國立交通大學

資訊科學與工程研究所

碩 士 論 文

Java 處理器上的記憶體分配電路設計

Memory Allocation Circuit Design for Java Processor

研 究 生：洪生迪

指導教授：蔡淳仁　教授

中 華 民 國　107 年　8　月

Java 處理器上的記憶體分配電路設計

Memory Allocation Circuit Design for Java Processor

研 究 生：洪生迪　　Student：Sheng-Di Hong

指導教授：蔡淳仁　　Advisor：Chun-Jen Tsai

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

August 2018

Hsinchu, Taiwan, Republic of China

中華民國 107 年 8 月

英文摘要

This thesis is developed based on a JAIP SoC, a Java processor with hardwired OS kernel. We propose a hardware memory allocator that will improve the memory utilization efficiency for dynamic memory allocation/free operations and allow many processors and IP cores to access a shared memory. Once our allocator circuit receives a request for memory allocation, it will return the pointer in a fixed time and manage the memory space released by the garbage collection unit. In addition, the released memory will be initialized to ensure the security. Our allocator circuit will not affect the processor's performance during the memory allocation and de-allocation because the data structure is updated in the background. The memory fragmentation problem will also be handled in this research. The final experimental results show that our hardware memory allocator is an order faster than the software memory allocator. The whole design is implemented and verified on a Xilinx FPGA and integrated into the JAIP SoC.

摘要

本論文以一顆內建電路化作業系統硬體的 Java 處理器, JAIP SoC, Java 為基礎，設計一個 hardware memory allocator，藉由此電路能有效地讓有限的記憶體空間使用上更有效率，同時也能讓處理器以及其他應用 IP 共享此電路達到記憶體共享的功能。而當處理器需要一塊記憶體時，能在固定的時間內回傳記憶體的起始位置，而當 garbage collection unit 進行回收而釋放的記憶體空間也是藉由此電路來管理，由於安全性的考量，會把釋放回來的記憶體初始化，而無論是分配記憶體或者是釋放記憶體所要維持的資料結構，通通都是在背景時間做更新，並不會影響處理器的效能，而在記憶體破碎的問題，在本論文中也有加以處理，最後的實驗數據中，hardware memory allocator 比 software memory allocator 快上數十倍。整個設計都是在 Xilinx FPGA 上實作及驗證，並整合至現有的 JAIP 中。

致謝

碩士論文的完成，我最感謝的就是蔡淳仁老師，這兩年的指導讓我學習非常多，像是相關專業領域知識以及擔任實驗課的助教，而每一次的會議都讓我有新的啟發，遇到困難時，老師總是很有耐心的教導我，並都能點出最關鍵的地方，簡潔扼要的說明問題，這也是我努力向老師學習的地方，

另外也要感謝實驗室的同學，在學習過程中一起討論並互相陪伴，讓我們能朝著目標一起前進，一路走來也感謝家人以及身邊的朋友支持我以及給我鼓勵，使我的目標更加明確。

目錄

英文摘要 i
摘要 ii
致謝 iii
目錄 iv
表目錄 vi
圖目錄 vii
第一章 前言 1
1.1 研究動機 1
1.2 研究目的與貢獻 1
1.3 論文架構 2
第二章 相關研究 3
2.1 Previous Work on JAIP 3
2.2 動態記憶體管理相關研究 4
2.2.1 Linklist 4
2.2.2 Bitmap 5
2.2.3 Buddy System 5
2.2.4 Newlib Allocator 7
第三章 Bitmap-based Hardware allocator 設計 10
3.1 Bitmap-based Hardware Allocator 架構 10
3.1.1 Bitmap 10
3.1.2 Bitmap Cache 11
3.1.3 Four Bit Searcher Pipeline 12
3.1.4 Bitmap Mark 15
3.1.5 Hardwar bitmap allocator FSM 16

3.2 Bitmap-based Hardware Allocator 分析 19
第四章 Newlib-based Hardware Allocator 設計 20
4.1 Newlib-based Hardware Allocator 架構 20
4.2 Newlib-based Hardware Allocator 設計 22
4.2.1 Main Controller 22
4.2.2 TOP Controller 24
4.2.3 EXACT Controller 24
4.2.4 OTHERS Controller 25
4.2.5 INSERT Controller 26
4.2.6 FREE Controller 26
4.2.7 REALLOC Controller 27
4.3 Fragmentation 處理 28
4.3.1 Internal Fragmentation 28
4.3.2 External Fragmentation 29
4.4 Application Programming Interface 29
第五章 與 JAIP Memory Manager 的整合 30
5.1 原本的 JAIP Memory Allocator 設計 30
5.2 新的 Memory Allocate Controller 設計 31
第六章 實驗結果 34
6.1 實驗環境 34
6.2 電路資源使用量 35
6.3 效能分析 37
6.3.1 Memory Allocate Sequence 分析 37
第七章 結論與未來展望 41
參考文獻 42

表目錄

表 1 FOUR BIT SEARCHER 紀錄資訊 13
表 2 ALLOCATOR API 29
表 3 JAIP BUS ARBITER 資訊 32
表 4 JAIP CORE 合成資訊 35
表 5 ORIGINAL GC 資源使用量 36
表 6 CURRENT GC 資源使用量 36
表 7 FREE CIRCULAR BUFFER 合成資訊 37
表 8 NEWLIB-BASED HARDWARE ALLOCATOR 合成資訊 37
表 9 ALLOCATE SEQUENCE 38
表 10 測試資料資訊 39
表 11 整體 ALLOCATOR 效能分析 39
表 12 單獨 ALLOCATE 效能分析 40
表 13 單獨 FREE 效能分析 40
表 14 記憶體使用量 40

圖目錄

圖 1 JAIP BLOCK DIAGRAM 3

圖 2 LINKLIST ALLOCATOR 5

圖 3 BITMAP ALLOCATOR 5

圖 4 BUDDY SYSTEM 6

圖 5 BUDDY TREE ORGANIZATION(圖源來自[7]) 7

圖 6 NEWLIB ALLOCATOR BIN 結構 8

圖 7 CHUNK AFTER ALLOCATING 9

圖 8 CHUNK AFTER FREE 9

圖 9 BITMAP-BASED HARDWARE ALLOCATOR 方塊圖 10

圖 10 BITMAP 方塊圖 11

圖 11 BITMAP CACHE 12

圖 12 FOUR BIT SEARCH PIPELINE 14

圖 13 符合配置大小的位置 15

圖 14 標記位置 16

圖 15 ALLOCATION OF FSM 16

圖 16 FREE OF FSM 18

圖 17 NEWLIB-BASED HARDWARE ALLOCATOR BIN 架構 20

圖 18 NEWLIB-BASED HARDWARE ALLOCATOR 方塊圖 21

圖 19 BIN CACHE 22

圖 20 MAIN CONTROLLER FSM 23

圖 21 TOP CONTROLLER 24

圖 22 EXACT CONTROLLER FSM 24

圖 23 OTHERS CONTROLLER FSM 25

圖 24 INSERT CONTROLLER FSM....26
圖 25 FREE CONTROLLER FSM....27
圖 26 REALLOC CONTROLLER FSM....28
圖 27 GARBAGE COLLECTION 架構....30
圖 28 GC TABLE 管理....31
圖 29 NEWLIB-BASED HARDWARE ALLOCATOR....31
圖 30 OBJECT ALLOCATION FSM....32
圖 31 GC STACK FSM....33
圖 32 KINTEX-7 (XC7K325T)....34
圖 33 ZEDBOARD....35
圖 34 XSDK ALLOCATE 分布圖....38

第一章 前言

1.1 研究動機

Java 程式語言具有許多重要的特性，物件導向、跨平台、安全、多執行緒等優點，只要把 Java 檔案透過編譯器所編出來的 class 檔，就可以在任何平台上執行，也因為有此跨平台的特性，在嵌入式系統中廣受歡迎，在安全性的方面，Java 程式語言不允許利用指標直接操控記憶體，也因此，記憶體的管理以及分配也是重要的一個環節。

Java 是一種物件導向的語言，每次產生的物件皆在 heap 上，thread 之間也能共享 heap 上的物件，因此 heap 空間的管理也是在設計上佔有非常重要的地位，垃圾回收(garbage collection)是記憶體管理的其中一種方式，而回收回來的記憶體該如何管理也是非常重要的一個環節，像是如何在再利用這些釋放回來的記憶體以及記憶體碎裂等等議題，這些議題若沒妥善處理，在有限的記憶體當中，對於一些大型應用程式當中很容易會有記憶體不足導致系統錯誤的可能。

1.2 研究目的與貢獻

本實驗室所開發的 Java Application-processor IP (JAIP)應用處理器已經完成垃圾回收的功能，但在回收回來的記憶體並沒有使用完整的機制去管理，本論文將以 JAIP 應用處理器[1][2]為基礎出發，目的是實作出能讓回收回來的記憶體空間能有效的管理以及再次使用，首先參考[4]的動態記憶體分配機制，此機制是目前嵌入式系統當中廣受歡迎的動態記憶體分配機制，因此我們把此機制完全利用硬體實作，並依據硬體能平行處理的特性修改了一些演算法，當處理器對此電路有要求時能快速處理並回應。

本論文的研究重點為當處理器需要一塊記憶體空間時要能快速搜尋並做回應，以及釋放回來的記憶體要如何管理並再次利用，因為會對釋放回來的記憶體以 size 為依據做分類，所以不管處理器要求多大的 memory size，都能在固定的

時間找到並回傳該記憶體的起始位置，而釋放回來的記憶體則是會先存在 buffer 當中，等到電路閒置的時候才會開始做 free 的動作，而上述的動作除了能快速回應以外，資料結構的維持的動作為了不影響處理器的速度，所以會在背景的時間去執行，此搜尋電路提供幾個重要的 Application Programming Interface(API)，像是 malloc、free、realloc，而因為安全性的考量，會把釋放回來的記憶體初始化，有了此電路就能讓處理器的記憶體能被有效的重複利用。

1.3 論文架構

本論文一共分為七個章節，第一章是概括性的導論，說明研究動機、背景及目的，第二章是介紹 JAIP 以及介紹普遍管理記憶體的相關研究及探討，第三章會介紹第一種實作方式 bitmap-based hardware allocator，和它如何管理記憶體，以及最後為甚麼不選擇此架構，第四章會介紹本論文所使用的記憶體管理，實作方法以及實際運作流程，第五章則會介紹如何把此架構整合至實驗室所開發的 JAIP 當中，第六章則是實驗數據分析，比較對象為軟體 newlib 的 allocator 以及 hardware bitmap，最後一章為本篇論文的結論以及未來展望。

第二章 相關研究

2.1 Previous Work on JAIP

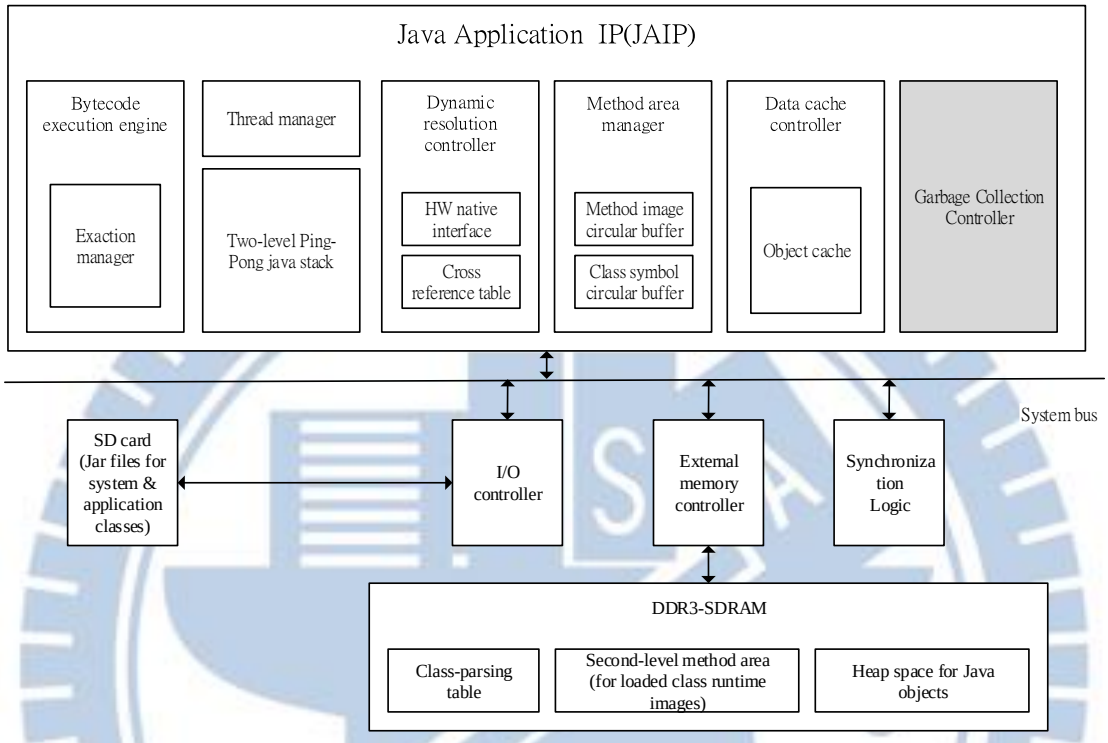

圖 1 JAIP block diagram

本論文使用的 JAIP 處理器架構如圖 1 所示，沿用[1] 優先權執行緒排程器之後的架構，深色的區塊則是本論文修改的區塊，Garbage Collection Controller。

JAIP 的架構包含了以下幾個部分，Bytecode Execution Engine(BEE)，負責執行 Java 經過編譯之後所產生出來的 class 檔，其中 BEE 是一個 4-stage pipeline，包括 Translate、Fetch、Decode、Execution 等 stage。其中 Translate stage 每次將會讀取兩個 bytecode 轉成兩組 j-code，接著送進 Fetch stage，而在此 stage，會根據 j-code 的資訊做分析，j-code 又可分為 simple 指令以及 complex 指令，若是 simple 指令的話，即是可執行的一組 instruction package，但若為 complex 指令的話，就會在此 stage 轉成一組 j-code sequence，送進 Decode stage 作解碼，最後送到 Execution stage 執行。

Dynamic Symbol Resolution Unit(DSRU)是負責符號解析的部分，讀取 Class Symbol Table(CST)以及 Cross Reference Table(XRT)。Class Symbol Table 是存著 class 檔中的 constant pool 資訊、Class ID、Field Type、及 method invocation 需要的 XRT index。而 XRT 則是 Method Invocation 及 Field Access 需要使用。

Thread manager 是管理執行緒的建立以及 context switch 的處理，當有執行緒要建立時會發訊號至 Multicore coordination controller(MCC)，而 MCC 會負責分配該執行緒該由哪顆處理器執行。

Garbage Collection Controller(GC)負責將以後不會參考到的物件進行回收，並把這些物件的資訊存入近 GC table 當中，當處理器需要一記憶體空間時，也會透過 GC 去分配，分配時會先去 GC table 搜尋，若沒有合適的空間則會分配全新的記憶體空間給處理器。

2.2 動態記憶體管理相關研究

動態記憶體管理的演算法須滿足以下兩種要求(1)當有 allocate request 進來時，必須搜尋符合需求大小的記憶體空間並回傳起始位置(2)當有 free request 進來時，必須管理這些記憶體，使得這些 free 的記憶體能重複利用。以下介紹幾種傳統上的記憶體管理的演算法[6]，以及分析各個優缺點，最後決定我們使用的演算法。

2.2.1 Linklist

利用雙向鏈結串列來管理 free 之後的記憶體，如圖 2，每一個節點會紀錄 4 個資訊，分別為 previous node 的 index、memory start address、memory size、以及 next node index，剛開始只有一個 node，也就是整個 heap size，當有 allocate request 進來時會從這個 linklist 搜尋，若有找到及分配該記憶體空間，以及根據分配的大小去決定是否做 split，相鄰兩節點為連續空間的話及會合併成一連續空間以減少記憶體空間碎裂的情形。優點是實作簡單，刪除、插入節點容易，缺點是搜尋速度慢，搜尋須從 free list 頭開始，每次往下搜尋節點都需一次的記憶體

的讀取。

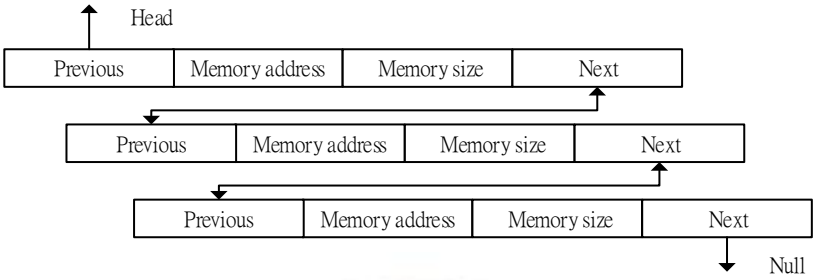

圖 2 linklist allocator

2.2.2 Bitmap

顧名思義利用 1 個 bit 去管理實際記憶體的一個區間，0 代表 free block，1 代表 allocated block，如圖 3，當有 allocate request 進來時，會根據 request size 去找到連續個 0，即可找到，之後把分配出去的記憶體所對應的 bitmap 標記成 1，而 free 的動作只需把對應的 bitmap 標記成 0 即可，由於相鄰兩個 bit 及代表連續兩塊空間，所以記憶體碎裂不用擔心。優點是實作簡單，找尋連續可用空間容易，缺點是需要額外記憶體空間管理，大型記憶體空間管理較不適用。此記憶體管理方法我們在本論文將會提出改良過後的 bitmap 並在第三章並在第七章做數據分析。

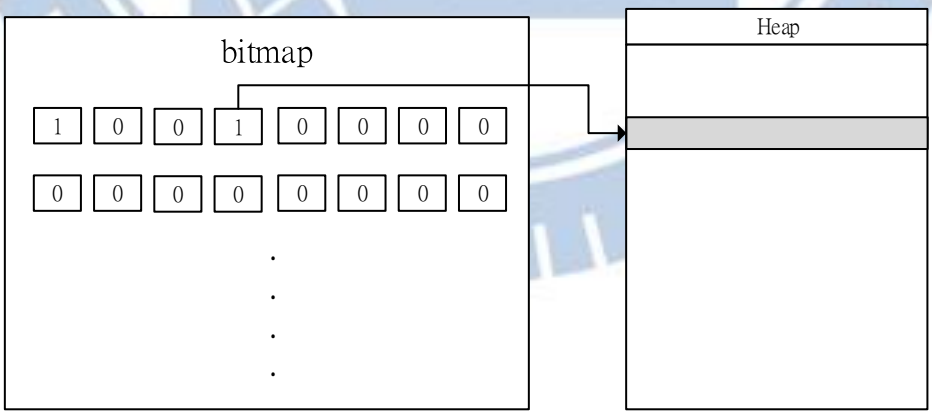

圖 3 bitmap allocator

2.2.3 Buddy System

Buddy system 是 linux 常用的記憶體管理方式[9]，每次再分配記憶體時會以

2^n 個 block 為單位，每個 block 為實體記憶體上的一連續空間，若 request size > 2^n block 的話，即會分配 2^(n+1)大小的空間，以下舉一個例子說明 buddy system，假設原先整個記憶體空間為 64 KB，當有一個 allocate request 為 8KB 時，首先會先去找尋 8KB 中的 free list 中有無可用的，若沒有就會向上找尋大 block，並做分裂，如圖 4，接下來如果有一個 allocate request size 為 10KB，就會去搜尋 16KB 的 free list，找到後直接分配，由此可發現此次的分配會浪費掉 6KB 的空間。Buddy system 的優點是搜尋快速，缺點是記憶體分配效率差，worst case 可能會造成將近一半的記憶體都是浪費的。

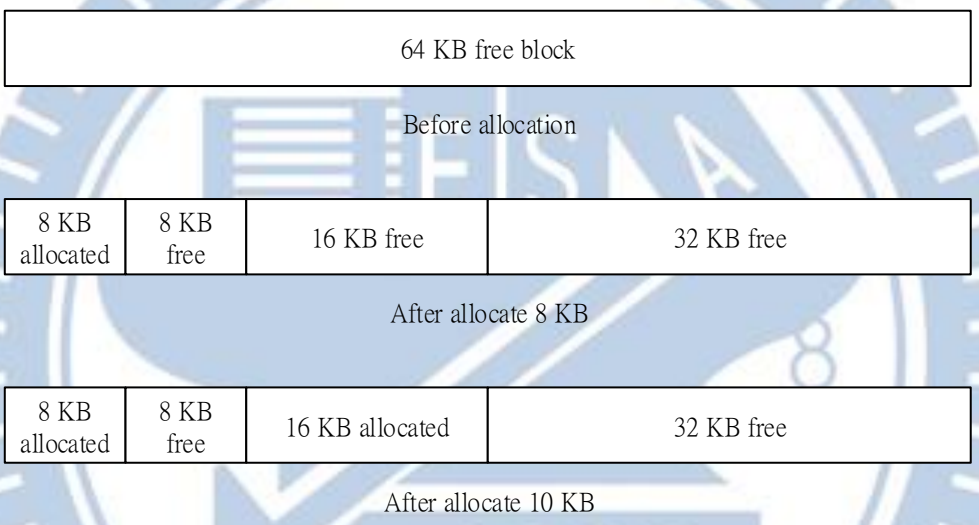

圖 4 Buddy system

有篇參考文獻發表了用電路實作 buddy system 的相關研究[7][10]，實作方法為引進 bitmap 作為 block size 的管理，建立起 binary tree，當 request size 有小於 2^n 的大小時會去第 n 層 binary tree 搜尋，找到的話就會將 tree 的第 n 層以下的 node 全部標記成 1，free 的話則是標記成 0 並向上檢查是否要做合併並更新資料結構，資料結構如圖 5。由於實現此資料結構所需的 FPGA 電路資源龐大，維持資料結構複雜，以及記憶體使用效率並不是這麼好，所以本篇論文並不採用此方法。

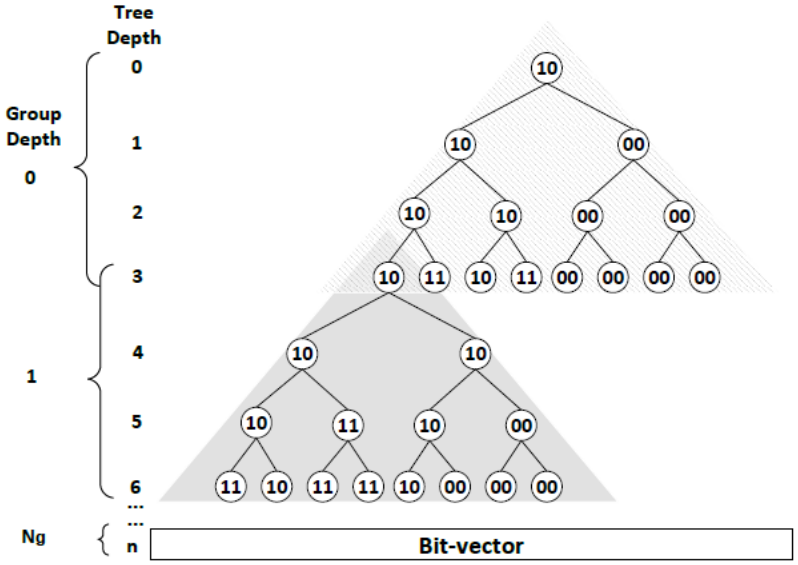

圖 5 Buddy tree organization(圖源來自[7])

2.2.4 Newlib Allocator

Newlib 是目前嵌入式系統當中廣為採用的軟體函式庫，且此函式庫為開源函式庫[4]，由 Jeff Johnson 所維護。在本篇論文中，我們參考了 newlib version 3.0.0 中的動態記憶體管理演算法，並開發出相對應的電路版本。而我們所參考的 allocator，程式位於 newlib-3.0.0\newlib\libc\stdlib\mallocr.c，此 allocator 為 Doug Lea 所開發，目前版本為 VERSION 2.6.5。

接下來介紹 newlib allocator 的演算法[5]，newlib allocator 會將 free 的記憶體空間根據 size 做分類，分類的名稱取為 bin，總共有 126 個 bin，第 2 bin 到第 64 bin 裡面都是放著相同 size 的 chunk(後面會介紹)，而第 65 bin 以後則會放著一個範圍的 size 的 chunk，每個 bin 裡的 chunk 則會用雙向鏈結串列串接起來，並且由小排到大如圖 6。

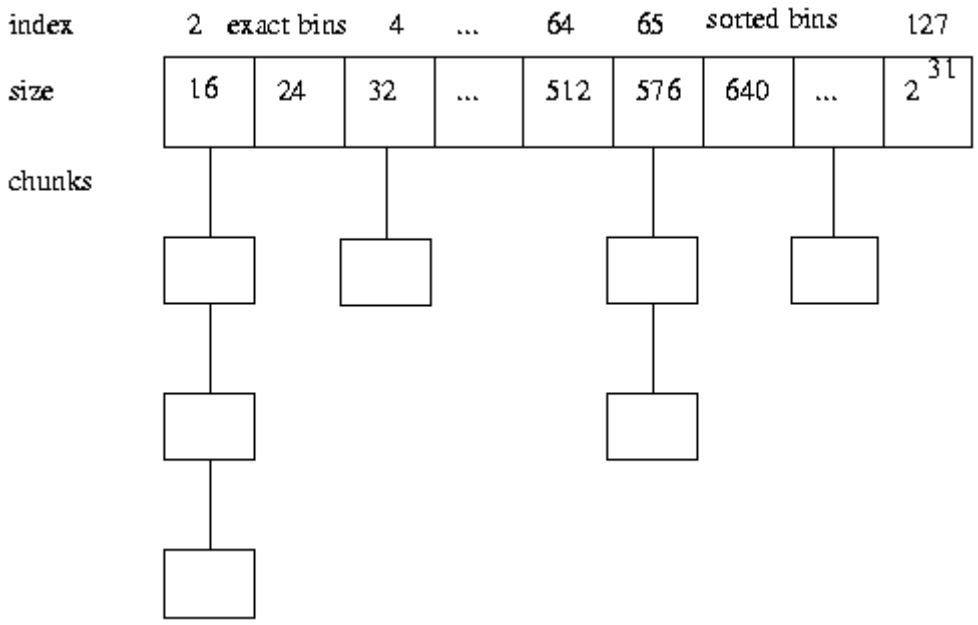

圖 6 Newlib allocator bin 結構

chunk 是一塊連續記憶體空間的單位，最小 size 為 16 bytes，當為分配出去的 chunk，則 chunk 的第一欄位會拿來紀錄該 chunk 的 size，此欄的最後一個 bit 為 previous in use bit，此 bit 為 0 時前一欄的 previous size 欄位才有效，分配出去的 chunk 此 bit 一定為 1，而剩下的空間為 user space，若 chunk 為 free 的話，第一欄一樣為 chunk size，第二欄為 FP(forward pointer)，為在同一個 bin 當中，前一個 chunk 的 pointer，第三欄為 BP(back pointer)，為在同一個 bin 當中，後一個 chunk 的 pointer，而 chunk 的最後一欄也會填上 chunk size，此欄的 chunk 的目的是要給下一個連續空間的 chunk 查詢 previous size 作使用的，而中間的記憶體空間則不會使用到，如圖 7 以及圖 8。

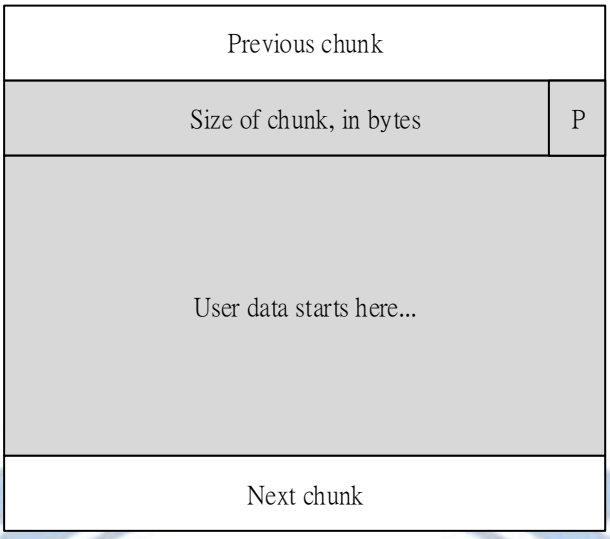

圖 7 chunk after allocating

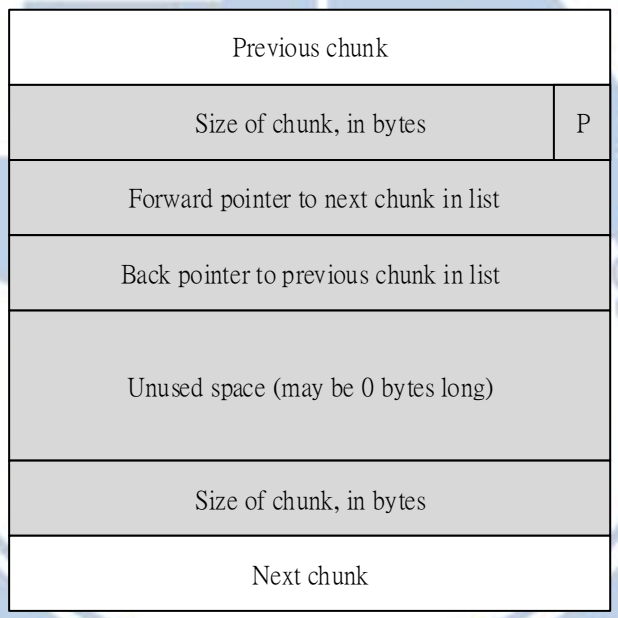

圖 8 chunk after free

Newlib 為本論文實作 hardware memory allocator 的參考架構，演算法及實作方式會在第四章 有更詳細的介紹，而最後的實驗數據將會與 hardware bitmap 以及軟體的 Newlib allocator 比較，而軟體是跑在 ARM Cortex™-A9 100Mhz 上，而實驗結果也能比前兩者快上數十倍。

第三章 Bitmap-based Hardware allocator 設計

傳統的 bitmap 大多用在檔案系統中，free space 的管理，因為實作簡單又能得到快速的回應，所以在早期相當的熱門。而在本論文中，我們沿用 bitmap 的概念，因為是要用電路實現 bitmap，所以會大量利用電路能平行搜尋的概念搜尋 free space，不是像軟體的循序搜尋方式，搜尋完成後的許多動作能在控制權還給處理器後在背景作執行，圖 9 為 bitmap-based hardware allocator 方塊圖。

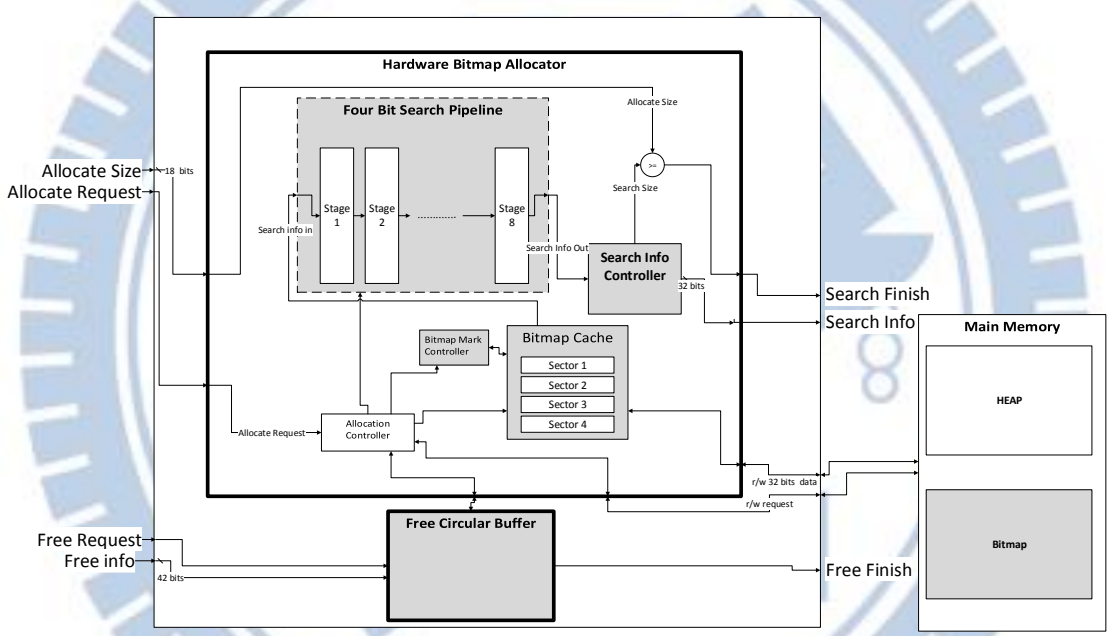

圖 9 Bitmap-based Hardware Allocator 方塊圖

3.1 Bitmap-based Hardware Allocator 架構

整個 bitmap-based hardware allocator 可以切成以下幾個部分，(1)Bitmap，(2)Bitmap cache，(3)Four Bit Search Pipeline，(4) Bitmap Mark Controller，(5) Search information controller，接下來會對這些部分一一介紹。

3.1.1 Bitmap

這邊預設 heap 空間為 512M，bitmap 裡的每一個 bit 為管理對應的一塊記憶

體空間，大小為 32 bytes，所以需要額外 2 MB 的記憶體來記錄 bitmap，而 bitmap 會被切割成 16384 個 sector，如圖 10，每一個 sector size 為 1024 bits。

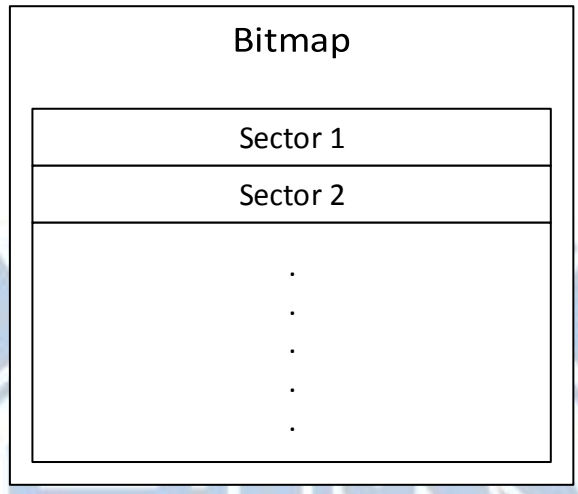

圖 10 Bitmap 方塊圖

3.1.2 Bitmap Cache

在 bitmap-based hardware allocator 中會 cache 其中 4 sector 進來，預設為整個 bitmap 的前四個 sector，當有 allocate request 進來時會啟動整個搜尋電路，每個 clock cycle 會送進 32 bits(sub sector)到 Four Bit Searcher Circuit 電路當中，圖 11 為 bitmap cache 的架構，當要做 allocation 時，會從第一組 sector 的第一個 sub sector 開始做搜尋，搜尋到第 32 個 sub sector 時會判斷下一個 sub sector，也就是第二組 sector 的第一個 sub_sector 是否為連續的，此時會去比較 sector tag，若是連續的 sector，則會接著搜尋，若沒有連續就會載入存在 main memory 的 bitmap 中對應的 sector 並接著搜尋，重覆以上動作直到搜尋完成。當要 Free 時，就會去比較要 free 的 address 是否存在 cache bitmap 四個 sector 當中，如果 miss 就會從 Main Memory 的 bitmap 載入對應的 sector，接著完成後續的動作。

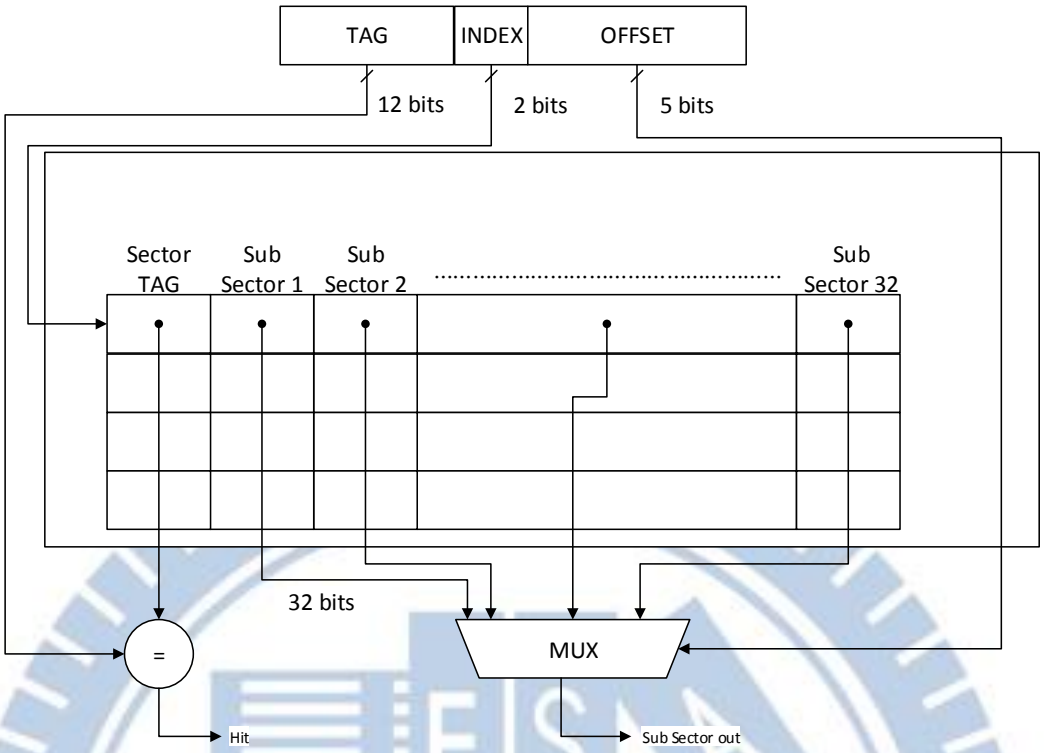

圖 11 Bitmap Cache

3.1.3 Four Bit Searcher Pipeline

在 bitmap 中負責找未使用空間的主要電路模組是 four bit searcher，每次操作是根據 sub sector 的其中 4 個 bits 以及 input 的資料去統計出目前最大的連續個 0 是落在此 sub sector 的哪個位置，會使用一次統計 4 個 bit 的原因為每個 sub sector 為 32bits，因此需要切成 8 組 Four Bit Searcher pipeline stage，完成 sub sector 的其中 4bit 來達到快速搜尋的功能，每一層 stage 有自己對應的 stage index，例如 stage0 就是負責一個 sub sector[0:3]，stage2 就是負責 sub sector[8:11]，不用 8 bits 的原因是因為 8bit 等於會有 256 種組合，每種組合都有不同的結果，會造成電路面積大量增加，所以選擇 4 bits 來實作，4 bits 會有 16 種不同的排列組合，表 1 為 Four Bit Searcher 紀錄資訊，簡單來說，Four Bit Searcher 就是一張查找表，每次要搜尋的資料會根據輸入的資料去表格裡面搜尋對應的輸出。

表 1 Four Bit Searcher 紀錄資訊

資料	描述
Leading Zero	統計 sub sector 搜尋從第一個 bit 到目前為止，一開始連續個 0 的數量。
Leading Flag	搜尋過程中，此 sub sector 開頭連續個 0 是否已經中斷，0 代表中斷，1 代表連續。
Trailing Zero	統計搜尋過程中連續個 0 的數量。
Trailing Zero Idx	統計搜尋過程中，連續個 0 的位置。
Continue Flag	紀錄目前的 4bit 資料是否跟下一筆資料為連續，1 代表連續，0 代表不連續。
Max Zero	每當 Trailing Zero 中斷時會更新是否這次的 Trailing Zero 有大於原先的 Max Zero 並做更新。
Max Zero Idx	每當 Trailing Zero 中斷時會更新是否這次的 Trailing Zero 有大於原先的 Max Zero 並做 Max Zero Idx 的更新。

Four Bit Searcher Pipeline 是由 8 組 four bit searcher 所組成的 pipeline stage，如圖 12 Four Bit Search Pipeline，每一層 stage 負責計算 sub sector 的其中 4 bit，例如:stage1 是計算 sub sector[0:3]，Stage2 是計算 sub sector[4:7]，以此類推，所以填滿這 8 個 stage 後，也就是 8 個 clock cycle 後，search information controller 就會開始統計是否有符合 allocate size 的空間了，統計的方式跟 Four Bit Searcher 類似，需要統計每個 sub sector 之間的 leading zero、trailing zero 以及 max zero 等等，因為 four bit searcher circuit 所輸出的資料就是整個 sub sector 所統計下來的資訊，所以根據 four bit searcher circuit 的輸出，search information controller 就能統計連續的 sub sector 中是否有符合的連續空間可以供使用。

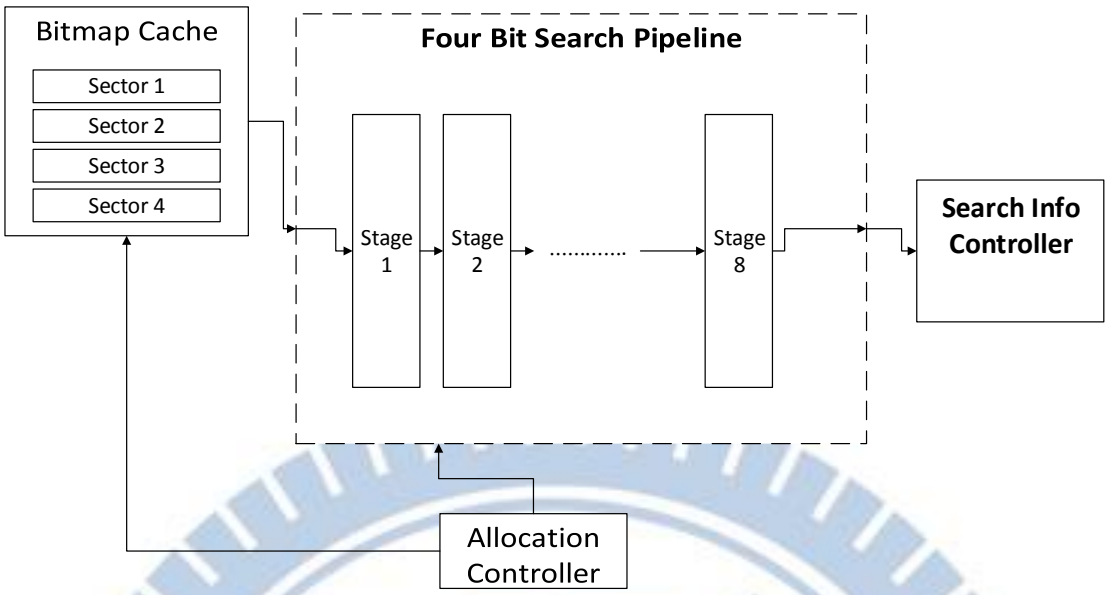

圖 12 Four Bit Search Pipeline

那在搜尋連續的 sub sector 過程當中，符合 allocate size 的連續的 0 有可能會落在三種地方，以下針對這三種地方做說明。第一種為符合 allocate size 的地方可能落在當前的 sub sector 中的 trailing zero 加上現在的 sub sector 的 leading zero，如圖 13 的 case 1，深色區塊為連續 0 的位置，2 第二種為符合 allocate size 的地方可能落在當前的 sub sector 中的 maximum zero 如 case 2，第三種為符合 allocate size 的地方可能落在當前的 sub sector 中的 trailing zero，只要以上三種組合的其中一種 size 符合 allocate size，即代表搜尋成功，並輸出相對應的 allocate address，前面有提到，為了節省搜尋的時間，會在背景時間事先填滿 8 個 pipeline stage，所以當有 allocate request 進來時，即可開始判斷是否有適合的 size。

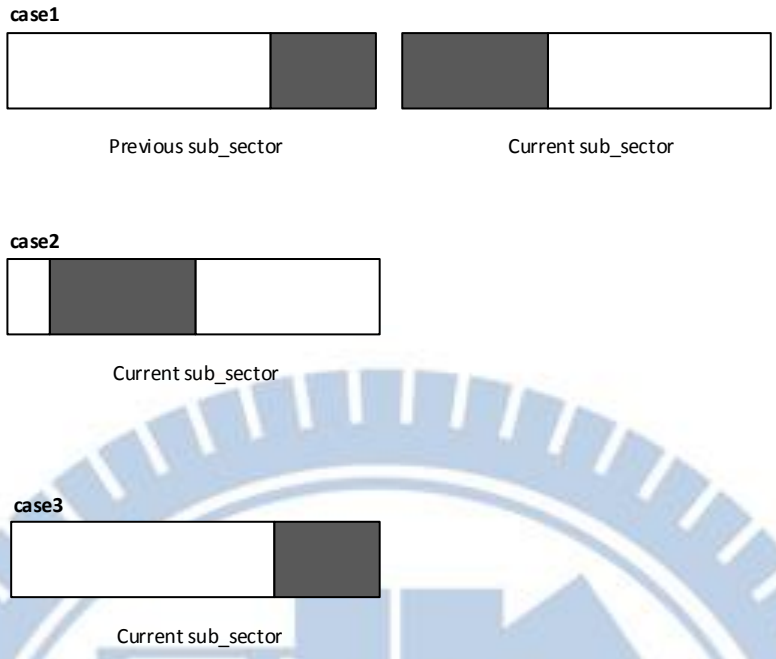

圖 13 符合配置大小的位置

3.1.4 Bitmap Mark

當 search 完成時會開始後續的作業 mark，也就是把分配出去的記憶體所對應到的 bitmap 從 0 標記為 1，標記的單位為一個 clock cycle 標記一個 sub sector，當找到 allocate size 時會紀錄 allocate address 的 start address 以及 end address，標記的過程中會從 end address 標記到 start address，則可以分為以下四種情況，第一種為 end address 出現在 sub sector 的某一點，則把 end address 左邊的值全部填為 1，剩下右半部的值都不變，如圖 14 標記位置 case 1，第二種為 start address 出現在 sub sector 的某一點，則把 start address 右邊的值全部填為 1，剩下左半部的值都不變，如 case 2，第三種為 start address 以及 end address 都出現在 sub sector，則把 end address 到 start address 中間的值全部填為 1，剩下的值都不變，第四種為 start addr 以及 end address 都沒出現在 sub sector，表示這 sub sector 落在 start address 以及 end address 的中間，值全部都得填為 1。

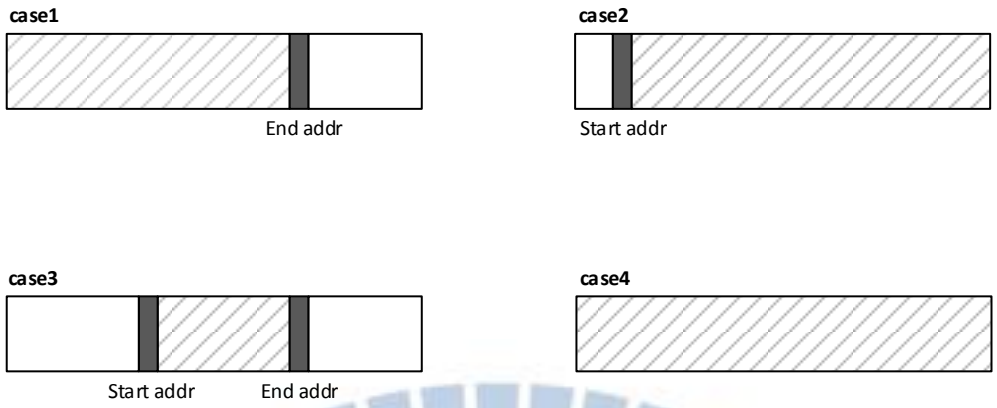

圖 14 標記位置

3.1.5 Hardwar bitmap allocator FSM

圖 15 為作 memory allocation 完整的 finite state machine，簡單的流程為當有 allocate request 進來時，會從 bitmap cache 搜尋，搜尋符合要求的空間時，就會輸出對應的記憶體起始位址，然後在背景時間做標記的動作才不會影響處理器的效能，標記完成後即回到 IDLE 等待下次的 request，有了基本流程概念後，以下是對每一個 state 做更詳細的解說。

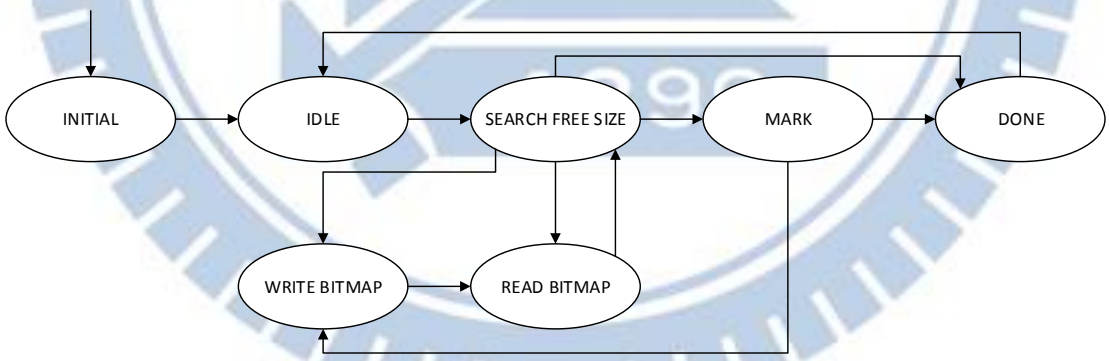

圖 15 Allocation of FSM

1. INITIAL:初始化整個 Allocate Free Controller，需要 2 個 clock cycles 初始時間，初始化完會進入下一個 state，IDLE。
2. IDLE:在這一個 state 時，有兩件事情需要做，第一件是若沒有 allocate request 進來時，在 IDLE 這 state 會事先花 8 個 clock cycles 的時間填滿 8 個 pipeline stage 的 four bit searcher，這樣當有 allocate request 進來時，

若 allocate size 屬於較小的 size(ex:32 bytes)且第一個 sub sector 不全為 1 時，就可以馬上輸出對應的 allocate address，第二件是當有 allocate request 進來時會馬上進入 SEARCH_FREE_SIZE state 進行搜尋。

3. SEARCH_FREE_SIZE:會啟動 four bit searcher 的 8 個 pipeline stage，且不斷地送新的 sub sector 到 stage 裡面，搜尋的過程中會一直統計目前最大的連續個 0 落在哪個位置且有幾個連續個 0，每一個 sector 有 32 個 sub sector，所以當這 32 個 sub sector 都搜尋完且沒找到適合的大小時，會去比對下一個 sector 跟目前的 sector 是否為連續的 sector，若為連續，則會繼續搜尋，若為不連續，則會去讀下一個連續的 sector 進來，此時要先判斷原先要被更換的 sector 是否為髒的(髒的意思是此 sector 的內容有被修改過)，若為髒的需先把此 sector 寫回 memory，若為乾淨的，就直接讀新的 sector 進來繼續搜尋，直到搜尋到適合的大小為止，若搜尋成功時，會送出 allocate finish 以及對應的 allocate address 訊號出去，代表搜尋完成，接著進入到 MARK state。
4. WRITE_BITMAP:此 state 是要把要原先的 sector 寫回記憶體，會送出 memory write request 給 memory controller，並將要寫入的資料寫進記憶體，寫完之後會進入 READ_ BITMAP。
5. READ_ BITMAP:此 state 要從記憶體讀下一個連續的 sector，會送出 memory read request 給 memory controller，並將要讀的 sector 寫入 cache 當中，讀完之後會進入 SEARCH_FREE_SIZE 繼續搜尋符合 allocate size 的空間。
6. MARK:將分配出去的記憶體所對應的 bitmap 做標記，每一個 bit 都代表一塊記憶體空間，所以原先分配前的記憶體空間所對應的 bit 應從 0 標記成 1，標記的順序為從尾巴標記到頭，當標記的過程中若是跨不同的 sector 的話，則需讀對應的 sector，此步驟則跟 SEARCH_FREE_SIZE state 相同，若為髒的則需先寫回記憶體在讀需要的 sector 進來繼續做標

記，直到標記完成後，會進入 DONE state。

7. DONE: 當搜尋及標記這些動作完成時，會在此 state 把部分的訊號做初始化，以便於下次搜尋

圖 16 為作 memory free 完整的 finite state machine，簡單的流程為當有 free request 進來時，會先分析此要被標記的 bitmap 是否在 bitmap cache 裡面，分析完後就開始做標記，標記完就回 IDLE 等待下一次的 request，因為有一部分的 state 跟 memory allocation 的 state 一樣，就不重複敘述，所以以下是對部分的 state 做更詳細的解說。

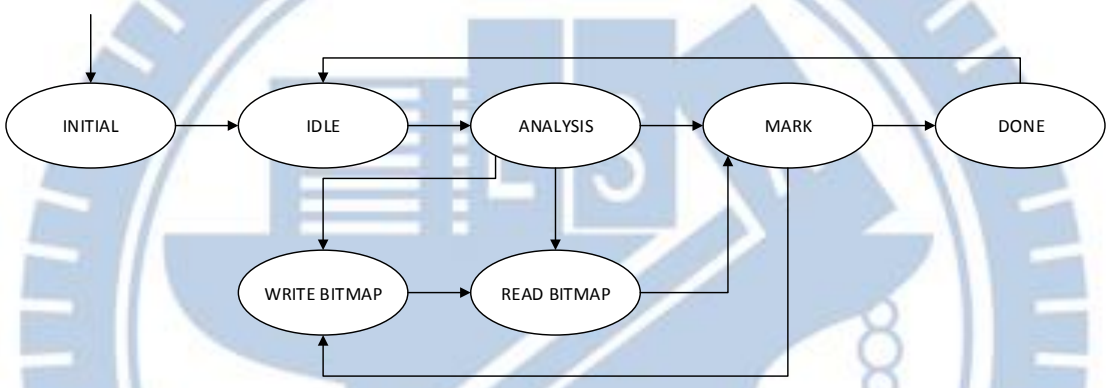

圖 16 Free of FSM

1. IDLE:此 state 為等待 free request 進來，若 allocate request 跟 free request，同時進來的話，allocate request 優先權會高於 free request，若只有 free request 的話則會進入 ANALYSUS state。
2. ANALYSIS:在標記的時候為了跟 allocate 標記的方式一樣以達到簡化電路，所以一樣要從尾巴標記到前頭，所以要分析此釋放的記憶體位址是否在 bitmap cache 裡頭，不在的話則須判斷要更換的 sector 是否須先寫回記憶體在讀所需的 sector 進來，此步驟跟 allocate 的 SEARCH_FREE_SIZE state 相似，分析完之後則進入到 MARK state。
3. MARK:將要被釋放的記憶體空間所對應的 bitmap 做標記，標記的順序一樣為從尾巴標記到前頭，將要被標記的 bitmap 從 1 標記為 0，標記完成後則下次可以重複使用這塊記憶體空間，接著到 DONE state。

3.2 Bitmap-based Hardware Allocator 分析

我們提供了用硬體實作的 bitmap-based hardware memory allocator，利用硬體平行搜尋的優勢提供了快速搜尋 free space 的功能，在第六章的實驗結果當中也有不錯的效果，而因為是使用 bitmap 管理的概念，在記憶體 fragmentation 的情況也不用擔心，external fragmentation 的部分因為在釋放記憶體之後會把 bitmap 所對應的位置填成 0，而相鄰的區塊若都為 0 即為連續的空間，所以不會因為記憶體使用久了之後就會造成嚴重的破碎情形，而 internal fragmentation 最糟糕的情況就是多分配將近 1 個 block 的大小，以 JAIP 為例，當有 allocate request size 為 129 byte，則須分配 5 個 block，也就是 160 byte，造成有 31 byte 的浪費。

而不採用此方法作為 JAIP 的 allocator 的原因是，bitmap-based hardware allocator 在數據分析看來，搜尋時間會隨著 allocate size 變大而增長，但這並不是我們想要的，所以在第四章中我們實作另外一種 allocator 並且得到更好的搜尋效果，且不論 allocate size 多大，都會維持一定的 performance。

第四章 Newlib-based Hardware Allocator 設計

在 2.2.4 中有提過 newlib allocator 的演算法，軟體的找尋方式為先從對應的 bin 開始搜尋，若沒有找到會往 size 較大 bin 逐一找尋直到找到為止，找到後需要更新資料結構後才會回傳 allocate address，然而如果用硬體實作此演算法，因為能平行對每一個 bin 搜尋，而搜尋完能馬上回傳 allocate address，資料結構的更新可以等到電路閒置時再來更新，free 的動作也是，可以先存入 free buffer，等到電路閒置時在做，以下為 newlib-based hardware allocator 的架構。

4.1 Newlib-based Hardware Allocator 架構

首先來在看一下此 allocator 如何管理釋放後的空間，如圖 17，每個被釋放或者是分配的空間我們都稱為 chunk，chunk 的資料結構跟原先的一樣，可以參考圖 7 以及圖 8，釋放後的 chunk 會以 chunk size 做分類，同一個 bin 裡面的 chunk 則會用雙向鏈結串列串接在一起，而在第 64 bin 以後原先的演算法是從小排到大，為了能快速平行搜尋，在本論文中我們把它改成從大排到小，在搜尋每個 bin 的第一個 chunk 時如果沒有符合就代表後面的 chunk 一定都不符合。

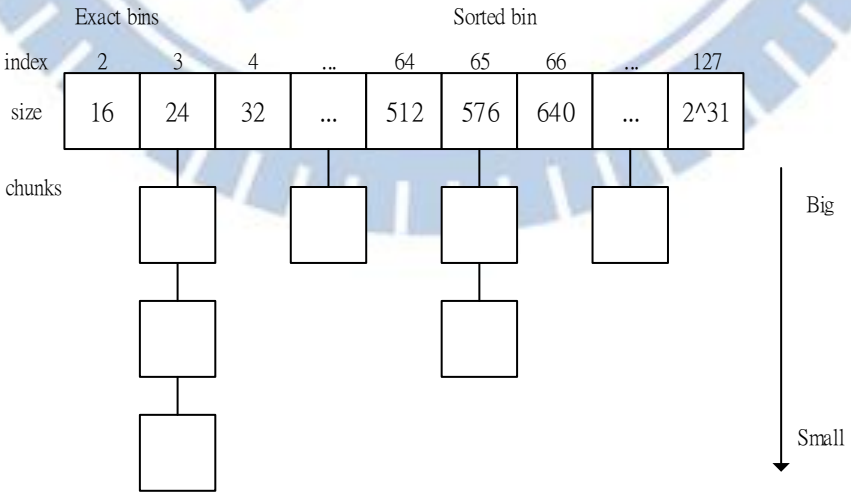

圖 17 Newlib-based Hardware Allocator bin 架構

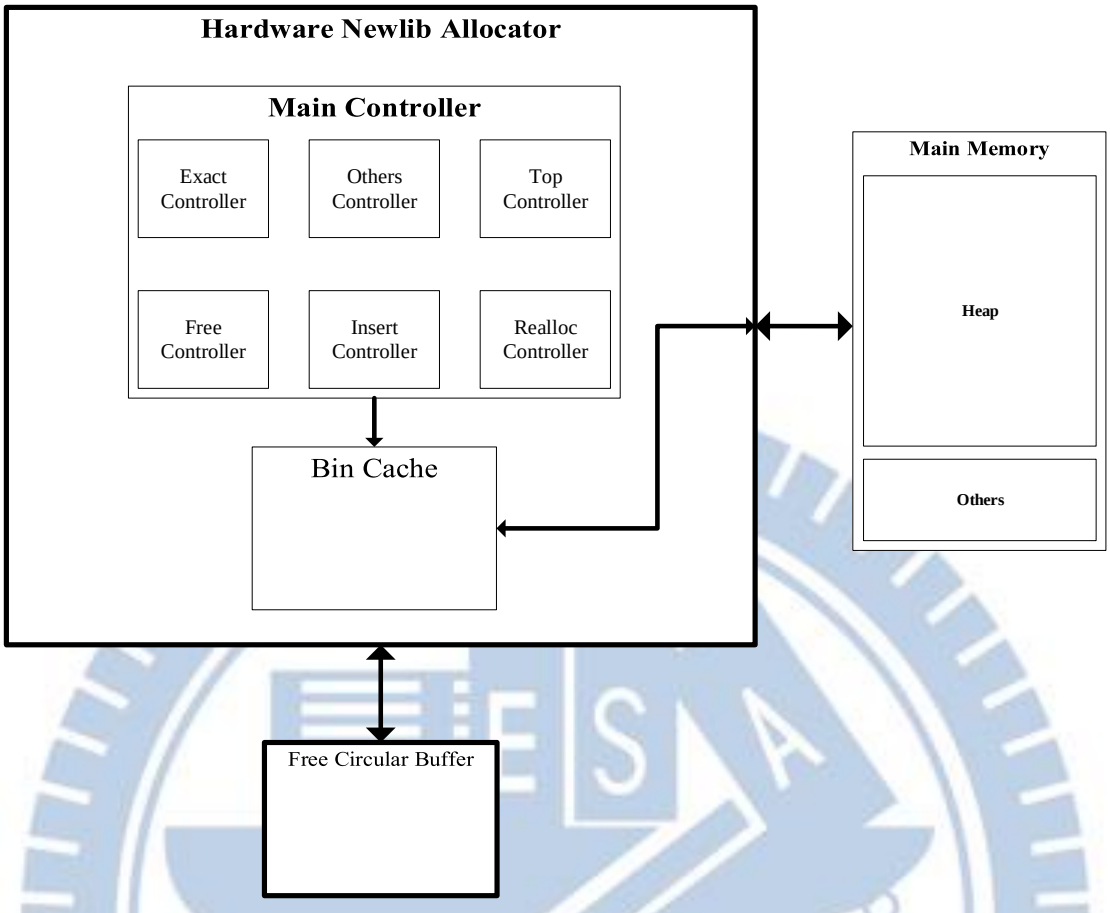

圖 18 Newlib-based Hardware Allocator 方塊圖

接下來介紹整個 newlib-based hardware allocator 的實作演算法，圖 18 為整個架構的方塊圖，整個 bin 的結構以及 chunk 的資訊都是存在 main memory 的 heap 當中，而在我們的設計中，我們會把每一個 bin 的第一個 chunk 存在 bin cache 當中，而第一個 chunk 在 exact bin 中會是最近一次放入進去的，而在 sorted bin 裡一定是 bin 裡面最大的 chunk，如此一來在做平行搜尋時只要搜尋 bin cache 裡面的 chunk 就能得知是否有符合 allocate request 的需求了，另外我們會用一個 128bit 的暫存器記錄每一個 bin 當中是否有 chunk，如圖 19，0 代表沒有，1 代表有，而 main controller 為 newlib-based hardware allocator 主要的 controller，當有 allocate request 或者是 free request 時，會先啟動 main controller，而 main controller 判斷完條件後會驅動裡面的 sub controller 完成動作，後面會對這一部分做更詳細的解說，而 free circular buffer 是先把要 free 的資訊暫存 buffer 裡面，並把控制權還給處理器，待 main controller 為閒置時再來做 free 的動作。

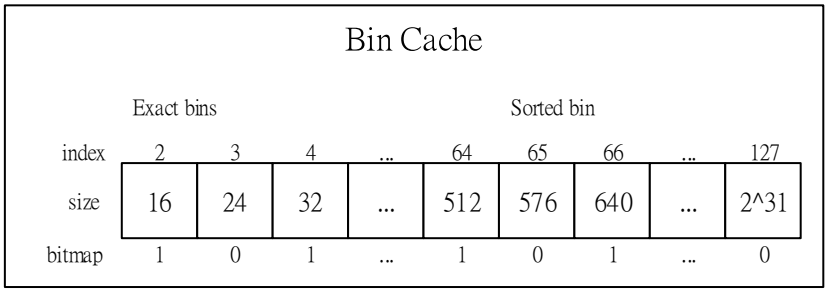

圖 19 Bin cache

4.2 Newlib-based Hardware Allocator 設計

整個 newlib-based hardware allocator 都圍繞在 main controller 上，而 main controller 會因為分配的條件不同而去驅動各個 sub controller，以下會介紹 main controller 以及其他幾個 sub controller 的運作流程。

4.2.1 Main Controller

當有 allocate request 時，會先去搜尋 bin cache 中對應 allocate size 的 bin，因為 bin cache 裡面一定會存著該 bin 裡面最大 size 的 chunk，所以對應的 bin 如果沒有就不用繼續往下找了，而是往更大的 bin 去找，而我們有用一個 cache bin bitmap 去記錄每一個 bin 是否有 chunk，所以我們就去看比 allocate size 更大的 bin 所對應的 bitmap 是否為 1，有的話就一定符合比 allocate size 更大的空間，如果 cache bin 都沒有我們需要的話就會從 top 去切割。而當有 free request 進來時，會去檢查上下連續的 chunk 是否要合併，合併完再重新插入到對應的 bin 裡面，或者需要合併到 top 中。圖 20 為 main controller 的 finite state machine(FSM)，接下來會對每一個 state 逐一解釋。

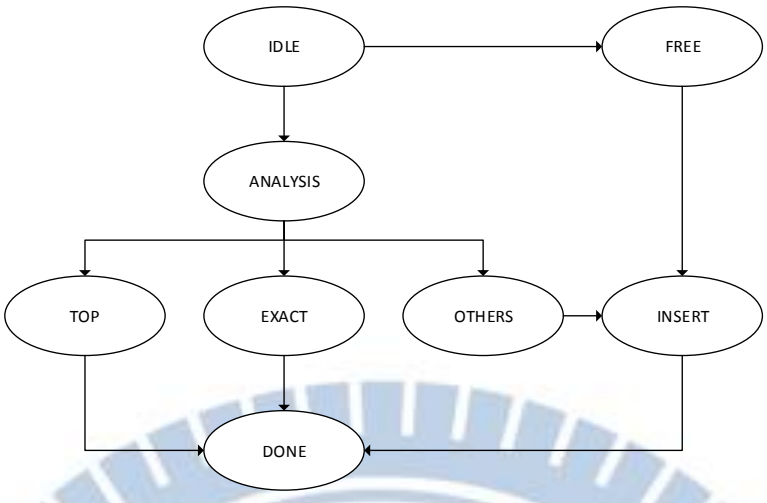

圖 20 Main Controller FSM

1. IDLE: allocator 電路閒置的時候，此時若有 allocate request 或 free request 時即會觸發此電路，並開始動作。
2. ANALYSIS:分析 allocate size 所對應的 bin 裡面是否已合適的 chunk 可以分配，沒有的話則會找尋更大的 bin，而若有找到的話會則會看找到的 chunk 是否需要做 split，split 的條件為找到的 chunk size 大於 allocate size 加 16 byte，這 16 byte 也就是 chunk 的最小 size，需要 split 的話就會跳到 OTHERS state，不需要的話則會跳到 EXACT state，如果都沒在 bin cache 找到的話代表 bin 裡面沒有符合我們要的 chunk，此時就會到 TOP state 切割一個新的 chunk。
3. EXACT:啟動 EXACT controller，結束後回 DONE。
4. OTHERS:啟動 OTHERS controller，結束後因為需要處理 split 的 chunk，所以會到 INSERT state。
5. TOP:啟動 TOP controller，結束後回 DONE。
6. FREE:啟動 FREE controller，結束到 INSERT STATE。
7. INSERT:啟動 INSERT state，結束後回 DONE。

4.2.2 TOP Controller

圖 21 為 TOP controller 的 FSM，功能為從 top bin 當中切一塊新的 chunk 作使用，並更新 chunk 的大小以及更新 top 的 size，以下對每個 state 逐一解說。

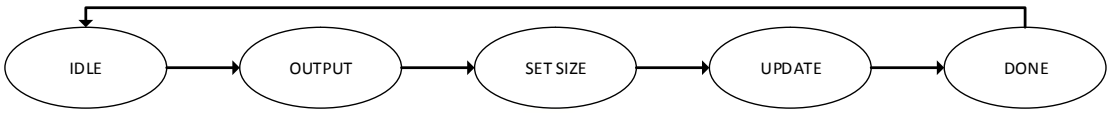

圖 21 Top controller

1. IDLE: controller 閒置，等待 main controller 觸發。
2. OUTPUT:此時就會回應處理器的 allocate request，並回傳 allocate address，剩下的動作皆在背景執行。
3. SET SIZE:此時分配出去的 chunk 為從 top 切割出去的新 chunk，而 chunk 並沒有任何資料，所以要在 chunk 的第二欄寫入 chunk 的大小。
4. UPDATE:更新 top 的 size 以及 address。
5. DONE:回應 main controller 此 controller 已執行完畢。

4.2.3 EXACT Controller

圖 22 為 EXACT controller 的 FSM，會從 bin 裡面分配並不用做任何的 split，但要更新 bin 裡的資料結構，以下對每個 state 逐一解說，若前面有重複功能的 state，則不多作說明。

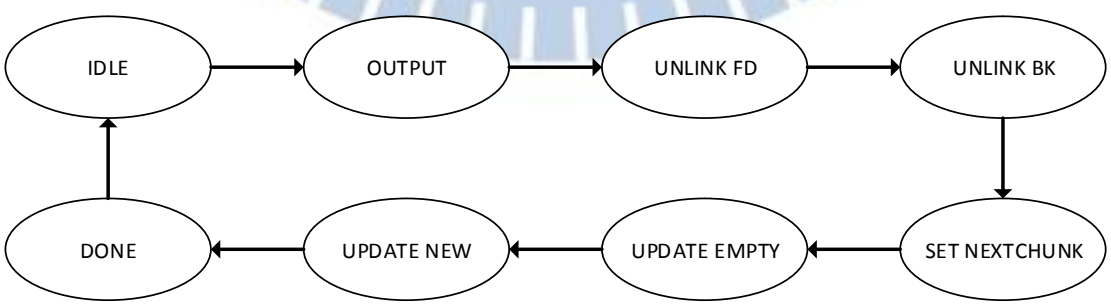

圖 22 EXACT controller FSM

1. UNLINK FD:因為分配出去的 chunk 原先是存在 bin 裡面並用雙向鏈結

串列串接著，所以當從裡面分配 chunk 出去後，必須維持原先的資料結構，此 state 為更新分配出去的 chunk 的 forward chunk。

2. UNLINK BK:與 UNLINK FD state 功能相同，為更新分配出去的 chunk 的 back chunk。
3. SET NEXTCHUNK:每一個 chunk 中的第二欄為 chunk size，而這一欄中的 LSB 為 previous in use bit，所以當此 chunk 分配出去後要更新下一個連續 chunk 的這一個 bit 由 0 改成 1。
4. UPDATE EMPTY:當從 bin cache 分配一個 chunk 之後，如果這一 chunk 為原先 bin 裡面唯一一個 chunk，則要把 bin cache 以及 bin bitmap 裡面的資料清除。
5. UPDATE NEW:若 chunk 裡面還有其他的 chunk 可以分配的話，則需要從記憶體載入新的 chunk 的資訊並更新 bin cache。

4.2.4 OTHERS Controller

圖 23 為 OTHERS controller 的 FSM，會從 bin 裡面分配並需作 split，並更新 bin 裡的資料結構，FSM 大多與 EXACT controller 相似，比較特別的地方為多一個 state 為 SET CHUNK SIZE，會多此 state 是因為因為是從比 allocate size 還要大的 chunk 分配，所以 split 後會有兩個 chunk，所以要更改 chunk size，而其他 state 大多與 exact controller 相似，所以這邊不多作說明。

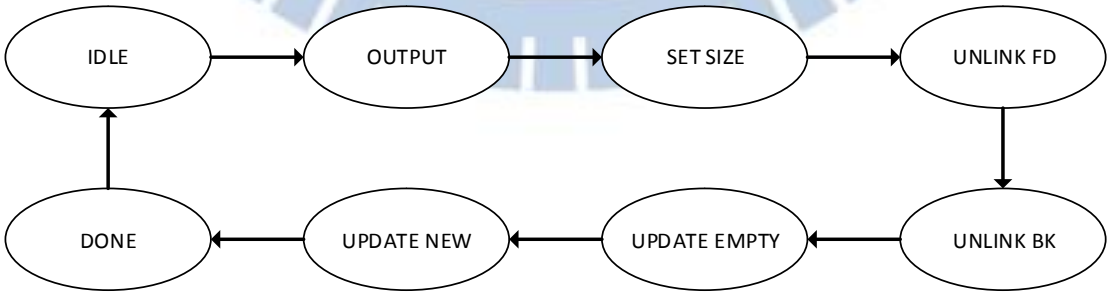

圖 23 OTHERS controller FSM

4.2.5 INSERT Controller

圖 24 為 INSERT controller 的 FSM，主要功能為將 FREE controller 處理完的 free chunk 以及 OTHERS controller 所 split 出來的 chunk 插入至 bin 的 controller，以下對每個 state 逐一解說。

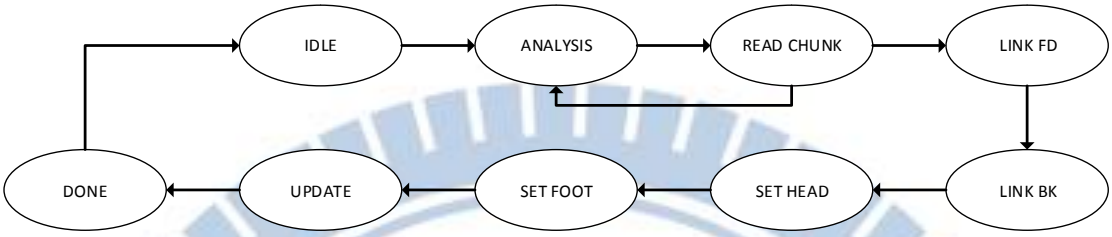

圖 24 INSERT controller FSM

1. ANALYSIS:如果是 exact bin 的話則 bin cache 會存著最近一次 free 或者是 split 的 chunk，如果是 sort bin，bin 裡面的 chunk 是根據 size 由大排到小，此 state 會分析目前要插入的 chunk 是否找到適合的位置，沒有的話則會讀取下一個連續的 chunk，所謂合適的位置能分為三種，第一種為插入空的 bin，第二種為插入兩個 chunk 中間，第三種為該 bin 裡面最小的 chunk。
2. READ CHUNK:從 memory 讀取目前 chunk 的 forward chunk 資訊。
3. LINK FD:更新要插入 chunk 的第三欄，並更新 forward chunk 的資訊。
4. LINK BK:插入 chunk 的第四欄，並更新 back chunk 的資訊。
5. SET HEAD:插入的 chunk 資訊更新到 memory。
6. SET FOOT:更新下一個連續 chunk 中的第一欄為，也就是 previous size。
7. UPDATE: chunk 若是插入空的 bin 或者是最大的 bin 又或者是 exact bin 的話，則都需要把此 chunk 的資訊更新到 cache 當中。

4.2.6 FREE Controller

圖 25 為 FREE controller 的 FSM，free request 來自 free buffer，因為安全性的考量，會對 free space 先作初始化的動作，也就是把記憶體的值初始化為 0 的

動作，之後會檢查上下連續空間的 chunk 是否也是 free，並決定是否要合併成一個 chunk，或者下一個 chunk 是 top 的話就直接併到 top 裡，以下對每個 state 作解說，若前面有重複功能的 state，則不多作說明。

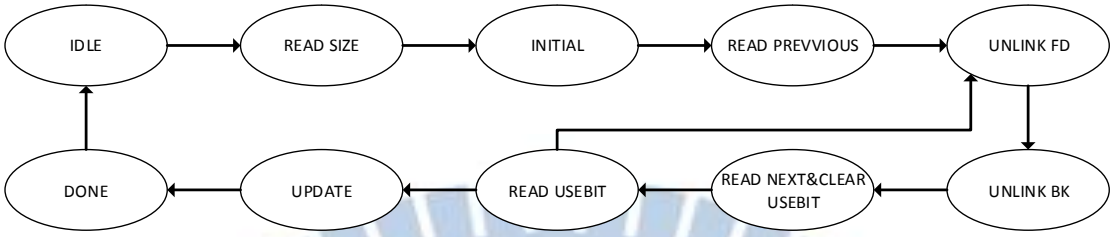

圖 25 FREE controller FSM

1. READ SIZE:根據 free address 得知此 chunk 的起始位置並讀取此 chunk 之 size。
2. INITIAL:根據 chunk size 開始作初始化動作，初始化完後會根據 chunk size 的 LSB(previous in use bit)得知是否要合併上一個連續 chunk，若需合併的話則需讀取上一個連續 chunk 的資訊。
3. READ PREVIOUS:讀取上一個連續 chunk 的資訊。
4. READ NEXT & CLEAR USE BIT:讀取下一個連續 chunk 的資訊並把第二欄的 chunk size 中的 LSB 清除成 0。
5. READ USEBIT:要得知下一個連續 chunk 是否為 free 必須讀取下下個連續個 chunk 的 previous in use bit 才能得知。
6. UPDATE:若有合併的 chunk，且此 chunk 是在 cache 裡面的話須更新 bin cache 裡的資訊。

4.2.7 REALLOC Controller

圖 26 為 REALLOC controller 的 FSM，當有 reallocate request 時會啟動這 controller，功能是為原先的記憶體重新分配，若重新分配的記憶體比原來的還要小的話，只需做 split 及可，而若重新分配的記憶體比原本大還要大的話就會重新找一塊符合 size 的 chunk 後並把原先的記憶體資料複製過去，以下對每個 state

作解說，若前面有重複功能的 state，則不多作說明。

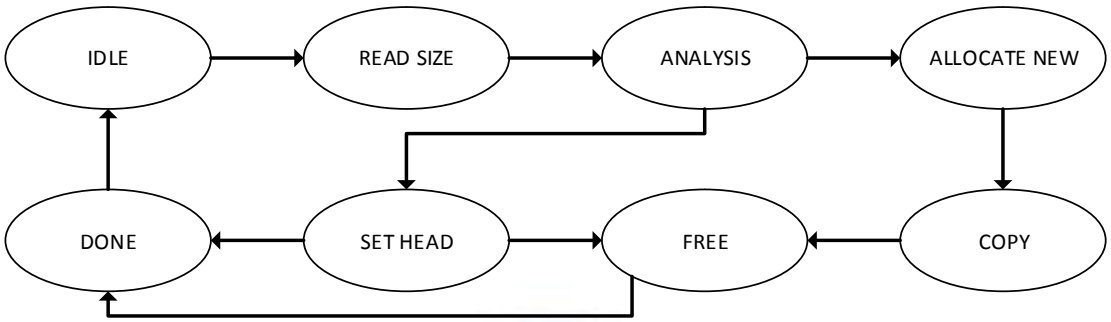

圖 26 REALLOC controller FSM

1. ANALYSIS:分析要重新分配的記憶體是否比原先的記憶體空間大或小，若比較小的話則到 SET HEAD state，若比較大的話會到 ALLOCATE NEW state。
2. ALLOCATE NEW:會把控制權交還給 MAIN controller 並重新找一塊符合重新分配的記憶體空間。
3. COPY:把原先的記憶體資料複製到新的記憶體空間，
4. FREE:會將來自原先的記憶體空間或者是 split 的記憶體空間釋放掉並記錄在 bin 裡面以便下次可以重複使用此空間。
5. SET HEAD:由於要重新分配的記憶體空間比原先的小，所以必須做 split 並修改 chunk size。

4.3 Fragmentation 處理

4.3.1 Internal Fragmentation

Internal fragmentation 就是當分配比 allocate size 還要大的空間造成多餘空間使用不到，但也沒辦法讓其他處理器使用，形成空間上的浪費稱之 internal fragmentation，而在本論文的設計中，當我們分配的空間大於等於 16 byte 時就會做 split，也就是每次在分配記憶體時，最糟的情況就是浪費 15 byte，而 split 的 chunk 就能再次丟入對應的 bin 並重複使用。

4.3.2 External Fragmentation

External fragmentation 是記憶體當中所有可使用空間的總和大於某個 allocate size，但因為不連續而造成空間上的浪費稱之為 external fragmentation，而在本論文中被釋放的記憶體，在 free controller 中會檢查只要是相鄰兩塊的 chunk，就會把這兩塊 chunk 合併成一個較大的 chunk 並重新插入對應的 bin，在第六章中會對記憶體使用量做分析，其使用量和原先的 software newlib allocator 相近。

4.4 Application Programming Interface

本論文提供了 3 種可使用的 application programming interface(API)，如表 2，第一種為 HW_malloc，要求分配一塊大小為 n byte 記憶體空間，第二種為 HW_free，釋放記憶體空間以及 HW_realloc，要求重新分配一塊大小為 n byte 的空間，realloc 的演算法為當要求的空間比原先的空間小時，直接做 split 即可，而當要求空間比原先大時，會直接分配一塊新的記憶體空間並把原先的資料複製過去。

表 2 Allocator API

HW_malloc(int nb)	分配一塊大小為 nb byte 的記憶體空間，並回傳記憶體起始位置
HW_free(pointer)	釋放記憶體位址為 pointer 的記憶體空間
HW_realloc(pointer,nb)	根據原先 pointer 的記憶體空間並重新分配大小為 nb byte 的記憶體空間，並回傳記憶體起始位置。

第五章 與 JAIP Memory Manager 的整合

本章討論如何將第四章所設計的 Newlib memory allocator 整合到 JAIP SoC 中，取代原本 JAIP 中的簡易 memory allocator 電路。

5.1 原本的 JAIP Memory Allocator 設計

在 Java 指令集當中有 new、newarray、multianewarray 以及 anewarray 會觸發 new request 的訊號，而 free 的動作則是由 GC 去蒐集以後不會再被參考到的物件做回收，在原先 JAIP 的架構當中，new 以及 free 的動作都是在 garbage collection controller 裡面執行，如圖 27。GC table 的管理如圖 28，用單向鏈結串列串接在一起，每次分配出去的記憶體空間都會記錄在這張 table，並把 flag 設為 1，當釋放空間是就會設為 0 以便下次可以重複利用，當有 allocate request 時，會先去 GC table 中搜尋是否有可用的空間，而搜尋方法為循序搜尋，table 大小為 2048 個 entry，搜尋一次需 2 個 cycle，也就是最糟的情況有可能是搜尋一個可用的空間需要 4096 個 cycle，而且還不一定搜尋得到，沒有搜尋到的話就會分配一個全新的空間出去。而當在回收記憶體時，會根據回收的位址去 GC table 當中搜尋並把 flag 標記為 1，以便可重複使用，所以最糟的情況也是 4096 個 cycle。

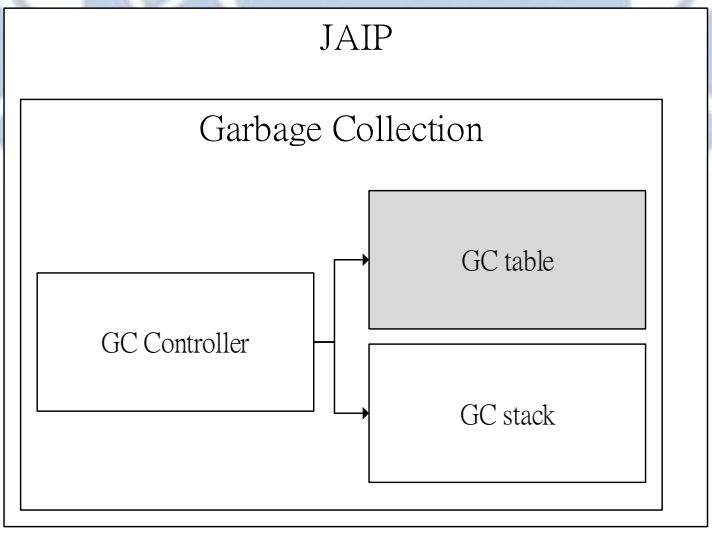

圖 27 Garbage collection 架構

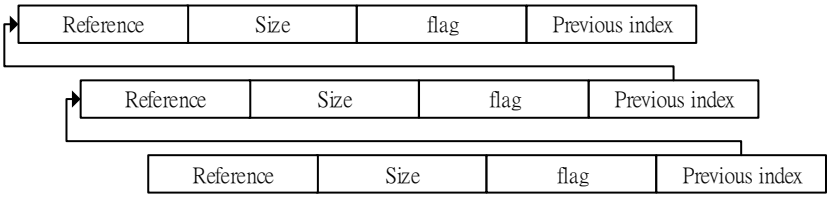

圖 28 GC table 管理

5.2 新的 Memory Allocate Controller 設計

本論文引進 newlib-based hardware allocator 整合進 JAIP 的 GC 中並取名為 Memory Allocate Controller，如圖 29，我們把原先架構中的 GC table 移除掉以及整理相關的控制訊號線後，電路面積明顯少了許多，而在記憶體讀寫的部分，原先有四個元件共同使用一個 bus，分別為 cache controller、context switch controller、native controller、JAIP pointer controller，如表 3，整個 allocator 是包含在 GC 當中，所以本論文新增了第 5 個元件共享 bus 的使用權，mst_lock 式記憶體控制權的 index，每個元件都有屬於自己的 index。整個 heap 的管理全部都由 Memory Allocate Controller 管理，所以當有 allocate request 進來時，會觸發 GC controller 運作，GC controller 會告知 allocator 開始搜尋，搜尋完畢後會將搜尋到的指標 push 至 GC stack 當中，未來當 GC controller 在回收時，會從 GC stack pop 要回收的指標且送入 allocator 中處理。

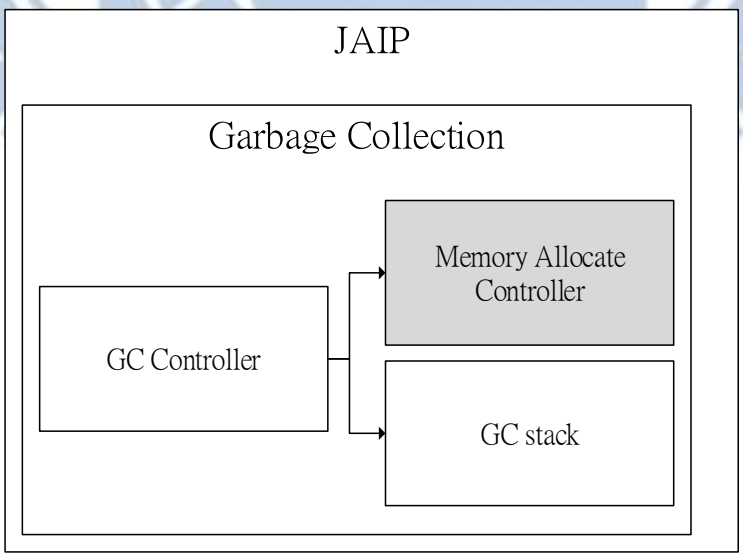

圖 29 Newlib-based hardware allocator

表 3 JAIP Bus Arbiter 資訊

mst_lock index	Description
000	閒置的狀態，表示任何原件都可以使用。
001	Cache controller
010	Context switch controller
011	Native controller
100	JAIP pointer controller
101	GC controller

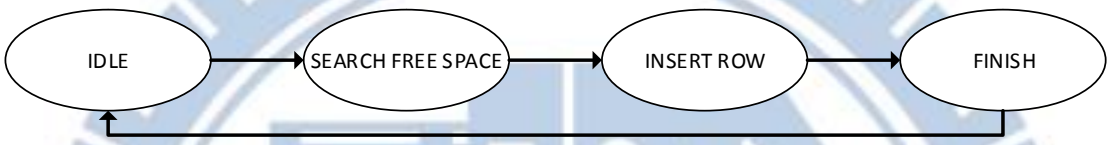

圖 30 Object allocation FSM

接下來會說明整個運作流程，當有 allocate request 進來 GC controller 時，會啟動 allocator 的 controller，並同時搜尋每一個 bin 裡面的第一個 chunk，若都沒有符合即會從 top 分配，而找完後即會回傳位址，所以無論 allocate size 為多少，都能在固定的時間回傳，而在回傳後的背景時間再來更新資料結構，而更新的內容會根據 allocate 的情況而定。圖 30 Object allocation 的 FSM 為接收 heap 分配空間的 controller，以下對這幾個 state 作說明。

1. IDLE:表示 FSM 閒置，若有 heap new request，例如 new 或 newarray，即會跳下一個 state 。
2. SEARCH FREE SPACE:會啟動 memory allocate controller，並開始搜尋能使用的記憶體空間，並回傳記憶體空間的起始位置。
3. INSERT ROW:把分配記憶體的記憶體 push 到 GC stack 當中，到時候等到 method exit 時即會一一 pop 出來做記憶體釋放的動作。

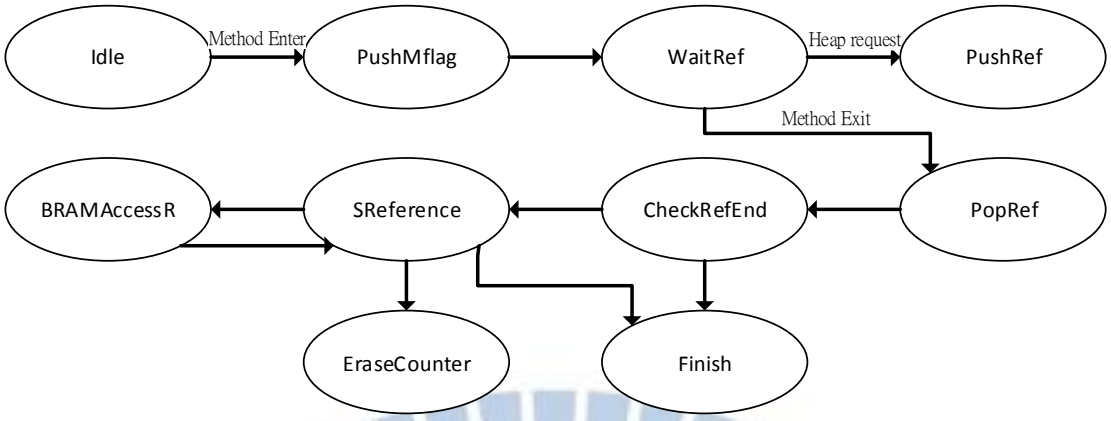

圖 31 GC stack FSM

圖 31 為原先控制 GC stack 的 controller[2]，我們修改了一些地方，當 state 為 PushRef 時，代表在作記憶體分配，會把分配出去的記憶體位址 push 到 GC stack 中，這邊所對應到的是 Object allocation FSM 的 INSERT ROW state，而當收到 method exit 的訊號時，會開始進行回收並把 GC stack 的 reference 一一 pop 出來，並在 SReference state 中把要回收的記憶體位址送入到 free circular buffer 當中並等到 main controller 閒置時再來更新至 bin 裡面。

第六章 實驗結果

6.1 實驗環境

本論文所提出之 JAIP 處理器為移除了 GC 中的 GC table，實作在 Xilinx KC705 Evaluation Kit Platform(XC7K325T)上，如圖 32，工作頻率為 100MHz，開發工具是 Xilinx 公司所提供的 Xilinx Platform Studio(XPS)，程式碼是以 VHDL 及 Verilog 所撰寫而成，用 Xilinx Synthesis Tool 合成，透過 Xilinx ISE Design Suite 提供的 Chipscope Pro Analyzer 與 debugging IP 來驗證我們實作的電路。

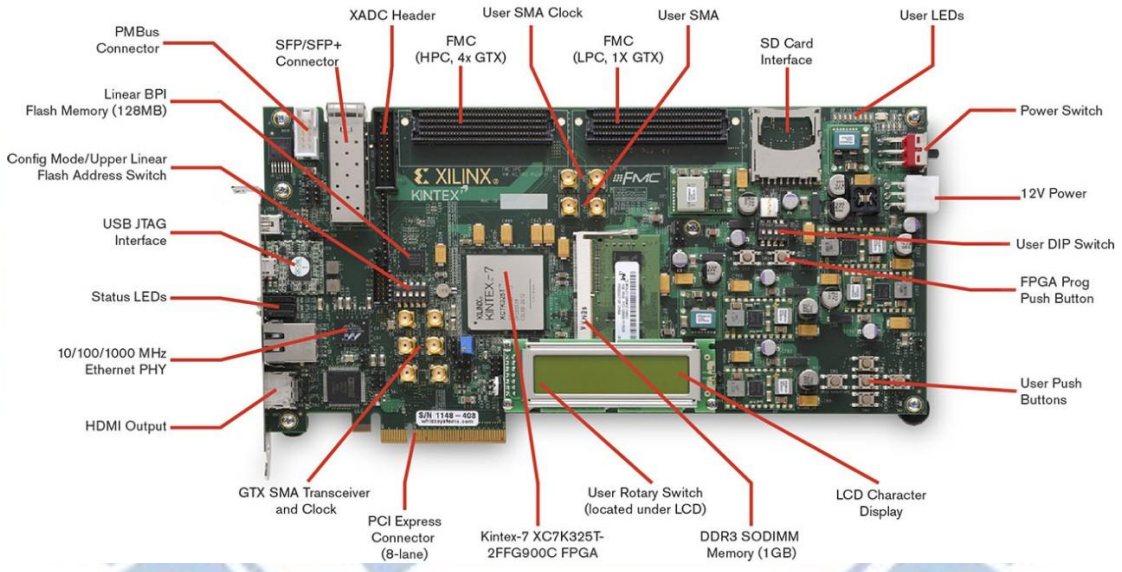

圖 32 Kintex-7 (XC7K325T)

Newlib-based hardware allocator 為可以 reuse 的 IP，並且在 vivado 2017.2 上開發，使用的 FPGA 板為 zedboard，如圖 33，使用的硬體語言為 verilog，而在最後與 software newlib allocator 作效能分析時是使用此 zedboard 上的 zynq 處理器，去做效能評估。

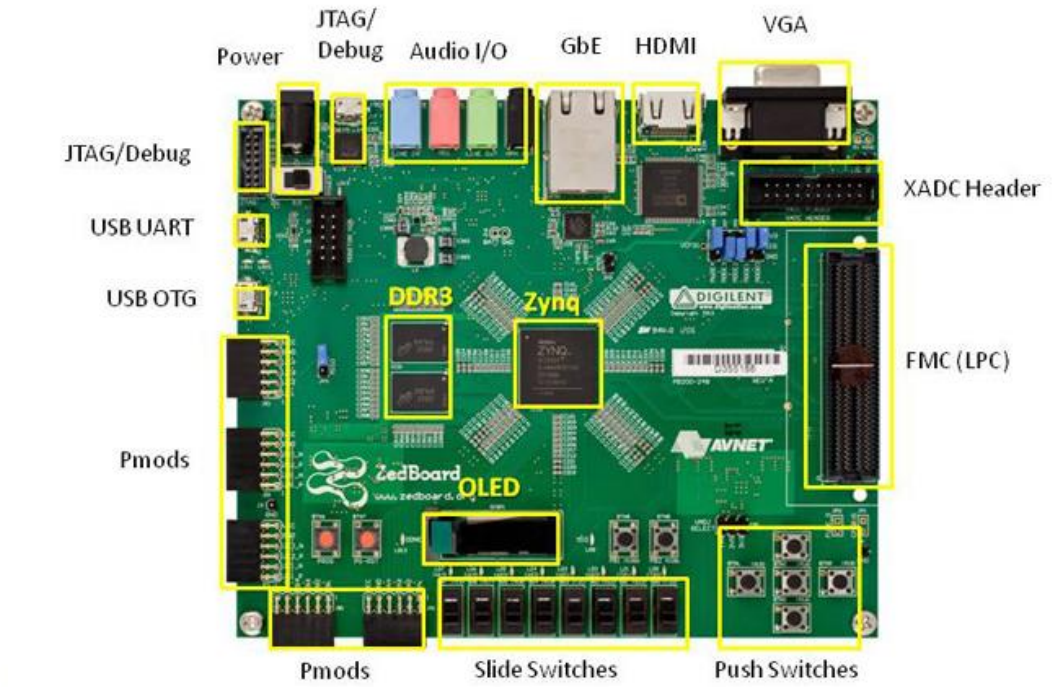

圖 33 Zedboard

6.2 電路資源使用量

在表 4 中為在 KC705 上的 JAIP core 資源使用量，表中包含了 look up table(LUT)，flip-flop(FF)以及 4KB 為單位的 block ram(BRAM)，而此表中是包含了修改後的 GC 所呈現的資源使用量。

表 4 JAIP core 合成資訊

Device : Kintex-7 (XC7K325T)	
Number of LUTs	24204
Number of Flip-flops	10880
Number of 4K BRAM	44
Maximum Frequency	100M Hz

而在表 5 為移除 GC table 前的資源使用量以及表 6 移除後的資源使用量，

由於 GC table 須記錄每次 allocate 的資訊，包括 object reference、object size、allocate flag 以及 previous index，而 free 之後還需一個一個 entry 去 GC table 找當初 new 的 entry，控制相當複雜，所以當移除掉 GC table 後面積少了許多。

表 5 original GC 資源使用量

Device : Kintex-7 (XC7K325T)	
Number of LUTs	523
Number of Flip-flops	664
Number of 4K BRAM	5
Maximum Frequency	100M Hz

表 6 current GC 資源使用量

Device : Kintex-7 (XC7K325T)	
Number of LUTs	133
Number of Flip-flops	241
Number of 4K BRAM	1
Maximum Frequency	100M Hz

在表 7 表 8 中，為 GC Controller 中的 newlib-based hardware allocator 以及 free buffer 的資源使用量，newlib-based hardware allocator 的 LUT 使用較多是因為在實作中會根據 allocate 的情況去啟動不同的 controller，每個 controller 都有自己的資訊要記錄，而裡面總共使用了 1 個主要的 controller 以及 5 個 sub controller，為面積增多的原因。

表 7 Free circular buffer 合成資訊

Device :Zedboard	
Number of LUTs	47
Number of Flip-flops	24
Number of 4K BRAM	1
Maximum Frequency	100M Hz

表 8 Newlib-based hardware allocator 合成資訊

Device : Zedboard	
Number of LUTs	2837
Number of Flip-flops	889
Number of 4K BRAM	1
Maximum Frequency	100M Hz

6.3 效能分析

6.3.1 Memory Allocate Sequence 分析

本論文中，為了呈現我們的設計符合實際的應用，所以使用了幾個實際應用程式並把程式中的 new 以及 free 的 workload trace 記錄下來，我們記錄五個應用程式，第一個為執行 linux wget 指令從網路上下載一份文件(3751 byte 文字檔)，第二個為執行 linux ls recursive 指令，第三個為用 Java virtual machine 執行 hello world 程式，第四個為用 python 執行 hello world，最後一個為執行 xilinx software developer kit(XSDK)。在表 9 為這五個應用程式的資訊，記錄了 allocate 以及 free 的次數，平均 allocate 的大小，還有總共 allocate 的大小以及執行過程中，同時存在的物件最多的數量。

表 9 allocate sequence

	Number of allocate & free	Average allocate size	Total size	Max allocate number
Wget	5446	125 bytes	332.4KB	512
ls recursive	632	2202.3 bytes	695.9KB	53
Java	35036	930.7 bytes	16304KB	562
Python	5940	1047.3 bytes	3110.4KB	321
Xsdk	209980	2231 bytes	234250.8KB	3930

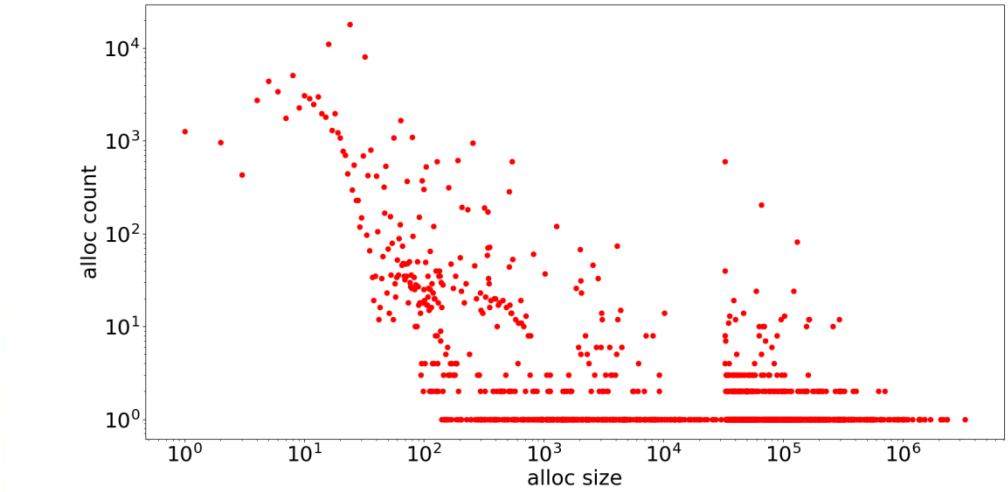

圖 34 Xsdk allocate 分布圖

圖 34 為 XSDK allocate 的分布圖，X 軸 allocate size，Y 軸為 allocate 數量，這是這五個應用程式當中 allocate 次數最多的，所以在此特別列出分布圖。

表 11 我們將比較 software newlib allocator(SW newlib)、bitmap-based hardware allocator(HW bitmap)以及最後放入 JAIP 的 newlib-based hardware allocator(HW newlib)，這三種環境如表 10。

表 10 測試資料資訊

	SW newlib	HW bitmap	HW newlib
Processor	ARM Cotex A9	Hardware IP	Hardware IP
Heap size	32 MB	32 MB	32 MB
Frequency	100 MHz	100 MHz	100 MHz

可以看出 HW newlib 在各個應用程式的 allocate 以及 free 的 workload 當中都明顯比其他兩者快上好幾倍，就連 HW bitmap 也都比 SW newlib 快上許多，但可以看出來當應用程式的 workload 較大時，HW bitmap 的速度會慢下來，原因是因為在經過多次的 allocate 以及 free 的動作後，整個 heap 會相對的破碎，要找到連續的 free space(找到連續個 0)會較不容易，所以可以看到 sequence 較少的測資(例如:ls recursive、wget)輸出都能比 SW newlib 還要快 10 倍左右，但 sequence 較多的測資(例如 XSDK)只快了 2 倍左右，所以這也是最後不選擇 HW bitmap 的原因，但 HW newlib 不會因為 sequence 的多寡而造成效能的輸出不穩，依舊保持一定的輸出，整體時間都能比 SW newlib 快上 15 到 25 倍。

表 11 整體 allocator 效能分析

	SW newlib	HW bitmap	HW newlib
Wget	3930.2 us	404.3 us	217.8 us
ls recursive	663.7 us	43.5 us	25.3 us
Java	24241.1 us	5403.7 us	1401.4 us
Python	4974.5 us	1838.4 us	237.6 us
XSDK	139203.2 us	68352.3 us	6299.4 us

而在表 12 及表 13 中，分別為 allocate 及 free 單獨作效能評估的統計，由於 HW bitmap 以及 HW newlib 的 free 都是先存入 buffer 當中，等到電路閒置時才會執行 free 的動作，所以執行時間上都一樣，在 XSDK 的 workload 中能看到 HW bitmap 的執行時間變慢許多，而 HW newlib 依舊能快速輸出。

表 12 單獨 allocate 效能分析

allocate	SW newlib	HW bitmap	HW newlib
Wget	2485.6 us	349.8 us	163.4 us
ls recursive	393.0 us	37.2 us	19 us
Java	16398.3 us	5053.3 us	1051.1 us
Python	3117.9 us	1779.0 us	178.2 us
XSDK	72954.2 us	66636.8 us	5146.4 us

表 13 單獨 free 效能分析

free	SW newlib	HW bitmap	HW newlib
Wget	1444.5 us	54.5 us	54.5 us
ls recursive	270.6 us	6.3 us	6.3 us
Java	7842.6 us	350.4 us	350.4 us
Python	1856.5 us	59.4 us	59.4 us
XSDK	66111.6 us	1715.5 us	1715.5 us

我們把原本 newlib allocator 的做法由循序搜尋改成平行搜尋且每個 chunk 的排序方法由小到大改成大到小之後，記憶體使用量並不一定會增多，如表 14，且 5 筆測資當中有 4 筆的記憶體使用量是比原先略少一些，原因是我們從每個 bin 裡的最大 chunk 開始搜尋，即使找到的 chunk size 比 allocate size 還要大上許多，但都會做完 split 動作後把 split chunk 重新插入 bin 裡面，而這些 split chunk 就能重新再利用。

表 14 記憶體使用量

	Total size	HW newlib	SW newlib
Wget	332.4KB	73.0 KB	72.0 KB
ls recursive	695.9KB	68.0 KB	69.0 KB
Java	16304KB	105.4 KB	106.4 KB
Python	3110.4KB	573.8 KB	587.1 KB
Xsdk	234250.8KB	16251.7 KB	17963.2 KB

第七章 結論與未來展望

本論文設計了 dynamic hardware memory allocator 電路，dynamic hardware memory allocator 是個 reusable master IP，也提供 slave 介面，可以讓異質多核心的嵌入式系統共享此電路，讓整個記憶體空間能有效地利用，並提供了 malloc、free、realloc 三種 API，以及在背景時間處理了記憶體破碎的問題。

我們也把此電路整合至 JAIP 處理器當中，讓 JAIP 的記憶體管理更加完善，本論文中並沒有跟整合前的 JAIP 比較記憶體分配的效能，原因是因為原本 JAIP 在 new object 時都需透過 GC controller 去不斷去 new 一塊新的記憶體空間，且並沒有做任何的 compaction 的動作，而整合此電路分配器後的 JAIP，會將釋放後的相鄰兩塊記憶體空間做合併，若找尋比 allocate size 還要大的記憶體空間就會做 split，有了以上幾個特性，就能更加利用記憶體空間。

目前 allocator 提供了三種 API，分別為 malloc、free、realloc，而每次在 free 之後的記憶體都已經有初始化，所以 calloc 的功能等同於 malloc，未來可以在新增幾個 API 讓整個 allocator 更加完整，像是 memalign、pvalloc 等等。目前 allocator 的資源使用量用了 2 千多個 LUT 以及將近 1 千個 register，原因是因為我們在 4.2 有提到，我們將不同的分配方式做了分別做了不同的 controller，每一個 controller 都有需要記錄的資訊，所以造成資源使用量增多的原因，如果能將這些 controller 合併成一個即能讓資源使用量降低。

參考文獻

[1] Yan-Hung, Lin *Priority Scheduler Circuit Design for Multi-core Java Processor*, Master thesis, NCTU, August 2017.

[2] Jun-Fu Wang, *The Design of Memory Management on a Java Processor*, Master thesis, NCTU, July 2014.

[3] Bill Venners, *Inside the Java Virtual Machine*, second edition, McGraw-Hill, 1998.

[4] Open-source Newlib C library, available on https://sourceware.org/newlib/

[5] D. Lea and W. Gloger, "A memory allocator," online document available on http://g.oswego.edu/dl/html/malloc.html, modified on Aug. 4, 2000.

[6] Abraham Silberschatz, Peter B. Galvin, Greg Gagne, *Operating System Concepts*, 8th Edition, Wiley, 2010.

[7] Z. Xue and D. B. Thomas, "SysAlloc: A hardware manager for dynamic memory allocation in heterogeneous systems," in *Proceedings of the 25th IEEE International Conference on Field Programmable Logic and Applications (FPL)*, 2015, pp. 1-7.

[8] J. M. Chang and E. F. Gehringer, "A high-performance memory allocator for object-oriented systems," *IEEE Transactions on Computers*, no. 3, pp. 357-366, 1996.

[9] J. L. Peterson and T. A. Norman, "Buddy systems," *Communications of the ACM*, vol. 20, no. 6, pp. 421-431, 1977.

[10] E. Von Puttkamer, "A simple hardware buddy system memory allocator," *IEEE Transactions on Computers*, vol. 100, no. 10, pp. 953-957, 1975.

[11] R. Burns and W. Hineman, "A bit-parallel search algorithm for allocating free space," in *Proceedings of the 9th IEEE International Symposium on Modeling,*

Analysis and Simulation of Computer and Telecommunication Systems, 2001, pp. 302-310.

[12] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review," in *Proceedings of International Workshop on Memory Management (IWMM95)*, Sep. 27-29, 1995, pp. 1-116.

[13] M. Shalan and V. J. Mooney, "A dynamic memory management unit for embedded real-time system-on-a-chip," *in Proceedings of the 2000 ACM International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2000, pp. 180-186.

[14] J. Shi, W. Ji, L. Zhang, Y. Gao, H. Zhang, and D. Qing, "Profiling and analysis of object lazy allocation in Java programs," in *Proceedings of the 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, 2016, pp. 591-596.

[15] M. S. Johnstone and P. R. Wilson, "The memory fragmentation problem: Solved?," in *ACM SIGPLAN Notices*, vol. 34, no. 3, 1998, pp. 26-36.

[16] Jari Nurmi (Ed.), *Processor Design: System-On-Chip Computing for ASICs and FPGAs*, Springer, 2007.