言實作。另外為了 HAL 介面實作上的效率，HAL 是用 C/ CPP 巨集實作，可以使用 inline C 語言，inline 組合語言，或外部呼叫 C 語言和外部呼叫組合語言。

HAL 含概三個不同的模組：Architecture HAL, Variant HAL, 和 Platform HAL。

第一個 HAL 模組定義 architecture。每個被 eCos 支援的處理器家族都被認定為

不同的 architecture。每個 architecture 模組會包含所需的 CPU 啟動程式碼，interrupt delivery 程式碼，context switching 程式碼和其他指令集架構特殊功能的程式碼。第二個 HAL 模組定義了 variant。一個 variant 是一個處理器家族當中的一個特定處理器。例如定義不同的 on-chip MMU 或是快取，也處理任何晶片上的週邊，如記 memory controller 和 interrupt controller。第三個 HAL 副模組是定義 platform。一個 platform 是一塊特別的硬體平台，它包括選擇處理器的 architecture 和 variant。傳統上這個模組包括平台啟動，晶片選擇設定，interrupt controller 和計時裝置。這三層的介定不必很清楚，因為各模組的功能性在不同硬體平台有不同的設定。例如快取和 MMU 可以在 architecture HAL 或 variant HAL。

Fig. 9是 HAL 啟動期間副程式被引用的流程圖[18]。啟動程序可能會依不同架構和平台的使用有些微的不同。此外，啟動程序可能也因為 HAL 的一些設定選項的不同而與此流程圖有所差異。

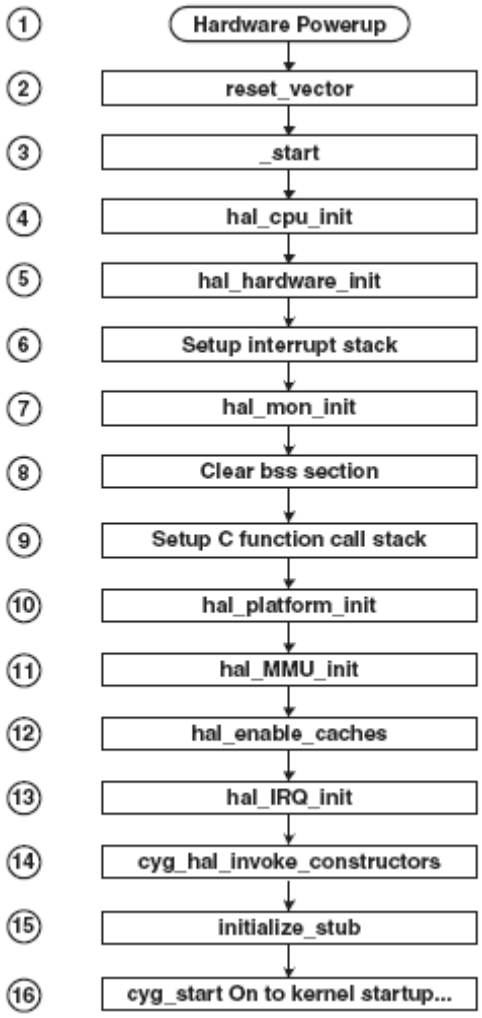


Fig. 9. HAL 啓動流程圖

1. 系統在電力週期啟動之後開始運作。此啟動程序也是 soft reset 的啟動方式。
2. 在 hard reset 或 soft reset 發生後，處理器會跳到 reset vector。Reset vector 對處理器會用到的最少數量暫存器做設定，使得系統可以繼續初始化程序。
3. 接著 reset vector 會跳到 _start，是 HAL 初始化的主要開啟點。
4. 呼叫 hal_cpu_init，這個程式設定處理器的暫存器，如關閉指令和資料快取。確保對於剩下的初使化程序而言，處理器在一個正常的狀態。
5. 下一個被呼叫的副程式叫 hal_hardware_init，硬體設定包含快取設定，設定插斷暫存器為預設狀態，關閉處理器的 watchdog，設定即時時鐘暫存器，和設定晶片選擇暫存器，這些都是基於平台特有的硬體而不同。
6. 接著是設定 interrupt stack 的區域，在

interrupt 發生時可以儲存處理器狀態。在整個初始化程序中，都是利用這一段 stack 做為呼叫 C 副程式會用到的 stack。因為此時 interrupt 是關閉的，不會有衝突發生。

7. 下一步執行 hal_mon_init 程式，確保預設的 exception 處理器安裝給每一個處理器支援的例外情況。
8. 接著清理 BSS 部份，它包含所有未初始化的區域和全域變數。
9. 然後設定 stack，以致於 C 程式呼叫可以被實行，不再使用 interrupt stack。
10. 呼叫 hal_platform_init 或呼叫 hal_if_init，初始 virtual vector table。
11. 初始化 MMU，處理 logical addresses 和 physical addresses 的轉換，同時提供保護和快取機制。
12. 接著啟動指令快取和資料快取。
13. 執行 hal_IRQ_Init，設定 Communications Processor Module (CPM)，它接受和按優先順序處理內外部 interrupt。
14. 下一步，cyg_hal_invoke_constructors 呼叫所有 global C++ constructors。Linker 會提供 global constructor 的名單，cyg_type.h 則用巨集定義這份名單上 constructor 被呼叫的順序。
15. 如果設定當中有除錯環境而且 ROM monitor 沒有提供除錯支援，下一個要呼叫的是 initialize_stub。initialize_stub 安裝 standard trap handler 並把硬體設定在適合除錯的狀態。
16. 最後一個步驟是把控制權轉給 kernel。cyg_start 就是控制權由 HAL 轉入 kernel 的點。

ii. The Kernel

Kernel 是 eCos 系統的中樞。Kernel 提供即時作業中的標準功能例如插斷和例外處理、排程、執行緒和同步。在 eCos 系統下可以對這些組成 kernel 的標準功能元件完全地設定，以達到特殊的需要。eCos kernel 以 C++實作，允許用 C++寫成的應用程式直接透過 C kernel API 介面和 kernel 溝通。eCos kernel 也支援標準 u-ITRON 和 POSIX compatibility layers 介面。為了符合即時性需求，eCos kernel 依循下列準則發展：

- Interrupt latency—interrupt 回應和開始執行 ISR 的時間要少而且 deterministic。
- Dispatch latency—執行緒準備完成可以執行的狀態到開始執行的時間要少並且 deterministic。
- Memory footprint—對一個設定完成的系統，程式或資料所需求的記憶體資源要保持最小化和 deterministic。而且要確保嵌入式系統的動態記憶體配置不會使用超出所有記憶體的量。
- Deterministic kernel primitives—所有 kernel 的行為都要是可預期的且符合即時性的需求。

eCos kernel API 不回傳錯誤訊息。在嵌入式系統當中，處理錯誤回傳訊息會導致許多問題，如消耗貴重的執行週期和程式空間來檢查回傳的訊息。為了程式發展的便利性 eCos kernel 提供 assertion，它可以被 eCos package 開啟或關閉。傳統上，assertion 在除錯階段會開啟，允許 kernel 程式顯示錯誤檢查。如果錯誤產生了，就會回傳一個 assertion 失敗並且會中止應用程式。除錯程序完畢之後，assertion 就在 kernel package 之中關閉。此方法有很多好處，如限制程式中錯誤檢查的 overhead，消除應用程式錯誤檢查的需要；如果有一個錯誤發生，應用就被暫停，可以立即知道發生錯誤的地點，而不是依賴回傳訊息再去檢查。

kernel 提供開發多執行緒應用所需的重要功能。

在硬體都啟動完成之後，HAL 中呼叫 Kernel 啟動的程序。cyg_start 是 kernel 啟重程序的啟始點。它呼叫其他預設的啟動程式來處理不同初始化的任務。只要在應用程式之中提供相同程式名稱，預設 kernel 啟動程式可以被簡單的替換，以完成使用者特殊的初始化工作。Kernel 啟動程序如圖 5 [18]。

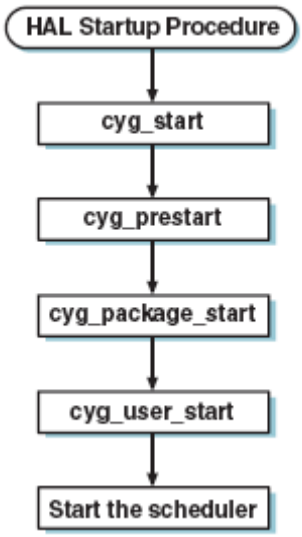


Fig. 10. Kernel 啓動程序

Core kernel 啟動程式，cyg_star。Cyg_start 之後下一個被呼叫的程式是 cyg_prestart。它預設不完成任何初始任務，讓使用者自行決定在其他系統初始化之前該被完成的初始化都可以在這裡執行。接著呼叫 cyg_package_start，在應用程式開始之前初始化將用到的 package，如 u-ITRON 和 ISO C 程式庫。之後呼叫 cyg_user_start，它是應用程式的進入點。建議 cyg_user_start 被使用來完成任何應用指定的初始化、產生執行緒、產生 synchronization primitives、設定鬧鐘、和註冊任何需要的 interrupt handlers。當 cyg_user_start return 時會自動執行呼叫排程器。

iii. The Scheduler

eCos kernel 的核心是排程器。排程器的工作是去選擇最適合的執行緒執行，提供執行中執行緒同步的機制和控制 interrupt 在執行緒執行的影響。在排程器程式程式碼執行期間，不會關閉 interrupt，因為 interrupt latency 十分短。排程器內存在一個計數器，它決定排程器是自由地執行或是關閉。如果計數器非零，排程器就被關閉，當計數器回到零，排程就會啟動。在 ISR 執行期間，HAL 預設 interrupt handler 會修改計數器來停止排程動作。執行緒也有能力開關排程器。

a. Multilevel Queue Scheduler

MLQ 排程器允許同一個 priority 有多個執行緒可執行。priority 可由 1 設定到 32，對應到數字是 0(最高)到 31(最低)。MLQ 排程器允許不同 priority 可有 preemption。Preemption 是指低 priority 執行緒的被 context switch 暫停執行，因此高 priority 的執行緒開始執行。在同一個 priority 中，MLQ 排程器有 time-slicing 功能。每一個執行緒在一段特定的時間內執行稱之為 time-slicing，統系開發者可以設定 time-slicing 的長度。MLQ 排程器的駐列實作上是用 double linked circular list 來連結同一個層級的不同執行緒和不同層級的執行緒。

b. Bitmap Scheduler

Bitmap 排程器的執行緒也是有多個 priorities；然而每層級只能有一個執行緒。這種設計簡化排程演算法，也令 bitmap 排程器非常有效率。Priority 的數量和設定和 MLQ 相同；駐列的實作有三種，可以是 8,16 或 32 位元值，依設定的 priority 而定。因此駐列中一個位元代表一個 priority。Bitmap 排程器可以有 preemption，但是因為每個 priority 只有一個執行緒，故 time-slicing 會失去功能。使用 bitmap 排程器時就會關閉 time-slicing。

比較這兩種排程器，bitmap 排程器是較簡單的排程策略。但是，MLQ 排程器可提供更多的選擇給執行緒操作，也可容納更多的執行緒，只要記憶體夠，就沒有執行緒數量的限制。系統開發者可依應用程式的特殊需求而決定使用哪一種排程器。

iv. Synchronization Mechanisms

eCos kernel 提供系統中執行緒溝通機制和執行緒對分享資源存取的同步機制。提供的機制有 Mutexes、Semaphores、Condition variables、Flags、Message boxes 和 Spinlock (for SMP systems)。

Kernel 提供同步 API，應用程式可以便利地使用這些同步機制。API 程式提供有 blocking 功能或 non-blocking 功能。Blocking 程式呼叫，如

cyg_semaphore_wait， 暫停執行緒的執行，直到 API 程式可以成功地完成。Non-blocking 程式呼叫有兩種，第一種如 cyg_semaphore_trywait，不論程式有沒有成功地完成，會回傳訊息指出呼叫的狀態，所以執行可以繼續執行。第二種 blocking 呼叫，必需先設定等待的時間，用時間長度為暫停長短的依據，如 cyg_semaphore_timed_wait。

Mutex 的目標和其他的都不同。Mutex 允許多個執行緒安全地存取共享的資源。它只有兩種狀態: locked 和 unlocked。並且 Mutex 有擁有者的概念，只有擁有者(上鎖者)可以解開 mutex。其他同步機制都是被使用來做為執行緒之間互相溝通或從 DSR 連接到相關的 interrupt handler 到一個執行緒。

Semaphore 是一種用計數來指示資源已上鎖或可以獲得的同步機制。Semaphore 有兩種型態，分別是 counting semaphore 和 binary semaphore。Binary semaphore 很像 counting semaphore，但是它的計數決不會大於一，所以它的狀態只有 locked 和 unlocked；和 mutex 不同的是，它沒有擁有者的概念。意即每一個執行緒都可以對 semaphore 上鎖和解鎖。Counting semaphore 依照計數值可以有多種狀態。當執行緒 post semaphore 時增加的數值，同時也是當一個執行緒 wait 一個 semaphore 減少的數值。當 semaphore 回到零時，只有最高 priority 的執行緒可以執行。

Condition variables 和 mutex 搭配使用允許多執行緒存取共享資源。傳統上，一個執行緒產生資料，一個或多個執行緒等待這筆資料。當資料備妥時，產生資料的執行可以發出 Condition variable 通知喚醒一個或喚醒全部執行緒。等待中的執行緒就可以拿到需要的資料並處理之。

Flag 用 32 位元表現的同步機制。每一個位元代表一個狀態，允許執行緒去等待一個或多個組成的狀態。當狀態符合，該執行緒就會被喚醒。

Message box 又叫 mail box，提供兩個執行緒交換資訊。因為 mailbox 的容積有限，交換的資訊一般都是資料結構的指標，或是有特別設計過的訊息。

在對稱式多核心平台上，eCos kernel 提供另一種額外的同步機制。同時其他的同步機制可以在對稱式多核心平台上運作。Spinlock 基本上是一個 flag，和其它同步機制比較起來是更底層的操作，會依不同的硬體有不同的實作方式。有些處理器提供 test-and-set 指令來實作 spinlock。在處理器執行一段特殊程式碼之前要去檢查此 flag。如果 spinlock 沒有上鎖，處理器可以設定 flag 和繼續執行此執行緒。如果 spinlock 被鎖上，執行緒就會持續地檢查 flag 直到它被解鎖，而沒有被 suspend。

v. Threads and Interrupt Handling

Kernel 利用一種 two-level 方法處理 interrupt。連結每個 interrupt vector 是一個 Interrupt Service Routine (ISR)，它會盡可能快速的執行來回應硬體 interrupt。ISR 只可以做少數的 kernel 呼叫，都是和 interrupt 系統相關，但不能做喚醒執行緒的動作。如果 ISR 偵測到 I/O 完成，一個執行應該被喚醒，ISR 就會讓連結的 DSR 去執行。DSR 可以使用更多 kernel 呼叫，如發 condition variable 訊號或 post a semaphore。關閉 interrupt 可以阻止 ISR 執行，但系統中很少機會關閉 interrupt，如果有也是很短一段時間。執行緒關閉 interrupt 的主要理由是改變一些 ISR 之間共享的狀態設定。例如，如果一個執行緒會增加 linked list 的 node，而且 ISR 可能在任何時候自 linked list 移除一個 node，此執行緒就要在操作 list 時關掉 interrupt。

vi. RedBoot

RedBoot 是 Red Hat Embedded Debug and Bootstrap 的縮寫。它是一個為嵌入式系統提供一個除錯和 bootstrap 環境而設計的程式。RedBoot 是一個以 eCos 為基礎的應用程式，並且使用 eCos HAL 為它的基礎。

RedBoot 包括 GDB stub，可以從 PC

上的 GDB host 連接到平台上除錯。連接方式可以是透過 serial port 或是 Ethernet port。除了除錯外，RedBoot 主要的功能是 booting，支援三種 booting 方式，分別稱為 ROM, RAM, 和 ROMRAM。其含意為 RedBoot 的 binary image 放置於何種記憶和由何種記憶體執行。ROM 和 RAM 即代表 image 放置和執行都在 ROM 或 RAM，ROMRAM 指的是放置在 ROM，欲執行前拷貝到 RAM，才在 RAM 執行。在 RAM 資源有限的情況下，經常使用 ROM 模式的 booting。Image 是被放置在 flash 或 EPROM。由於 flash 命令不能寫入 RedBoot image 本身存放的區塊，所以為了更新 RedBoot，必需使用 RAM 模式來達成。可以使用 ROM 和 ROMRAM 模式直接 booting，除非配合其他 ROM monitor，否則不能直接使用 RAM 模式 booting。

RedBoot 會在 memory map 的底部，保留 RAM 空間結 run time 的資料和 CPU exception/ interrupt tables。在 memory map 頂空的空間則保留給 net stack、zlib 解壓空間、等等平台的特殊需求。

依設定選項和 package，可以建立最經濟的 RedBoot，只包含該平台所需的最小數元件。圖 8 顯示出 RedBoot ROM monitor 部份特徵的區塊圖[16]。由此圖，可以看到 RedBoot 提供的功能全包含在 RedBoot 程式 image 之中。RedBoot 可以只提供簡單的 image 載入和執行功能，提供簡易 flash file system，或是透過命令列對應用程式做監控和除錯功能。

3.7 移植 eCos 到 OMAP 5912

因為本計畫所設計的動態分工異質雙核心排程器是架構在 eCos 系統之下，所以實作之前必須先把 eCos 移植到目標平台上。目前 eCos 的公開程式碼計畫並沒有支援 OMAP 5912 OSK，所以本章先描述如何將 eCos 移植到目標平台上。移植的流程分為三個階段，分別是移植 architecture HAL、variant HAL、和 platform HAL。每個 HAL 的功能及含義已在上一章敘述。這裡大略介紹移植流程，之後段落將每個細節再詳加說明。

Architecture HAL 的移植：這部份的程式是根據處理器核心的大體架構而設計的。因為 5912 OSK 的核心處理器是 ARM，由於 ARM 的 architecture HAL 已經存在現有的 eCos 中，所以我們可以直接延用。

Variant HAL 的移植：Variant HAL 主要是處理每一類嵌入式處理器核心在架構上的小調整。比如同樣是 ARM 9 的處理核心，有些有 MMU，有些沒有，有些有訊號處理指令集，有些則支援 Java 加速。由於目前在 eCos 下還沒有 ARM 926EJS 核心的移植，所以我們以 ARM 925T 的程式碼為基礎開始進行移植工作。首先取得需要的硬體資料（詳見下一節），接著修改各部份的內容設定，例如 instruction cache 和 data cache。還要在 eCos.db 檔之中新增 ARM926EJS package。

Platform HAL 的移植：由於 OMAP 5912 OSK 是承續 OMAP Innovator 的設計，因此可以用 OMAP Innovator 為基礎來移植。但仍舊必需取得所需之硬體資料。5912 OSK 的 Ethernet 是 LAN91C96，與 Innovator 相同，故可以使用，但必需新增 Ethernet_5912 package 於 eCos.db 之中，並指出其使用 LAN91C96，此外還得給定其 base address。Flash 是兩顆 Micron Q Flash MT 28F128J3，並沒有現存的 package，但是它和 Intel Strata Flash 28F128J3 相容，可以借用其 package，之後我們必需新增 Flash_5912 package 於 eCos.db 之中，並指定使用 Intel Strata 28F128J3。Serial port 使用 UART， Innovator package 中沒有檔案可以直接使用，但我們可以用相近的 Integrator 做為參考，但需做些許修改，當然也要加入 eCos.db 的 package。Platform start up 包括很多的項目，如 internal memory, external memory, MMU, register 等等的設定，由於平台的改變，因此能延用 innovator 的部份有限，在這邊，我們做了很大的修改。

上面三個 HAL 的設定若有衝突，將以 platform HAL， variant HAL， architecture HAL 的優先順序來決定那一個設定是有效的。例如 architecture HAL

可能設定了 instruction cache 的大小，variant HAL 也可以設定之，此時就以 variant HAL 的設定為準。

4. 異質多核動態分工排程器設計

本計畫研究的中心概念為異質雙核心平台上的動態分工。如之前所述，動態精細分工是以同時考量整個平台的各異質核心的即時計算狀態為排程的依據，以期達到最高的效能。Fig. 11是我們設計的系統架構。原本利用 eCos 為 kernel 的系統架構包含有：應用程式 application(APP)，應用程式可以透過 eCos kernel 和 HAL 利用底層平台的硬體，如 GPP 和 DSP。其中 eCos kernel 最重要的元件是 MLQ scheduler。在 eCos kernel 中加入 Dispatcher、Service Registrar、和 core service table。應用程式經過 Scheduler API 可以和 kernel 或 Service Registrar 溝通，包括註冊 dual-core services，和下達執行 services 的命令。這些新加入的單元，即是 scheduler 的核心。以下，我們把這個異質雙核心動態分工排程器稱做 Heterogeneous Multi-Processor (HMP) scheduler。

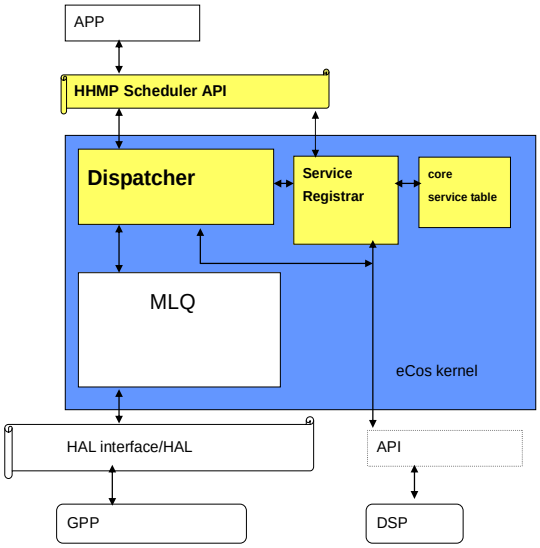


Fig. 11. 系統架構

接下來會逐一介紹 scheduler 的每一個元件、它們各別的功能、以及運作方式。scheduler 的每個元件溝通都是雙向的動作，例如 Dispatcher 可能會向 Service Registrar 索取特定 service 的執行指標，Service Registrar 就會回覆之。圖中 eCos kernel 會透過一組 API 和 DSP 溝通，這一組 API 是同一研究團隊成員提供的介面，因為它不是本研究的重點，僅在最後做簡單介紹。

4.1 Scheduler API

這個介面提供 scheduler 使用上的便利性。Scheduler API 是新的 system call，它的目標就是令使用者在移植單核心的應用程式為異質雙核心應用程式時，只需做最小的修改，便能使用 scheduler。因 [illegible]heduler Interface 提供兩類三個系統[illegible]程式，它們是註冊和執行這兩類：

- [illegible]egister_service()
- [illegible]voke_service()
- [illegible]move_service()

[illegible]_register_service()和 HMP_remove_service()歸屬於註冊類。它們扮演的功能是向 kernel 註冊某項 service 為 dual-core service 或是移除之。一旦註冊了某項 service 為 dual-core service，之後使用這個 service 時，就會被引入 HMP scheduler，由 GPP 或由 DSP 負責執行都有可能。使用 service 的方法，在註冊後，HMP Scheduler API 會給使用者一個 ID。使用 service 就要呼叫 HMP_invoke_service()，填入 ID 指定註冊的 service 和填入原 service 的參數。HMP_invoke_service()被設計為可容納不同數量參數的介面，因此各種數量參數皆可以處理。舉一個簡單的例子，在之後的實驗裡，測試的應用程式為 H. 263 decoder。圖 11 中 Idct 是 decoder 其中一個程式，接受的參數是一個 short。欲使用 HMP scheduler 只需把程式改為圖 12 的寫法。

```
    void idct(short *block)
    { … }

void main()
{  …
       idct(block);
   …
}
```

Fig. 12. 一般單核心 idct 函數的使用方法

Idct 程式不必做任何修改。在主程式中呼叫 idct 之前，先用 HMP_register_service(idct_image)註冊取得 id。之後要使用 idct 就改呼叫 idct(block)為呼叫 HMP_invoke_service(id, block)。id 是向 kernel 取得的資料，而 block 是原本主程式的指標資料。全部就只要做到這些修改，應用程式幾乎看不到有單核心雙核心程式的不同，移植的便利性非常大。

```
    void idct(short *data)
     { … }

void main()
{
   int   id;
   …
      id = HMPHMP_register_service(idct_image);
   …
   HMPHMP_invoke_service(id, block);
   …
}
```

Fig. 13. HMP scheduler 下 idct 使用方法

OMAP 5912 是屬於雙核心的晶片，因此 HMP Scheduler 在執行排程工作之外，要負責 DSP 的啟動。當然 DSP 的啟動可以在 booting 時完成，考量並非每一種應用程式都需要用到 DSP，只在 HMP Scheduler 被啟動時才自動啟動 DSP，以減少 booting 的 overhead。另外有一個做法也是可以實行的，那就是將 DSP 啟動轉換成 bootstrap (Redboot)的指令，欲使用 HMP Scheduler 時，要先下該指令啟動 DSP，再執行應用程式。

4.2 Service Registrar 和 Core Service Table

應用程式向 Service Registrar 註冊 service，Service Registrar 收到 service 的資訊如下：

- Service function pointer
- Service DSP binary image pointer

得到所需資訊的 Service Registrar 開始進行註冊動作。首先在 core service table 內紀錄 function pointer。接著透過 DSP API 將 DSP binary image 傳入 DSP internal RAM，完成之後向 DSP 註冊該 service，得到 DSP 回傳的 service DSP ID。在 core service table 紀錄此 ID。之後都用此 ID 向 DSP 做指定 service 的動作。未來 function pointer 會改成 ARM 的 image。ARM 的 image 和 DSP 的 image 會包成一個資料結構。

完成上述程序的 Service Registrar 會產生一個 service kernel ID，回傳給應用程式。Service kernel ID 和 service DSP ID 是不同的。前者是應用程式和 kernel 互動時，用來指明 service 的依據。應用程式要執行 service 時，若 Dispatcher 判斷該 service 為 GPP 執行，kernel 會根據 service kernel ID 到 core service table 找到正確的 function pointer，由 GPP 端執行 service；若 service 被判斷為 DSP 執行，kernel 依 service kernel ID 在 core service table 內找對應的 service DSP ID。在通知 DSP 執行某個 service 時，一併傳入 service DSP ID，如此一來 DSP 便了解該執行哪一個 service。所以 service kernel ID 和 service DSP ID 是完全不相同。

Service Register 另外一項功能為移除 service。接受移除指令的 Service Register 會自 core service table 中移除紀錄。並將 service image 從 DSP 內移除。

4.3 Dispatcher

Dispatcher 負責在應用程式要求執行 dual-core service 之後，判斷由哪一顆處理器來執行會得到最短的執行時間，進而決定由哪一顆處理器執行。判斷由 GPP 執行，就直接將 service 交給

MLQscheduler，依原本 eCos kernel 的路徑執行，但結束時會有些許改變；或是由 DSP 執行該 service。Dispatcher 的架構圖如下：

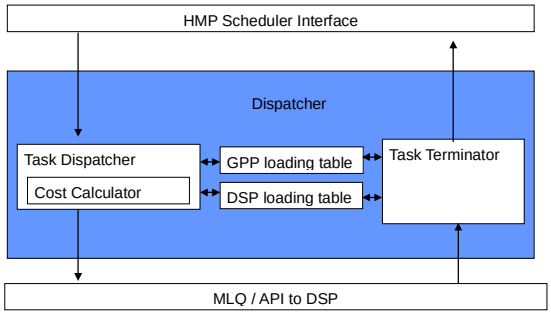


Fig. 14. Dispatcher 架構圖

Dispatcher 是由三大單元組成，它們分別是 Task Dispatcher、Task Terminator、和 loading tables。執行 dual-core service 的資訊經 HMP Scheduler Interface 傳入 Dispatcher。一旦 Task Dispatcher 接收到資訊，會喚醒 Cost Calculator。Cost Calculator 開始計算 cost value，接著便回傳 Cost value 給 Task Dispatcher。Task Dispatcher 就以 cost value 判定由哪一處理器執行 service。Task Dispatcher 判定 service 的執行處理器，在屬於該處理器的 loading table 新增 service loading 資料。只要 service 執行完畢，就由 Task Terminator 來結束整個程序，包括通知應用程式 service 執行完成，和自處理器的 loading table 移除紀錄。由 loading table 的觀點來看，其 producer 是 Task Dispatcher，而 Task Terminator 為 consumer。此外，當 service 必須對大量資料做運算使用到 pointer，而此 service 為被決定為 DSP 執行時，Dispatcher 得負責 DSP 可以得到整筆資料，之後有段落會專門介紹 Dispatcher 如何令 DSP 得到完整資料的實作方式。

4.4 Task Dispatcher

Task Dispatcher 得自應用程式的 service ID 是 service kernel ID。Task Dispatcher 的工作主要是依處理器的 loading 狀態，分配 service 給某一顆處理器負責執行，我們所使用的動態排程方式，基本上是根據過去的執行狀態來預測何者未來的 loading 比較低，便將工作交由該處理器執行。Task Dispatcher 會從 Cost Calculator 得到被呼叫的 service 在不同處理器所需的執行時間的估測，分別記錄在 time_ARM_prediction 和 time_DSP_prediction 這兩個變數中。此動態排程的判斷方法是將估測出的時間相比，來決定誰該執行該工作：

- time_ARM_prediction > time_DSP_prediction：service 判定給 DSP 執行。
- time_ARM_prediction < time_DSP_prediction：service 判定給 ARM 執行。
- time_ARM_prediction = time_DSP_prediction：service 判定給 DSP 執行。

在相等的情況下，service 會由 DSP 負責執行。其原因在於本平台系統專為處理多媒體資料而設計，dual-core service 都是數位訊號處理的形式，在 DSP 和 ARM 的速度和耗電力比較，和忽略雙核心溝通的 overhead 的前提之下，DSP 更適合擔任執行角色，因此判定由 DSP 執行。

在 Cost Calculator 的部份，是利用指數平均數(exponential average)的方法來預測 service 的執行時間。依公式：

$$T_{n+1} \equiv \alpha \times t_n + (1-\alpha) \times T_n$$

T_{n+1}：預估這一次執行時間。

t_n：上一次實際執行時間。

T_n：上一次預估的執行時間。

α：比例常數，常用 0.5；這裡使用 0.7 加重實際時間的比重。

每一類 service 執行時間的預測都

是獨立的，換言之，每一類 service 都有專屬的一套預測變數。而且每一個 service 實際上是有兩份時間紀錄，一份為 ARM 時間紀錄，一份為 DSP 時間紀錄。一個 service 第一次執行時，t_0 及 T_0 都是預設值，所以前幾次的預測值不十分準確，但隨著執行次數的增加，不斷有實際執行時間可以修正預測的結果，提升精準度。

但是這種方式會出現一個嚴重的缺點。假設平均 DSP 在正常情況下處理 service idct 比 ARM 處理還快，其速度分別為 500 個 time tick 與 800 個 time tick。一般情況下，也許有時候 DSP 慢一點而 ARM 又快一點，Cost Calculator 計算的結果是 ARM 較小，該 idct 就由 ARM 來執行；如果又回到 DSP 快的情況下，Cost Calculator 計算的值 DSP 會較小，變由 DSP 來執行 idct。實作過程中，發現一種特殊的情況，會極端地使效能變差。有些不明因素讓 DSP 執行 idct 的時間由平均的 500 個 time tick 大大提升到 2000 個 time tick，自此之後都不會由 DSP 執行 idct，也不會更新 t_n 和 T_n。這麼一來，系統上的 idct 開始就會判定給 ARM 來執行。每一次 idct 平均執行的花費就會增加 300 個 time tick。

為了解決這個問題，當上述情形發生時，Cost Calculator 會自動將此次的預測值除以特定值。以便讓下一次的預測值縮小。當我們假設 α 為 0.7，而假設預測十分精準使得實際值和預測值都是 k。如下列公式：

$$T_{n+1} = \alpha \times t_n + (1-\alpha) \times T_n$$
$$= \alpha \times k + (1-\alpha) \times k$$
$$= k$$

經過實驗當預測值被除以 2 之後，公式改變如下。每次新值就會依 0.85 的比例向下修正，可以達到解決這個問題的需求。

$$T_{n+1}' = \alpha \times t_n + (1-\alpha) \times T_n / 2$$
$$= \alpha \times k + (1-\alpha) \times k / 2$$
$$= 0.7k + 0.3k / 2$$
$$= 0.7k + 0.15k$$
$$= 0.85k$$
$$= 0.85T_{n+1}$$

利用上述方法完成執行處理器的指定之後，Task Dispatcher 要在對應處理器 loading table 增加 service 紀錄，和更新處理器全部執行時間。接著交付 service 給處理器執行。

4.5 Task Terminator

Task Dispatcher 和 Task Terminator 在系統中的角色是相反的。它接受 ARM 或是 DSP 結束 service 執行的訊息，自 service loading table 移除 service 紀錄，和更新處理器全部執行時間。下一步再通知應用程式”已經完成 service”。

4.6 Loading Tables

Loading table 用來紀錄各處理器的 loading 狀態，和各處理器上執行的 services。意義上 ARM(DSP) service table 是紀錄 service 種類，而 ARM(DSP) loading table 是紀錄每一個處理器上正在執行的 service。換句話說，同一種 service 種類在 ARM(DSP) service table 只有一筆資料，但在 ARM(DSP) loading table 上，只要應用程式對同一種 service 要求執行幾次，同一種 service 種類就會有幾筆資料，可是這些資料互相是不同。圖 14 為 loading table 的結構：

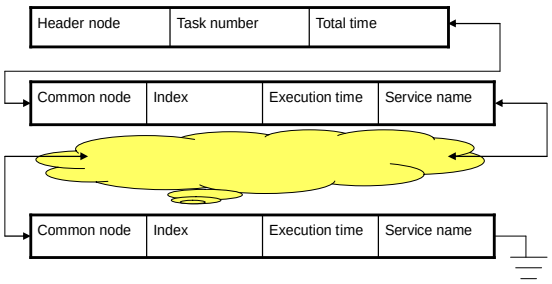


Fig. 15. Loading table

Loading table 分為 header node 和

common node。Header node 紀錄其負責處理器上所有執行中 task 的數目(task number)，和所有的執行時間(total time)。未來我們會增加不同 cost function 考慮的因素，概括說來是不同種類的核心 loading 狀態，如電力消耗。在 header node 預留一些欄位可以用來記錄新增的 loading 狀態。每一個 common node 代表一個執行中的 service，紀錄著在 loading table 上的位置(index)、執行時間(execution time)、和名稱(service name)。這是一個 double direction linked list 資料結構，有鑑於不同的 service 有不同的執行時間，亦即 service 加入 table 的順序不會和離開 table 的順序相同，使用 linked list 處理這個特性。

4.7 雙核心溝通方法

使用 mailbox 和 shared memory 的組合。OHMP 5912 上有四組 mailbox，其中兩組為 ARM to DSP，另兩組為 DSP to ARM。顧名思義，ARM to DSP 為 ARM 發給 DSP 的 mailbox，DSP to ARM 是 DSP 發給 ARM 的 mailbox。每組 mailbox 上有兩個 16 位元 register，一個設定為 command register，另一個為 data register。在 ARM to DSP mailbox 上，ARM 有完全的讀寫權利，DSP 只有讀取的權限；反之亦然。ARM 填完第二個 register 時，硬體會自動設定一個 flag，因此 DSP 會有 interrupt 產生。DSP 開始讀取兩個 register 的資料，讀完第二個 register 之後，flag 會再由硬體設回 0，表示 DSP 讀取完畢。利用 mailbox 傳遞做為雙核心 service 執行的開端。

設計三種 ARM to DSP mailbox action command，如下：

- Register： 註冊 service。
- Invoke： 執行 service。
- Remove：移除 service。

以及 DSP to ARM mailbox action command，是 ARM to DSP 的回應指令如下：

- Return register： 完成 service 註冊。
- Return invoke： 完成 service 執行。
- Return remove： 完成 service 移除。

ARM 發 register mailbox action command 註冊 service 之前，ARM 利用 DSP API 把 service image 放入 DSP space internal RAM。完成註冊動作，DSP 會回 return register mailbox action command，並利用 data register 儲存 service DSP ID。再由 Service Registrar 存入 DSP service table。相同的運做模式，但功能相反是 remove。

最後一種指令為 invoke 指令。ARM 在 command register 填入 invoke 指令，data register 填入自 DSP service table 取得的 service DSP ID。依此行為 DSP 便會開始執行要求的 service。著眼於 mailbox 只提供兩個 16 位元的 register，沒有辦法利用 mailbox 傳入其它參數或資料。為了解決這個問題，shared memory 正可派上用場。Shared memory 指得是 OMAP 5912 on-chip SRAM，共有 250 k-byte 可以使用。設計了 parameter table，結構如下：

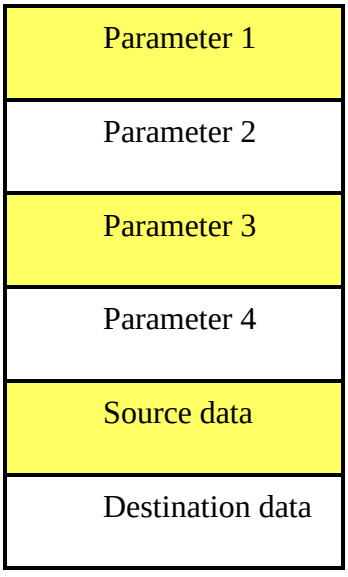


Fig. 16. Parameter table

Parameter 1 到 parameter 4，每一個佔用 4 位元組。接下來 source data 及 destination data 每一個佔 4000 位元組。這兩塊大記憶體是選擇性使用，如果使用這兩塊記憶體，可以利用前面的 parameter 1

或 parameter 2 分別指到 source data 和 destination data。給應用程式設計者很大的彈性空間。如前述 H.263 idct(short block)的例子，可以把整個 short block 從 ARM 的 heap 搬到 source data，把 Parameter 1 指到 source data，並另用 Parameter 2 指定一塊記憶體。DSP 端的 idct.image 就可以從 Parameter 1 拿到整筆資料，運算完畢後，再將結果存到 Parameter 2 指到的位置。

每一個 service 會有一份此 parameter table。使用完之後可立即釋放記憶體空間。ARM 的定址模式為 byte addressing，而 DSP 的定址模式為 word addressing(每個 word 2 bytes)。兩種位址的轉換如下：

DSP_address = (ARM_address – ARM_base_address) / 2 + DSP_base_address

因此 ARM 在選擇 parameter table 的起始位址，必需在 16 位元制下以 0、4、8、或 C 結尾。DSP 才能正確讀取。否則會有不可預期的錯誤。

4.8 DSP API

表 I 簡介 DSP API 的名稱與其功能。

TABLE I. DSP API

API name	description
`dsp_init()`	啓動 DSP 處理器
`dspmmu_for_sdram()`	啓動 DSP MMU
`pmem_init()`	啓動 DSP internal ram
`pmem_allocate()`	取得 DSP internal ram
`pmem_free()`	釋放 DSP internal ram

這部份的設計是配合 DSP 的 real-time kernel 設計的。

5. 實驗結果

本論文一直闡述想的法在第五章介紹了實作的過程。在這一章將會做五組不同的實驗來驗證 HMP Scheduler 的動態排程功能和系統效能。首先我們會介紹實驗的環境，接著再介紹各組實驗，包括實驗的方法和討論。

5.1 實驗環境

在這一段落先介紹實驗使用的的應用程式，它是 H.263 的 decoder。 H.263 decoder 是用來將已經壓製好的 m4v 檔案解回 yuv 檔案。主要的 function 也就是將在實驗中註冊為 service 的 function 如下：

- idct：對所有 frame 做 inverse DCT。
- dequant_inter ： 對 inter frame 做 inverse quantization。
- deqaunt_intra：對 intra frame 做 inverse quantization。
- interpolation：對 inter frame 做垂直點之間、水平點之間、和對角線點之間的 interpolation。

GPP 核心和 DSP 核心都設定為 96MHz。使用的 bit-stream 是 foreman。我們準備了三種不同的 bit rate，其特徵如表 II 所示：

TABLE II. 實驗 bit-stream

Name	Type	Frame	Frame rate	Bit rate
Foreman	QCIF	300	30	192k
Foreman	QCIF	300	30	128k
Foreman	QCIF	300	30	64k

測定用的 timer 是 OMAP 5912 平台上 ARM private peripheral timer 2，它的頻率為 12MHz。time tick 與 micro second 的轉換如下：

(time tick +1) * 2 / Frequency * 1000 = micro second

ARM Private Timer 2 Registers

BYTE ADDRESS	REGISTER NAME	DESCRIPTION	ACCESS WIDTH	ACCESS TYPE
FFE:C600	MPU_CNTL_TIMER_2	Timer 2 Control Timer Register	32	RW
FFE:C604	MPU_LOAD_TIM_2	Timer 2 Load Timer Register	32	W
FFE:C608	MPU_READ_TIM_2	Timer 2 Read Timer Register	32	R

表 8 為 timer register 列表。第一欄紀錄每個 register 在 ARM 記憶體空間的 address。接著兩欄位是 register 名稱和描敘。第四欄代表每個 register 的長度是 32 位元。第一列是 control register，用來開啟或關上 timer，名為 MPU_CNTL_TIMER_2。第二列的 register 是 MPU_LOAD_TIM2，使用 timer 前可以填入自訂的值，由該值開始倒數，等於是控制 timer 計時的全部長度。第三列是 timer 被啟動之後，依時間 MPU_LOAD_TIM_2 遞減所得的新值會出現在這個 register 中。此 register 稱為 MPU_READ_TIM_2。此外 MPU_CNTL_TIMER_2 尚可控制當 timer 倒數到零時的動作，共有兩種可以選擇，其一是狀態自動全部重設，再開始倒數；另一種是直接停止。本實驗將 MPU_LOAD_TIM_2 設為最大值 2^32，足夠每一次實驗的時間，故當 timer 倒數到零時就直接停止，不再動作。自下節開始有各組實驗目的和結果的探討。

5.2 動態排程實驗

這個實驗的目的在於觀察動態排程的實作結果。在相同環境下，使用 bit rate 192 K-bps 的 bit-stream，執行 HMP Scheduler。預測每一種 service 各自分佈在 ARM 和 DSP 上的比例會相差無幾，但是不會每次的數字都不變。下面表 9 到表 11 是相同的實驗執行三次的結果：

動態排程實驗 1

	time(ms)	ARM(次)	DSP(次)	sum(次)
idct	3888	18043	3659	21702
dequant inter	2221	14693	3385	18078
dequant intra	455	2949	675	3624
interpolation	4158	25646	5843	31489
service time	10724			
app time	14256			
sum(次)		61331	13562	74893

動態排程實驗 2

	time(ms)	ARM(次)	DSP(次)	sum(次)
idct	3886	18059	3643	21702
dequant inter	2220	14699	3379	18078
dequant intra	454	2947	677	3624
interpolation	4162	25625	5864	31489
service time	10723			
app time	14258			
sum(次)		61330	13563	74893

動態排程實驗 3

	time(ms)	ARM(次)	DSP(次)	sum(次)
idct	3894	18033	3669	21702
dequant inter	2218	14693	3385	18078
dequant intra	455	2941	683	3624
interpolation	4164	25632	5857	31489
service time	10733			
app time	14271			
sum(次)		61299	13594	74893

每張表前四列代表不同 service，第一欄是每個 service 所花費的時間，單位是毫秒。第二和第三欄分別是該 service 在 ARM 和 DSP 上處理的次數。而最後一欄是每個 service 總共會處理的次數，因為所用的 bit stream 相同，故這個值不會改變。

第五列是所有 service 耗去的時間。第六列代表整個 decoder 執行的時間。第七列則總和 ARM 和 DSP 的執行次數。

以 interpolation 為例，ARM 上執行的次數和 DSP 執行的次數三個實驗下來都是約為 5：1；ARM 和 DSP 執行的總次數為 6：1。首先看到動態排程在執行時期確實達到動態分配工作的能力。然而以執行時間為分配工作的依據，在這個實驗中可以看到 ARM 所得到的工作量是 DSP 的 6 倍。這是不是意味著 ARM 處理這些 service 的速度較快呢？要探討這個問題，設計了下面一個相異處理器實驗。

5.3 相異處理器實驗

對於 bit rate 為 192 K-bps 的 bit-stream，實驗在 Pure ARM、Pure DSP、和 Dual-core 時的效率。在每個處理器上跑 10 次 decoder 再做平均，觀察其效能的不同。本實驗可以達到兩個目的。第一，探討 ARM 和 DSP 對 decoder service 的相對速度。第二，驗證 dual-core 是否可以達到雙核心加成的能力：

Pure ARM

	service time(ms)	application time(ms)
1	12454	15855
2	12451	15853
3	12459	15850
4	12451	15853
5	12452	15845
6	12452	15855
7	12451	15855
8	12450	15853
9	12454	15857
10	12452	15852
average	12453	15853

Pure DSP

	service time(ms)	application time(ms)
1	14510	17911
2	14638	18039
3	14892	18301
4	14775	18184
5	14560	17970
6	14475	17876
7	15252	18652
8	13558	16966
9	13577	16987
10	14739	18039
average	14498	17893

Dual core

	service time(ms)	application time(ms)
1	11003	14446
2	11055	14502
3	12471	15924
4	14868	18365
5	10170	13598
6	10581	14030
7	11041	14496
8	11803	15287
9	10708	14117
10	11494	14381
average	11520	14915

上方三張表分別是 pure ARM、pure DSP 和 dual-core 三種執行的結果。看到 pure ARM 與 pure DSP 的比較。大家都知曉 DSP 就是為了處理數位訊號而特別設計的處理器，其效能應該會比 ARM 來得好。在這組比較中得知卻是相反的。其中的原因在於 HMP Scheduler 系統的 overhead。細數 overhead 有下例幾項：

- DSP invocation overhead
- Memory copy overhead

由第一項開始說明。ARM 如何通知 DSP 開始工作以及執行哪一個 service？靠著 mailbox 來達成。ARM 如何得知工作已結束？也是靠 mailbox 來達成。因此每個 service 交由 DSP 處理無可避免得承受 mailbox overhead。另外在 DSP 端 kernel 要處理 DSP 上排程問題，也必須執行 context switch，這些都是 overhead。測量之下得到一次 DSP invocation overhead 平均時間要 450 個 time tick。比較 ARM 和 DSP 對 idct 最長的執行時間，ARM 為 600 個 time tick；DSP 為 2080 個 time tick。DSP invocation overhead 佔了 DSP 執行時間的五分之一。

另外，第二項 overhead，是 memory copy overhead。DSP 被限制無法存取 ARM 的記憶體空間，除了共用記憶體外。所以 HMP Scheduler 需負責將資料由 ARM heap 區搬到共用記憶提供 DSP 使用；相同地，DSP 運算完畢後，HMP Scheduler 也要負責填回到 heap 區。當然這一段時間包含在 service 給 DSP 處理的執行時間內。就 idct 而言，必需搬動 64 筆 short 資料兩次，共要花 100 個 time tick。

表 12、13 和 14 記錄了所有 service 執行時間的總和以及應用程式執行的時間。現在讓我們檢視各 sercive 在不同處理器上的執行時間，將表 12 和表 13 各自第一筆實驗的細部資料列出如下二表；

service time

ARM	time(ms)	DSP	time(ms)
idct	3795	idct	4788
inter	2895	inter	3088
intra	591	intra	635
interpolation	5172	interpolation	5998
service time	12454	service time	14510
app time	15855	app time	17911

Reference profile

	ARM clock cycles per MB	DSP clock cycles per MB	ratio
(Y)DC pred. & comp.	44268	8465	5.229533373
(Y)Transform	188250	23178	8.121925964
(Y)Quant	247201	39521	6.25492776
(Y)Inv. Quant	238948	30002	7.964402373
(Y)Inv. Transform	202184	27863	7.256361483
(Y)Reconstruct	106392	17835	5.965349033
(DC)DC pred. & comp.	50458	10142	4.97515283
(DC)Transform	88928	11379	7.815097988
(DC)Quant	134689	20814	6.47107716
(DC)Inv. Quant	124131	15410	8.055223881
(DC)Inv. Transform	100497	13907	7.226360825
(DC)Reconstruct	51913	8861	5.858593838
cavlc	396022	46204	8.57116267
ILF	420018	37209	11.28807547
Total	2393899	310790	

在表 15 裡面可得到各項 service 的執行時間、全部 service 總和執行時間、和應用程式執行時間。我們著重於各項 service 在不同處理器上執行時間的討探，藉此可以了解不同處理器對同一個 service 的處理速度。以 idct 為例，ARM 和 DSP 的比例是 1：1.26(3.7：4.7)。但是根據研究室學長過去的的測試[24] ARM 和 DSP 對 idct 的執行時間比為 7.226：1(Inv. Transform 欄位)，這個數字是在 ARM 沒有開啟 cache 的情況下產生的。估計 ARM 的 cache 打開後，ARM 的處理時間可能會下降到為 DSP 的 2 到 4 倍。它意味著 DSP 對 idct 的執行速度比 ARM 快。比對我們的實驗是 ARM 比 DSP 快。這個差別的來源是因為我們的 service 是引用 reference software 用 C 寫成，但是[24]的 DSP 部份是特別以組語寫成，效率上佔極大優勢。此外 OMAP 5912 的 DSP 是 C55 系例，它的乘法運算是 16 位元 X16 位元的運算。在 reference software 中卻有 32 位元 X32 位元的乘法運算，這使得 DSP 必須利用兩次 16 位元乘法運算來模擬 32 位元乘法運算。因此我們測量的結果 DSP 效能下降很多。

何以我們不也改用組語來寫 DSP 部份的程式和改變乘法的使用方式？其原因是 HMP Scheduler 設計的重點在於減少 platform dependence 以及方便程式的移植。忠重這個想法，我們對 reference

software 只做最少且必要的修改，其他一律依原程式的寫法忠實呈現-- 不特別將 C 語言改為組合語言也不特別對程式做最佳化。所以在這裡 DSP 的效能不如[24]所測量出來的效能。

5.4 相異 bit rate 實驗

在這個實驗中，考量不同 bit rate 對 HMP Scheduler 效能的影響。以 idct 為例，在 bit rate 較低的情況下，會有較多的 0 出現；反之在 bit rate 較高的情況下，0 就比較少。對於底層硬來說，執行乘以 0 和乘以非 0 的動作所花費的時間也有所不同。希望藉由這個實驗探討不同 bit rate 對 HMP Scheduler 的效能有何不同。下面三個表格依順由 bit rate 高到底排列：

bit rate 實驗：192k bps

	service time(ms)	application time(ms)
1	11004	14447
2	11056	14502
3	12471	15925
4	14869	18365
5	10171	13598
6	10582	14031
7	11042	14496
8	11804	15288
9	10708	14117
10	11494	14381
average	11520	14915

bit rate 實驗：128k bps

	service time(ms)	application time(ms)
1	9514	11967
2	9139	11565
3	10204	12660
4	9598	12039
5	9525	11969
6	8682	11084
7	8760	11163
8	8633	11033
9	8526	10936
10	9031	11455
average	9161	11587

bit rate 實驗：64 bps

	service time(ms)	application time(ms)
1	7082	8541
2	8555	10050
3	7320	8788
4	6367	7789
5	6496	7925
6	7371	8836
7	6356	7781
8	7658	9127
9	8444	9935
10	9074	10559
average	7472	8933

每一張表各執行 10 次 decoder，然後再取平均值。第一欄 service time 是每一個 service 所花費的時間的總和；第二欄 application time 代表整個 decoder 完成的時間，但不包括 I/O 的部份。192 K-bps 的 service time 是 11520 毫秒；128 K-bps 的 service time 是 9161 毫秒；64 K-bps 的 service time 則為 7472. 毫秒。很明顯地解 bit rate 較低的 bit-stream 所使用的時間比 bit rate 較高的 bit-stream 少。以這個 300 張 frame bit-stream 為例，decoder 處理 192 K-bps 的效能為每秒 26 張 frame；128 K-bps 是每秒 33 張 frame；最後，最高效率的 64 k-bps 則是每秒 40 張 frame。可以想像得

到對於 bit-rate 較高的 bit-stream，其運算量會較高。對於較高的運算量，相較之下 DSP invocation overhead 和 memory copy overhead 這兩個 overhead 和運算量的比例會降低。可以減低 overhead 對較能的影響。

5.5 DSP delay 實驗

HMP Scheduler 發展的概念是動態的精細分工排程。換言之，HMP Scheduler 會動態地分辨處理器的狀態。理論上當 DSP 處理 service 的速度快於 ARM，service 會被分配到 DSP。但如果 DSP 同時又因為有其他工作在做--不是屬於 decoder 的工作在執行中。在 DSP 上會發生 context switch 輪流處理 service 和另一項工作。這麼一來 service 的處理，從開始到完成的時間就會變長。這個影響會使得 HMP Scheduler 將 service 分配給 ARM 執行，以取得相較之下最好的效能。

我們在 DSP 端隨機地增加不同工作，分別會使 DSP 上的 service 延遲 1000~5000 的 time ticks 不等。因為測試用的 services，單次 service 執行的時間在正常情況下最大不會超過 2500 個 time tick。選擇延遲最大 5000 個 time tick 就會對 DSP 造成很大的影響，可以達到模擬的目的。

在實驗中為了使 DSP 延遲，設計除了 decoder 的 service 之外的另一個給 DSP 執行的延遲用 service，稱之為 D-service。Kernel 利用隨機的方式決定是否使 DSP 延遲，如果決定延遲，就同時將 decoder service 和 D-service 一併透過雙核心溝通機制通知 DSP 執行。D-service 會到上一節介紹過的 parameter table 讀取 kernel 準備的參數。此參數告知 D-service 要做多少次 loop，換言之可以決定延遲的 time tick。

經過測量，一次 mailbox 的往返會花費平均 450 個 time tick。通知 D-service 延遲的時間必需減掉 mailbox 往返的花費，所以是 550~4550 個 time tick。D-service 內每個 service 設計為消耗 550 個 time tick，於是 550~4550 個 time tick 延遲分別是 loop 1 ~ 9 次。實驗結果如表 18、19、20、21 和 22：

dual-core without delay

	time(ms)	ARM(次)	DSP(次)	sum(次)
idct	3889	18043	3659	21702
dequant inter	2221	14693	3385	18078
dequant intra	456	2949	675	3624
interpolation	4158	25646	5843	31489
service time	10724			
application time	14257			
sum(次)		61331	13562	74893

表 18 為對照組，數據內容和表 9 相同。下兩組表格則是本實驗的實驗組。Dual-core with delay 和 delay distribution 兩張表格為一組。首先看到 dual-core with delay，第一欄為 service 單獨的執行時間、service time、和 application time。接著兩欄分別是各 service 執行在不同處理器的次數。再來看到 delay distribution 這張表，delay 這一列顯示的數字是 delay 的 tick 數，如 1000 是指 delay 1000 個 time tick 到 1999 個 time tick。2000 是指 delay 2000 個 time tick 到 2999 個 time tick。但是 5000 的欄是不同的，只有到 5050 個 time tick。因為 5000 對於系統已經是很大的 delay，故只使用到 5050 個 time tick。

dual-core with delay 1

	time(ms)	ARM(次)	DSP(次)	sum(次)

idct	3889	18043	3659	21702
dequant inter	2221	14693	3385	18078
dequant intra	456	2949	675	3624
interpolation	4158	25646	5843	31489
service time	10724			
application time	14257			
sum(次)		61331	13562	74893

Delay distribution 1

delay(tick)	1000	2000	3000	4000	5000
次數	945	1038	788	893	40

Dual-core with delay 2

	time(ms)	ARM(次)	DSP(次)	sum(次)
idct	4348	18247	3455	21702
dequant inter	2608	15039	3039	18078
dequant intra	532	3006	618	3624
interpolation	4823	26206	5283	31489
service time	12310			
application time	15837			
sum(次)		62498	12395	74893

Delay distribution 2

delay(tick)	1000	2000	3000	4000	5000
次數	918	1033	824	865	60

比較加入 DSP delay 和沒有加入 DSP delay 的數據，可以看到 DSP 處理的次數都下降。Dequant inter 的數字由 3385 降到 3039 和 3045，約有 350 次的減少。但是看到 dequant intra 的下降卻只有 70 左右，這是因為會呼叫到 dequant intra 的時機很少，只有 intra frame 上的 macro block 和 inter frame 上的 I macro block 會呼叫。這個 foreman bit-stream 的 intra frame 只有第一張，其他都是 inter frame。所有呼叫 dequant intra 的總次數也只有 3624 次，和其他 service 呼叫量有一個位數之差，所以加入 DSP delay 改變的幅度相比之下不大。

在之前的實驗結果 dual-core 的效能大於 pure ARM 和 pure DSP。當 DSP delay 發生，HMP Scheduler 就認為 DSP 上 service 的執行時間增加，若執行時間大於 ARM 的時間，就會判定 service 由 ARM 執行。既便是原本由 DSP 執行較快的 service 也因 DSP delay 的影響，就不再交由 DSP 負責。Dual-core 加入 DSP delay 因素使得 service time 由 10724 1 毫秒上升到 12325 毫秒，效能降低了。換另一方面看，若系統測試到 DSP 方面的速度下降後，仍執意由 DSP 執行，效能肯定更下降。為了避免這情形發生，HMP Scheduler 會判斷由 ARM 負責執行 service。再者，看到 pure ARM 的平均 service time 是 12453 毫秒，dual-core 加入 DSP delay 的效能下降到和 pure ARM 差不多，但又比 pure ARM 好一些。因為由 ARM 負責的 service 量上升，更接近於 pure ARM 的工作清形。

原始期望的效能是 dual-core > pure DSP > pure ARM。由於前述的 overhead 尚未克服，使得效能成為 dual-core > pure ARM > pure DSP。倘若上述 overhead 都能克服，直接影響到 DSP 在系統中的能力會大幅上升，進而使得 dual-core 也得到幫助。

6. 結論與展望

本篇論文目的在證實一個概念，這個概念就是動態精細分工的排程方式可以在多媒體雙核心平台上有較好的運作效能。為了證實此想法我們提出一個全新的 HMP Scheduler。有鑑於 eCos 作業系統是以元件組成的作業系統，有極高的彈性和修改的便利性，系統開發者可以自由地依

需要的功能選擇對應的元件。利用此特點本論文在 eCos kernel 中加入新的 HMP Scheduler 元件來實作。我們將 HMP Scheduler 設計成依據整個系統的狀態來動態地分配 task 給 GPP 處理器或是 DSP 處理器。這裡考慮的系統狀態是一個抽象的處理器 loading 狀態，排程的標竿是儘量做出使整個系統 loading 最小的排程。因此考量使用預測出最小處理器執行時間來分配工作：只要工作預測出來在 GPP 處理器有比在 DSP 處理器有更短的執行時間，就把工作分配給 GPP 處理器；反之亦然。這裡只用了最短執行時間的因素作為排程依據，未來將會考慮更多的因素來控制動態的排程處理，例如電力消耗和 deadline 等等，以達到更精確更多功能性的考量。在 HMP Scheduler API 方面，HMP Scheduler API 的設計使得現存的單核心程式可以做最少的修改，便可以移植到使用 HMP Scheduler 的雙核心平台上。

除了動態精細分工排程，我們也提出一個雙核心溝通的方法。以有效率的溝通為設計目標，儘量簡化一切可以略去的資訊來往。利用 mailbox 和共用記憶體的組合來實現這個設計。除了一道通知 DSP 執行 service 的 mailbox 動作，其他不能省略的資料都以設計好的資料結構方式，放置在共用的記憶體上。如此一來，便可以不必靠好幾次的 mailbox 來回傳遞資訊。此外，eCos 在本論文實作之前，沒有移植到 5912 OSK 的版本。這個移植工作完成之後，eCos 便多了一個可以直接利用的 5912 OSK 版本。

然而本論文實作之後，發現了一個大障礙，那即是雖然本論文提出的雙核心溝通方式已經減去許多不必要的雙核心資訊往返，但其它的 overhead 沒辦法有效地最小化或甚至消除。此 overhead 包含 DSP invocation 的花費，記憶體搬移的花費。第一個 DSP invocation 的花費在 mailbox 方面是無可避免的支出，但在 DSP kernel 的部份可以再做最佳化。第二個問題可以利用新的 compiler 來改善。記憶體搬移問題，可以將會共同使用的資料就直接配置在共用的 SRAM 上，如此可以減少搬動的時間花費；其他尚有必需與應用程式分開事先建立給 DSP 使用的 service image 的麻煩，但是利用新的 compiler 可以將應用程式和 service image 一併產生，全自動化增加便利性。

一旦能有效地解決上述提出的 overhead，在雙核心平台上精細分工的排程方式將會革命性地取代目前廣泛被做用的鬆散式結合排程方式。進而為日益複雜的嵌入式應用提供更高品質更高效能的環境。

7. 參考文獻

[1] C-N Chiu, C-T Tseng, and C-J Tsai, "Tightly-Coupled MPEG-4 Vedeo Encoder Framework on Assymetric Dual-Core Platforms," IEEE INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS Vol. 3, 2005.

[2] "OMAP5912 Application Processor Data Manual," Texas Instruments, Dallas, Texas, [Online], Available: http://www.ti.com.

[3] R. L. Cancian, L. F. Friedrich, "Performance Evaluation of Real Time Schedulers for a Multicomputer," IEEE International Workshop on Distributed Simulation and Real-Time Applications, 2002.

[4] Satoshi Kaneko et al, "A 600-MHz Single-Chip Multiprocessor With 4.8-GB/s Internal Shared Pipeline Bus and 512-kB Internal Memory," IEEE Journal of Solid-State Circuits, Vol. 39, NO. 1, January 2004.

[5] M. J. Harrold, "Performing Data Flow Testing in Parallel," IEEE, 1994.

[6] M. Annavaram, E.Grochowski, J. Shen, “Mitigating Amdahl’s Law Through EPI Throttling,” IEEE International Symposium on Computer Architecture, 2005.
[7] J. W. Wendorf, R. G. Wendorf, Hideyuki Tokuda, “Scheduling Perating System Processing on Small-Scale Multiprocessor,” IEEE, 1989.
[8] A. G. Greenberg, “Design and Analysis of Master/ Slave Multiprocessor,” IEEE Transactions on Computers, Vol. 40, No. 8. August 1991.
[9] G. Manimaran, C. Siva Ram Murthy, “A Fault-Tolerant Dynamic Scheduling Algorithm for Multiprocessor Real-Time Systems and Its Analysis,” IEEE Transactions on Parallel and Distributed Systems, Vol. 9, No. 11, November 1998.
[10] A. Avritzer et al, “The Advantage of Dynamic Tuning in Distributed Asymmetric Systems,” IEEE, 1990.
[11] S. Majumdar, “Performance Scalability in Multiprocessor Systems with Resource Contention,” IEEE, 2000.
[12] S. Saewong, R. Rajkumar, “Cooperative Scheduling of Multiple Resources,” In Proceedings of 20th IEEE Real-Time Systems Symposium, 1999.
[13] K. K. P. Research, “Increasing Functionality in Set-Top Boxes,” In Proceeding of IIC-Korea, Seoul, 2001.
[14] A. Ferrari et al, “The Design and Implementation of a Dual-Core Platform for Power-Train Systems,” Convergence 2000, Detroit (MI), USA, October 2000.
[15] Paolo Gai, Luca Abeni, G. Guttazzo, “Multiprocessor DSP Scheduling in System-on-a-chip Architectures,” IEEE Proceedings of the 14th Euromicro Conference on Real-Time Systems, 2002.
[16] Bart Veer, John Dallaway, “The eCos Component Writer’s Guide,” Red Hat, 2002.
[17] N. Garnett, J. Larmour, A. Lunn, G. Thomas, “eCos Reference Manual,” Red Hat, 2003.
[18] “eCos User Guide,” eCosCentric, 2003.
[19] A. J. Massa, “Embedded Software Development with eCos,” PERNTICE HALL, 2002.
[20] “OMAP Starter Kit (OSK) OMAP5912 Target Module Hardware Design Specification,” OMPA, Revision 2.3, July 2004.
[21] “OMAP5910 Dual-Core Processor Data Manual,” Texas Instruments, Dallas, Texas, [Online], Available: http://www.ti.com.
[22] C-P Wang, Design and analysis of a Unified Asymmetric Multiprocessors Scheduler, master thesis, NCTU, June 2005
[23] The University of North Carolina, “Feasibility Analysis of Preemptive Real-Time Systems upon Heterogeneous Multiprocessor Platform,” Prceedings of the 25th IEEE Internal Real-Time Systems Symposium, 2004.
[24] Cheng-Nan Chiu, H.264 Video Encoding Optimization on Dual-Core Platform, master thesis, NCTU, June 2005.

附錄三、Java 處理器加速機制的設計

1. Introduction

In this appendix, we first presented a popular method called dynamic code optimization (DCO) for speeding up Java VM. Using DCO in a hardware/software co-design approach is examined in section 2. In section 3, we list the advantages of DCO and hardware/software co-design for embedded Java applications. Finally, the overview of this thesis is given in section 4.

1.1 Why Dynamic Code Optimization (DCO)

Code optimization for dynamically typed object-oriented languages is more difficult than statically typed object-oriented languages. Research shows that the main bottleneck is in the unpredictability of dynamic message sending, which is determined at runtime for dynamically typed object-oriented languages.

In this section, we first illustrate the differences between statically and dynamically typed object-oriented languages, and then we focus on dynamic message sending. Optimizing code dynamically on this topic will significantly improve the efficiency of the system.

1.2 Dynamically-Typed OO Languages

Dynamically typed object-oriented languages, such as Smalltalk and Java, are much slower than statically typed languages like C++. The reason is that the reference variables in dynamically typed languages may potentially reference to any objects in the program at runtime. Therefore type checking of the references can only be done at runtime. Furthermore, the addresses of the dynamic objects are also unknown at compile time. As a result, indirect access must be used, which is again very expensive at runtime. [4]

Consider the Java program segment in Fig. 17, integer i is a local variable in method m(B), and f is an object field in class B. Object cc is sent to method m(B), and the field f of object cc is retrieved and assigned to local variable i. Because the address of object cc is unknown at compile time, the address resolution of cc.f must be done at runtime. When executing the statement i = cc. f, the address of object cc is retrieved first, and then the address of field f is calculated based on the address of the object cc. As a result, there are two indirect accesses in order to get the value of cc.f. These accesses cause the inefficiency of executing dynamically-typed object-oriented programs.

```
class A {
 public void m(B cc) {
  int i;
  i =    cc. f;
  …
 }
```

Fig. 17. Indirect Access Example

1.3 Dynamic Message Sending

In object-oriented languages, message sending is the most frequent operations. When we invoke a method, a message is sent to a class or an object, which selects the method to be executed. Message sending is

also called method invocation in some languages.

Polymorphic operations from dynamic binding and inheritance make it easy for object-oriented language programmers to develop well-designed systems, but also result in the difficulty of efficient execution of these programs. Because the address of the method can only be determined at runtime. To perform a message sending we must extract the name of the method, use it as a key to find the method in the current class (or in the superclass that this method is inherited), continue in this way up the class hierarchy until we find the corresponding method or the top of the inheritance hierarchy is reached.

In Fig. 18, we will show how the polymorphic operations make the execution of object-oriented programs more difficult.

Class A is the superclass of class B, and the m1() method of class B override the m1() method of class A. m0(A) is a method of class A, and m1() method is invoked in it. m2() is also a method of class A, which method is just directly inherited in class B. Note that in the main program, the two statements x.m0(y) and x.m0(z) will invoke a.m1() while execute method m0(A). In the first message sending, the class of y is A, so the statement a.m1() will invoke the m1() of class A. While in the second message sending, the class of z is B, so the statement a.m1() will invoke the m1() of class B. Inheritance property also makes it difficult to determine the access addresses in object-oriented programs. Consider the statement z.m2() in Fig. 18. The class of z is B, but we can not find the method m2() in class B, so we try to look it up in the superclass of B, i.e., class A. The address of method m2() in class A is then retrieved in order to execute this statement. From this example, one can realize that the dynamic message sending is the crucial property of dynamically typed object-oriented languages.

By analyzing the message sending behavior, we can develop dynamic code optimization techniques to improve the efficiency of the language systems. Using the caching mechanism, some duplicated method lookup procedure can be prevented. In this thesis, an adaptive dynamic code optimization mechanism for a java virtual machine is developed. By modifying the runtime behavior, method invocation can be more efficient and the extra memory required for this technique is limited.

```
Class A {
 public void m0(A a) {
    a.m1();
 }

 public void m1() {
 …
}

 public void m2() {
…
}
}

Class B extends A {
 public void m1() {
 …
}
}

main() {
 A x = new A();
        A y = new A();
        B z = new B();

        x.m0(y);
        x.m0(z);
           z.m2();
       }
```

Fig. 18. Polymorphic Operations Example

1.4 DCO for Java VM Using HW/SW Co-design Approach

Java is also a dynamically-typed pure object-oriented language developed by Sun Microsystems in the early 1990. It has many features of modern programming languages, such as simple, object-oriented, robust, secure, architecture neutral, automatic garbage collection, dynamic linking, multi-threaded, and portability. However, it loses the efficiency. Slow execution speed makes Java incapable of handling multimedia applications efficiently without resort to native code or hardware accelerator.

Pure hardware implementation approach, such as java processor, can improve the execution speed greatly. The disadvantages are high design cost and low upgradeability. Hardware/software co-design takes the advantages of both approaches: low cost, flexibility and efficiency, but the execution speed can not be as fast as the pure hardware approach. DCO can significantly improve the system efficiency, which makes this HW/SW co-design approach more useful and powerful.

2. Related Work

In this chapter, we first list some papers and systems about dynamic code optimization. Then we introduce the Java platform including Java execution flow, Java class file format, JVM and its instruction set. In the next section, popular implementation approaches of JVM are discussed, including Java interpreter, Just-In-Time compiler, HotSpot, and Java processor.

2.1 Previous DCO Mechanisms

In this section, we will discuss several dynamic code optimization mechanisms for various dynamically typed object-oriented programming language systems. This concept was first proposed in 1983 [1], with implementation of the smalltalk-80 system. It is called lookup cache. In 1984, an efficient implementation of the Smalltalk-80 system that used a modified cache mechanism (called inline cache) was presented by Deutsch and Schiffman [2]. The inline cache concept now is adapted into many object-oriented language systems. One classical example is polymorphic inline cache, which is implemented in SELF system [3]. Another famous implementation is in the Java programming language. The Java virtual machine and K virtual machine of Sun’s reference implementation which adopts this mechanism will be discussed in the end of this chapter.

2.2 Lookup Cache Mechanism in Smalltalk-80

The Smalltalk definition specifies that the source code is translated into a sequence of primitive operations called byte codes. Smalltalk-80 was originally run on virtual machines which implemented the byte codes in microcode. Early implementations of Smalltalk-80 on hardware interpreted the byte code in software, which led to poor performance [5]. Ungar and Patternson proposed a lookup cache mechanism that can improve the performance of message sending for Smalltalk. [1]

Lookup caches are used to cache the previous lookup result. Method addresses are retrieved from the lookup cache, a hash

table of the most recently used method addresses, via the pair (receiver class, message selector) as the key. The receiver class is the class that the called object belongs to, and the message selector selects the method to be executed. Fig. 19 illustrates the selection mechanism of the lookup cache. When a method is invoked, the pair (receiver class, message selector) is used as a key to the lookup cache. If it hits this hash table, the message address will be extracted and the method lookup procedure can be avoided. Otherwise, the method lookup routine will be processed. And then the new address information will be kept in the lookup cache for next method invocation.

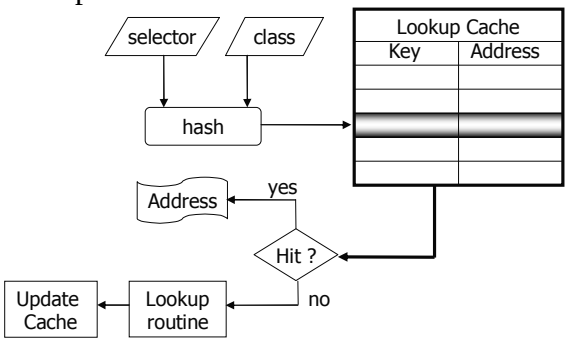


Fig. 19. Selection Mechanism of Lookup Cache

Lookup cache is very effective in reducing the lookup overhead. Berkeley Smalltalk [1], for example, would have been 37% slower without a cache. Furthermore, if the hit ratio of the lookup cache is high, this advancement will be more observable.

2.3 Inline Cache Mechanism in Smalltalk-80

The inline cache mechanism proposed in 1984 [2] predicts the method addresses and places them in the message send site. Even with a lookup cache, sending a message still takes considerably longer than calling a simple procedure because the cache must be probed for every message sent. However, send operations can be sped up further by the observation that the class of the receiver at a given call site rarely varies; that is, if a message is sent to an object of class X at a particular call site, it is very likely that the next time the send is executed will also have a receiver class X.

This locality of receiver class usage can be exploited by caching the most recently look-up method address at the call site (e.g. by overwriting the call instruction). Fig. 20 (see [5]) shows the modification using this technique. Subsequent executions of the sent code jump directly to the cached method, completely avoiding any lookup. Of course, the class type of the receiver could have changed, so the calling method procedure must verify that the receiver class is correct and call the lookup routine if the type test fails. After updating the method code of the receiver class, it may be matched and the method lookup cost can be saved next time. This form of caching proposed by Deutsch and Schiffman is called inline cache since the target address is stored at the sent point. [2]

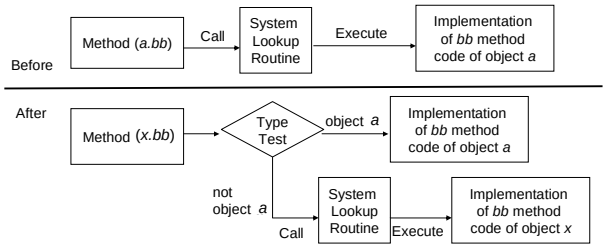


Fig. 20. Inline Cache

Inline caching is surprisingly effective, with a hit ratio of 95% for Smalltalk code [2]. SOAR, a Smalltalk implementation for a RISC processor, would be 33% slower without inline cache [6]. Nowadays all compiled implementations of Smalltalk that

we know is integrated with inline cache mechanism.

2.4 Polymorphic Inline Cache in SELF System

Inline cache mechanism is effective only if the receiver class remains relatively constant at a call site. Although it works very well for the majority of sends, it does not speed up a polymorphic call site with several equally likely receiver classes because the call target switches back and forth between different methods. Worse, inline cache mechanism may even slow down these sends because of the extra overhead associated with inline cache misses.

Based on the inline cache technique, Polymorphic Inline Cache (PIC) caches all method addresses, if the degree of polymorphism is less than ten [3]. The example in Fig. 21 (see [3]) illustrates this.

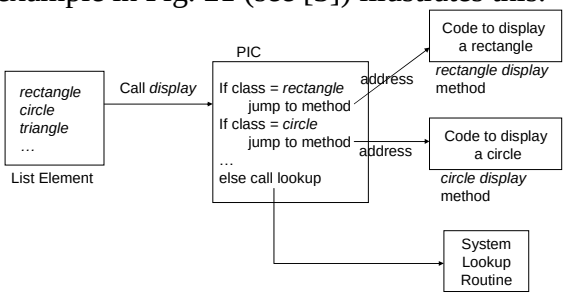


Fig. 21. Polymorphic Inline Cache (PIC)

Suppose that the method display is sent to all classes in the list, the polymorphic inline cache mechanism will handle this method invocation. First, the list element is a rectangle class. Similar to the normal inline cache, the method address will be extracted and the calling code will jump to the direct method code to display a rectangle. It is the same with the class circle. Following the type test, a triangle class is passed. When the system finds that it is a new receiver class type that does not exist in current cache the Polymorphic Inline Cache handler will call the method lookup routine and construct a new branch routine for the display method to rebind the receiver class triangle. Next time the receiver class triangle is called, it can just branch to the corresponding code of the method.

If the cache misses again, the Polymorphic Inline Cache will simply be extended to handle the new case. Eventually, the Polymorphic Inline Cache handler will contain all cases seen in practice, and there will be no more cache misses or method lookup procedures. Thus, a Polymorphic Inline Cache is not a fixed-sized cache similar to a hardware data cache; rather, it should be viewed as an extensible cache in which no cache item is ever displaced by another newer item.

Since many methods are very short, the Polymorphic Inline Cache can be modified to be more effective and more space can be saved. At polymorphic call sites, short methods could be integrated into the Polymorphic Inline Cache handler instead of being called by it. For example, suppose the lookup routine finds a method that just loads the receiver's x field. Instead of using the stored method address to call this method from the handler, its code can be copied directly into the handler, eliminating the calling and return procedure. The figure in Fig. 22 (see [3]) explains this example.

The hit ratio of the Polymorphic Inline Cache depends on the runtime behavior of the programs. In [3], this mechanism is implemented for SELF, a typical dynamically-typed pure object-oriented language. In SELF, all operations including

variable accesses and basic arithmetic operations are implemented by dynamically bound procedure calls.

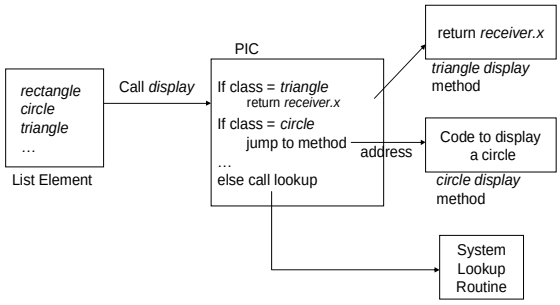


Fig. 22. Inlining a Small Method into Polymorphic Inline Cache

Fig. 23 (see [3]) shows the individual execution time with several benchmark programs. PolyTest is an artificial benchmark with only 20 lines that is designed to show the highest possible speedup with Polymorphic Inline Cache while all the others are produced by software in order to cover a variety of programming styles. The median speedup for the benchmark programs (without PolyTest) is 11%. And the space overhead of Polymorphic Inline Cache is very low, typically less than 2% of the compiled code.

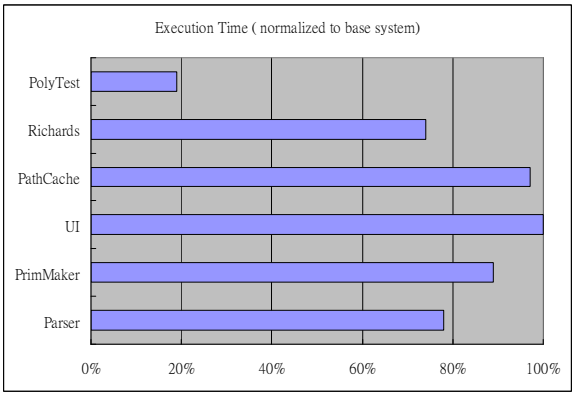


Fig. 23. Impact of Polymorphic Inline Cache

This research also found an interesting observation. In Fig. 24 (see [3]), there is no direct correlation between cache misses and the number of polymorphic call sites. For example, in these benchmark programs, one receiver type dominates at most call sites in PathCache, while the receiver class frequently changes in Parser's Inline Caches. Thus, ordering a Polymorphic Inline Cache Mechanism may win with programs like Parser.

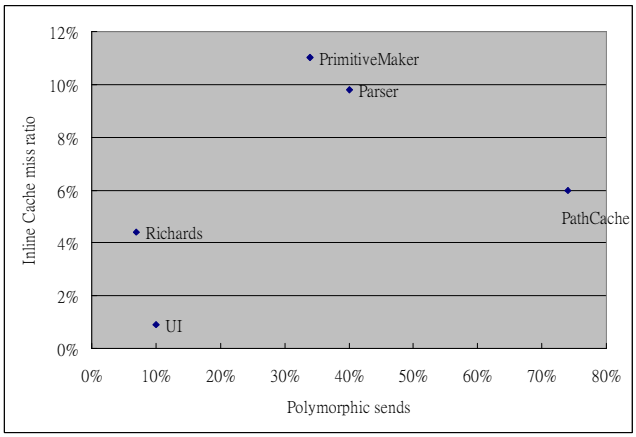


Fig. 24. Inline Cache Miss Ratios

2.5 Java Virtual Machine Reference Implementation

The Java programming language relies on the simulated machine, known as Java Virtual Machine (JVM). JVM allows the computer programmer to communicate with the virtual machine instead of the real hardware system. This is advantageous, because it allows for portability. If the individual JVM are installed on two completely different machines, the Java programs should work well on both machines without any code modification, because it relies on the JVM and not the hardware system it is running on.

Sun Microsystems developed this powerful language system, and this language becomes very popular nowadays. Various Java VM were constructed by different teams that conform to the Java Virtual

Machine Specification [7] but have independent implementations. For a reference implementation, Sun Microsystems also develop a Java Virtual Machine and a K Virtual Machine for a part of the Java 2 Micro Edition (J2ME) called Connected Limited Device Configuration (CLDC) [8]. Dynamic code optimization is also used in these reference implementations to improve the efficiency of Java VM execution.

In Sun's version of the Java Virtual Machine, compiled java Virtual Machine code is modified at runtime for better performance. This optimization takes the form of a set of pseudo-instructions that are distinguishable by the suffix _quick in their mnemonics. These are variants of normal Java Virtual Machine instructions that take advantage of information learned at runtime to do less work than the original instructions.

TABLE I. Fast Bytecodes in Sun's Java VM Reference Implementation

203	ldc_quick
204	ldc_w_quick
205	ldc2_w_quick
206	getfield_quick
207	putfield_quick
208	getfield2_quick
209	putfield2_quick
210	getstatic_quick
211	putstatic_static
212	getstatic2_quick
213	putstatic2_static
214	invokevirtual_quick
215	invokenonvirtual_quick
216	invokesuper_quick
217	invokestatic_quick
218	invokeinterface_quick
219	invokevirtualobject_quick
221	new_quick
222	anewarray_quick
223	multianewarray_quick
224	checkcast_quick
225	instanceof_quick
226	invokevirtual_quick_w
227	getfield_quick_w
228	putfield_quick_w

To learn from inline cache mechanism [2], the Reference Implementation (RI) of Sun's JVM also uses the concept of caching the previous method lookup information and stores them in the instruction space. Only standard java bytecode instructions numbered from 0 to 201 may be generated by the java compiler. The optimization works by dynamically replacing occurrences of certain instructions by the reserved instructions (in the range of 202-255) after the first time they are executed. These new instructions listed in Table 1 have been loaded and linked the first time the associated regular instruction is executed.

Note that these new instructions (referred to as fast bytecodes) are not specified in the Java Virtual Machine Specification [7]. However, for the implementation of Java Virtual Machine the adoption of the fast bytecodes has been proven to be an effective optimization technique.

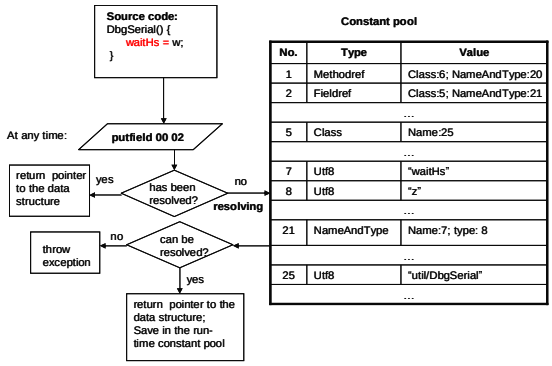


Fig. 25. Original Execution Flowchart

Fig. 25 shows the original execution flowchart if we do not enable fast bytecodes. Consider the assignment instruction waitHs = w in Function DbgSerial ( ). A sequence of java bytecodes will be generated after compilation, and putfield 00 02 is the core instruction of this assignment. When Java VM fetches this instruction for execution, first it will check that if this constant pool component, indexed by 2 in this case, has been resolved. The java constant pool inside the java class file format is designed to support dynamic linking. When the Java Virtual Machine encounters a use of a constant pool entry for the first time (e.g., when you first use the new statement to create a new object of a class, or in the first use of getfield to get a field), the constant pool entry is resolved [9].

The actions the JVM performs to resolve a constant pool entry depend on its type. Resolution of an entry involves two basic steps: checking that the item you are trying to access exists (possibly loading or creating it if it doesn't already exist), and checking that you have the right permissions to access the item (i.e., making sure that you don't access private fields in other classes, etc.). In Fig. 25 the constant pool of the class DbgSerial is listed. The Java VM checks that the index 2 points to a field that belongs to the class util/DbgSerial (index 25), its name is waitHs (index 7), and its type is an integer ("z" in index 8). If any illegal situation happens, an exception will be thrown by the Java VM. After the entry is resolved, the address of this constant pool item will be returned for execution of the Java VM. At the same time, this address will be stored in the runtime constant pool of that class. Next time this constant pool is used, the Java VM will find that it has been resolved and use the direct address in the runtime constant pool.

If fast bytecode is enabled, many duplicated subroutines can be avoided. A flowchart in Fig. 26 shows the modification.

At the first execution of the Java instruction, the Java VM resolves the item address or gets the constant pool item address from run-time constant pool if it has been resolved. JVM then overwrites the instruction with the _quick or _quick_w pseudo-instruction listed in 0 with corresponding new operands which may be one byte or two bytes determined by the length of the item address. The instructions putstatic, getstatic, putfield, and getfield each have two _quick versions, chosen depending on the type of the field being operated upon (i.e., putstatic2_quick if the type is long or double). From this point on, the subsequence execution of that instruction instance is always the _quick variant and can be execute directly without any check and runtime constant pool consulting.

The operands of these new instructions are invisible outside of the Java Virtual Machine. Sun Microsystems provides a possible solution, but the decisions such as

the format of operands are left up to the implementer. Just remember that the operands of the _quick pseudo-instruction must fit within the space allocated for the original instruction's operands.

With this dynamic code optimization, a significant amount of time is thus saved on all subsequent invocations of the pseudo-instruction.

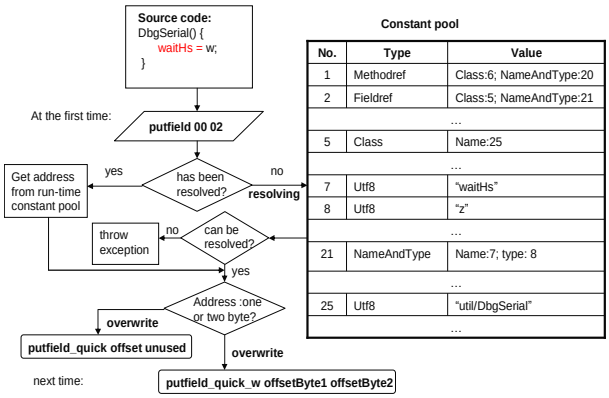


Fig. 26. Execution with Fast Bytecodes

2.6 Sun's K Virtual Machine Reference Implementation

Recognizing that one size does not fit all, Sun Microsystems has grouped its Java technologies into three editions, and each of them aimed at a specific area of today's vast computing industry. Java 2 Enterprise Edition (J2EE) is for enterprises needing to serve their customers, suppliers, and employees with solid, complete, and scalable Internet business server solutions. While Java 2 Standard Edition (J2SE) is for the familiar and well-established desktop computer market. The Java 2 Micro Edition (J2ME), targeted at two broad categories of products: CDC (Connected Device Configuration) and CLDC (Connected, Limited Device Configuration), is specified for the consumer and embedded device manufacturers, service providers, and content creators.

For these three different Java editions, the underneath Virtual Machine also have different execution speed and ability. The K Virtual Machine (KVM) is developed for CLDC in Java 2 Micro edition, which is a compact, portable Java Virtual Machine specifically designed from the ground up for small, resource-constrained devices. The high-level design goal for the KVM was to create the smallest possible complete Java virtual machine that would maintain all the central aspects of the Java programming language, but would run in a resource-constrained device with only a few hundred kilobytes total memory budget.

In Sun's Reference Implementation, dynamic code optimization is also used in the K Virtual Machine. The implementation details are just like Sun's JVM RI, but _fast suffix is used as the new instructions instead of _quick. By caching the method lookup result in the call site, the time of searching constant pool and method table can be saved.

As the restrictions of DCO in Java VM, the KVM implementers also need to assure that the executed java instructions are stored in RAM or other memory types that the stored data can be modified at runtime. The other important restriction is, the operands of the _fast pseudo-instruction must fit within the space allocated for the original instruction's operands. Instead of just saving the corresponding address, KVM provides a second technique to save more execution time. Some instructions need much information to be executed, such as

invokevirtual, which instruction will invoke a method of an object instance. The information (e.g. parameter, method's return type, etc) now can be stored in an external memory called inline cache, and an index to the inline cache is used in the instruction operands. The new instruction format is illustrated in Fig. 27.

This additional dynamic code optimization technique in Sun's KVM RI requires about 100 Kbytes extra memory, but it has been proven to be very efficient. The execution time with fast bytecodes enabled is two or three times faster than without it. Because using inline cache to execute method invocation, the performance of Sun's KVM RI is much better than the JVM RI.

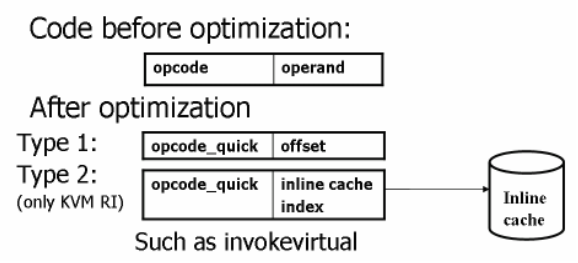


Fig. 27. Java Instruction Format Using DCO

3. Proposed Dynamic Code Optimization System

Due to the demand of efficiency in DVB-MHP applications, we need to further improve the performance of the JOP system. By analyzing the execution frequency, we observed an important feature and use it to design our new dynamic code optimization scheme.

In this chapter, we first discuss the data structure using our framework. Then we analyze the bytecode execution frequency and give an overview to our scheme. Finally, the hardware and software modules of our design are respectively illustrated.

3.1 Data Structure Using in Our Dynamic Code Optimization

In this section, the data structure using dynamic code optimization is given. These include the data arrangement in the external memory, method cache and each of the runtime data structure.

i. Data Arrangement in the External Memory

The application programs are compiled into Java class files by the Java compiler (javac), with all the linked library programs recompiled, and then passed to JavaCodeCompact (JCC).

In conventional class loading, javac is used to compile Java source files into Java class files, which are loaded into a Java system, either individually, or as part of a jar archive file. Upon demand, the class loading mechanisms resolve references to other class definitions. JCC provides an alternative means of program linking and symbol resolution. First the multiple input class files will be combined, and JCC will determine the layout and size of an object instance. Only the designated class members will be loaded and linked with the Java Virtual Machine in order to reduce JVM's bandwidth and memory requirements. Resolution of symbols is also performed in this stage, which reduces the start-up time of JVM.

The output of JCC is a C file and its format can be arranged by the user-defined writer. In JOP system, the writer is redesigned to have JCC output a data layout file like the data arrangement in the external memory (SRAM in Spartan-3) and loaded it

directly to the external memory. An illustration of it is shown in Fig. 28.

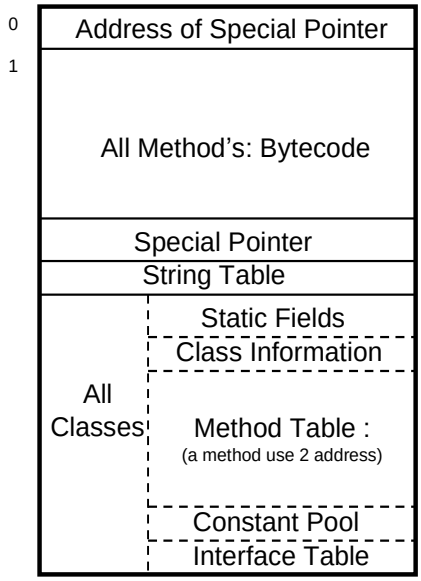


Fig. 28. Data Arrangement in the External Memory

All of data in this output file are united in 32 bits of an address. This means that the address 0 has 32 bits data, and the 33rd bit is the first bit in address 1. After collected all the designated method bytecodes, JCC has the bytecode size in 32-bits. The JOPWriter writes this size added one in the first address, and then all the designated method bytecodes. Finished all the writing of bytecodes, the next writing address must be the data saved in address 0, because it is the size of bytecodes added one.

Then we save four special pointers: a pointer to boot code, a pointer to first non-object method structure of class JVM, a pointer to first non-object method structure of class JVMHelp, and a pointer to main method structure. We can easily get special pointers by using the data in address 0, because it is also the address of first special pointer. For example, the data in address 0 adds three is the address of main method structure.

The next area is the string table area, followed by the all-class data area. The all-class data area contains the static fields, class information, method table, constant pool, and interface table if this class has interfaces. In this area, the data related to all the classes are listed one after another.

All of the information in the output file (the same as in external memory) will be used while execution. The method table (Fig. 29) of a class is the key data structure to get the address to other class information. Note that a method table occupies two address spaces, and an address is 32 bits.

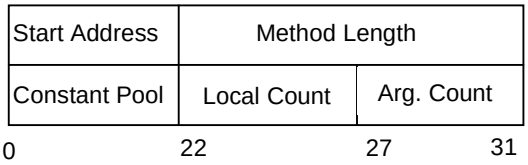


Fig. 29. Method Table Structure

The highest 10 bits in the first address of method table are the length of method bytecodes with 32 bits a unit. By shifting right 10 bits of the first address we can get the method bytecodes' start address that points to the second block in Fig. 28. The start address has 22 bits and it is in Big-Endian byte order. The second address stores the constant pool pointer in 22 bits, the number of local variables in 5 bits, and the number of arguments in 5 bits.

ii. Method Cache

Method cache is also called bytecode cache. Because the fetch of external memory is very expensive, the concept of method cache is created in JOP. During one external memory fetch, the whole bytecodes of one executing method are fetched and loaded to

the method cache, which is usually a memory area synthesized on FPGA. The external memory fetch time can be smaller than fetching one address a time. For example, assume that we fetch one address in external memory takes 3 microseconds. We will spend 30 microseconds if we want to fetch a method with 10 units (32 bits a unit) address bytecodes. However, if we fetch all bytecodes of that method (10 units address) one time, we may just spend 22 microseconds in fetching external memory.

Method cache is designed to cache just one method bytecodes. Consider this example program [14]:

```
Foo () {
    A();
    B();
   }
```

We will have the following cache loads:

1. method *Foo* is loaded on invocation of *Foo*()
2. method *A* is loaded on invocation of *A*()
3. method *Foo* is loaded on return from *A*()
4. method *B* is loaded on invocation of *B*()
5. method *foo* is loaded on return from *B*()

It should refill the method after returned from its internal method. This is the main drawback of the method cache. But by that we can almost make sure that the method cache will reload when executing the same method next time. As a result, we do not need to reflash the method table when we modified the executing method bytecodes in our dynamic code optimization scheme. This also saves much time in doing optimization.

iii. Runtime Data Structure

As we mentioned before, memory is addressed as 32 bits data, so the memory pointers are incremented for every four bytes. No single or 16 bits access is necessary in our JOP system. The reference data type is a point to memory that represents the object or an array, which is pushed on the stack before an instruction operating on it. A null reference is represented by the value 0 [14].

In the following we are going to see each runtime data structure.

a. Stack Frame

First we look into the stack frame. On a method invocation, the information of the invoker is saved in a newly allocated frame on the stack. It is restored when the method returns. The information consists of five registers: SP (Stack Pointer), PC (Program Counter), VP (Variable Pointer), CP (Constant Pool Pointer), and MP (Method Table Pointer).

SP, PC and VP are registers in JOP while CP and MP are local variables of JVM. Fig. 30 (see [14]) provides an example of the stack change before and after invoking a method. The caller has two arguments and the called method has two local variables. The arguments that we want to pass into the invocated method can be accessed in the same way as local variables. As in this example, the arguments arg_0 and arg_1 will become var_0 and var_1 with the original var_0 and var_1 shifted to var_2 and var_3. The start address of the frame can be calculated with the information from the method table:

Frame address = VP + Arg. Count + Local

Count

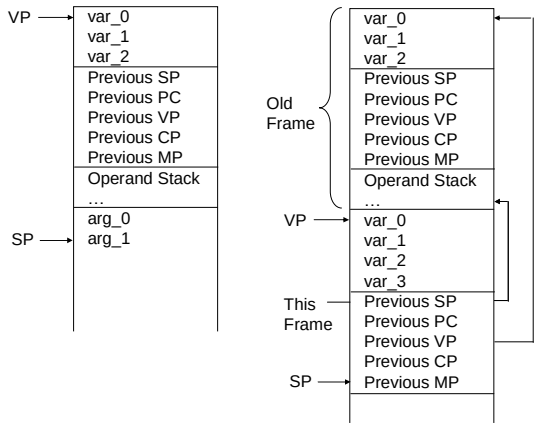


Fig. 30. Stack Change on Method Invocation

b. Data layout

In JOP, objects are stored in memory during runtime in the Fig. 31 (see [14]) format. Note that the object reference points directly to the first reference of the object to speedup. We can access the class information pointer by object reference subtracted one.

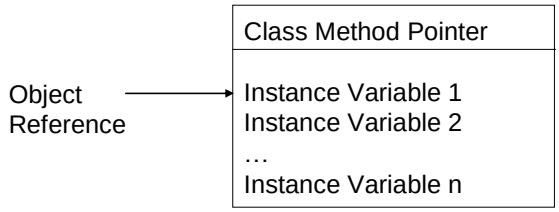


Fig. 31. Object Format

The array layout in memory is just like an object. We showed the array format in Fig. 32 (see [14]). Also, if we want to access the array length, just take object reference subtracted one.

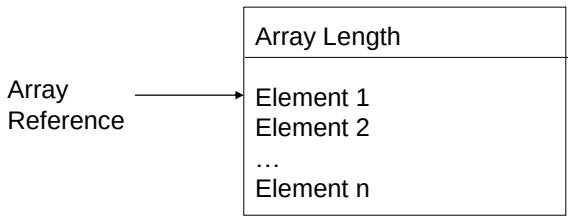


Fig. 32. Array Format

c. Runtime Class Structure

The runtime class structure of JOP is shown in Fig. 33 which had discussed in i as all classes' information. This class structure is stored in the external memory. For indicating the pointers in previous data structure, we drew this class structure again with pointers Class Reference, Class Method Pointer, MP, and CP.

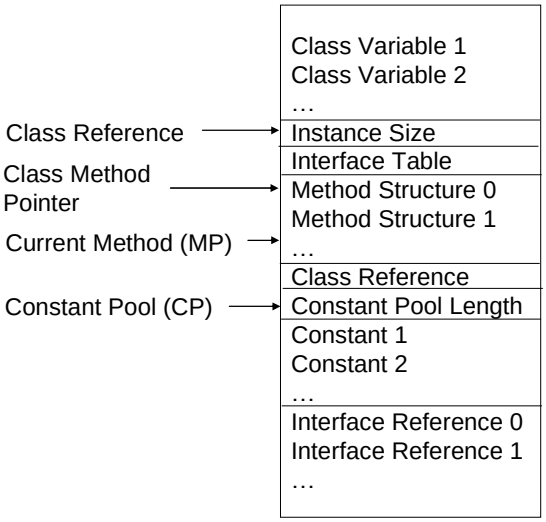


Fig. 33. Runtime Class Structure

3.2 The Proposed Dynamic Code Optimization Scheme

In this section, we propose our dynamic code optimization scheme. First we analyze the bytecode execution frequency, from which we get the new idea of improvement. Then we compare the access time of the external memory and the internal memory. By reducing the number of dynamic code modifications that do not improve the performance, we can make the system more efficient. The architecture overview is illustrated in the last subsection.

i. Analysis of Bytecode Execution Frequency

We have mentioned that Hotspot uses optimistic compilation which can dynamically choose which instructions needs to be compiled and the rest are executed by the interpreter. The decision is based on the execution frequency. This concept is used in many systems. For example, the famous code morphing processor from Transmeta also uses execution frequency to decide whether the code is to be interpreted or to translated. As Fig. 34 (see [22]), the translation threshold is decided by the code execution frequency. When the number of executions of a section of x86 machine code reaches a certain threshold, its address is passed to the translator.

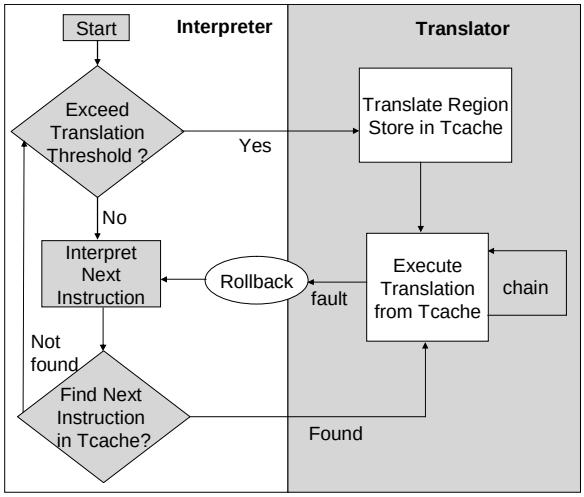


Fig. 34. Transmeta Code Morphing Software Control Flow

We analyze the bytecode execution frequency using three common benchmark programs. The distribution of bytecode execution frequency is shown in Fig. 35. The number of bytecodes is counted under the given execution frequency. Consider the following analysis data. When executing the UDP/IP program, there are 385 bytecodes that are executed 9 times and 210 bytecodes are executed 13 times. For the same bytecodes (e.g. aload_0), they are different in different methods or sequences.

Look at the curves in Fig. 35. We observe a very important rule. The bytecodes are almost executed exactly once or much more than twice. This observation is the critical point in our design.

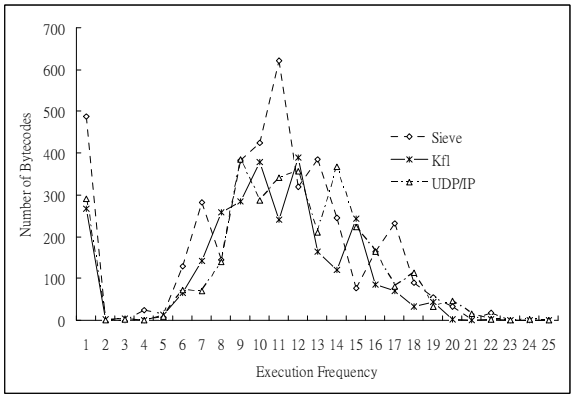


Fig. 35. Distribution of Bytecode Execution Frequency

ii. Access Time of External Memory & Internal Memory

As we mentioned before, typical dynamic code optimization can speed up the execution of embedded Java VM, but it suffers from the overhead of external memory accesses.

Consider the JOP system. The clock frequency of both FPGA and SRAM is 50MHz, so the clock time is calculated as following.

$$\frac{1}{50M} = \frac{1}{50} \times 10^{-6} = 0.02 \times 10^{-6} = 2 \times 10^{-8} \text{seconds}$$

The internal memory access only needs 1 cycle. But if it is the external memory, it needs 5 cycles for memory read and 7 cycles for memory write on JOP because JOP is

designed for various developing boards. The microcode sequence of external memory read is shown in Fig. 36.

```
stmra
nop
wait
wait
ldmrd
```

Fig. 36. Microcode Sequence of External Memory Read

Upon execution of a memory read, the address is stored and the processor waits for the value to arrive and then pushed the value to the top of the operand stack as in Fig. 30. Each microcode executes in a single cycle, so the external memory read needs 5 cycles. For the microcode sequence of external memory write shown in Fig. 37, it needs 7 cycles.

```
stmwa
nop
stmwd
nop
wait
wait
nop
```

Fig. 37. Microcode Sequence of External Memory Write

As a result, if we can reduce the number of dynamic code modifications that do not give us any advantages, e.g. the codes that are exactly executed once, we can make a big improvement of execution time and cut down the power consumption. In next subsection we are going to introduce the design of our dynamic code optimization module.

iii. Architecture Overview

From previous experiment, we knew that bytecodes are almost executed exactly one time or much more than two times. Then in subsection ii, we analyzed the memory access time, and found that the access time of external memory is a big overhead of the traditional dynamic code optimization scheme. Based on these two observations, we designed the new dynamic code optimization architecture called JDCO.

To speed up the execution and cut down the power consumption, we only modify the codes when it is necessary. That is, if the code is executed exactly one time, we do not do the dynamic code optimization – constructing a new bytecode to replace the original bytecode and storing the field or method offset in the operand of new bytecode. Because the method bytecodes are stored in external memory in most embedded system and also our JOP system (described in subsection 3.1 i), this new module can execute the Java programs with dynamic code optimization in a more efficient way.

However, if the execution frequency can not be determined upon the first encounter of a bytecode (unless we do a "fast-forward" to check whether the bytecode will be executed again, which has unacceptable overhead). Another possible way is to perform a pre-pass counting of the execution of the bytecodes, but this is also very expensive. We proposed a simple algorithm that reduces unnecessary modifications with very low overhead. The proposal is as follows. A small memory is synthesized in the FPGA to count the number of execution of each bytecode during execution. For the first execution, no dynamic code modification is performed.

The DCO is only done at the second time the code is executed, because we assume that it will be executed again and again base on the observation of subsection i. For third execution and above, we can directly use the operand of new bytecode to speed up the performance and cut down the power consumption.

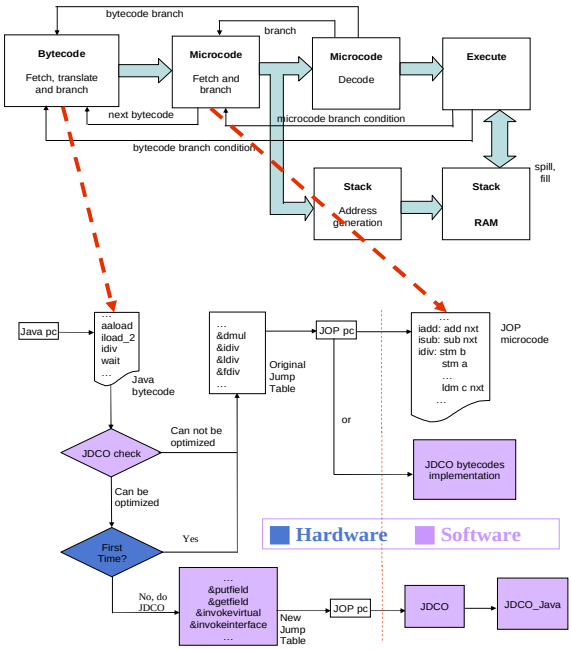


Fig. 38. Proposed JDCO Architecture

The flowchart of our JDCO architecture is shown in Fig. 38. We mark our new modules in colored background with distinguishing hardware and software implementation modules. In the beginning of the first stage, bytecode fetch, a bytecode is pointed by Java pc to be executed. The bytecode will pass to a JDCO check module, which will check if this bytecode can be optimized or not. For example, if the bytecode has the information that can be recorded for speeding up the next execution (e.g. getfield. putfield. etc.), we say that it can be optimized. If the answer of JDCO check is yes, our system will further check if it is the first time to execute this bytecode to decide whether we should perform DCO or not. If it is not the first time of execution, the new JDCO optimization will look up the bytecode in our new jump table to get the JOP pc, which points to our new JDCO module in the second stage, microcode fetch. JDCO will execute this bytecode and get the runtime information depending on the specific bytecode. It may be the offset of an object field, or of the class method that will not change when next time we execute the same bytecode. The runtime information will be passed to a JDCO Java program which will construct a new bytecode to replace the original bytecode in external memory, and store the runtime information in the operand of this new bytecode.

If the answer of the JDCO check is no, or it is yes but this is only the first time of execution of the byte code, our architecture will follow the original procedure. Looking up in the jump table, the JOP pc is retrieved for execution. The corresponding bytecode implementation is executed whether it is a newly implemented bytecode that we constructed or not. The implementation of the bytecode may be the VHDL implementation, microcode implementation, or Java Code implementation.

3.3 Implementation Details

In this section, we are going to look into more details of the implementation. The description is divided into two parts: hardware implementation modules and software implementation modules, which distinguished in Fig. 38.

i. Hardware Modules

In our design, a hardware module is needed for first time of execution checking.

We need to synthesis a small on-chip memory that can count the execution times of each bytecodes, and then decide to do the original bytecode implementation or the JDCO module.

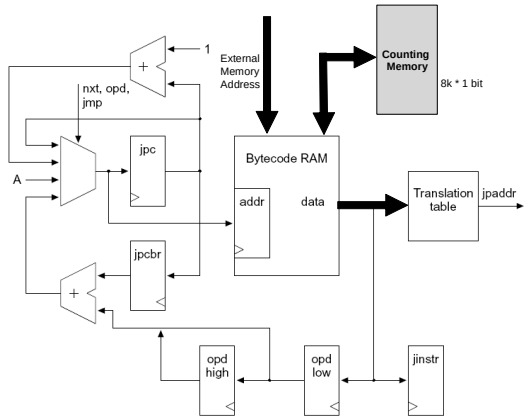


Fig. 39. Java Bytecode Fetch Stage of The Proposed JDCO

As we mentioned before, JOP has four pipeline stages. In the first pipeline stage as in Fig. 39, the Java bytecodes are fetched from the internal memory (Bytecode RAM). The bytecode is mapped through the translation table into the address (jpaddr) for the microcode RAM in next stage.

We synthesize an 8K * 1 bit memory called Counting Memory, in which one bit map to an address of method bytecode in external memory (see Fig. 28). When Java bytecodes are fetched from the internal memory, we use its start address of method as an index to see if the bit in Counting Memory is set or not. If it is set, we know that it is the second time executed. Then the address of the modified bytecode implementation (e.g. putfield_modify in next subsection) is mapped through the translation table and passed to next stage. If the bit is zero, then the address of original bytecode (e.g. putfield) is mapped and passed. Finally the corresponding bit in Counting Memory is set.

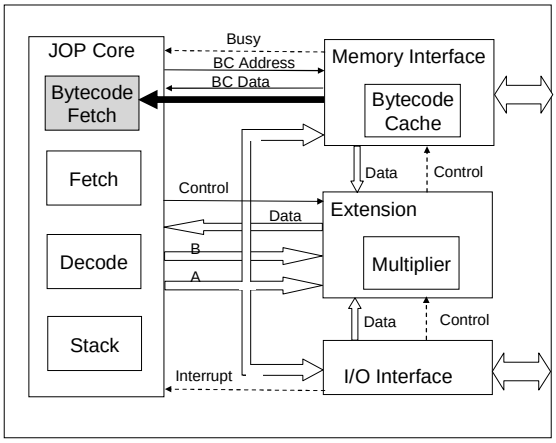


Fig. 40. Block Diagram of The Proposed JDCO

The main modification to the JOP core is in the Bytecode Fetch stage. But we do not have the method start address of external memory because method bytecodes are fetched from the bytecode cache missing the original address in external memory. So we need to map this port (external memory address) from memory interface to JOP core, and map to the Bytecode Fetch stage. We summarize our modification and redraw this as in Fig. 40.

ii. Software Modules

In bytecode level, first we should figure out what bytecodes are needed to do our JDCO. Based on Sun's JVM Reference Implementation, we may have 25 bytecodes (0) that can be considered. But most operands of them are not modified. The new bytecodes of these bytecodes just indicate that they have been resolved. All bytecodes are passed to JavaCodeCompact (JCC) first, and then the output is loaded into the external memory in JOP system. In other words, all method bytecodes in external memory have been resolved, and the DCO is not useable for these. As a result, we only

have four bytecodes needed to do JDCO: getfield (180), putfield (181), invokevirtual (182) and invokeinterface (185).

To fulfill our JDCO modules, two bytecodes need to be constructed for one bytecode without changing the instruction length. We list the new bytecodes of our architecture with their format in Table II.

Upon the first time of executions, we will execut the original bytecodes. Modified bytecodes are used in the second executions and above, and replaced itself in the new bytecodes. The offset of object field or class method will be stored in the operand of the new bytecodes for next execution.

TABLE II. Proposed JDCO Bytecodes

	bytecode	format				
Original Bytecodes (not changed)	180	getfield	indexbyte1	indexbyte2		
	181	putfield	indexbyte1	indexbyte2		
	182	invokevirtual	indexbyte1	indexbyte2		
	185	invokeinterface	indexbyte1	indexbyte2	nargs	0
Our JDCO Bytecodes	228	getfield_modify	indexbyte1	indexbyte2		
	229	putfield_modify	indexbyte1	indexbyte2		
	230	invokevirtual_modify	indexbyte1	indexbyte2		
	231	invokeinterface_modify	indexbyte1	indexbyte2	nargs	0
	233	getfield_new	offserbyte1	offsetbyte2		
	234	putfield_new	offserbyte1	offsetbyte2		
	235	invokevirtual_new	offserbyte1	offsetbyte2		
	236	invokeinterface_new	offserbyte1	offsetbyte2	nargs	0

4. Performance Study

In this section, we first introduce our development environment – Xilinx Spartan-3 Developing Board, and then we state the Java benchmark used in this research. Finally, the experiment results are shown and discussed. We analyze the performance on both execution time and power consumption.

4.1 Xilinx Spartan-3 Development Board

The Xilinx Spartan-3 Developing Board is used for the development of the proposed Java VM accelerating algorithm. The detail spec. of the board is in [21]. The equivalent gate counts of the target Spantan-3 device are 200,000 gates, and the logic utilization of JOP on the FPGA is 64 percent. The data path of Spartan-3 is 32 bits with an 8-bit memory interface. Shift instruction can be computed in exactly one single cycle. The external memory devices of JOP on Spartan-3 is a 32-bit SRAM blcok of 1M bytes and an 8-bit flash of 2M bits. Java program is compacted by JCC to *.jop file which is loaded into SRAM. Configuration data is stored in flash. Finally, the maximum working frequency of this processor is 194.621 MHz, according to the synthesizer.

4.2 Java Benchmark Programs

In this research, we use three small Java benchmark programs, which contain a synthetic benchmark (Sieve of Eratosthenes) and two application benchmarks, Kfl and UDP/IP. [14] We describe them in the following subsection.

i. Sieve of Eratosthenes

This program will produce a list of prime numbers. The algorithm is proposed by Erastosthenes. His method is as following. First, write down a list of integers. Then mark all multiples of 2. The next step is, move to the next unmarked number, in here is 3, and mark all its multiples. Continue to mark all multiples of the next unmarked number until there are no new unmarked numbers. The numbers which survive from this marking process (the Sieve of Eratosthenses) are primes.

ii. Kfl

Kfl is adopted from a real-time application which is taken from one of the nodes of a distributed motor control system. The motor control system is a solution to rail

cargo. During loading and unloading goods from wagons, a large amount of time is spent due to the obstacle of contact wires. Balfour Beatty Austria developed and patented a technical solution called Kippfahrleitung to tilt up the contact wire. An asynchrony motor on each mast is used for this titling. However, it has to be done synchronously on the whole line. [23]

Each motor is controlled by an embedded system. This system also measures the position and communications with a base station [14]. The base station need to control the deviation of individual positions during the tilt. It also includes the user interface for the operator. In technical term, this is a distributed, embedded real-time control system, communication over an RS 485 network.

A simulation of both the environment (sensors and actors) and the communication system (commands from the master station) forms part of the benchmark, so as to simulate the real-time workload.

iii. UDP/IP

UDP/IP benchmark is composed of a tiny TCP/IP stack (Ejip) for embedded Java. This benchmark contains two UDP server/clients, exchanging message via a loopback device.

4.3 Experiment Results

We simulated our dynamic code optimization scheme on Spartan-3. The percentage of logic utilization increment is less than 1%, but we have made a big improvement in both execution time and power consumption. Now we are going to discuss in these two aspects.

i. Execution Time

We synthesize our JDCO system with comparisons to DCO (no frequency check) and the original JOP system. The execution time is listed in Table III and shown in Fig. 41. In the table, we can see that the average speedup of our system is 13.8%, and compare to DCO system, we also have 7.1% execution time speedup.

Let us focus on the results of UDP/IP benchmark. In our JDCO system, it has 9.7% speedup compared to DCO system, while other two benchmarks only have 6.0% and 5.6% speedup. The reason is that the UDP/IP benchmark has many initialization and executed-only-once code, so our JDCO system can make a big improvement by avoid that cases. Actually, the performance of this system is dependent on the Java program behavior.

TABLE III. Execution Time

system / benchmark	JOP	DCO	JDCO	DCO/JOP	JDCO/JOP	JDCO/DCO
Sieve	11813	11043	10382	0.935	0.879	0.940
Kfl	2719	2419	2284	0.890	0.840	0.944
UDP/IP	4813	4627	4179	0.961	0.868	0.903
average	6448.3	6029.7	5615.0	0.929	0.862	0.929

Unit: millisecond

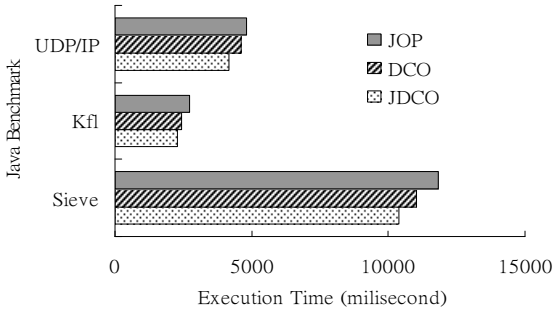


Fig. 41. Execution Time

ii. Power consumption

To estimate the power consumption savings, we can analyze the microcode

execution cycles and the external memory access times. We discuss the two aspects in the following subsections.

iii. Microcode Execution Cycles

As we know that the less microcode execution cycles, the less power consumption will be. We analyze the microcode execution cycles of each bytecode and separate them by the number occurrences.

Because we have different microcode execution cycles in different number of occurrences, we should know the total execution times of the modified bytecodes of each benchmark separating by the number of occurrences, which is listed in Table IV. But these are the sum of the four modified bytecodes (putfield, getfield, invokevirtual, and invokeinterface), we should know the percentages of each of them. By analyzing the benchmark programs, we assume the percentages of the bytecodes as following:

180 : 181 : 182 : 185 = 40 : 20 : 20 : 1

TABLE IV. Microcode Execution Cycles of Each Bytecode

# occurences / bytecodes	For JDCO			For DCO	
	first	second	third and later	first	second and later
getfield	20	33	7	33	7
putfield	23	36	10	36	10
invokevirtual	106	119	98	119	98
invokeinterface	118	131	110	131	110

Unit: cycles

We can calculate the microcode execution cycles by the following formulation:

$$\sum_{\#\ occurrence} (T * (\sum_{bytecode} (cycles * P)))$$

T is the execution times in Table V, and P is the percentage of bytecodes. For example, P of getfield is 40 / (40+20+20+1).

The principle of this formulation is to calculate the sum of the execution cycles multiply the execution times. The execution cycles are calculated according to the percentage of each bytecode. Note that the microcode execution cycles of original JOP are always the same as the first time of JDCO.

TABLE V. Execution Times of Bytecodes 180. 181. 182. 185

# occurences / bytecodes	For JDCO			For DCO		For JOP
	first	second	third and later	first	second and later	all
Sieve	1874	1592	17462	1874	19054	20928
Kfl	1129	1001	12384	1129	13385	14514
UDP/IP	1342	1128	14287	1342	15415	16757

We still calculate the execution cycles of our JDCO system with comparison to DCO and original JOP system. The experimental results are listed in Table VI and shown in Fig. 42. Because we only calculate on the modified bytecodes, we need to know the percentage of them of all bytecodes. By analyzing the benchmark programs, we get that the roughly percentage is 1/2. That is, $(180 + 181 + 182 + 185) \div \text{all} = \frac{1}{2}$.

As shown in Table VI, our JDCO has average 20.8% less execution cycles for the modified bytecodes, so for the all bytecodes, we have 10.4% less execution cycles than the original system. However, our JDCO has a little more microcode execution cycles than DCO system. This can be easily explained. By comparing between our JDCO and DCO system, we have less execution cycles for the executed-only-once bytecodes, but the needless first time overhead is happened to all the other bytecodes.

TABLE VI. Microcode Execution Cycles of Bytecodes 180. 181. 182. 185

benchmark \ system	JOP	DCO	JDCO	DCO/JOP	JDCO/JOP	JDCO/DCO
Sieve	903779.6	705139.2	720105.5	0.780	0.797	1.021
Kfl	626789.8	484812.7	494864.1	0.773	0.790	1.021
UDP/IP	723654.1	560687.6	571107.3	0.775	0.789	1.019
average	751407.8	583546.5	595359.0	0.776	0.792	1.020

Unit: cycles

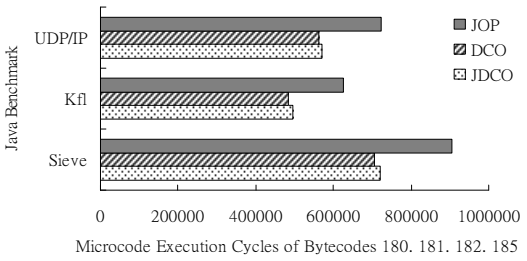


Fig. 42. Microcode Execution Cycles of Bytecodes 180. 181. 182. 185

iv. External Memory Access Times

In addition to the microcode execution cycles, there is another important factor of power consumption. That is the external memory access times. Like the microcode execution cycles, the less external memory accesses, the more power saving.

The calculation is similar to the microcode execution cycles. We also list the external memory access times of each bytecode and separate them by the number occurrences as in Table VI, in which we calculate the sum of memory read and memory write. The times 3 or 5 is based on the number of address we modified because an address is of 32 bits. For example, if the address of modified bytecode is "42 1 2 181", we should modify the next address because it contains the operand of bytecode 181. For calculating, we use the average 4. Use this information and the total execution times of the modified bytecodes of each benchmark separating by the number occurrences in Table V, we can calculate the external memory access times by the following formulation:

$$\sum_{\#\ occurrence} (T * (\sum_{bytecode} (times * P)))$$

TABLE VII. External Memory Access Times of Each Bytecode

# occurences / bytecodes	For JDCO			For DCO	
	first	second	third and later	first	second and later
getfield	2	2+3/5	1	2+3/5	1
putfield	2	2+3/5	1	2+3/5	1
invokevirtual	4	4+3/5	3	4+3/5	3
invokeinterface	6	6+3/5	5	6+3/5	5

Unit: times

The experiment results are listed in 0 and showed in Fig. 43. Our JDCO system has 22.2% less external memory access times of the modified bytecodes, so for the system of total bytecodes, we have 11.1 % less external memory access. If comparing to DCO system, we still have a little more external memory access times. The reason is as we mentioned in the previous subsection.

TABLE VIII. External Memory Access Times of Bytecodes 180. 181. 182. 185

benchmark \ system	JOP	DCO	JDCO	DCO/JOP	JDCO/JOP	JDCO/DCO
Sieve	53224.3	41666.3	42130.3	0.783	0.792	1.011
Kfl	36912.2	28043.1	28532.1	0.760	0.773	1.017
UDP/IP	42616.6	32569.6	32841.6	0.764	0.771	1.008
average	44251.0	34093.0	34501.3	0.769	0.778	1.012

Unit: times

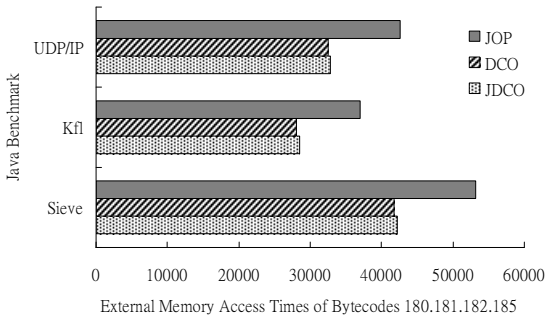


Fig. 43. External Memory Access Times of Bytecodes 180. 181. 182. 185

5. Conclusion and Future Work

In this thesis, we propose a dynamic code optimization scheme which can

significantly improve the efficiency of Java program execution and cut down on the power consumption for a hardware/software co-designed Java VM. As we mentioned above, typical dynamic code optimization can save method lookup and constant pool searching time using the runtime information at the first time a bytecode is executed. However, in the embedded system such as DVB-MHP terminal, code modification and saving of the runtime information is very expensive due to the overhead of external memory accesses. By analyzing the execution frequency of Java code segment, we can dynamically decide if the dynamic code optimization is needed. This JDCO architecture can make Java execution more efficient and more suitable to the DVB-MHP terminal due to less power consumption.

We implement this architecture based on the Java Optimized Processor (JOP) and verified the design on a Xilinx Spartan-3 development board. It is shown by our experimental results that the proposed dynamic code optimization scheme for Java VM hardware/software co-design has 13.8% average speedup of execution time. Furthermore, the power consumption of the proposed system can be reduced due to 10.4% less microcode execution cycles and 11.1% less external memory accesses compared to the original system.

Future researches can improve on recognizing the pattern of the relationship between frequency code and non-frequency code (maybe can learn from HotSpot). By doing this, the overhead of needless first time searching as describe in subsection 4.3 ii can be avoided. It may give a great improvement in power consumption. And then if other target systems do not use JCC, the format of other bytecodes will be designed and implemented. In the future, the proposed system will be port to other more powerful developing board, such as the Xilinx ML 310. It can be expected to have better performance for the Java VM.

6. REFERENCES

[1] David Ungar and David Patternson, "Berkeley Smalltalk: Who Knows Where the Time Goes? ," In *Smalltalk-80: Bits of History, Words of advice*, Addison-Wesley, Reading, MA, 1983.

[2] Peter Deutsch and Alan M. Schiffman, "Efficient implementation of the Smalltalk-80 system," In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 297-302, ACM Press, January 1984.

[3] Urs Hölzle, Craig Chambers, and David Ungar, "Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches," In *Proceeding America, editor, Proceedings ECOOP '91,* LNCS 512, pages 21–38, Geneva, Switzerland, July 15-19 1991. Springer-Verlag.

[4] Ming Chan, Lin, "Runtime Profiling and Analysis of Java Program Execution," Master Thesis, Computer Science and Information Engineering, National Chiao-Tung University, Taiwan, June 1998.

[5] Anders Dellian, "Dynamic Code Optimization for Statically Typed OO Languages in An Integrated Incremental

System," In *Proceeding of NWPER'94, Nordic Workshop on Programming Environment Research*, Lund, Sweden, June 1994.

[6] David Ungar, "The Design and Evaluation of a High Performance Smalltalk System," In *MIT Press*, Cambridge, MA, 1986.

[7] Sun Microsystems Inc., "The Java Virtual Machine Specification," [Online] Available: http://java.sun.com.

[8] Sun Microsystems Inc, "The K Virtual Machine White Paper," [Online] Available: http://java.sun.com , June 1999.

[9] Jon Meyer and Troy Downing, "Java Virtual Machine," published by *O'REILLY*, 2000.

[10] DVB project, "Digital Video Broadcasting (DVB): Multimedia Home Platform (MHP) Specification 1.1.1," [Online] Available: http://www.mhp.org, Jun 2003.

[11] "The Unicode Standard: Worldwide Character Encoding, " [Online] Available: http://unicode.org

[12] "UCS Transformation Format 8 (UTF-8), " [Online] Available: http://www.stonehand.com/unicode/standard/wg2n1036.html

[13] Eric Armstrong, "HotSpot : A New Breed of Virtual Machine," [Online] Available: http://www.javaworld.com/jw-03-1998/jw-03-hotspot.html, 1998.

[14] Martin Schoberl, "JOP: A Java Optimized Processor for Embedded Real-Time Systems", Vienna, Jan 2005.

[15] J.Michael O'Connor and Marc Tremblay, "picoJava-I: The Java Virtual Machine in Hardware," In *IEEE Micro*, 17(2):45–53, 1997.

[16] ARM, "ARM Jazelle Technology," [Online] Available: http://www.arm.com/products/solutions/Jazelle.html

[17] M. Schoeberl, "Restricitons of Java for Embedded Real-Time Systems, " In *Proceeding of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, ISORC 2004, Austria, Vienna, May 2004.

[18] P. Puschner and A.J. Wellings, "A Profile for High Integrity Real-Time Java Programs," In *Proceeding of the 4th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, 2001

[19] A.Burns and B. Dibbing, "The Ravenscar Tasking Profile for High Integrity Real-Time Programs," In *Proceeding of the 1998 annual ACM SIGAda intertional conference on Ada*, pp.1-6, Washington, USA, 2002.

[20] J.Kwon, A. Wellings and S. King, "Ravenscar-Java: a High Integrity Profile for Real-Time Java," In *Proceeding of the 2002 joint ACM-ISCOPE conference on Java Grande*, pp. 131-140, Seattle,Washington, USA, 2002.

[21] Xilinx, "Spartan-3 Starter Kit Board User Guide", Jul 2004.

[22] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson, "The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address

Real-Life Challenges," In *Proceeding of the First Annual IEEE/ACM International Symposium on Code Generation and Optimization*, San Francisco, California, 27-29 March 2003.

[23] Martin Schoeberl, "Using a Java Optimized Processor in a Real World Application," In *Proceeding of the First Workshop on Intelligent Solutions in Embedded Systems (WISES 2003)*, pages 165–176, Austria, Vienna, June 2003.