行政院國家科學委員會專題研究計畫 成果報告

Heterogeneous multi-core platform--應用於嵌入式異質多核心平台之爪哇虛擬機器(嵌入式軟體關鍵技術開發分項)

研究成果報告(精簡版)

計 畫 類 別 ：整合型
計 畫 編 號 ：NSC 97-2218-E-009-029-
執 行 期 間 ：97 年 08 月 01 日至 98 年 10 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：楊武
共 同 主 持 人：雍忠、單智君

處 理 方 式 ：本計畫可公開查詢

中 華 民 國 99 年 03 月 09 日

行政院國家科學委員會補助專題研究計畫 ■ 成 果 報 告 □期中進度報告

記 Heterogeneous multi-core platform－應用於嵌入式異質多核心平台之爪哇虛擬機器(嵌入式軟體關鍵技術開發分項)

計畫類別：■ 個別型計畫　　□ 整合型計畫
計畫編號：NSC 97－2218－E－009－029－
執行期間： 97 年 8 月 1 日至 98 年 7 月 31 日

計畫主持人：楊武 教授

計畫參與人員：陳裕生、沈柏曄、呂禮君、黃帥維、黃致超、孫信慶

成果報告類型(依經費核定清單規定繳交)：□精簡報告 ■完整報告

本成果報告包括以下應繳交之附件：
□赴國外出差或研習心得報告一份
□赴大陸地區出差或研習心得報告一份
□出席國際學術會議心得報告及發表之論文各一份
□國際合作研究計畫國外研究報告書一份

處理方式：除產學合作研究計畫、提升產業技術及人才培育研究計畫、列管計畫及下列情形者外，得立即公開查詢
■涉及專利或其他智慧財產權，□一年■二年後可公開查詢

執行單位：國立交通大學資訊工程學系(所)

中　華　民　國　98　年　7　月　31　日

目錄

一、中文摘要

今日無論在小型裝置、個人電腦，甚至大型工作站都可以發現 Java 的蹤跡，Java 為各種應用的需求提供了良好的解決方案，也已經有許多廠商開發了大量 Java 程式，尤其在小型裝置上更是普遍。相對於傳統程式而言，Java 程式能夠僅以一次撰寫及編譯，而得以在不同的平台上執行。這是因為 Java 程式在編譯時並不直接轉換成與平台相關的機械碼，而是先轉換成中介碼 bytecode，再仰賴位於各種不同平台上的 Java Virtual Machine(JVM)來翻譯執行。

然而，若僅以 interpreter 來實作 JVM 的執行引擎，勢必無法獲得良好的效率，因此 Just-In-Time (JIT)編譯技術因應而生。JIT 編譯器會在 Java 程式執行時期將其對應的 bytecode 先轉換成一個中介形式 IR(Intermediate Representation)，並在此階段先進行一些與平台無關的初步優化(例如 copy propagation、constant folding 等)；然後 native code generator 會再根據 IR 與平台架構選擇合適指令、配置暫存器、…等，來產生與平台相依，可在處理器上直接執行的原生程式碼。

JIT 編譯技術不會破壞 Java 具有的可攜性與安全性，但卻能夠帶來極大的效能提升。現今 JIT 編譯技術已經被廣泛地應用到伺服器、個人電腦，甚至嵌入式系統中，來克服 Java 為人詬病的效率問題。

二、報告內容

(一) Introduction

此成果報告將展示今年度計畫的成果也就是在 Andes 處理器架構上移植適合的 JVM 之詳細過程。下面的內容(二)敘述此計畫的目的，內容(三)講述 CVM JIT 背景知識，內容(四)講述移植方法與困難解決的過程，內容(五)呈現一份對於移植成果的測試報告。

(二) Research Objectives

本計劃之主要目的是為 Andes 處理器架構移植嵌入式系統上常見的 JVM 。此外，亦會探討針對 Andes 處理器架構設計優化技術的可能性。最終目標就是為 Andes 處理器提供高效能的嵌入式系統用 JVM，使其成為適合的嵌入式 Java 應用平台。

我們以過去實作記憶體受限 JIT 編譯器的經驗，將中型嵌入式系統上常用的 CVM[1] 移植到 Andes 處理器的 Linux 平台上。主要工作包括：

1. Andes ISA 及處理機微架構的研讀
2. CVM 程式以及程式碼架構的研讀
3. 移植 CVM 到 Andes 架構
4. 針對 Andes 架構研究適合的優化技術

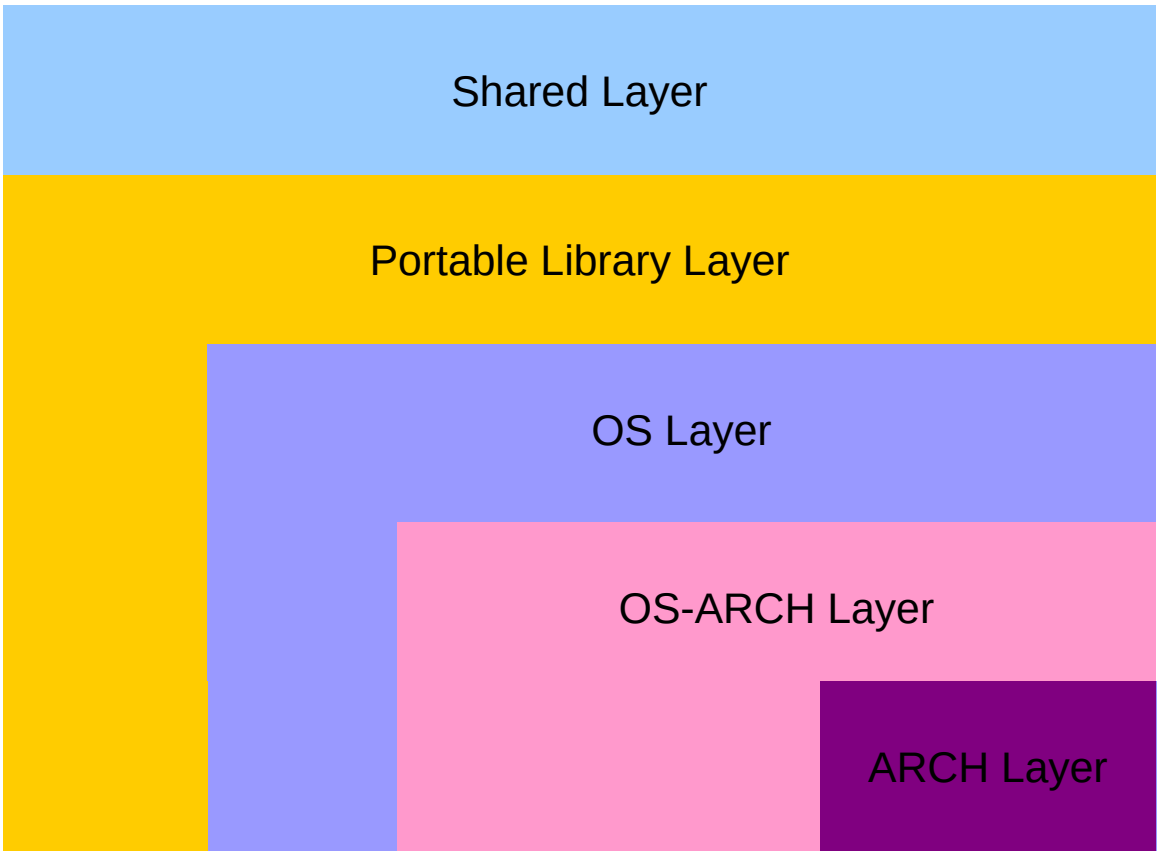
我們採用 Embedded CaffineMark、CLDC evaluation kit 及 EEMBC 提供的 Grinder benchmark

suite 作為評估移植結果的驗證工具。同時也利用這些應用程式作為調整 JIT 編譯器效能以及設計優化技術的依據。

(三) Background

3.1 CVM 架構

CVM 是一個架構設計良好的嵌入式系統用 JVM。其採用分層的實作方式，將不同平台相依性的程式碼做切割，使得移植時的工作量減少到最低[6]。如圖一所示，CVM 中大部分的程式碼屬於與平台無關的 Shared Layer，或相依於常見的 portable library，在移植時通常不需多做考慮；CVM 也於 OS Layer 實作了作業系統相關的程式碼，支援了常見的作業系統(e.g. Linux, windows…)。



圖一. CVM 實作架構圖

在移植時需要投注心力的部分在於與底層平台相關的部分，也就是在圖一中最低的兩層。我們的目標在於移植 CVM 到 Andes 平台上，與底層平台相關的部分才是我們研讀的重點。這部份包含的程式碼不多(與整個 CVM 相較之下)，主要是一些平台相關的定義、原生呼叫的程式碼以及最重要的 JIT compiler。

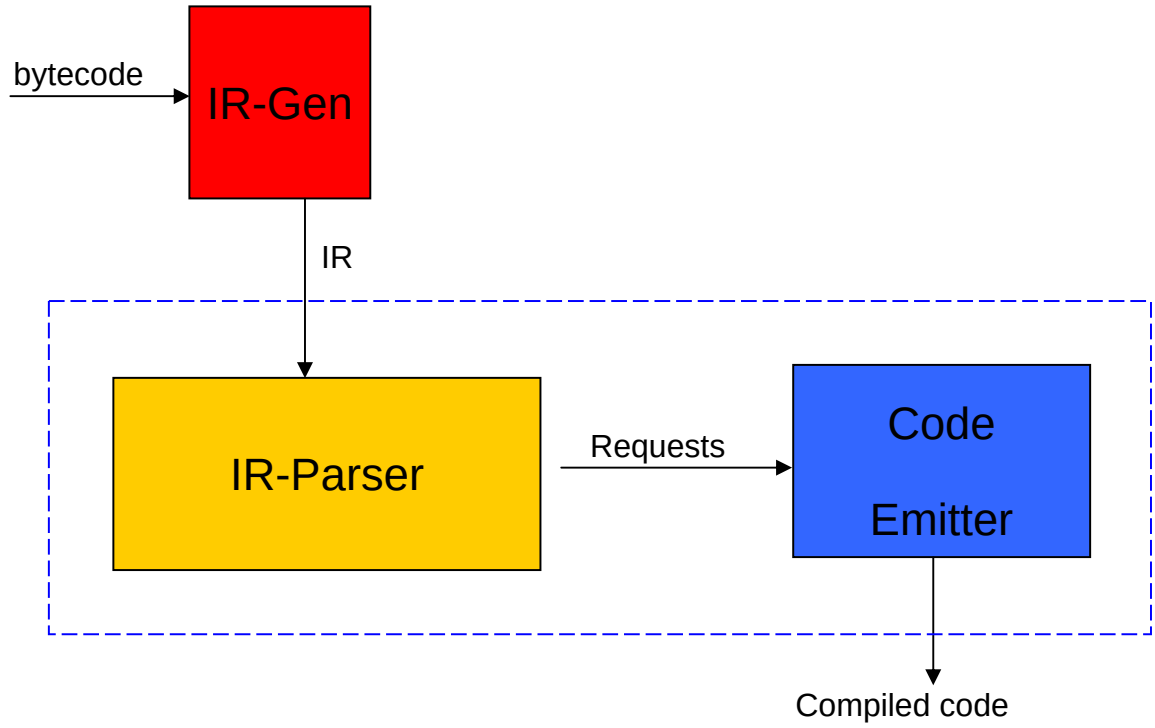
3.2 CVM JIT compiler 架構

CVM 的 JIT compiler 在設計上十分良好，如圖二所示，主要包含兩個部份：與平台無關的 front-end 及與平台相關的 back-end[7]。

Front-end 的工作主要為將函式中所有的 bytecode 轉換成 DFT(data flow tree)式的 IR，

並做一些簡單的最佳化(如：Constant folding、constant propagation、copy propagation 等)。

Back-end 又可以分成兩個部份： IR pattern matcher 及對應的 instruction emitter。IR pattern matcher 是由工具根據事先定義的規則(JCS rules)所產生，這些規則定義了各種 IR 的 pattern 及其在從 DFT 中被選擇時所必須執行的動作(包含呼叫對應的 instruction emitter)。Instruction emitter 則是一組事先定義的函式，負責產生對應的原生指令。



圖二. CVM JIT Compiler 架構圖

(四) Research Methods

在計畫的先期，我們主要的工作以研讀相關資料及文獻為主。包含 Andes ISA[2]及應用程式介面[3]、CVM 架構及程式碼[4][5][6][7]。後期則著重於設計並改寫原本 CVM 中的程式碼，使其可以在 Andes 平台上順利執行。以下分段描述移植工作相關的背景知識，及移植工作的簡述。

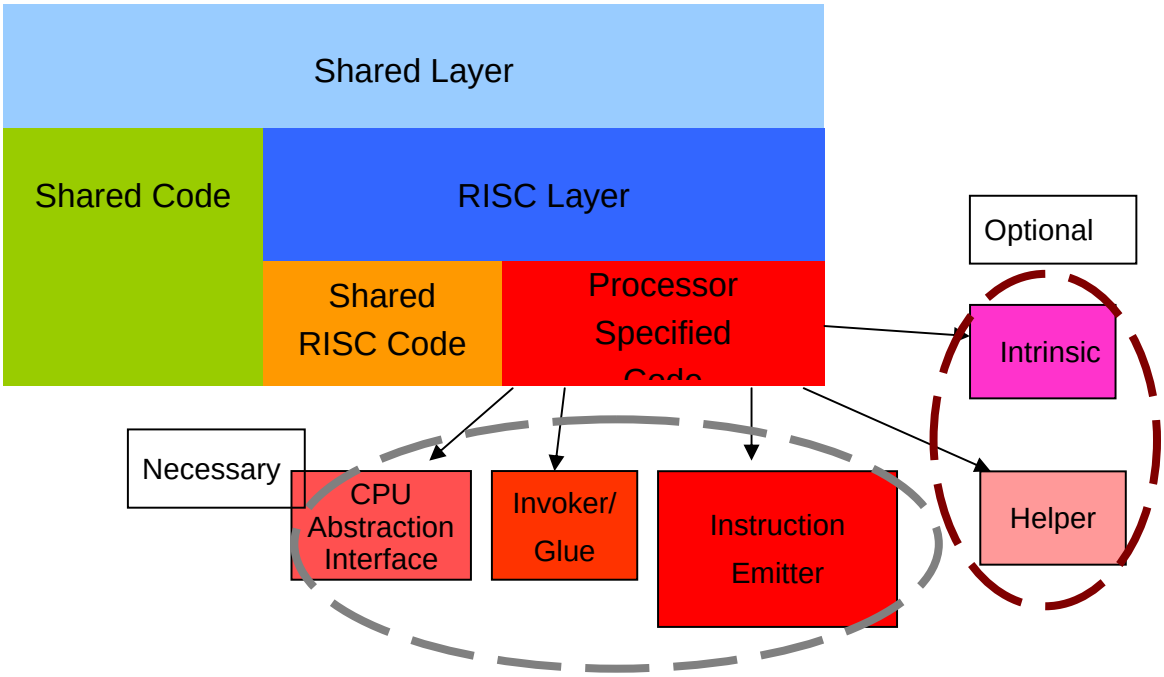
4.1 CVM 移植工作

由於 CVM 的分層架構設計良好，移植到不同處理器平台所需要修改或實作的程式碼並不多，主要可以分為兩個階段：直譯器及 JIT 編譯器[6]。

直譯器與處理器相關的程式碼不多，主要包含原生函式的呼叫者及平台相關的設定。原生函式的呼叫者需要使用組合語言來編寫，其主要工作依序為：解析原生函式的 signature、將存放於 java stack 的參數移動到 c stack 或暫存器中、將控制權轉移到原生函式中及將原生函式的回傳值存回 java stack。平台相關的設定包含了位元組排列的方式、浮點運算的相容性（是否需要額外的檢查或設定程式碼）、同步方式等。

值得一提的是，因為 Andes 平台的 ABI[3]規定必須使用 register 0 傳遞儲存空間的指標供 callee 來存放 64 位元回傳值，因此會造成與回傳 32 位元值的 callee 不同的參數存放順序，這一點在實作原生函式呼叫者時必須特別注意；另外一個比較需要注意的地方同樣因為 ABI 中的規定：在傳遞 64 位元參數時，必須對齊暫存器的編號於偶數。除了 ABI 之外，目前我們並未實作 CAS(compare-and-set)為基礎的同步機制[6][7]，而改用 pthread 中的 mutex。主要的原因是為了加速移植的過程，並避免不完整的實作造成測試上的困難，這部份預期在下個年度完成。

移植 JIT 編譯器的工作較直譯器複雜，雖然也是分層式的設計(參考圖三)，但其中包含較多的組件必須實作以符合平台的特性。為了加速移植過程，我們選擇與 Andes 架構較為相似的 MIPS 版本 CVM 做為基礎來修改。主要修改部分為圖三中紅色的部份(Processor specified code)中標示為必需的部份。可以分成幾個比較重要的部份：



圖三. CVM JIT Compiler 實作架構圖

4.1.1 CPU Abstraction Interface

這部份包含了描述處理器的特色(如 delay slot、post increment load 等)、可用暫存器的數目及其用途(參數、Java stack pointer 等)、及各種功能的選擇(如 null pointer 的處理方式)[6]。處裡器特色的設定與處理器本身相關，直接根據目標平台定義即可。暫存器的設定是最複雜的部份，除了必須的特定用途外(如 JSP 及 JFP 等)，許多設定都是由移植者自行決定的，這部份我們將配合下年度的計畫，調校效能時詳加考慮。另外，為了避免偵錯的困難，我們決定暫時將許多選擇性的設定/優化關閉，這部份也預期下年度會實作完成。

4.1.2 Instruction Emitter

CVM 定義了一系列的 JIT-compiler-related functions 稱為 Emitter APIs[6]，負責產生特

定功能對應的原生程式碼到記憶體(Code Cache)中。必須修改 MIPS 版本中與平台相關的 emitter APIs 使之能產生 Andes 平台的原生程式碼。相關的修改大致可分成兩類：

a. 指令格式的不同：

不同的 ISA 對於指令的編碼格式不相同，且指令的定義亦不相同，因此必須改變暫存器及立即值等的偏移量，使之對應到正確的位置，並改變操作的代碼使之正確運作。

b. 平台架構的差異：

根據不同平台所有的架構不同，會有不同的特殊設計，例如：MIPS 架構中的 Zero 暫存器專門用來存取立即值 0，而 Andes 架構下並沒有此功能。因此移植時，必須避開專用的特殊架構，修改成正確的運行方式。

移植的過程是由 MIPS 架構作為來源，移向 Andes 平台，舉例如下：

```
/* Purpose: Emits instructions to do the specified 32 bit unary ALU operation. */
void
CVMCPUemitUnaryALU(CVMJITCompilationContext *con, int opcode,
                                    int destRegID, int srcRegID, CVMBool setcc)
{
    switch (opcode) {
    case CVMCPU_NEG_OPCODE:
        CVMCPUemitBinaryALURegister(con, CVMCPU_SUB_OPCODE,
                destRegID, CVMMIPS_zero, srcRegID, setcc);
        break;
    default:
        CVMassert(CVM_FALSE);
    } CVMJITdumpCodegenComments(con);
}
```

此函式的目的為產生 negative 操作的原生程式碼。原來的版本產生的是 Zero 暫存器減去來源暫存器的結果存到目的暫存器的程式碼。因為 Andes Architecture 並沒有所謂的 Zero 暫存器，因此必須將該函式改寫如下：

```
/* Purpose: Emits instructions to do the specified 32 bit unary ALU operation. */
void
CVMCPUemitUnaryALU(CVMJITCompilationContext *con, int opcode,
                   int destRegID, int srcRegID, CVMBool setcc)
{
    CVMassert(!setcc);
    switch (opcode) {
    case CVMCPU_NEG_OPCODE:
        emitInstruction(con, CVMNDS_SUBRI_OPCODE    |
                             destRegID << NDS_RT_SHIFT |
                             srcRegID << NDS_RA_SHIFT);
        CVMtraceJITCodegenExec({
            printPC(con);
            CVMconsolePrintf("subri %s, %s, 0",
                             regName(destRegID),
                             regName(srcRegID));
        });
        break;
    default:
        CVMassert(CVM_FALSE);
    }
    CVMJITdumpCodegenComments(con);
}
```

我們將其改為產生以立即值 0 減去來源暫存器並將結果放到目的暫存器的指令，達到同樣的效果。

除此之外，因為兩個平台的架構不同，還有相當多的問題必須解決。主要的議題如下：

a. Calling convention[3]：

因為回傳 64 位元值的方式不同於傳統的方式，在呼叫某些 helper 時必須特別調整，使用 compiled frame 上的暫存空間來傳遞回傳值。

b. Endianness：

在 Andes 架構下指令永遠為 big endian 的排列順序。由於 JIT 編譯器概念上是把指令當作資料來操作，當設定 CVM 的資料 endian 為 little 時就會發生問題。因為在此時產生的指令是以 little endian 的順序存放於記憶體中，但處理器卻會假定這些指令的排列順序為 big endian。所以在產生指令到 code buffer 之前必須多一個簡單的步驟將其調整成正確的 endianness。

c. Delay slot：

Andes 架構並沒有 Delay slot 的設計，因此必須移去原本 MIPS 程式碼中關於 delay slot 的部份。這部分主要包含：產生放置於 delay slot 的指令、調整相關指令長度的計算方

式。

所有必須支援的 emitter API 均已完成移植的工作。由於程式直接從 MIPS 版本修改過來，因此我們預期會有部分的程式必須針對 Andes 平台做優化的工作，才能有比較好的效能。目前我們只完成 32 位元指令集的支援，未來將會繼續實作 16 位元指令集的支援。另外，CVM 中沒有支援 Andes 特定指令(如 load/store multiple words 等)及特定的 addressing mode，這部份我們將配合下年度的計畫，在調校效能及實作 16 位元指令集的支援時加以評估並以適合的方式實作。

4.1.3 Invokers/Glue code

在移植 CVM 的 JIT 編譯器裡和處理器相關的程式碼時，有幾乎一半的工作是編寫 assembly code。這些 assembly code 負責處理一些使用 C 語言無法描述的工作(如暫存器的設定)以及負責部分的優化工作[6][7]。這些 assembly code 主要可分為兩個項目:

a. Invoker

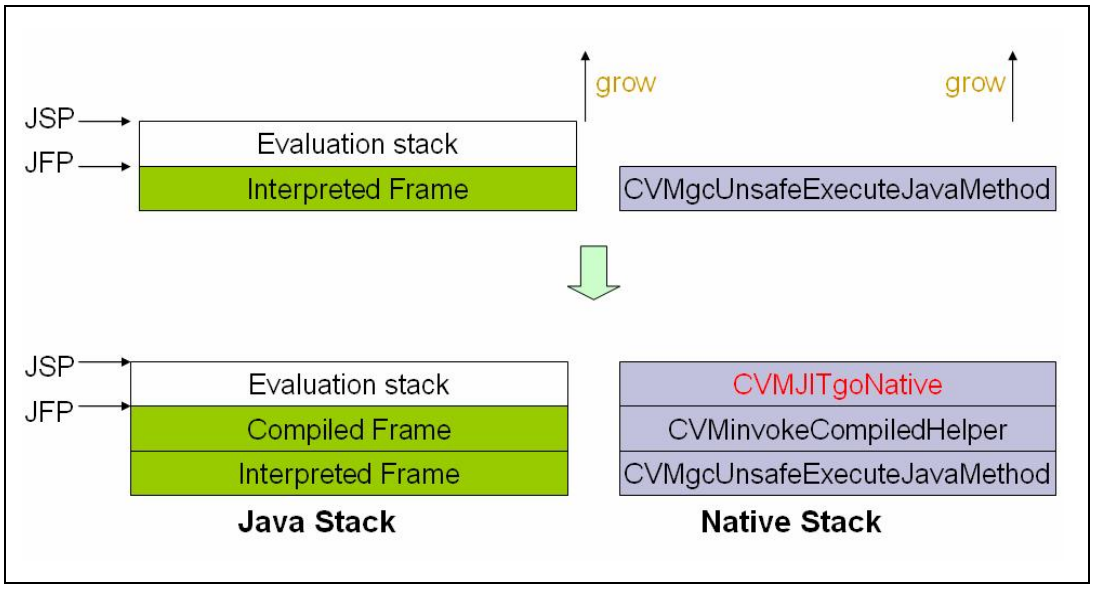
這個項目的程式碼負責處理從 compiled code 到 interpreter/原生程式碼(CNI/JNI)以及反方向轉移時必需的設定(包含堆疊及暫存器等)[6][7]。以下介紹相關的兩個檔案中的內容。

- jit_cpu.s

此檔案內的程式碼負責設定 interpreter 和 compiled code 間轉換執行權時的設定。當 interpreter 欲切換至 compiled code 時，VM 會呼叫 CVMJITgoNative。CVMJITgoNative 是一段必須由目的平台提供的 assembler glue code，主要是用來設定供 compiled code 使用的 native stack frame，而 native stack frame 必須保留記憶體供 CVMCCExecEnv 結構使用，在移植程式時，我們必須針對 Andes calling convention 的規定，將 callee-saved 暫存器(s0~s8)依序存至堆疊中。同時由於我們暫不使用 trap-based 和 patch-based 的 GC check[7]，所以必須將 explicit GC check 會使用的 CVMglobals 資料結構的記憶體位址載入至 GLOBAL_REG 中，才能正確執行 GC 動作。

另外一個重要的 assembly function 是 CVMJITexitNative，其用途是將 CVMJITgoNative 所設定的 stack frame 做展開的動作，使其回到 interpreter frame。這段 glue code 通常是使用在當 compiled code 發生 Java exceptions 的時候。

下圖(圖四)是由 interpreter 中呼叫 compiled method 的範例。圖中顯示 java stack 和 native stack 各自變更的情況。當 Interpreter 呼叫 compiled code 時，CVMCompiledFrame 會被放入至 Java stack 中，而 native stack 記錄著由 CVMinvokeCompiledHelper 呼叫 CVMJITgoNative 的 frame。



圖四. Calling from an interpreted method to a compiled method

目前此部分相關的程式碼已經移植完成，但仍有少部份的程式碼尚未經過測試，因為這些程式碼負責的工作，主要是將選擇性優化技術使用到的暫存器初始化。這部分將待明年度實作相關的優化技術時再驗證。

- ccminvokers_cpu.S

CCM invoker扮演著compiled code與其他不同形式程式碼間轉換的中介角色，例如用來呼叫 JNI/CNI methods 或是透過 interpreter 去呼叫 interpreted method[6][7]。

當從 compiled code 中呼叫 JNI method 時，CVMCCMinvokeJNIMethod 會先被呼叫，這個 assembly function 內主要包含了環境的設定、stack frame 的處理以及參數的傳遞等工作，然後會透過CVMinvokeJNIHelper來呼叫JNI method。而當從 compiled code 中呼叫 CNI method 時，則 CVMCCMinvokeCNIMethod 會被呼叫，此 assembly function 內主要包含了環境的設定及參數的傳遞等工作，並會直接呼叫 CNI method。

至於透過 interpreter 去呼叫 interpreted method 主要是透過 CVMCCMletInterpreterDoInvoke 來達成。如下圖五所示，在從 compiled code 中呼叫 interpreted method 時，interpreted frame 會被 push 到 Java stack 中，並透過 NDSexitNative0 將 native stack 內所保留的 frame (經由呼叫 CVMJITgoNative 產生) 做 unwinding 的動作後回到 interpreter，然後執行 interpreted method。

在 ccminvokers_cpu.S 這個檔案中，有數個與執行緒同步相關的 invoker，目前我們尚未移植優化過的版本，而是交由直譯器來處理。

圖五.Calling from a compiled method to an interpreted method

b. Glue code

在 CVM 的 JIT 編譯器設計裡，有部分較為複雜的工作(如 object allocation)是由 C 程式碼撰寫的 helper 完成的，當 compiled code 需要執行這些功能時就會直接呼叫或透過 glue code 來呼叫 helper。

這部份的程式碼負責執行部分的 helper 工作(C 語言無法描述的工作或者為了優化)，並在必要時轉移執行到 helper 中完成要求的工作。這部份可依採用 assembler 實作 glue code 的原因分成三類[7]：

i. 由於需要透過 Helper 完成的 work，可區分成具有 fast path 或 slow path 兩類。Fast path 的 work 較容易實作，slow path 的 work 則較為複雜且不易實作，為了效率起見，我們將 fast path 透過 ASM helper 的方式實作，而 slow path 則採用 C helper 方式實作。

Example : CVMCCMruntimeCheckCastGlue
使用 Checkcast 檢查物件是否為預定資料類型時，首先會透過 CVMCCMruntimeCheckCastGlue 檢查 object class block 的 address 是否為 null，若不為 null 將 guess class block 載入後，再呼叫 C helper 進行 check cast 的動作。

ii. 用於呼叫 C helper 之前，重新配置變數的 ASM helper。

Example : CVMCCMruntimeThrowNullPointerExceptionGlue
當發生 Null pointer exception 時，系統會先呼叫 CVMCCMruntimeThrowNullPointerExceptionGlue 對 compiled frames 進行修正的動作，接著將 exception message(0)存放至$a3 後，呼叫處理 java exceptions 的 C helper function。

iii. 用於改寫呼叫此 ASM helper 的 compiled code。

Example : CVMCCMruntimeRunClassInitializerGlue
Class 初始化時，先透過 CVMCCMruntimeRunClassInitializerGlue 設定好相關的變數後，再呼叫 CVM 中對應的 C helper function，進行 class 初始化動作。待 class 初始化動作結束後，將 NOP 指令 patch 至原本 compiled method 中呼叫此 helper 指令的位址，避免再次執行初始化的動作。

這部分主要包含了兩個檔案：ccmglue_cpu.S 和 ccmallocators_cpu.S。在第一個檔案裡，除了和 monitor 相關的函式尚未經過完整的測試外(因此尚未啟用，而直接轉呼叫 C helper)，其他部份都已經完成實作及測試的工作。至於第二個檔案裡的所有函式，雖已經實作完成，但同樣尚未經過完整的測試。

大部分的 glue code/invoker 已經移植完成，只有少部份因為複雜度較高且不容易除錯(如同步相關的 invoker)尚未移植優化的版本。這些函式目前實作的方式是直接轉呼叫 interpreter 或相關的 c helper，由實驗結果得知，這樣的實作方式會影響系統的執行效能，因此我們計劃在明年度完成這些尚未優化的 glue code/invoker 的移植工作來提升效能，並做完整的測試來確保正確性。

4.2 測試及驗證工作

計畫前期我們缺乏一個完整的開發環境，沒辦法編譯以及執行程式碼。因此前期使用人工驗證的方式，直接檢查 emitter 產生出來的程式是否如預期；glue code 的部份則是人工複查。同時也累積了不少測試程式，在開發過程為了測試及驗證的需要，我們整理了一套共 33 個功能性測試程式套件。此套件在除錯過程中幫了不少忙，減少許多除錯的時間。

當我們取得完整的模擬環境後，除了直接執行我們設計的測試套件外，我們另外以 CVM 內建的測試程式作輔助，確認移植的工作的完成度。此內建程式包含了 411 個小測試。

由於 CVM 的複雜度頗高，即使通過了上述的測試，也不代表我們的工作完全相容於規格，因此我們希望能在未來取得昇陽的 TCK 來測試我們的移植工作，確保完成性以及相容性。

4.3 遭遇之困難及解決方式

在開發移植的過程中，我們遭遇許多困難及問題，也一一克服。這些問題大致上可以分成以下幾類：

1. 開發環境相關的問題
   - 開發早期動態聯結的功能並無法正常使用，因此我們在編譯 CVM 時設法使其與系統函式庫靜態連結。但此方式可能會帶來一些問題，因此後來工具較成熟且經過測試後，我們才調整回原來的編譯方式。
   - Sid 模擬器曾經在執行 cctl 指令時發生問題，經過回報後已經修正。
   - 使用 pthread 產生新的 thread 時，可能會使此執行緒的初始 stack pointer 不對齊於 8 bytes，因此會造成 JNI 函式呼叫者處理 64 位元的參數時發生錯誤。此問題經回報後已修正。

2. Andes 架構與 ABI 相關的問題

- Andes 早期的 ABI 規定，在呼叫 PIC 函式時，必須以暫存器 ta 作為 jump 暫存器才能正確執行。我們經過檢視程式碼後已修正此問題
- 在移植過程中，我們只專注在與平台相關的程式碼，但由於 CVM 內部的假設與 Andes 的 ABI 不同，因此在處理 64 位元回傳值時，我們不得已只好修改一部分與平台無關的程式碼。在未來 ABI 修改之後，這部份會調整回來。
- 位元組排列方式造成的問題。因為在 Andes 架構下指令的位元組排列永遠是由小到大，資料則可以自由設定。但 JIT 編譯器其實是將指令當做資料來處理，當儲存產生出來的指令到記憶體時必須確認其排列方式為由小到大。這部份的修正可能會造成效能的問題，我們會在未來評估後做改進。

3. 與 MIPS 架構的差異造成的問題

- MIPS 架構有專用的 Zero 暫存器，在需要使用 0 立即值時，在 Andes 平台上必需使用額外的 move immediate 的指令來載入 0。
- Andes 平台上並不需要考慮 delay slot 的處理，MIPS 則需要。
- Branch 及 load/store 等指令的範圍及計算方式與 MIPS 不同，必須小心設定。
- 在 MIPS 平台上，Load/store immediate 指令的 offset 在編碼時不需做對齊的動作(捨棄最小的數個位元)，在 Andes 上則是必須的。雖然這樣可以讓 offset 的範圍增加，卻必須小心處理。
- 在 Andes 架構裡，有兩類的 conditional branch 指令，一類可以與 0 比較大小及是否相等、另一類則只能比較兩個暫存器是否相等。此兩類的 immediate 範圍也不相等。另外，只有兩個 conditional branch 指令支援 link 的功能。這些都會使得 conditional branch 指令處理上較為複雜。
- MIPS 架構的 load/store 指令不支援以暫存器當作 index 的 addressing mode，這是 Andes 平台可以加以利用的優點。
- Andes 架構目前不支援除法及浮點運算等指令，在執行相關的 benchmark 會有極差的效能。
- Andes 架構支援了相當多 MIPS 架構不支援的指令，如乘加、conditional move、data prefetching、load/store multiple、sign/zero extension 等指令，雖然可以增加效能，卻會增加實作的困難度，在目前的實作中暫不使用這些特別的指令。
- 其他的差異，如 MIPS 只有一個長乘法用的暫存器，但 Andes 上有兩個且還可以用來作乘加的操作。

目前還存在一個尚未解決的問題：在執行多緒的程式時有可能會出現死結的狀況。目前還在追蹤測試，因為模擬器上不會出現這個問題，所以我們猜測可能是函式庫或者作業系統的問題。

(五) Results

此章節包含了目前執行效能的評估比較以及討論。主要項目包括 JIT compiler 帶來的效益，以及與現有完整的移植做比較。由於 ARM 是目前 CVM 支援最完整的平台，因此我們採用 ARM 的平台做為比較對象。軟硬體設定如下表，要注意的是 ARM 平台的 CVM 設定和 NDS32 不同，主要是因為我們目前尚未實作完所有的選擇性優化。在後面的實驗結果我們會看到，因為這些未完成的優化，可能會造成效能上不小的差距。

處理器	N12	FA526
指令集架構	NDS32	ARMV4
執行頻率	150 MHZ	150 MHZ
CVM 設定	Workable settings	All possible settings

表一. 軟硬體設定

我們採用三套中小型的 benchmark 來評估執行效能，其中一個多緒的程式因為 4.3 最後提到的問題因此不採計其結果。使用這些中小型的 benchmark 主要是因為其較符合 CVM 的定位，且經常被拿來評估中小型的 VM。以下是此三套程式的簡介：

Name	**Brief Description**
Sieve	The classic sieve of Eratosthenes finds prime numbers.
Loop	The loop test uses sorting and sequence generation as to measuree compiler optimization of loops.
Logic	Tests the speed with which the virtual machine executes decision-making instructions.
Method	The Method test executes recursive functional calls to see how well the VM handles method calls.
String	String comparison and concatenation.

表二. Embedded CaffineMark 3.0 簡介

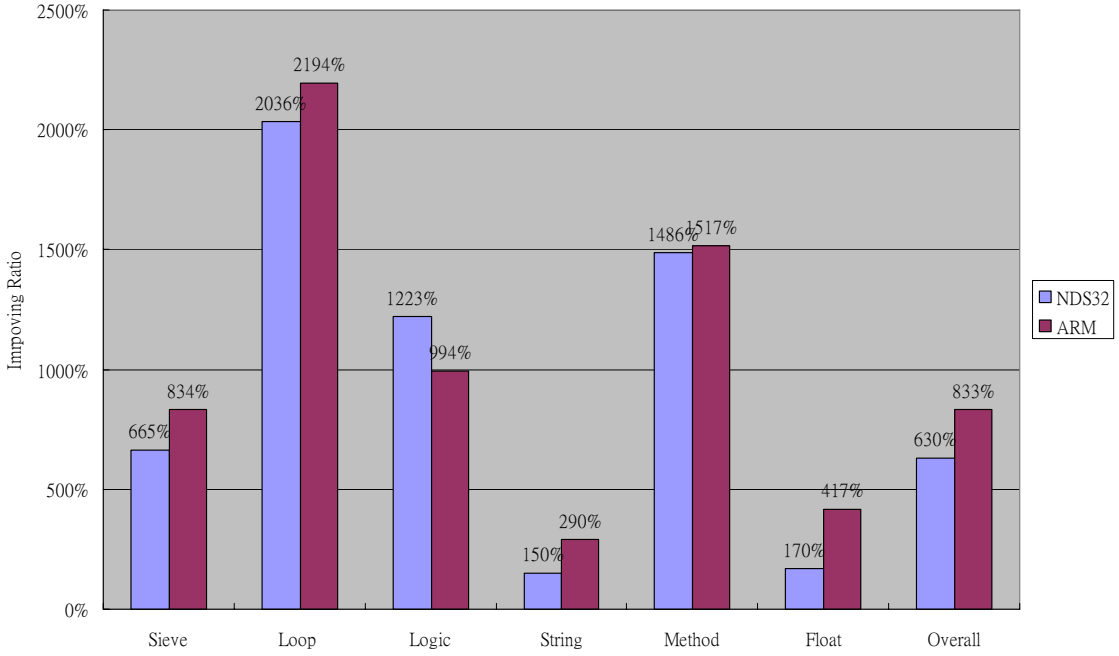
Name	Brief Description
Richards	Richards is a benchmark that simulates the task dispatcher in the kernel of an operating system.
DeltaBlue	DeltaBlue solves one-way constraint systems.
Queens	A solver of the n-queens problem. It is a classical problem used to illustrate several techniques such as general search and backtracking.
Image Processing	The Image Processing benchmark reads an image file and performs various transformations on it, such as Sobel, threshold, 3x3 convolver, and so forth.

表三. CLDC Evaluation Kit 簡介

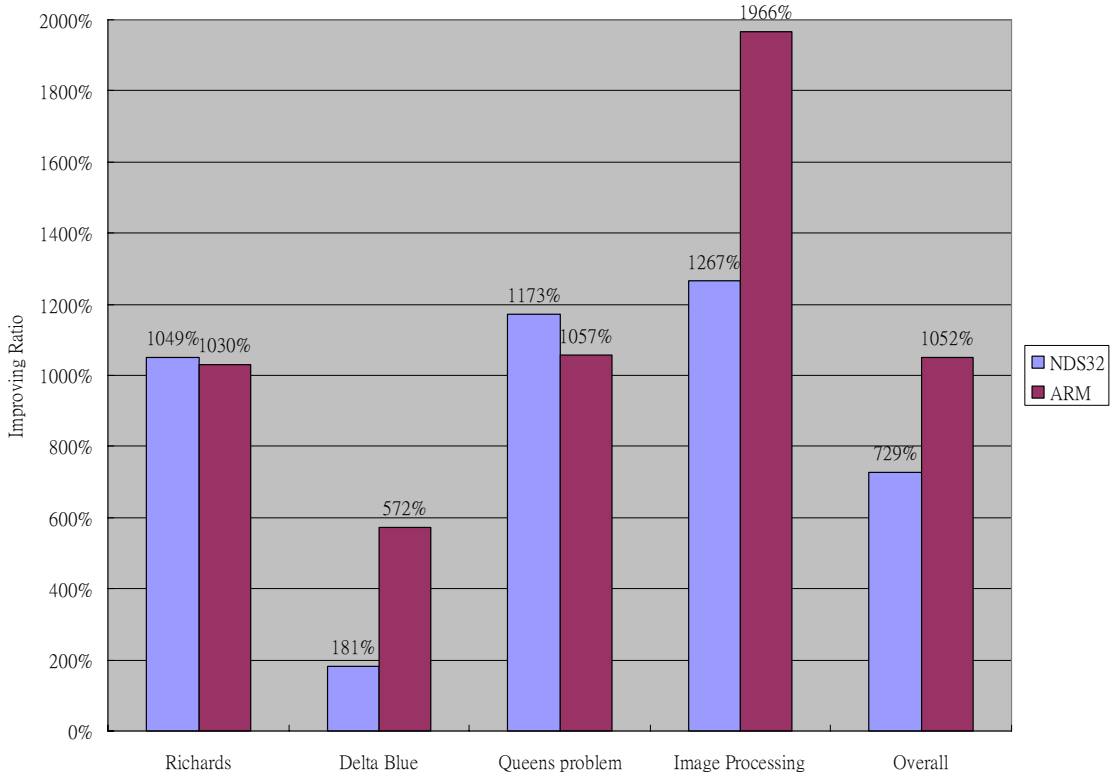
Name	Brief Description
Chess	A complete chess playing engine that is used to determine a set of chess moves.
Crypto	This suite of algorithms measures the performance of Java implementations in cryptographic transactions.
kXML	Measures XML parsing and/or DOM tree manipulation.
PNG	Shows how fast a Java implementation can decodes a PNG photo image of a typical size used on a mobile phone.

表四. GrinderBench 簡介

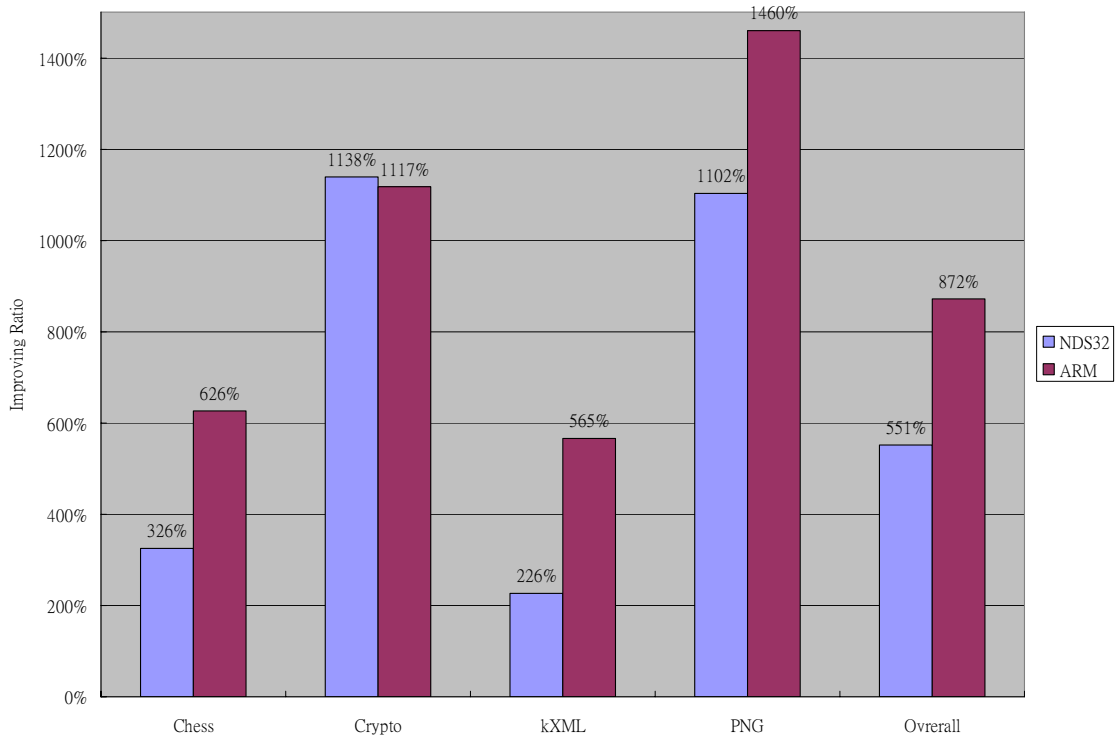
首先我們要評估的是 JIT 編譯器帶來的效能增進，直接與直譯器的執行效能做比較。以下三張圖分別是三套 benchmark 的執行結果，縱軸代表加入 JIT 編譯器後的執行效能與直譯器執行效能的比例，橫軸代表不同的 benchmark，Overall 則代表了幾何平均的結果。



圖六. 加入 JIT 編譯器後 Embedded CaffineMark 的效能改進



圖七. 加入 JIT 編譯器後 CLDC Evaluation Kit 的效能改進



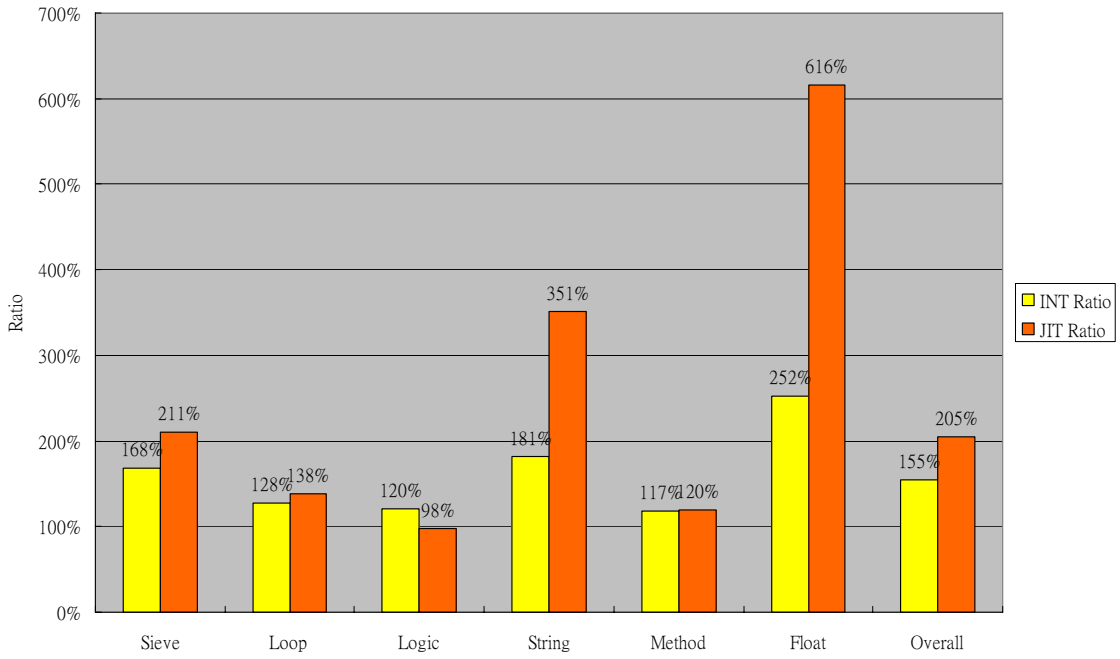
圖八. 加入 JIT 編譯器後 GrinderBench 的效能改進

由上面三張圖可以觀察到,在 Andes 平台上 JIT 編譯器帶來的平均效能改進為 551%到 729%，總平均大約為 633%。而在 ARM 平台上的效能增進更多，其平均約為 914%。會有這樣的差距當然是因為目前的移植還有調校的空間，且還有許多選擇性的優化還沒完成。整體而言，平均執行效能為直譯器的 6 倍。

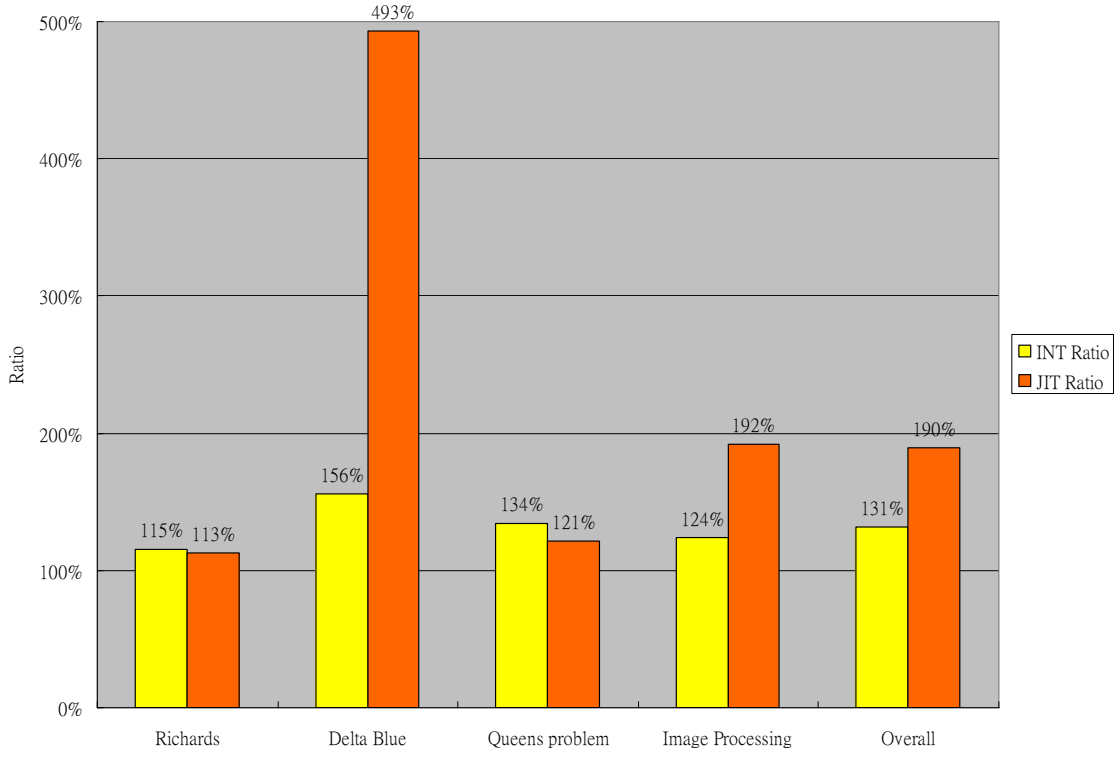
接著我們將評估與在 ARM 平台上與在 Andes 平台上執行效能的比例，分別評估直譯器以及加入 JIT 編譯器後兩種組態的執行效能，請參見圖九到圖十一。

我們觀察到直譯器的效能差距從 15%到 152%，範圍極大。經過分析執行的程式碼和相關的 glue code 和 helper 後，我們發現那些差距極大的程式如圖九中的 Sieve、Float 及圖十中的 Delta Blue，執行了大量的整數除法、浮點運算及同步的動作。在 ARM 實作中，前二者是以特別調校過的 assembly 程式碼來實作。另外 ARM 的版本也實作了 CAS 為基礎的快速同步。在扣除這些因素後,我們相信效能的差距是來自於靜態編譯器產生程式碼的品質，因為在直譯器上，與 ARM 版本的差別只有上述三者。

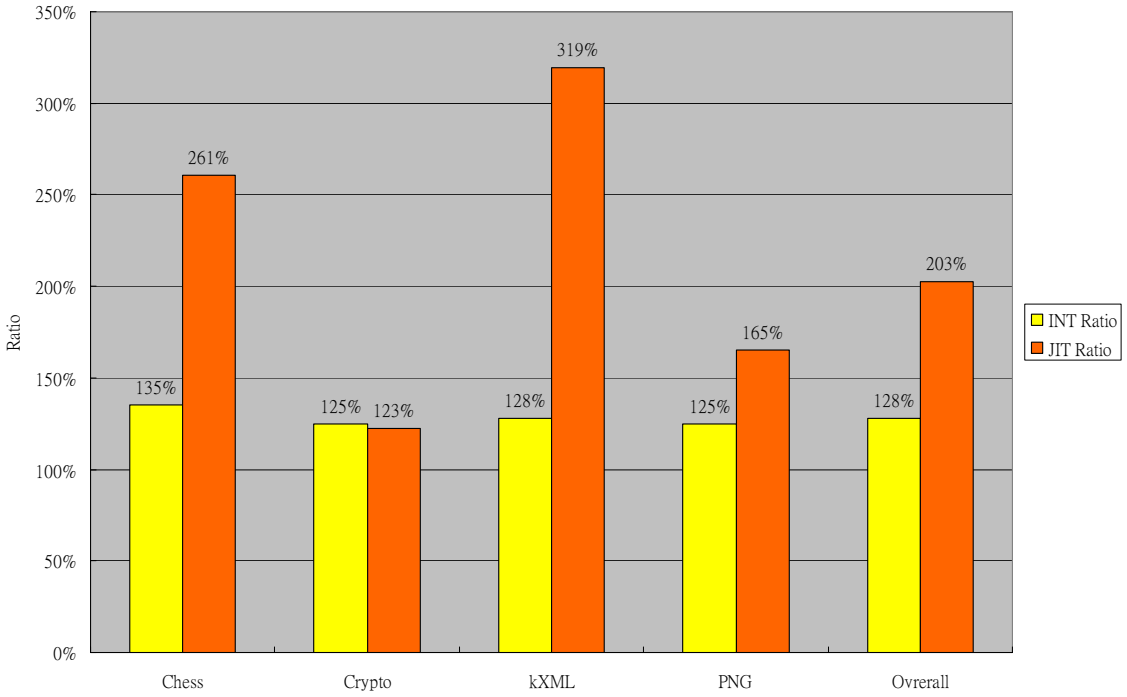
在加入 JIT 編譯器後，影響效能的因素更多了，與 ARM 平台的效能差距則更大了。在我們觀察產生的程式碼後，我們發現除了上一段敘述的因素外，還有 object allocatioon、null pointer check、GC Check 及程式碼品質等變因。而且因為目前同步相關 glue code 的實作方式為直接切換回直譯器，待完成工作後再切換回 compiled code，使得同步的速度當緩慢。值得一提的是，圖九中的 Logic 在 Andes 平台上效能較好，我們相信是因為可用暫存器數量不同造成的差異,也因此我們預估未來在較複雜的程式上,Andes 平台也許有機會較 ARM 平台適合執行 CVM 等級的應用。



圖九. ARM VS. NDS32 for Embedd CaffineMark



圖十. ARM VS. NDS32 for CLDC Evaluation Kit



圖十一. ARM VS. NDS32 for GrinderBench

目前在ARM平台上與在Andes平台上平均的執行效能比為190%到205%，總平均約為199%。與 ARM 平台效能的差距，我們相信在未來完成 CVM 中額外的優化技術後，應該可以拉近不少，加上預期的靜態編譯器改進，這些差距應該可以弭平。

另外，我們統計了兩個平台下，CVM 執行檔的所佔的空間比例、JIT 編譯器產生的程式碼大小比例及 JIT 編譯器產生程式碼的速度，由於變異性不大，我們將只列出總平均的結果。平均而言，Andes 版本的 CVM 執行檔較 ARM 版本略大 3%，JIT 編譯器產生的程式碼大概較 ARM 版本大 24%，而編譯速度(bytecodes/sec)則稍快 6%。這部分主要還是因為額外的優化和程式碼品質的差異造成的結果。

三、參考文獻

[1] Sun Microsystems. CDC HotSpot Implementation Virtual Machine, 2004.

[2] Andes Technology. Andes Instruction Set Architecture Specification, 2007

[3] Andes Technology. ANDES Whitiger Application Binary Interface, 2007

[4] Sun Microsystems. CDC Build System Guide, 2005

[5] Sun Microsystems. CDC Runtime Guide, 2005

[6] Sun Microsystems. CDC Porting Guide, 2005

[7] Sun Microsystems. CDC HotSpot Implementation Dynamic Compiler Architecture Guide, 2005

四、計畫成果自評

本計劃為 Andes 處理器架構移植嵌入式系統上常見的 JVM，且探討了針對 Andes 處理器架構設計優化技術的可能性。經過對實驗結果的觀察，我們發現在 Andes 處理器上運行 JIT 技術是可行的且獲得了很好的加速。而且跟在 ARM 上運行 JIT 技術比較下，測試結果顯示 Andes 處理器在幾處的評比跟 ARM 的結果是相當接近的甚至優於 ARM，我們認為除了部分硬體上的先天限制較難以突破以外，透過持續對 JIT on Andes 技術做最佳化將可以完成我們的最終目標也就是為 Andes 處理器提供高效能的嵌入式系統用 JVM，使其成為適合的嵌入式 Java 應用平台。