探討程式在不同編譯最佳化對快取記憶體設計空間之影響-以H.264為例

許望毅
大葉大學資訊工程研究所
R9506038@mail.dyu.edu.tw

張峻銘
大葉大學資訊工程研究所
R9406013@mail.dyu.edu.tw

摘要

本文探討如何提升嵌入式系統對多媒體應用的效能，且以 StrongARM 及多媒體視訊標準 H.264 應用為例，透過對 H.264 進行測描以取得 H.264 最常出現函式然後分別在不同最佳化條件下編譯此函式，再將編譯後之函式使用 Sim-Panalyzer 模擬分析，進而取得不同程式在 StrongARM 處理器內第一階快取記憶體運作內容最後探討分析以上模擬資料，以探究編譯器優化選項對於效能與功率消耗的影響，進而找出對於 H.264 編碼器在多媒體的應用上，其成本、效能及功率消耗這三方面在記憶體架構上的最佳設計。

關鍵字：嵌入式系統，功率消耗，Sim-Panalyzer，StrongARM，H.264

1. 簡介

數位科技日新月異舉凡遊樂影音等生活配備或是運輸系統及製造生產的自動化，隨處可見嵌入式系統的運用，有別於傳統效能導向的桌面式系統平台開發嵌入式系統的可攜性、價格敏銳、電池供電量、長時運作等需求使得嵌入式系統開發必須兼顧低價、省電及高效率以上三項設計準則特別是視訊多媒體應用需要消耗大量 CPU 運算資源及大量記憶體資料的存取，而上述皆是造成嵌入式處理器耗電的主因。

傳統遷入式多媒體應用程式開發多以效能及程式大小為最佳標化的標的，因此本文旨在探討透過編譯器最佳化後的程式，在嵌入式晶片架構上執行效能及功率消耗的影響以及記憶體架構之分析。

本文以現今嵌入式系統常用的處理器 StrongARM 及多媒體視訊標準 H.264 的應用為例，透過使用模擬器以觀察 H.264 在第一級資料與指令快取記憶體上的運作情形，進而改善第一級快取記憶體設計以減少失誤率。設計並加入成本與功率消耗的因素考量，進而達到快取記憶體效能、成本、與功率消耗這三方面最佳化要求。在嵌入式應用程式開發上傳統式以效能及程式大小為目標，做法上通常最快捷的方法是採用編譯器實現，所以編譯器在嵌入式應用程式開發佔了一個重要的地位。本文採用功能強大的自由軟體 GCC 編譯器，探討透過此編譯器最佳化後的 H.264 解碼器程式在嵌入式晶片架構上執行效能及功率消耗的影響。

本文對編譯器最佳化的研究分為三個階段，首先將 H.264 解碼程式對於 StromgARM 的快取記憶體架構的效能影響，最後再探討 GCC 編譯器最佳化程式後效能與功率消耗的分析討論。

實作方面本文採用軟硬體協同設計(Hardware/Software Co-design)的方式而不再是傳統硬體或軟體個別獨立處理的方式[16]，藉由軟硬體整合設計的方法是透過軟體模擬及最佳化分析以找出影響整體設計的效能瓶頸，並且以硬體最佳化方式去改善。首先測試程式方面透過傳統函式層測描的方式找出針對 H.264 解碼器應用上執行時間最久的函式，並且針對此函式做測描分析根據此函式輸入與輸出的資料開發出單一函式測試程式，以加速實驗模擬時間，並減少大量數據在分析上的困難，其次在模擬分析方面就上述開發的單一函式測試程式透過 Sim-Panalyzer 模擬分析整個 StrongARM 處理器內第一階快取記憶體運作情形，並探討分析 GCC 編譯器優化選項對於效能與功率消耗的影響關係，進而找出對於 H.264 解碼器在多媒體的應用上其成本效能及功率消耗這三方面在記憶體架構上的最佳化設計。

2. 相關研究

H.264/AVC 的編碼為 ITU-T 與 ISO/IEC 聯合制訂的最新視訊編碼，它是先由 ITU-T 的 VCEG 於 1997 年提出[1]，目標是提出一種更高性能的視訊編碼。

H.264 與其他視訊標準不同的地方為[2]:

- 高壓縮效能
- 簡單化的回歸基礎方法
- 編碼器與解碼器有複雜度的可調性(scalabilility)
- 除錯免疫力強

H.264 的技術包含畫面內預測編碼、畫面間預測編碼預測、迴路中使用方塊效應濾波器、使用不同的區塊大小及形狀使用多個參考畫面、使用高精確的次像素、動作像量、整數 DCT 轉換及新的熵編碼等下圖 2.1 為 H.264 解碼器的架構。

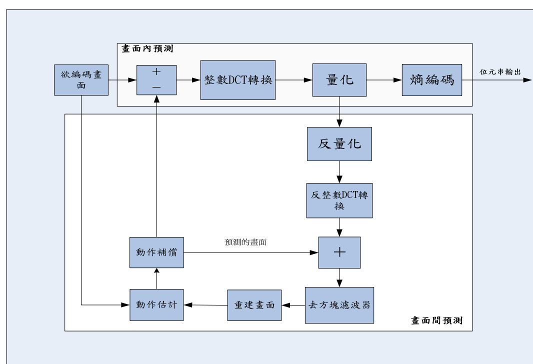

圖 2.1 H.264 編碼器架構

動作預測與估算：其主要包含畫面內部預測與畫面間預測畫面內部預測，主要是利用一張畫面裡的空間累贅做編碼而畫面間預測。

基本方塊大小和形狀：在 H.264 中一個動作估計基本單元的大小可以是 16×16、16×8、8×16、8×8、8×4、4×8 或者 4×4。根據巨方塊的內容是否變化劇烈，可以選擇最適當的模式做動作估計

整數 DCT 變換：H.264 的標準中所使用的是 4×4 的整數 DCT 轉換。相較於之前的標準所使用的 8×8 浮點數 DCT 轉換，整數的 DCT 轉換不會產生因為小數做四捨五入而產生的誤差問題

熵編碼:兩種不同的編碼方法，分別是可變長度編碼 VLC (Variable Length Coding)以及以前文為基礎的適應性的二進位算數編碼 CABAC(Context-based Adaptive Binary Arithmetic Coding)。

VLC(Universal Variable Length Coding)，簡稱 UVLC。不管資料的總類為何，H.264 都使用這個 UVLC 來編碼所有在編碼端要編碼的符號。其方法的優點是簡單，缺點是沒有考慮符號間的相關性，對於中高碼率效果並不是很好。

CABAC 可以配合符號出現機率分布，來機動性地調整符號的編碼，其中 CABAC 最大的特色是前文模式，利用前面剛剛編碼過的符號來決定估算下一個符號出現的機率，因此可以大大地降低符號間累贅。

3. 實驗方法及架構

3.1 H.264/AVC 程式碼

本研究使用的 H.264 官方測試程式碼為 JM 12.2，它是由德國 Heinrich-Hertz-Institute 研究所負責開發且 JM 能實現 H.264 所有的功能[10]。但由於 JM 主要設計考慮是引入各種新功能以提高其編碼性能，而忽視了編碼複雜度，造成其編碼複雜度高且程序結構冗長，使得執行效能不彰的缺點針對上 JM 的缺陷，許多現有研究分別提出新的軟硬體架構設計，藉以提昇其執行效能。

本研究的主旨即在提出一個透過函式層側描方法找出影響整個 JM 解碼器的關鍵函式(critical function)，並以此關鍵函式就效能及功率最佳化的情況下，找出最佳的快取記憶體設計，以下就我們使用的函式層側描方法作一探討。以求測試結果的精確。

3.2 函式測描法

當我們執行一個應用程式所花的時間其中有約有 90%的時間可能花費在 10%的程式碼中，而這 10%的程式碼可能來自應用程式中的一個函式且這個函式可能幾乎都在執行迴圈的動作，因此把這個站總程式執行時間最長的函示找出來並且對此函式做最佳化所得到效益是最顯著的。

在本文中我們使用 Gprof[11] 與 OProfile[12]此 2 種 profiling 工具去偵測整個 JM 解碼器，找出在 JM 解碼器程式中佔整個程式最長時間的函式。

Gprof 為一個 GNU profiling 工具[17]，它可以在程式執行時側描出每個函式呼叫次數、函式所佔的 CPU 時間，也可以找出每個函式之間呼叫關係及呼叫函式的執行時間。由於 gprof 簡單且容易了解，因此使用此工具可以很快的來幫助我們找到一個應用程式中所花費最久時間的函式，再來進行對此函式的最佳化。首先將 JM 的 Benchmark 中的解碼器部分作側描，使用 gprof 工具可以獲得 flat profile 數據表部分檢測結果顯現於下圖 3.1。

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 11.11      0.01     0.01     2416     0.00     0.00  itrans
 11.11      0.02     0.01     2316     0.00     0.00  GetStrengthNormal
 11.11      0.03     0.01     1348     0.01     0.01  get_block_chroma
 11.11      0.04     0.01     1281     0.01     0.01  read_significant_coefficients
 11.11      0.05     0.01     1162     0.01     0.01  EdgeLoopLumaNormal
 11.11      0.06     0.01      674     0.01     0.01  get_block_luma
 11.11      0.07     0.01      297     0.03     0.03  ercWriteMBMODEandMV
 11.11      0.08     0.01        1    10.00    10.00  CleanUpPPS
 11.11      0.09     0.01        1    10.00    10.00  compute_colocated
  0.00      0.09     0.00    30773     0.00     0.00  biari_decode_symbol
  0.00      0.09     0.00    30696     0.00     0.00  getNonAffNeighbour
  0.00      0.09     0.00    19272     0.00     0.00  arideco_bits_read
  0.00      0.09     0.00     9636     0.00     0.00  readSyntaxElement_CABAC
  0.00      0.09     0.00     9300     0.00     0.00  getLuma4x4Neighbour
  0.00      0.09     0.00     6497     0.00     0.00  readRunLevel_CABAC
  0.00      0.09     0.00     5544     0.00     0.00  biari_decode_symbol_eq_prob
  0.00      0.09     0.00     2613     0.00     0.00  get_mb_block_pos_normal
  0.00      0.09     0.00     2532     0.00     0.00  biari_init_context
  0.00      0.09     0.00     1909     0.00     0.00  read_and_store_CBP_block_bit
  0.00      0.09     0.00     1522     0.00     0.00  EdgeLoopChromaNormal
  0.00      0.09     0.00     1280     0.00     0.00  readIntraPredMode_CABAC
  0.00      0.09     0.00     1120     0.00     0.00  intrapred
  0.00      0.09     0.00      808     0.00     0.00  getChroma4x4Neighbour
  0.00      0.09     0.00      604     0.00     0.00  readMVD_CABAC
  0.00      0.09     0.00      594     0.00     0.00  CheckAvailabilityOfNeighbors
  0.00      0.09     0.00      495     0.00     0.00  CheckAvailabilityOfNeighborsCABAC
  0.00      0.09     0.00      493     0.00     0.04  perform_mc
  0.00      0.09     0.00      426     0.00     0.00  SetMotionVectorPredictor
  0.00      0.09     0.00      364     0.00     0.00  get_mem2Dshort
  0.00      0.09     0.00      299     0.00     0.00  biari_decode_final
  0.00      0.09     0.00      297     0.00     0.07  DeblockMb
  0.00      0.09     0.00      297     0.00     0.10  decode_one_macroblock
  0.00      0.09     0.00      297     0.00     0.00  exit_macroblock
  0.00      0.09     0.00      297     0.00     0.00  get_mb_pos
```

圖 3.1 部分 Gprof flat profile 結果清單

表 3.1 此測描結果的數據欄位意義如下:

函式名稱	功能說明
%time	表示此函式執行時間佔總程式執行時間之百分比同時包含此函式呼叫的副程式所執行時間之比重
cumulative seconds	以秒為單位累計函式所花費的時間
self seconds	以秒為單位函式本身所花費的時間
calls	函式被呼叫的總次數
self ms/call	函式每次被呼叫的平均時間
tota ms/call	函式每次被呼叫的平均時間和呼叫其他函式所花費的時間
name	函式名稱

依上圖 3.1 及表 3.1 之數據可以發現到共有九個函式各占整個 JM 解碼器總時間的 11.11%，這表示此九個函式為整個 JM 解碼器執行時間最久的函式。

Gprof 得優點是簡單易用，缺點是掛載在作業系統下的一個應用程式所以無法測描系統程式執行結果且其取樣頻率只有 1000Hz 對於現代 CPU 處理速度來說不夠精確，特別是函式實際執行時間若小時 0.001 秒時 Gprof 可能無法偵測到該函式執行時間進而產生統計上的誤差；為了克服 gprof 的缺陷必須借助 Oprofile 去分析 JM 解碼程式。

Oprofile 是一個新開發使用於 Linux 的側描和性能監控工具，他具有全系統且低負載的系統效能監視特性。使用時，它以核心頻率透過處理器的效能監視硬體以取樣方式取得關於系統上執行程式的資訊。由於它對系統的負載低而且可以對 Linux 核心進行側描，因此目前被建立在 Linux 2.6 版的核心中。此工具幫助使用者收集有關處理器事件的信息，其中包括 Cache 的失誤率、Memory 的存取次數、分支的預測錯誤率等等 OProfile 使用 CPU 中的 Times Stamp Counter (TSC)搭配 Advanced Programmable Interrupt Controller (APIC)。OProfile 執行時，TSC 會以內頻數率遞增，當 TSC 累計值達到使用者設定值時，APIC 會發出一中斷訊號啟動對應中斷服務以記錄 CPU 狀態，這個資料會透過緩衝器儲存到磁碟中特定目錄中。爾後使用者可以使用 OProfile 提供的分析器自該目錄中擷取所需資訊以產生系統與應用程式效能表現的分析數據，來找出造成效能瓶頸的函式程式碼，進而改善之。

本研究使用 Oprofile 分析工具中的 opreport 與 opannotate 首先使用 opreport 將 JM 解碼器做完整分析後得知 JM 解碼器程式碼共又 56 個函式被執行其中花費時間最久的函式為 get_block_luma，其函式層分析結果列表 3.2 如下。

表 3.2 Oprofile 測描結果如下:

sample	%	symbol name
50	15.1976	get_block_luma
26	7.9027	EdgeLoopLumaNormal
16	4.8632	biari_decode_symbol
15	4.5593	getNonAffNeighbour
13	3.9514	EdgeLoopChromaNormal

13	3.9514	find_snr
11	3.3435	bu2img
11	3.3435	get_block_chroma
11	3.3435	Itrans
11	3.3435	readRunLevel_CABAC

第一個欄位為 samples 為該函式的樣本數，第二個欄位%為此函式的樣本數對所有函式樣本數的百分比第，三個欄位 symbol name 為函式的符號名稱。

其次再用 Oprofile 中 opnnaotate 找出 JM 解碼器程式中執行最長時間的檔案為 erc_do_p.c，且在 erc_do_p.c 程式碼中發現 get_block_luma( )函式，且此函式被呼叫多次。因此確定 get_block_luma( )函式為整個 JM 解碼器效能瓶頸的關鍵函式，期檢測結果顯現於下圖 3.2。

```
for(j=0;j<MB_BLOCK_SIZE/BLOCK_SIZE;j++)
{
    mingplus=0;
    joff=j*4;
    j4=img->block_y+j;
    for(i=0;i<MB_BLOCK_SIZE/BLOCK_SIZE;i++)
    {
        mingplus++;
        ioff=i*4;
        i4=img->block_x+i;

        vec1_x = i4*4*mv_mul + mv[0];
        vec1_y = j4*4*mv_mul + mv[1];

        printf("%d:vec1_x=%d\n",mingplus,vec1_x);
        printf("%d:vec1_y=%d\n",mingplus,vec1_y);
        get_block_luma(ref_frame, listX[0], vec1_x, vec1_y, BLOCK_SIZE, BLOCK_SIZE, img, tmp_block);

        for(ii=0;ii<BLOCK_SIZE;ii++)
            for(jj=0;jj<MB_BLOCK_SIZE/BLOCK_SIZE;jj++)
                img->mpr[LumaComp][jj+joff][ii+ioff]=tmp_block[jj][ii];
    }
}
```

圖 3.2 erc_do_p.c 部分程式碼

3.3 關鍵函式分析

使用 Oprofile 找到關鍵函式 get_block_luma ()後做進一步分析此關鍵函式與整個 JM 解碼器的關係，得知此函式位於 mc_prediction.c 檔案中，此程式碼的功能在做移動補償預測。get_block_luma 在整個 H.264 解碼器中的功能是以各種不同區塊方式讀取參考畫面的亮度 (luminance) 資料以計算該區塊的移動補償預測值。所以大量的算數計算與記憶體的讀取這些影響效能及功率的關鍵幾乎都在 get_block_luma 函式內所完成。

為了分析此關鍵函式 get_block_luma 所造成的效能及功率消耗關係，我們分析了 get_block_lum 執行時的對應資料，再將這些資料與關鍵函式 get_block_luma 重新包裹成一個獨立的測試程式以進行指令層模擬分析

3.4 實驗環境與模擬工具

為了估算 ARM 處理器上的功率消耗本研究使用 Sim-Panalyzer ARM[3][4]模擬器做為計算實驗功率消耗的模擬工具；Sim-Panalyzer 擴充 SimpleScalar[5][6] 模擬器的功能，增加了對快取記憶體功率消耗的計算模組，因此 Sim-Panalyzer 工具除了具有 SimpleScalar 所有功能之外，還增添了功率消耗的計算。

Sim-Panalyzer 在 Power 模組的部份包括：aio,dio, irf, fprf, il1,il2, dl1,dl2, itlb,dtlb, btb, bimod, ras,logic, clock, alu, mult, fpu, uarch。其中快取記憶體模組的部份包括：il1, il2, dl1, dl2, itlb, dtlb, btb。而每個模組所估算出的功率消耗包含 switching、internal、leakage、pdissipation、peak 五部份，其各代表的意義如下：

- Switching：在輸入與輸出轉換時所造成的功率消耗。
- Internal：單獨 input 或 output switching 所造成的功率消耗。
- Leakage：漏電流的功率消耗(靜態)。
- Pdissipation ： switching ＋ internal ＋ leakage。
- Peak：最高峰時的功率消耗。

為了讓測試程式能在 Sim-panalyzer ARM 模擬器上執行我們使用 ARM Cross Compiler Tool Chain 裡的 GCC 做跨平台編譯，且本實驗為了探討 GCC 編譯器最佳化選項對於 ARM 嵌入式系統效能及功率消耗之關係，故使用 GCC 最佳化選項 O0、O1、O2、O3[14][15]各別去編譯測試程式，分別產生執行檔，再分別使用 Sim-panalyzer 來模擬 ARM CPU 內部運作過程，藉以產生所需的各項數據以供分析並詳列模擬環境於表 3.3。

本研究使用 ARM 平台的 GCC 優化選項 O0、O1、O2、O3 說明及使用軟硬體平台如下:

- O0:預設值不使用任何最佳化選項
- O1:壓縮執行檔大小同時提升執行效能
- O2 :會進行所有不包含效能、空間折衷考量的最佳化動作。
- O3: 指定這個最佳化選項會讓 GCC 進行所有效能最佳化的處

理，即使會讓程式碼變大亦然

表 3.3 模擬環境列表如下:

模擬環境
◆ CPU ： Intel(R) Xeon(TM) CPU 3.40GHz。
◆ Memory：4GB DDR RAM。
◆ OS：SuSE Linux Professional 9.3。
◆ ARM Cross Compiler BINUTILS Tool Chain：
■ GNU GCC v4.0.2
■ GNU binutils v2.16.1
■ Newlib v1.14.0
◆ Sim-Panalyzer Version 2.0.3 Release (ARM target)：
◆ Gprof ：2.15.94.0.2.2
◆ Oprofile：0.9.3
◆ H.264/AVC Software：JM 12.2

4.實驗與結果分析

4.1 測試程式開發流程

由於 sim-panalyzer 模擬器是以指令層模擬的方式去紀錄整個 ARM CPU 內部運作過程，因此若以整個 JM 解碼器去做模擬分析，它所花費的時間過於長久，也會產生龐大的數據資料，處理如此龐大的數據必會造成分析上的困難。基於以上理由本文將影響整個 JM 解碼器效能瓶頸所的關鍵函式 get_block_luma 擷取出，並將此關鍵函式包裹成一個可在 sim-panalyzer 模擬器上執行的二位元可執行檔，並命名為 JMHS(JM Hot Spot)作為本文實驗的 H.264 解碼器測試程式測試流程如下圖 4.1 所示。

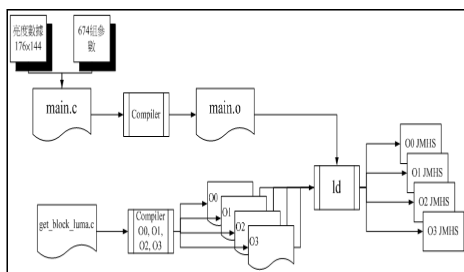

圖 4.1 測試程式開發流程

4.2 快取記憶體規範

探討程式最佳化、快取記憶體設計與 CPU 消耗功率三者的影響關係。本文中快取記憶體設計組態的部份分別使用以下幾種[7][8][9]：

1. 快取記憶體大小(Cache Size)：16Kbyte、32Kbyte、64Kbyte、128Kbyte。
2. 區塊大小(Block Size)；8byte、16byte、32byte、64byte。
3. 關聯集合數(Set Associative)：1-way、2-way、4-way
4. 置換法則(Replacement Rule)：Least Recently Used (LRU)。

根據上述快取記憶體組態設定，快取記憶體大小有 4 種組合，區塊大小有 4 種組合，關聯集合數有 3 種組合，LRU 置換法明顯優於其他種類的置換法，所以本論文使用 LRU 演算法為本實驗的置換法則。所以本實驗共有 4×4×3×1＝48 種組合作為本文快取記憶體的設計組態。

一般在快取記憶體效能分析實驗中，失誤率(Miss rate)是一個重要的參考指標。本文根據 average memory access time(AMAT)的 CPU 執行時間計算公式，對於快取記憶體效能在相關 AMAT 公式轉換如下[19][20][21]：

$$Performance \propto Average\ memory\ access\ time^{-1}$$
$$AMAT_{L1} = HitTime_{L1} + MissRate_{L1} \times MissPenalty_{L1}$$

$HitTime_{L1}$：快取記憶體命中時的存取時間。
$MissRate_{L1}$：快取記憶體失誤率。
$MissPenalty_{L1}$：快取記憶體失誤懲罰。

根據 AMAT 公式表示，記憶體的存取為主要影響整個 CPU 效能，而記憶體的存取來自於快取記憶體上的失誤，所以快取記憶體的失誤率為快取記憶體上效能的評估指標。

4.3 實測結果

首先在不進行優化的測試程式下來做模擬測試分析，也就是使用 GCC 編譯器 O0 優化的 JMHS 來測試，最後將模擬數據紀錄並繪圖比較已找出最佳的指令與資料的快取記憶體大小設計如下圖 4.2、4.3 所示，最佳的指令與資料快取記憶體配置為記憶體大小 16Kbytes，區塊大小為 64byte，2-way 集合關聯數。

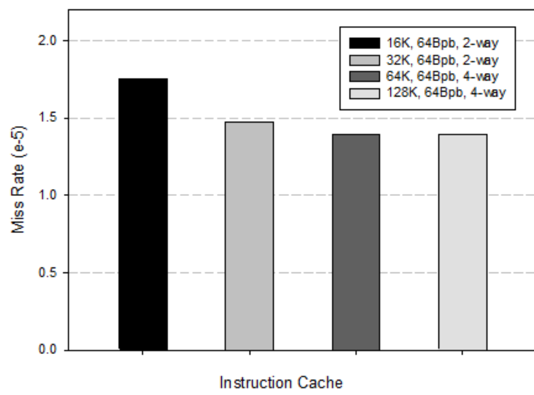

圖 4.2 最佳指令快取記憶體與失誤數的比較

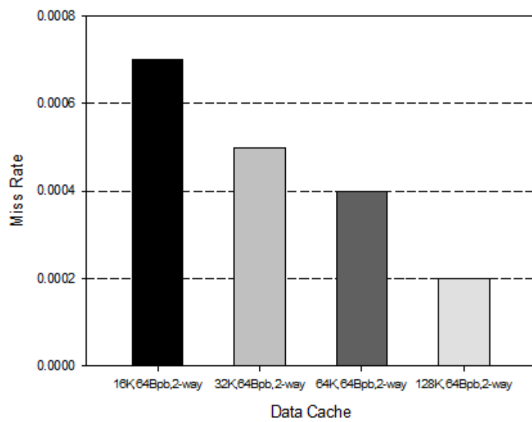

圖 4.3 最佳資料 快取記憶體與失誤數的比較

依上述使用記憶體大小 16Kbytes，區塊大小為 64byte，2-way 集合關聯數做為 O0、O1、O2、O3 測試檔之記憶體配置，產生下列結果依圖 4.4 、4.5、 4.6 所示[13][18]。

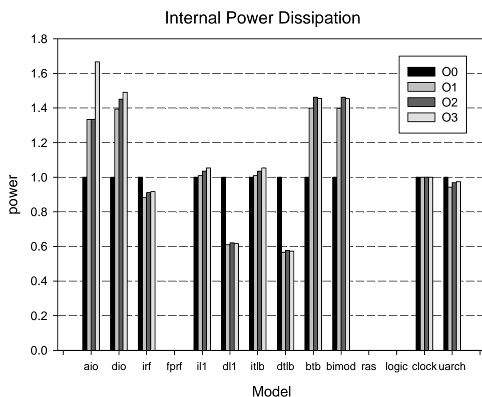

圖 4.4 動態 internal 消耗功率與程式優化選項之比較

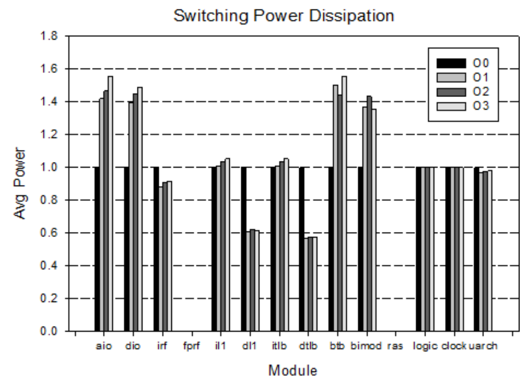

圖 4.5 動態 switch 消耗功率與程式優化選項之比較

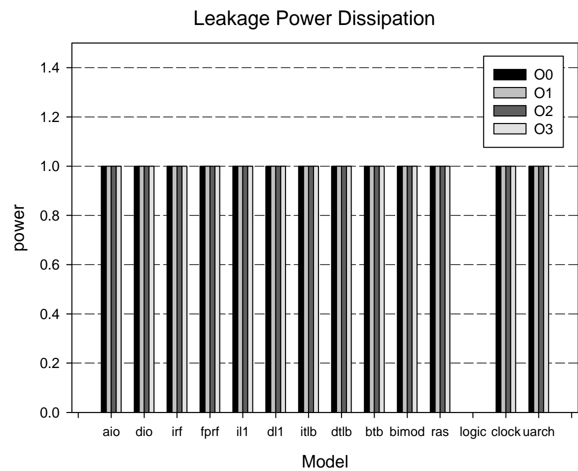

圖 4.6 leakage 消耗功率與程式優化選項之比較

一般而言功率消耗分為兩種分別是動態功率消耗(Active Power)為 internal 與 switch 的加種以及靜態功率消耗(Static Power)為 leakage

本研究依上列 Sim-Panalyzery 在各模組上消耗數據相加計算比不使用優化(O0)都有改善，並以 O1 優化選項對於功率消耗的改善效果為最佳其統計結果顯現於下圖 4.7。

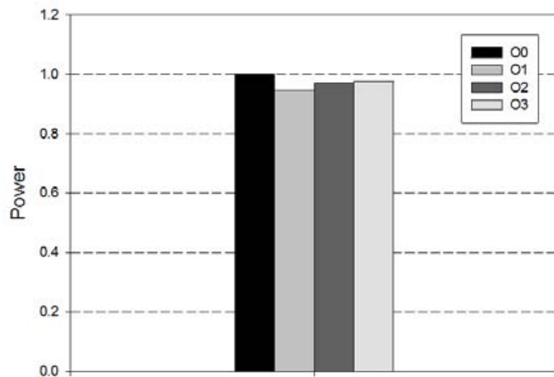

圖 4.7 編譯器優化選項對於測試程式總功率消耗

5.結論

本研究在於透過資料側描方式達成對嵌入式系統低功率消耗的設計方式，並針對多媒體影像 H.264 解碼器為例。透過本文提出函式層側描方法可找出佔用整個 H.264 解碼器執行時間最久的關鍵函式，並以此關鍵函式制作出針對函式層級的測試程式 JMHS。製作過程中爲了與標準 JM 解碼器程式所用資料上的一致性，採用分析整個關鍵函式與相關函式之程式碼，並將整個標準 JM 解碼器程式相關輸入資料萃取出來，重新將輸入資料與關鍵函式包裹成為可於模擬器 sim-panalyzer 上執行的二位元檔。本研究為了探討編譯器優化選項與功率消耗之影響，使用 ARM 的 GCC 在編譯過程將關鍵函式各別以 O0、O1、O2、O3 優化，最後再與整個 JMHS 測試程式連結與載入成為此四種優化選項下的執行檔。經實驗結果對於此關鍵函式最佳低功率消耗的指令與資料快取記憶體之設計，分別為 16 Kbytes、64 bytes 區塊大小、4-way 集合關聯數的指令快取記憶體與 32 Kbytes、64 bytes 區塊大小、4-way 集合關聯數的資料快取記憶體。再根據最佳低功率消耗之快取記憶體設計上探討編譯器優化選項對於功率消耗之影響，從實驗結果得知幾點結論：

1. 編譯器優化選項對於靜態部分的功率消耗沒有影響。
2. 編譯器優化選項只影響動態部分的功率消耗。
3. 對於整體 CPU 功率消耗，使用編譯器優化選項有助於降低功率消耗。

综合以上討論，編譯器優化選項對於低功率消耗為一重要考量因素。在未來後續低功率消耗研究中，對於如何降低靜態與動態的功率消耗為重點，雖然 GCC 編譯器優化選項對於靜態功率消耗無任何影響，卻可以用硬體設計方式降低靜態功率消耗。

參考文獻

[1] Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG, International Standard of Joint Video Specification (ITU-T Rec. H.264 ISO/IEC 14496-10 AVC), JVT-G050, March 2003.

[2] B. Stabernack, Kai-Immo Wels, H. Hübert, "***A System on a Chip Architecture of an H.264/AVC Coprocessor for DVB-H and DMB Applications***", IEEE Transactions on Consumer Electronics, Vol. 53, No. 4, NOVEMBER 2007.

[3] Sim-Panalyzer 2.0 Reference Manual.

[4] Sim-Panalyzer, The SimpleScalar-Arm Power Modeling Project. [Online]. Available: http://www.eecs.umich.edu/~panalyzer/

[5] SimpleScalar Tool Set. [Online]. Available: http://www.simplescalar.com/

[6] D. Burger and T. M. Austin. ***The simplescalar tool set, version 2.0. SIGARCH Comput.*** Archit. News, 25(3):13–25, 1997.

[7] J. L. Hennessy and D. A. ***Patterson. Computer Architecture A Quantitative Approach.*** Morgan Kaufmann Publishers, Inc, 2003.

[8] C. R. Baker and S. Bandypadhyay. "***Energy Based Analysis of Embedded Microprocessor Cache Design***", EECS, University of California, Berkeley, CS 252 Project Presentation.

[9] A. J. Smith, "***Cache memories***", ACM Computing Surveys 14, No.3, pp.473-530, 1982.

[10] H.264/AVC Software Coordination. [Online].Available: http://iphome.hhi.de/suehring/tml/

[11] Sysprof, System-wide Linux Profiler . [Online].Available:http://www.daimi.au.dk/~sandmann/sysprof/

[12] Gprof, The GNU Profiler. [Online]. Available: http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html

[13] Oprofile, OProfile manual. [Online]. Available: http://oprofile.sourceforge.net/doc/index.html

[14] Kurt Wall, William von Hagen 著，鄧偉敦譯，"***GCC 完全指南***"，博碩文化，2005 年 4 月。

[15] GCC, GCC 3.3.5 Manual. [Online].

Available: http://gcc.gnu.org/onlinedocs/gcc-3.3.5/gcc/

[16] Dinesh C. Suresh, Frank Vahid, Greg Stitt, Jason R. Villarreal, and Walid A. Najjar, "***Profiling tools for hardware/software partitioning of embedded applications.***" Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems, pp.189-198, 2003.

[17] Make, GNU Make Manual. [Online]. Available: http://www.gnu.org/software/make/manual/html_node/index.html

[18] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: ***simple techniques for reducing leakage power***. SIGARCH Comput. Archit. News, 30(2):148–157, 2002.

[19] C.-G.Kim,J-W.Park,J.-H.Lee,and S-D.Kim, "***A small data cache for multimedia-oriented embedded systems***" Journal of Systems Architecture,In Press,Corrected Proof, Avaiable onlibe 16 Mzy 2007

[20] J.-H.Lee,J.-S.Lee,S.-D.Ki, "***A new cache architecture based on temporal and spatial locality***" Journal of Systems Architecture, Vol.46,Number 15,31 December 2000,pp.1451-1467(17)

[21] J. L. Hennessy and D. ***A. Patterson. Computer Architecture A Quantitative Approach.*** Morgan Kaufman Publishers, Inc,2003.