國立交通大學

資訊科學與工程研究所

碩士論文

Java 處理器上的記憶體管理與設計

The Design of Memory Management on a Java Processor

研究生：王俊富

指導教授：蔡淳仁教授

中華民國 103 年 7 月

在 Java 處理器上的記憶體管理與設計

The Design of Memory management on a Java Processor

研究生：王俊富　　Student：Jun-Fu Wang

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

NationalChiaoTungUniversity

inpartial Fulfillment of the Requirements

for the Degree of

Master

in

ComputerScience

July 2014

Hsinchu, Taiwan, Republic of China

中華民國 103 年 7 月

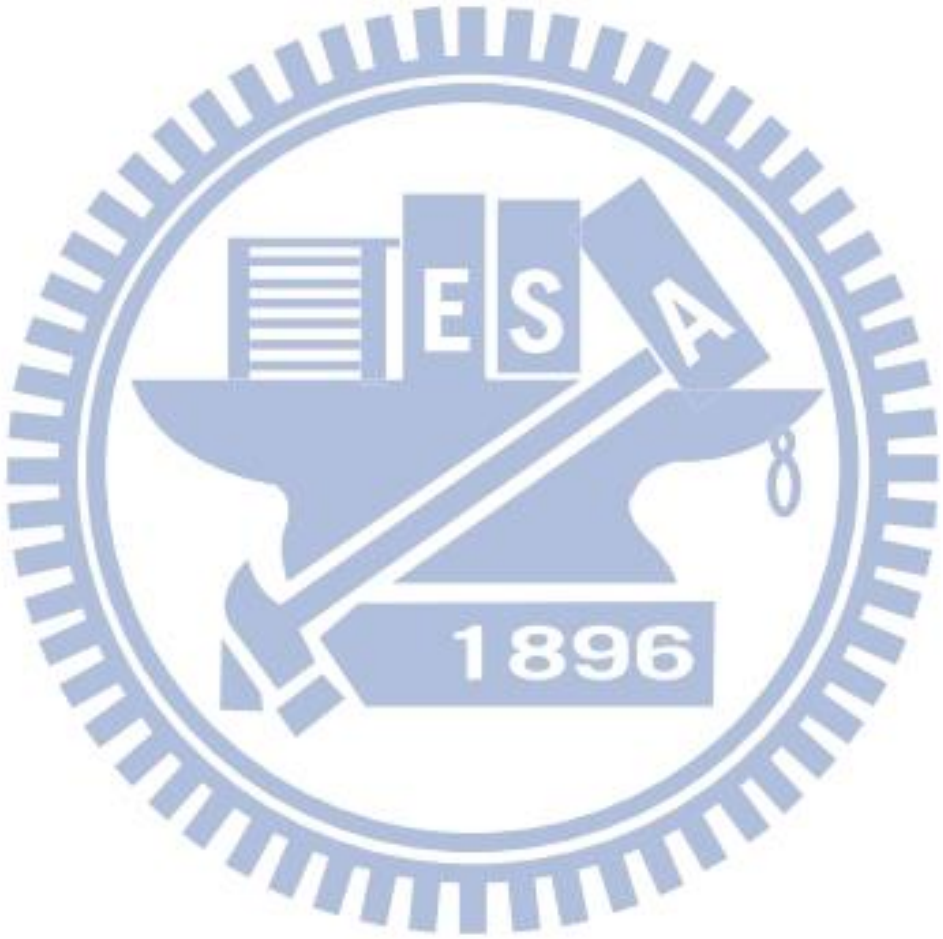
摘要

在本篇論文中設計了兩個管理記憶體的元件，一個是 Object Cache Controller 另一個是 Garbage Collection，藉由此設計不僅可以在效能上得到更好的提升，外加記憶體的使用率可以更好。在 Object Cache Controller 方面, 我們採用了 2-way set associative cache 以及 write-back policy，Object cache controller 會截取讀 object heap 記憶體的要求，先判斷有無事先被快取住，如果有就可以節省花在讀取外在記憶體的時間。在 Garbage Collection 方面我們著重在不影響效能的同時還可以收集垃圾物件，在選取演算法時都以效能為第一考量，因此我們引進 2-port memory manager table 設計來讓回收記憶體和釋放記憶體可以同時進行。

致謝

這篇論文的完成，最需要感謝的是蔡淳仁老師，每次研究上遇到困難老師都會一起幫我們找解法，或者自己陷入困惑時老師都會適時的提點我們，讓我們改進自己的盲點，而不會陷入迴圈無法自拔，這兩年過程中從老師身上學到很多經驗，看到老師不管在教學上或是再解決問題上都是值得我去學習和效法的。很多時我們可能懈怠或懶惰，老師也會舉很多例子或講目前業界的情況來激勵我們，使我們重新熱情燃起更確立自己的目標，努力在自己的研究上。

接下來要感謝父母親支持我繼續完成碩士學位，也感謝MMES實驗室的大家，也感謝胖嘟嘟，有大家才能成就今日的我，沒有大家就沒有今日的我。



圖目錄

表目錄

第一章 前言

1.1 研究動機

Java 語言具備有跨平台，物件導向，安全...等優點，在程式語言中一直是備受眾人所愛，最主要的，把 java 檔案透過 java 編譯器可以轉換成 class 檔，只要有安裝 java 虛擬機器這些 class 檔可以在任何平台上執行例如 UNIX 平台、windows 平台 OS/2 平台，此跨平台的特性越來越受嵌入式系統的喜愛，可見 java 語言的重要不可言喻。

Java 是一個物件導向的語言，每產生出來的物件都是放在 Heap 空間上，Thread 也會共享 Heap 空間上的資料，因此可知道對 Heap 的管理也是在設計上重要的探討，一個高效能的 Java 處理器，除了要有適當的 object caching 機制來增加物件存取的速度之外，也要有 Garbage Collection 的能力。否則對物件產生頻率高的應用程式，很容易就會發生 Out of Memory 引發執行錯誤，加上沒有處理一些用不到的物件 Out of Memory 更容易出現。因此對於 Garbage Collection 探討更為重要，這也是 java 語言的特色之一，程式設計師可以專心於設計，對於物件的回收只要交給 Garbage Collection 去判斷。大部分的研究也指出物件的存活時間很短，只有少部分會存活到程式執行結束，物件如果不再被 reference，則 Garbage Collection 則可以對此物件回收，對於越來越多區塊被一一回收可能也會衍生出另外一個問題，外部碎裂，也就是在連續配置記憶體的策略下，所有可用記憶體空間加總大於目前所要求的大小，但由於因為這些空間不連續所以無法使用，這個也是一個值得探討的問題，我們在接下來的章節將會提出我們的解決方法。

當資料放到記憶體時我們所想到的優點就是空間大，但與暫存器和 BRAM 相比速度卻是我們要考量的課題，因此對於 cache 的引進是必需的，如此一來可以兼顧空間和速度上這兩個考量。現今大部分的系統，不管是個人主機，伺服器，

或是嵌入式系統，cache 幾乎是無所不在，使用時的 coherence 也是最值得探討的議題，不管是在多核心上的 coherence 或是單核心的 memory 和 cache block 之間的 coherence 都是現在主要研究的重點。

1.2 研究貢獻及目的

本論文是在異質雙核心 JAVA 應用處理器的嵌入式系統進行測試與效能提升的改進。實驗平台皆在 ML507，簡單來看此處理器只是一個 IP，是一個高度可移植性的 IP core，也是相當完整的 java virtual machine，但還有些功能還可以修改，因此本論文就在這邊提出改進與實際整合進到 IP 測試。

首先因為之前沒有對 Heap 空間實施管控，簡單來看只有一個累加器一直對 Heap 空間作使用，實作程度來看就是當拉起 Enable 信號線在傳入要分配的 size 在與 current_heap_pointer 即可完成此次的 Request，沒有回過頭去釋放不會用到的物件空間，因此使用空間只會無限制的增加，要是有一個需求 Heap 空間很大的程式，就不能在我們的 IP 上執行，而且 java 語言中的 garbage collection 本來就是主要的特色之一，所以我們引進了完全用硬體實作的 garbage collection，可以大幅提升對 memory 的使用率，一些沒有用到的物件可以適時的回收，不會造成空間上的浪費，而且為了效率考量我們採用的演算法也是最適合硬體實作的，雖然整體的效能有稍微下降，但這是不可避免的，不過這些所換取來的記憶體空間使用率，是絕對可以接受的。

第二個改進就是，原先的 Heap 空間是放在 on chip memory 所以空間大小有極大的限制，之前的設計只有 32Kbit，因此我們在測試程式時，常常要注意不能執行會用到太多的 Heap 空間，因此無法測試太多程式，而且大部分的測試程式都會有一個迴圈一直執行某個區段的程式碼，所用到的 Heap 空間就會用越多，因此本論文就在此提出改進，使我們的 java application IP 可以得到更完整的測試，不用去擔心 Heap 空間大小的問題。

現在 Heap 空間我們已經移到 DDR memory 空間大小已經放大成 32MB，所得到的優點是空間大大的增加，但缺點是存取的速度卻大幅下降，原先去 OCM 存取只需要兩個 cycle 現在透過 BUS 到 DDR memory 存取須更多 cycle，在不讓效能差距太多的們決定引入 Object Cache Controller 好讓速度和空間皆可以兼顧，我們就設計 2-way set associative cache，因為我們有兩個核心，coherence 也會在本論文看到如何處理，做了以上的改進後效能就可以提升到與放在 OCM 相當的效率，因為放在 OCM 是一個 upper bound 只要能達到與 OCM 相符的效率就是一個成功的設計。

1.3 論文架構

本論文一共包含六個章節。第一個章節是蓋括性的導論，包含了研究動機、背景及目的。第二章，會先介紹 JAIP 相關的研究，與先前的架構，和提供一些不一樣的 Garbage Collection 演算法，第三章為 garbage collection 的詳細介紹，也是本論文重點貢獻之一，首先會把架構完整呈現，之後會把每個重要的元件逐一的介紹，和重要的機制將會在此章節介紹，第四章為 object cache controller 會介紹如何用硬體實作 cache，整體架構圖，第五章為則會分析效能，第六章則為結論和未來發展的研究方向。

第二章 相關研究

2.1 Previous Work

要執行 Java 程式的話必須先要有 java virtual machine(JVM)，是由 sun 公司所推出的 java platform，當我們寫好 java 的程式就可交由編譯器變成 class file，此 file 就可以放到 JVM 上執行，下圖是一個傳統的 JVM 設計，class loader 就是負責載入指定的 class，Runtime Data 提供 JVM 在執行時所需要的資料，Execution Engine 負責去執行所載入方法的指令。

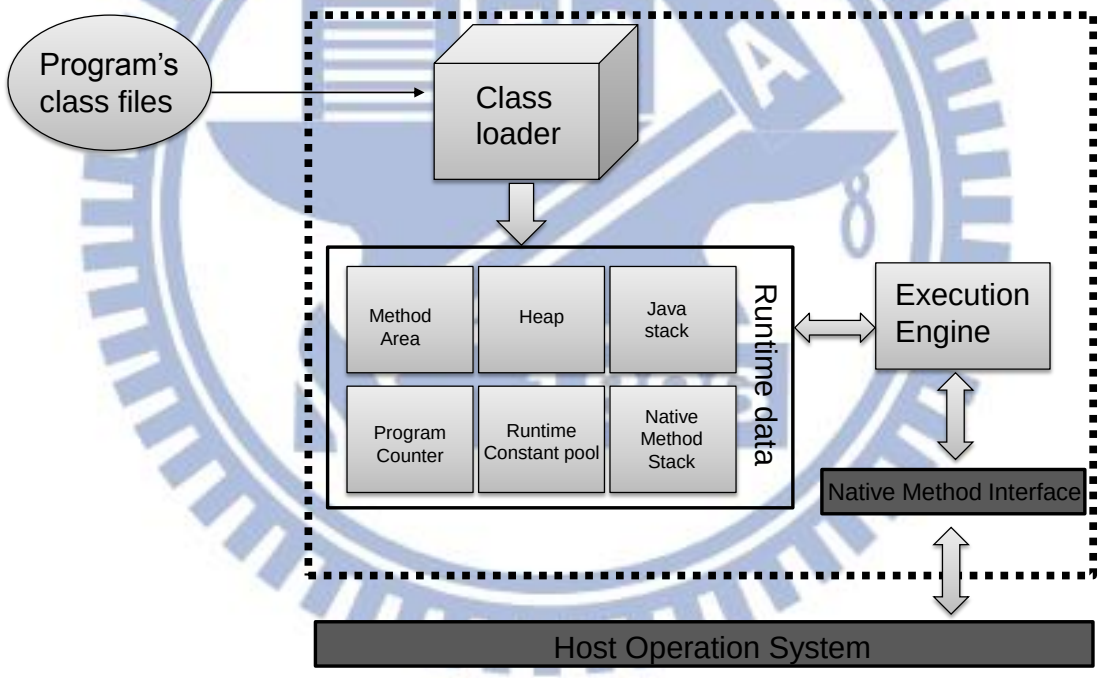


Figure 1. 傳統 java 執行環境

JVM 在實作上有定義規範，但留了很多選擇給程式設計師，因此下面我們會介紹我們實作出來的 JVM。本論文所採用的 java 處理器架構是異質雙核心 java 處理器[10][11]，我們稱之為 Java Accelerator IP，簡稱為 JAIP，架構如下圖。

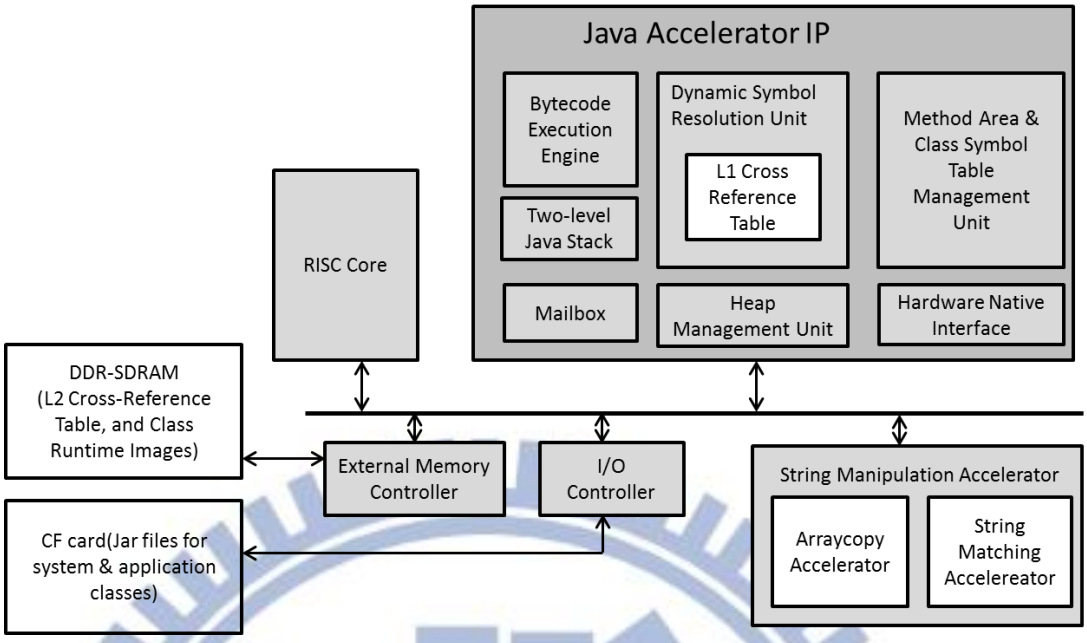


Figure 2. 異質雙核心爪哇處理器與字串加速器

我們主要有兩顆核心，第一顆核心為 RISC core 主要功能是作為一個 parser 對 class 作 parse 的動作，和負責系統的初始化，或當 JAIP 發出 interrupt 要求要執行 native method 也是 RISC core 的功能。第二顆核心也是最主要的。其中的 Bytecode Execution Engine 分成四個 stage 的 pipeline，Translate、Fetch、Decode、Execution stage。

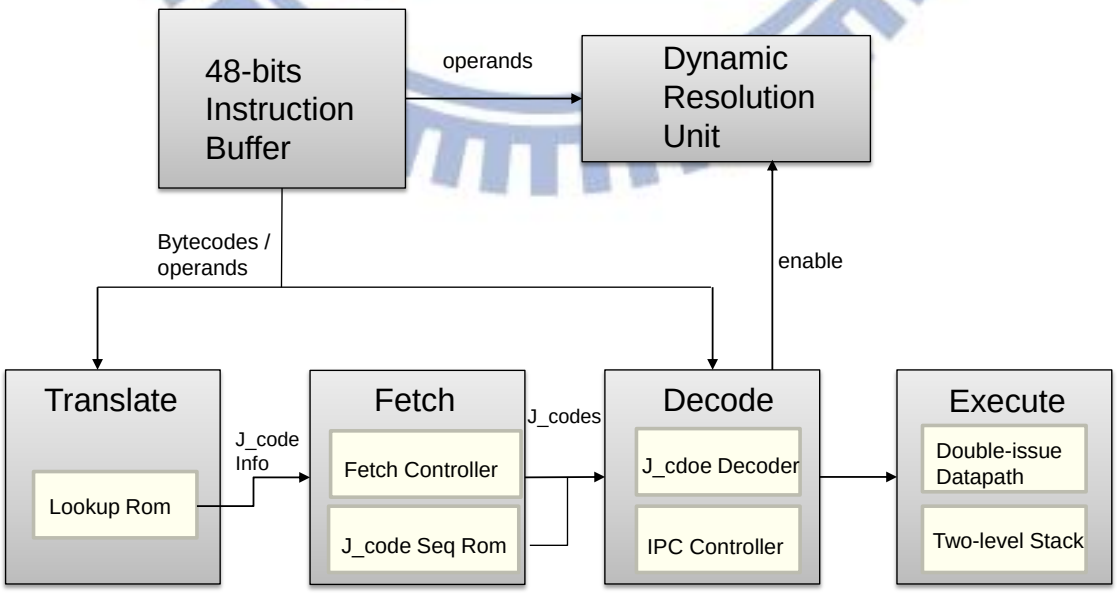


Figure 3. Four-stage pipelines of BEE

Translate stage 會去 Instruction Buffer 取得要執行的 bytecode，此時會分出兩種類型，第一種是 simple bytecode 就可以迅速的轉換成 j_code 送到 Fetch stage，第二種是 complex bytecode 此時傳過去的是對應 Fetch stage 的 j_code sequence 的起始位址，之後到了 Decode stage 會產生這些 j_code 相對應的控制信號線傳到 Execution stage 執行。JVM 是以 stack based 所設計的，因此 JAIP 在實作中堆疊架構與堆疊運算的設計也是設計重點，我們提出了 Two-level Java stack memory[10]，第一層由三個暫存器組成，第二層的 java stack 由兩塊 Dual-port On-chip BRAM 來實作交錯式的記憶體架構。並且使用四個暫存器作為存取為頻繁的四個區域變數，如此一來速度可以增快。

2.2 JAIP heap 空間分配管理

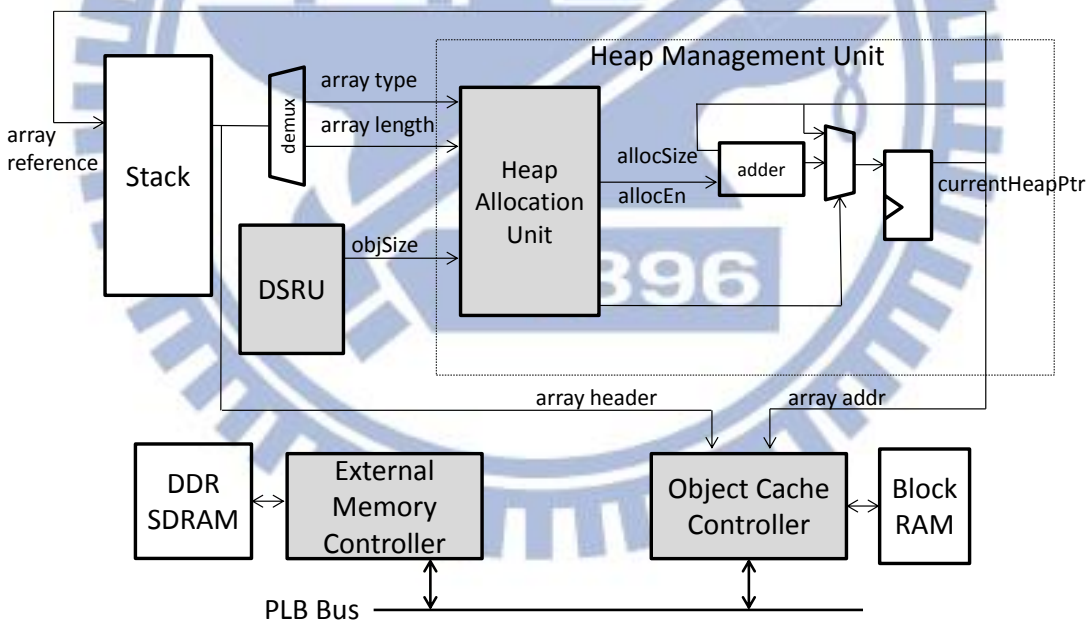


Figure 4. 未加入 GC 的 JAIP heap 空間分配管理

原先的 Heap 空間分配管理詳細結構如 Figure 4 我們可以清楚看到當 allocSize 和 allocEn 從 Heap Allocation Unit 取出後，會傳送到一個 adder 直接相加，中間並沒有經過任何的處理，此時 Heap 空間只會不斷的增長，當遇到需求 Heap 空間很大的程式我們就有機會當機，因為 Heap 空間不夠用的緣故，所以接下來

的章節我們就會介紹一個 Garbage Collection 的機制來回收已經使用不到的 Heap 空間，透過回收使用可以確保 Heap 不會無限制的增長空間，而且會反覆的利用。

2.3 Garbage Collection

Java Virtual Machine 的 heap 是用來存放物件，這些物件透過 new、newarray、anewarray 和 multianewarray 等 bytecode 所建立，但卻沒有明確的指令可以釋放記憶體空間，有人會透過把物件指定成 null 但此法不一定會釋放，如果此物件還有其他人會參考到就不會釋放所持有的 heap 空間，另外一個是透過呼叫 system.gc() 去釋放記憶體，但這只是給系統建議參考用，系統會觀看目前的情況決定是否要釋放記憶體。Heap 是由 Garbage Collection 來負責的，儘管 JVM 的規格書中沒有要求特殊的 Garbage Collection，甚至不需要，但由於記憶體空間有限，而且嵌入式系統的記憶體所提供的容量更小，所以 Garbage Collection 仍然是不可或缺的技術，它是自動釋放不再被參考的物件，按照各個不一樣的演算法來實行自動回收的功能。aaabGarbage Collection 也可以清除記憶體資料斷離(Fragmentation)由於建立物件和 Garbage Collection 釋放無參考物件，就會在 Heap 中出現 Fragmentation，下面我們會繼續探討 Garbage Collection 的演算法還有如何去處理 Fragmentation 問題。

2.3.1 Algorithm of Garbage Collection

任何一種的演算法必須做到以下兩點(1)可以發現無被參考的物件(2)回收這些無參考的物件，使得該 heap 空間可以再重覆的使用。大部分的演算法都採用 root set 的觀念，也就是從 root 開始，區分哪些是可以到達，哪些不可以到達，可到達的就是有效物件還會被參考到，不可到達就會被回收起來，以下會介紹各種不同的演算法。

第一種 Reference Counting Collector，是唯一沒有採用 root set 的概念。作法

是每當物件創立時就會賦與一個變數，初始值是一，當此物件又被其它物件所參考到就會對變數再加一，但如果當物件出了範圍後就是代表不再被參考使用，就會對變數做減的動作，一旦變數變為零則可以被收集起來，然而因為每個物件都要給一個變數而且隨時都要加或是減，還有不會偵測 cycle 的關係，如果有父和子物件有存在彼此參考的關係，則變數永遠不會被設為零，就不會被收集起來，基於以上原因 JVM 不會採去此演算法，而是採用 tracing 演算法。

第二種 Tracing Collectors，也稱為 mark-and-sweep，此演算法採用 root-set 概念，實作時每個物件都會有一個 flag 來指出是否可回收，在 mark phase 時，會過掃描整個 root set，判斷出哪些是還會被參考到那些是可以被回收起來，再 sweep phase 時，把剛剛判斷出來可以被回收起來的一起執行 Free 記憶體的動作。此方法還是會導致整個系統 stall 住，導致效能下降，因此有衍生出另一個演算法，Tri-color marking。

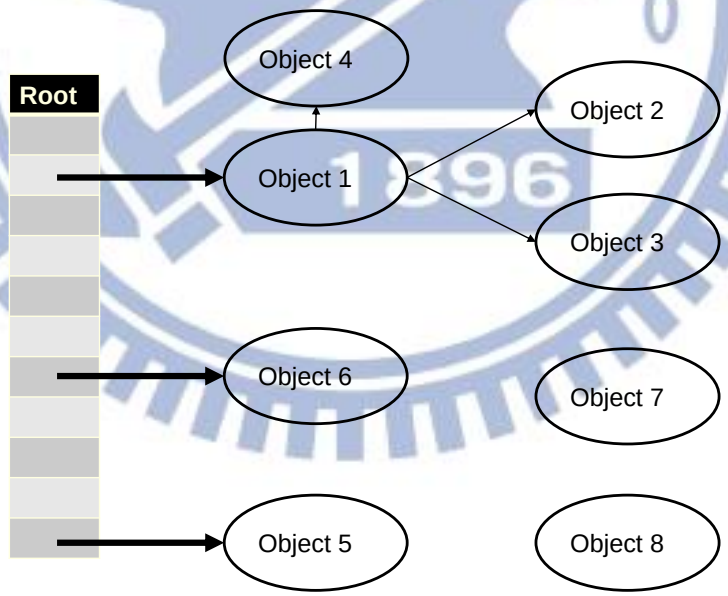


Figure 5. Mark and Sweep

上圖中除了 object 7 和 object 8 會被收集起來外，其他的物件因為都可從 root 到達皆會被標示成 in-use。

第三種為 Generational Garbage Collection，分成三個部分: Young Generation、Old Generation 和 Permanent Generation，Young Generation 又有分 Eden space 專門給新創出來物件放置用，和兩個 survivor spaces，接下來會根據物件所再不同的區域，依不同的演算法去處理垃圾收集，例如 Copying Collector 不會搬移垃圾物件，如果執行的區域大部分都是要被收集的，它的搬移量就會減少，執行效率就可以提升。另一個主要概念是物件在 Heap 空間中存活越久的物件越不可能是垃圾，然後給予每個物件一個欄位來代表物件所處區域，根據統計，約有 80-98%的物件在一開始創建時，也就是在 Young Generation 時，之後就不會再被使用，因此很快就變成垃圾，如果沒有變成垃圾就會晉級到下一個 generation，下圖可看到物件如何得到晉升的機會而到不同的世代。

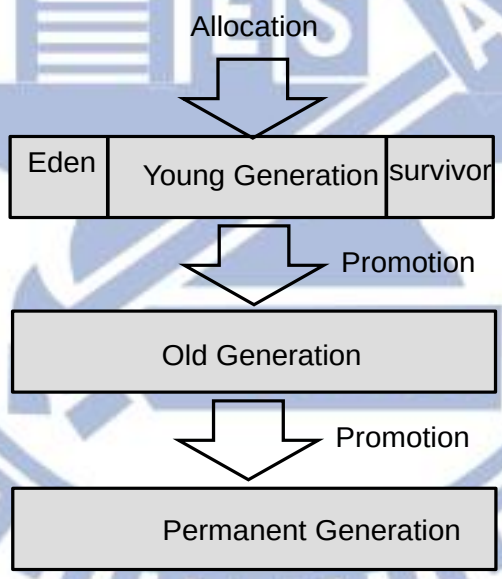


Figure 6. Garbage Collection Area

2.3.2 處理 Fragmentation

在處理 Fragmentation 上，一般有兩種作法，想法都是去移動 Heap 使得物件都可以連再一起使用 Heap 空間，第一種 Compacting Collector 作法是透過新增一張表格(object handles)，不是直接可以存取到 heap 位址，第一次的存取只是拿到表格位址，再透過表格才可以知道真正再 heap 的位址，因此在移動上只要對表格

更新即可達到搬移的效果，而不需要真正去對 Heap 執行搬移的動作，節省大部份的時間，但此法對於每次要對物件存取時會有額外的負擔，實作上也要注意指標所指的 Heap 空間是正確的，否則就會造成系統當機。第二種是 Copying Collector[21]，會將 Heap 分成兩個區域，分配物件的時候全配置再同一區域，記憶體不足時，掃描該區域活著的物件並移到另一個區域，搬完後剩下不用到的物件直接回收整個區域,再將兩個區域角色互換,也就是物件區段和空閒區段對換。缺點是一次只能使用一半的 Heap 空間，空間越大掃描的時間也越久，暫停程式的時間就越久，優點是配置簡單。以上兩種作法都會造成系統上效能的下降，因此本論文會參考這些作法衍生出另一種作法，以便不會影響系統效能。

第三章 Garbage Collection 設計

開始本章節之前，我們會先探討如何選用我們現在的演算法來設計 Garbage Collection。第一個我們用的是 Reference Counting Collectors 也就是每當物件被創建的時候就會指定一個變數給它且設定為一，每當有別的物件在參考到此物件時就會對此變數一直累加，當變為零的時候就可以回收此物件的 Heap 位址，在 java 處理器實作時，我們要到 bytecode 層級去設計，每當要產生物件時，會有四行 bytecode，依序是 new、dup、invokespecial、astore_1。

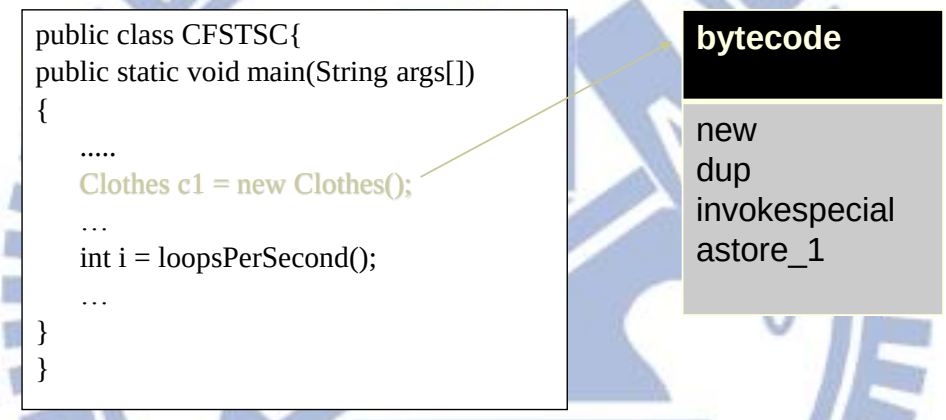


Figure 7. 創建物件所產生的 bytecode

當執行到 new 的 bytecode 時，會去 Heap 要求一塊位址，dup 會把此時的 reference 在堆疊上在複製一份，invokespecal 會對這個物件作一個初始化的動作，最後 astore_1 會存到第一號 local variable 中，反過來如果最後沒有出現 astore_1 就代表產生出來而沒有使用到，所以不會對變數加一，當不使用此物件時我們只要簡單的指定 null 就會對變數減一，因此當此變數為零就是代表可以回收。上述的方法很容易實作，但實作在 JAIP 時發現有下列三個缺點，第一點，很少有 java 程式設計師會習慣把沒有用到的物件指定成 null，也就是大部分的程式碼很少看到會指定沒有用到的物件為 null，因此此功能就會很少被用到，第二點，對變數加或減也是種額外的負擔，實作時我們要去 GC table 中找到此物件在哪需要一次的搜尋，而且所花費的 cycle 會隨著物件變多而變多，額外負擔特別重，後來我

們有想到改進的方法，先試想如果會指定變數加一通常會連在一起執行所以要記錄上一次使用的 GC table 這樣就可以少掉此次的搜尋，因此我們也有實作此功能，皆下來開始去測試效能，光是 Jembench 中的 UdpIp 就掉了 50%的，第三點，不是所有的 astore 的 bytecode 之前都會搭配 new 的 bytecode 因此也會造成額外的搜尋，例如說 String s = getclass.getName()，因為找不到所以會自動掃完整個表格，浪費的 cycle 更是巨大，綜合以上三點原因我們實作時不採用 Reference Counting Collectors 的方法來回收物件。

以下是我們設計 java hardwire garbage collection 的方法，每當創建出新的物件時，我們首先會去表格中尋找是否有回收的空間可用，有的話即可使用，沒有的話就會額外分配新的 Heap 空間，因此有相對應的表格儲存其目前狀況，包括記錄其 flag 判斷是否可以回收，再來就儲存此次分配在 Heap 中的真實位址、和分配的大小，Flag 的更改是當一個 local method 返回時在執行過程中如果有產生物件都必須被回收，因為跳離此 method 後，這些物件都不會再被使用到，除非返回時傳物件的 reference，該物件就不能回收。

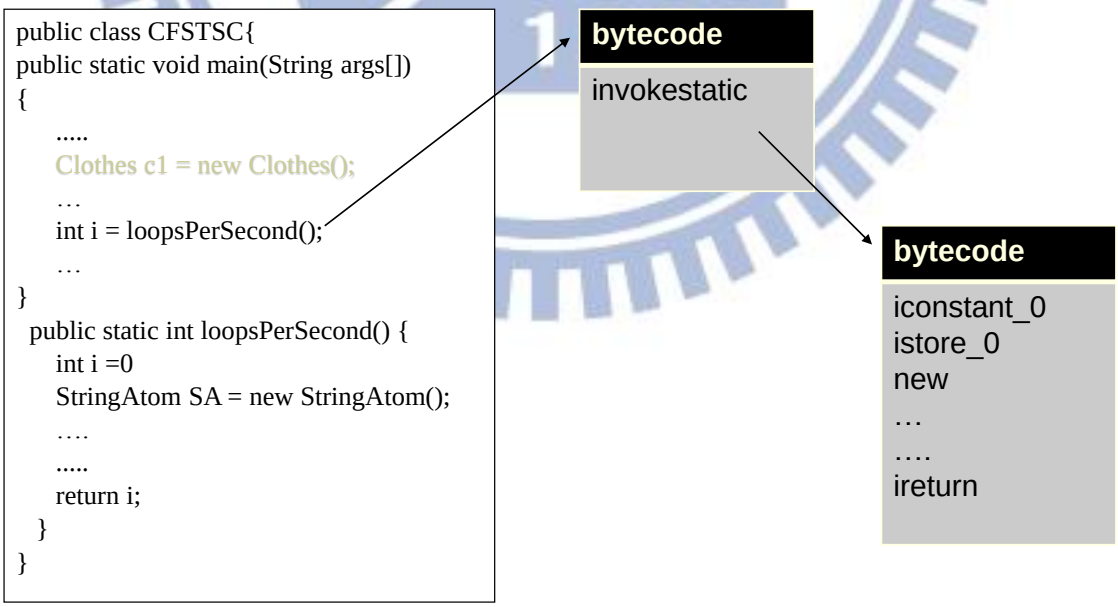


Figure 8. Method 所產生的 bytecode

由上圖我們知道 loopPerscond 的方法中一開始會先產生出的 bytecode 為

invokestatic 之後才會跳過去 method 所產生的 bytecode 開始，執行第一個是 iconstant_0，就是圖中最右邊的 bytecode 表格。此例最後離開 method 會回傳整數，對應到 bytecode 是 ireturn，因此會通知控制器可以開始回收在 loopPerscond 的方法中所產生出來的物件。

3.1 Garbage Collection 系統架構

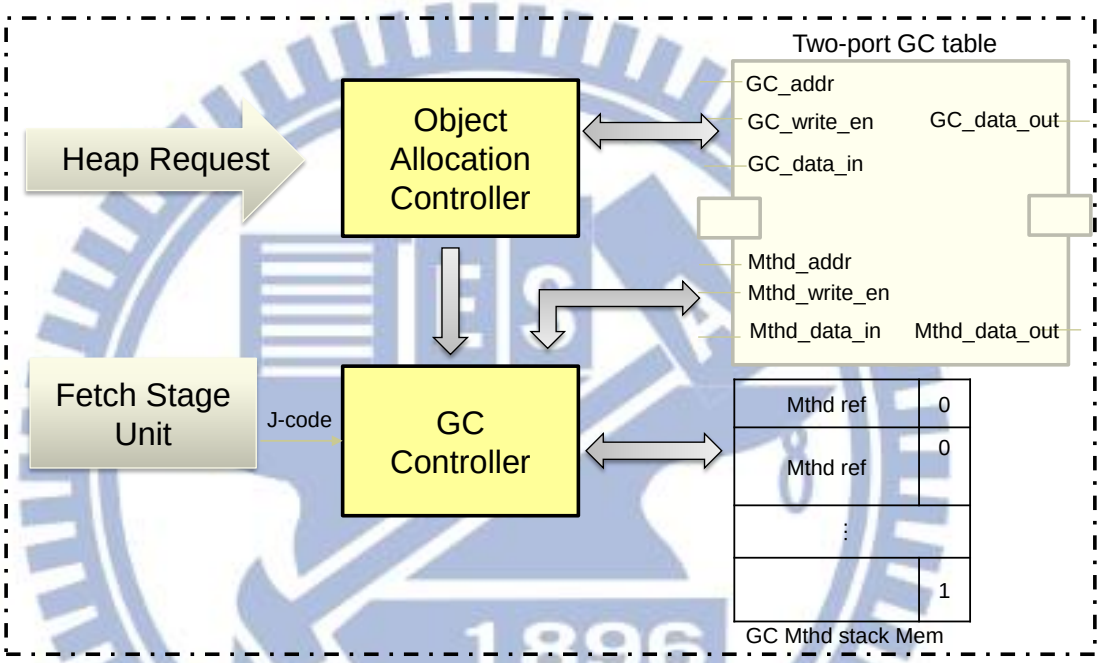


Figure 9. Garbage collection 架構圖

我們使用兩個控制來主導整個系統，一個是 Object Allocation Controller 也是整個架構中最重要的 controller 將再下一小節介紹，另一個是 Garbage Collection Controller，使用到兩個 BRAM 來存取資料，第一個 Garbage Collection Table(GC table) 且使用 Two-port BRAM 來實作，主要原因是兩個 Controller 可以同時存取不需要互相等待而造成時間上的浪費，舉例來說，當有一個要求到，Object Allocation Controller 可以去 GC table 尋找是否有可用空間，同一時間點 GC Controller 可以把上次要寫入的 flag 寫回 GC table 中標示此欄位已經被回收，此作法可大大提升效率，因此我們 GC table 不會採取單一 port，實作時當 GC Controller 正在回收某些空間時，因為尚未回收完畢還不能使用，不過由於 BRAM 存取只需

要兩個 cycle，所以下個 Request 來之前就可以回收完畢而被使用。另一個是 Garbage Collection Method Stack Memory，主要由 GC Controller 負責 push 和 pop 資料，這些資料就是用來判斷是否該回收此 Heap 空間與否。

3.2 Object Allocation Controller

此 Controller 主要是接收所有要對 Heap 空間作存取的請求，比如說 new 或是 newarray，接下來就會開始去 GC table 找尋是否有已經回收的空間可以使用，找到後就可以使用已回收的空間，如果沒有找到就必須分配新的 Heap 空間，分配完後，仍需把這些資料寫到 GC table 中作記錄，以便日後可以回收再使用，回收的動作我們是在 GC Controller 作相關的處理。

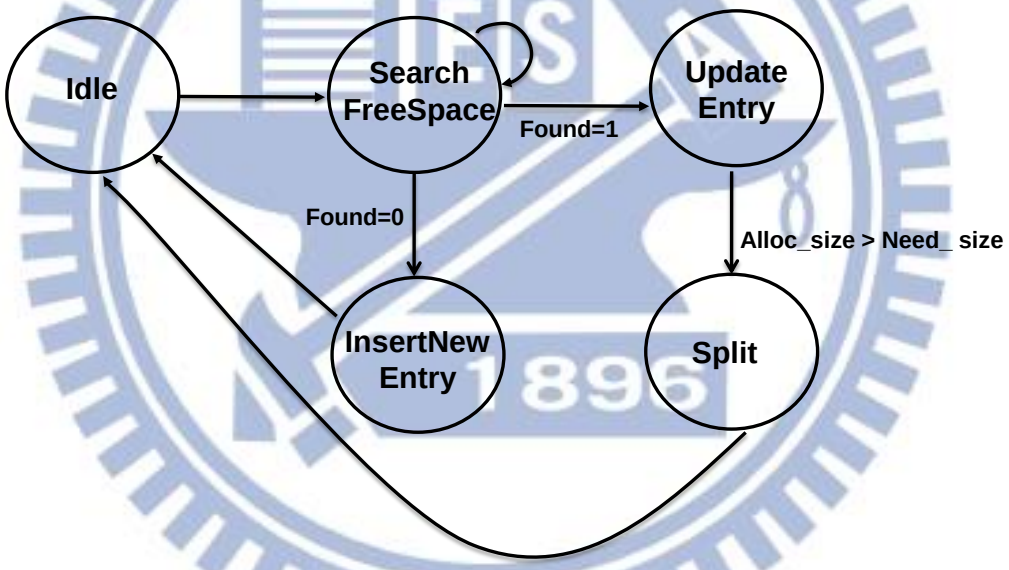


Figure 10. The FSM of Object Allocation Controller

Figure 10 為 GC Controller 的狀態示意圖。大致上可分為五個狀態，分別為 Idle、SearchFreeSpace、UpdateEntry、InsertNewEnter、Split。接下來是對各個狀態的詳細解說。

(1)Idle 狀態時，表示目前沒有任何的 Heap space request 到來，將會一直維持此狀態，直到有 request 來就會進到下一個 state，SearchFreeSpace。

(2)SearchFreeSpace 狀態時，會一直對 GC table 做搜尋，如果有找到就會跑到

UpdateEntry 沒有就會一直找直到資料的尾巴，最後就會產生一欄新的欄位，由此可知這個狀態是最會花時間的，假設今天找了 1024 個欄位都還沒找到，最後寫入第 1025 欄因此花費了 2048 個 cycle 在搜尋 BRAM，所以光是一個 Heap 的存取就要花這麼多 cycle，因此在選擇合適的記憶體區塊我們採用的演算法是 First-Fit，即只要找到第一個符合空間大小需求的就把此記憶體區塊分配出去，因此可以得到更好的效率。因為我們的機制是每次有物件要分配時，我們就會去搜尋所有 GC table 中所有的資料，終止條件有兩個(1)當找到有回收的空間而且 size 符合需求(2)搜尋到資料的底部，代表在表格中沒有找到可以使用的 Heap 回收空間，因此隨著使用欄位越多時，每次要分配物件所花的時間就越多，為了增進效能，我們對於所收集的物件就必須有所限制，在一開始 size 小於 7 個 word 我們不會收集(因為一次的 Burst 為八個 word)，等到使用到一定的欄位，限制的大小也會隨之而增加，以便得到更好的利用率。

(3)UpdateEntry 狀態時，表示有找到符合大小的 Heap 空間而且此空間已經無任何物件會使用即已經被收集起來，會更改 GC table 中的 collection bit 還有視情況更改大小，如果符合大小就不需要更改，如果超過其需求則會到 Split 的狀態更改其值。

(4)Split 狀態時，代表現在要求的 Heap 空間超過我在 GC table 所找到的空間，因此會造成差值，為了避免造成內部碎裂的問題，我們用此狀態來處理這些剩餘空間的問題，我們會把這些空間寫到 GC table 中的另外一個欄位，也意味者在 GC table 中所存的空間不一定會連續，因此 GC table 資料結構上的設計我們也有增加指標，來做記憶體的合併，詳細在 3.4 會作介紹。

(5)InsertNewEntry 狀態時，代表皆無可用空間可以符合我這次的 Heap Request 所以必須寫入新的欄位，而且為了避免寫入過多的欄位，而造成效率的低落在此我們新增了幾個 flag 來標示現在已經寫到哪些欄位，因此隨著欄位越寫越多，我們所要收集的空間就會開始有所限制，比如說現在已經寫到第 500 欄，我們就會

限制小於 50word 大小的 Heap size 就不會收集起來，畢竟寫到 500 欄之後每次的存取都必須至少要 1 千個 cycle 才能完成一次的 Heap request。

3.3 Garbage Collection Controller

主要功能就是處理儲存在 GC table 中的資料，藉由 GC method stack Memory 裡面的資料可以知道 GC table 中有哪些是 local Method Return 之後被回收的 Heap 空間，接下來會標示此 Heap 空間可以被回收使用，下面的圖是整個的 finite state machine。

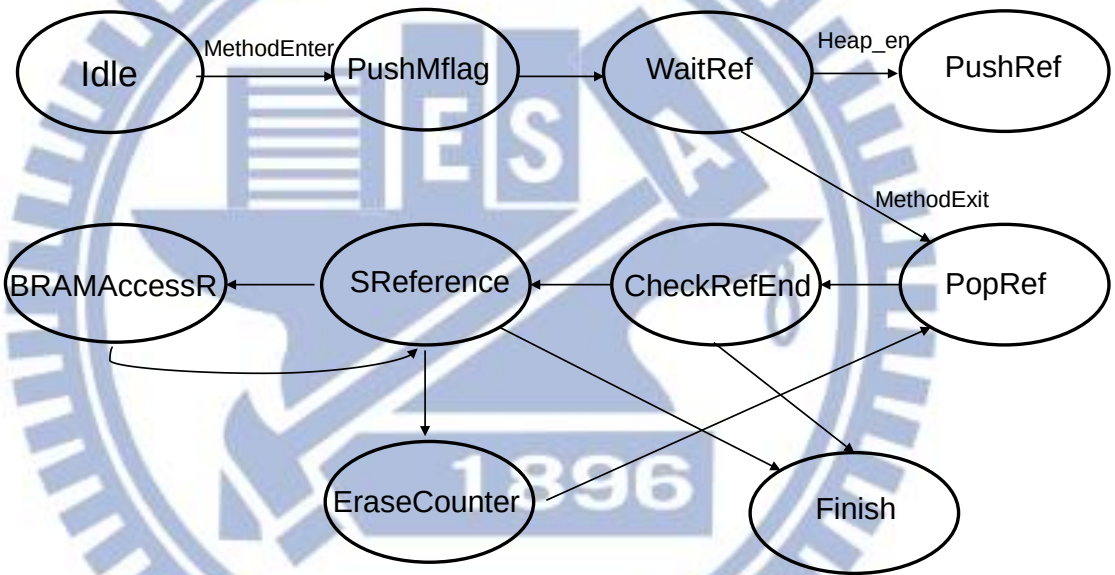


Figure 11. The FSM of GC Controller

一開始會接收來自 Fetch stage 的 j-code 得知目前有要進入一個 local method 所以就會開始收集，過程中 Object Allocation Controller 也會傳送 signal 說現在要分配哪些記憶體區塊，所以 GC controller 也會記錄此記憶體區塊的位址以便之後 method return 時就可以把這些位址回收。

在 java 中會觸發 method 的 bytecode 分別有 invokestatic、invokevirtual、invokeinterface、invokespecial，但是我們不會收集 invokeinterface 和 invokespecial 所引發的 method，前者是因為不會引起分配 Heap 空間的需求，後者是因為回收

時機點設在此有機率會觸發錯誤，舉例來說，因為每次的新增物件都會有三個 bytecode 分別是 new 緊接者 invokespecial 最後是 astore，意思也就是說這個 invokespecial 只是一個初始化物件的功能而已，不能算是一個 local method 的返回，所以只要裡面有要分配 Heap 空間就會引發錯誤，因此在我們的實作裡面只會有 invokestatic 和 invokevirtual 這兩個 bytecode 會經由 fetch stage 在進入 Object Allocation controller 判斷哪些可以回收再利用。Method Return 的 bytecode 為 areturn、ireturn、return，這邊要注意的是不是遇到 return 就要開始回收，因為我們要先確認這個 return 是一開始引發 controller 開始收集的 method 的 return。遇到 areturn 我們要注意在這個 method 中如果有新增物件，這些都不能收集，因為未來此物件會被使用到。這邊要注意的是在 JAIP 中我們的 invokespecial 和 invokestatic 在 j-code sequence 是會產生一樣的 bytecode，而且 invokeinterface 和 invokevirtual 在 j-code sequence 也會產生一樣的 bytecode，因此我們必須作修改，也就改成會產生不一樣的 j-code sequence 即可，才能區分出哪些可以傳到 controller 可以觸發回收機制，準確的回收物件。

有了上述的認知就可以對 Figure 11 作一個詳細的介紹，當 j-code 傳來 method enter 的 flag 時就會進入到 PushMflag 的 state 就會放置一個 flag 到 GC stack Memory 接下來就會進入 WaitRef 開始等待這個 method 是否會產生出物件，此部分會和 Object Allocation Controller 會通知是否有要收集此物件，假設有就會把位址寫入到 GC stack memory 中，一直到 method 離開後就會開始對 GC stack memory 一直 pop 裡面的資料，在 EraseCounter 時，就會去 GC table 把相對應的 Heap 位址標示成可以回收使用，然後到了 CheckRefEnd 就會判斷是否到這個 method 的底端了，也就是說上次 pop 出來的資料就是一開始進入 method 所 push 進去的 flag。

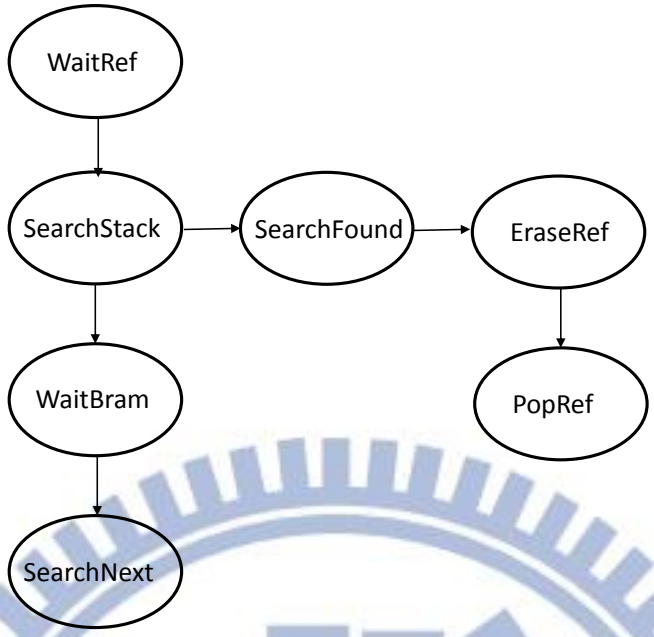


Figure 12. 移掉未來會再用到的 reference 的 FSM

並不是所有的方法回傳，裡面所新產生的空間都可回收，有些會在指定到別的地方使用，因此這些必須保留在 Heap 中，上述的 FSM 就是在處理此問題，當皆收到 getstatic 和 getfield 的 J-code，而且此時正在對某個方法做收集時，就可以知道這些參考不可以收集起來，必須把這些資料從 GC Stack 中移除。

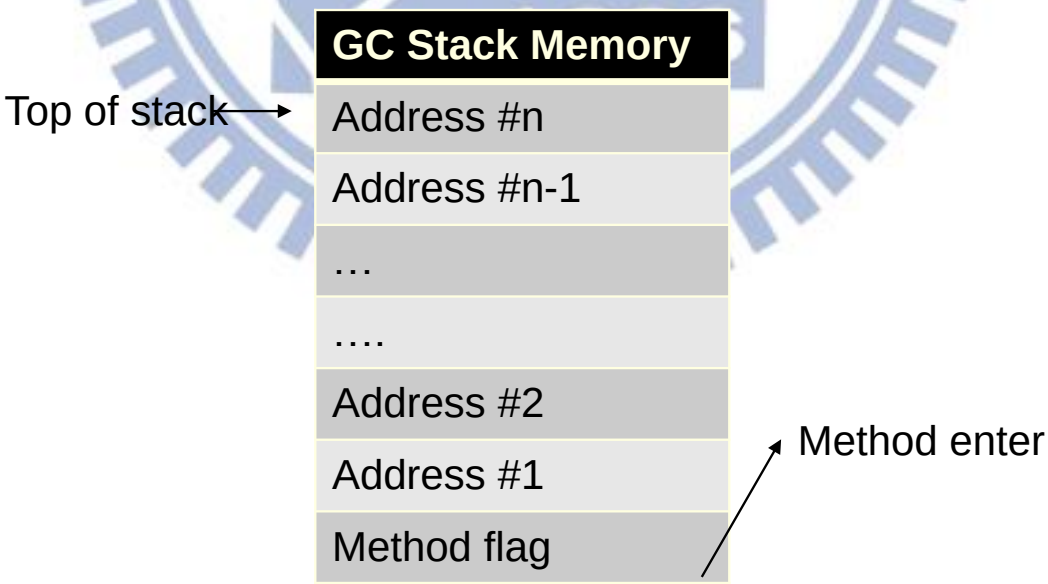


Figure 13. The Content of Garbage Collection Stack Memory

由上 Figure 13 是描述 GC Stack Memory 如何運作，假設今天收到一個 method

return 的 bytecode 且是一開始觸發 stack 收集的 method return 的返回，就會開始從 stack memory pop 出 Address#1 到 Address#n，這些資料就是將要被回收的物件，會由 GC Controller 去更新 GC table，標示這些物件空間都可以被回收使用，最後再 pop 的話就會得到 Method flag 就表示，下面部分是屬於其它 method 所以到此就會停止，此時 state 就完成此次的 local method return 的收集因此 state 會再次回到 Idle 等待下次的收集。

3.4 處理 External Fragmentation 的設計

考慮到此問題我們再在設計 GC table 中我們有四個欄位 reference、size、flag、previous node pointer 第一個是指出寫到 Heap 位址的哪些地方，第二個是要放置的大小，第三個是指出目前是否有被收集，0 代表可以在被重複使用 1 代表目前還在使用中，第四個是指標，指到上一個位址，借由此指標我們可以消除 External Fragmentation，合併一些零星的記憶體區塊重新組成一個更大的記憶體區塊。

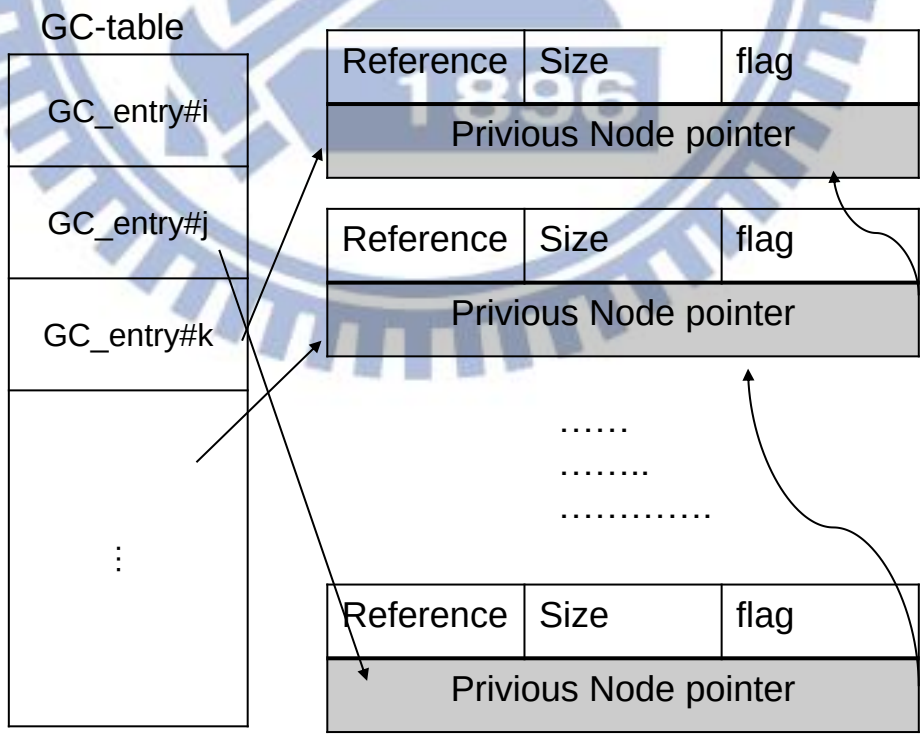


Figure 14. The Content of Garbage Collection Table

我們也有處理 Internal Fragmentation，也就是分配的 size 大於我所求的 size 就會產生一塊記憶體區塊都不會被使用到，因此我們在 Object Allocation controller 有設計 split 的功能，如果找到的 size 大於所要求的就會額外再寫到另一外一個欄位，但是如果差額的 size 小於 n 個 word 我們就會把整塊區域分配出去，n 可以端看此系統是是用何種功用，比如說很常會使用到物件此 n 值必須要夠大，目前預設值是設定為 5，每多一個欄位就會多一次搜尋，一次搜尋的代價是兩個 cycle，還有 FSM 狀態轉換也需要額外的 cycle，因此只要是會新增 GC table 的資料我們必須慎重思考其演算法，避免造成過度浪費表格的使用，也由於 split 功能的設計我們可以了解到每個相鄰的欄位不一定記憶體是連續的，所以當我們在合併鄰近被回收的區塊就必須要再額外檢查上一個的位址加上所分配大小，是否等於現在這個欄位的位址，以下有一個圖來舉例說明。

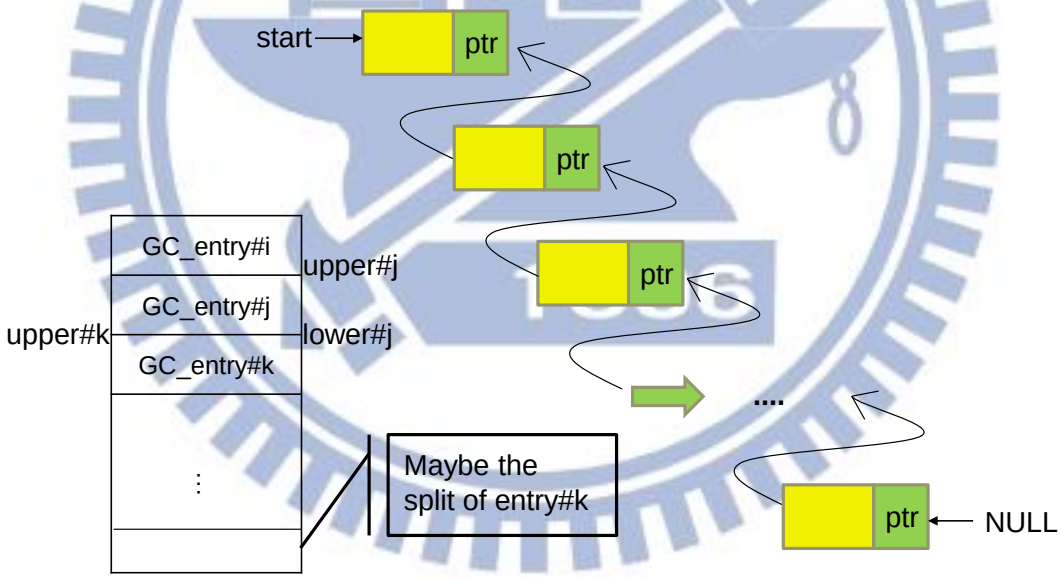


Figure 15. 發生 split 時的 GC table 情況

因為在表格中所相鄰的欄位不一定在空間上就是相鄰，所以是實作上我們有建立一個鏈結的結構才能把相鄰的 Heap 空間串起來，GC_entry#k 如果要合併時會檢查上一個，也就是 GC_entry#j，j 所以會先檢查位址就是 upper address #j 加上大小就會等於 lower address #j，一但 lower address#j 等於 upper address #k 就可以將此兩欄位合併。

3.5 Garbage Collection Table 的設計

由於使用 xilinx 所提供的 dual-port Bram，有些限制，就是當所要存取的 data 資料越大，所能用的位址也就越少，以下表格是可以說明，

Table 1. Port Aspect Ratio

Data Width	Address Width	Depth	DI/DO BUS	DI/DO P Bus
9	11	2048	7:0	0
18	10	1024	15:0	1:0
36	9	512	31:0	3:0

基於表格的設計我們是以 Address Width 去選擇，因此我們選 Address Width 是 11 所以可以有 2048 個欄位，因為如果表格寫到兩千個欄位以後，每次的創立物件最壞的情況就是會需要 4 千個 cycle 才可以完成此次的要求。然而 data port 只有八個 bit 可用，因此我們會串接 7 個 BRAM，因此我們可以有 56 個 bit 可以同時使用，也就是當傳入一個位址到此串接好的 BRAM 我們就會有七個資料同時輸出，之後再將其串接起來就是等於有 7 個 8 個 bit 資料也就是 56bit 資料。

在 port A 是用來處理從 fetch stage 傳來的要求，port B 是用來回收記憶體區塊使用，因為 dual-port 特性可以使兩者同時處理，不用因為 port A 在處理 Heap 要求而造成等待，因此我們所設計的 Garbage Collection 所造成的負擔只有在每次創建物件的時候，在回收物件的時候是不需要和系統搶資源，可以同時並行所以不會造成額外的時間浪費，在處理記憶體的零散空間時，我們是在 JAIP core 發送一個 interrupt 要求到 RISC core 執行 native method，才會開始判斷 GC table 中已回收的空間是否可以合併，當然這部分也不會影響到額外的時間，因為我們以效率為優先，所以當執行全返回到 JAIP 時，合併空間還沒有處理完我們就會停止，

等待下次的 RISC core 去執行 native method。

3.6 Garbage Collection 運作和對 JAIP 修改

前面的章節都是在著重在每個元件單一的介紹，有了前面的認知，接下來會說明我們所設計 GC 的整體運作過程。

一開始收到分配空間的需求,就會到 GC table 中找有沒有回收空間可以利用，如下圖所示，0 代表可以在重複利用此空間，1 代表此空間能被其它 process 所佔領,每個欄位都會標記所佔用的 size 多大,比如說,下圖第一欄就是 50 個 word，各個 GC 表格中每個欄位都會對應的 DDR2 的 heap memory 的實體位址。查詢完 table 就完成創建物件的動作。

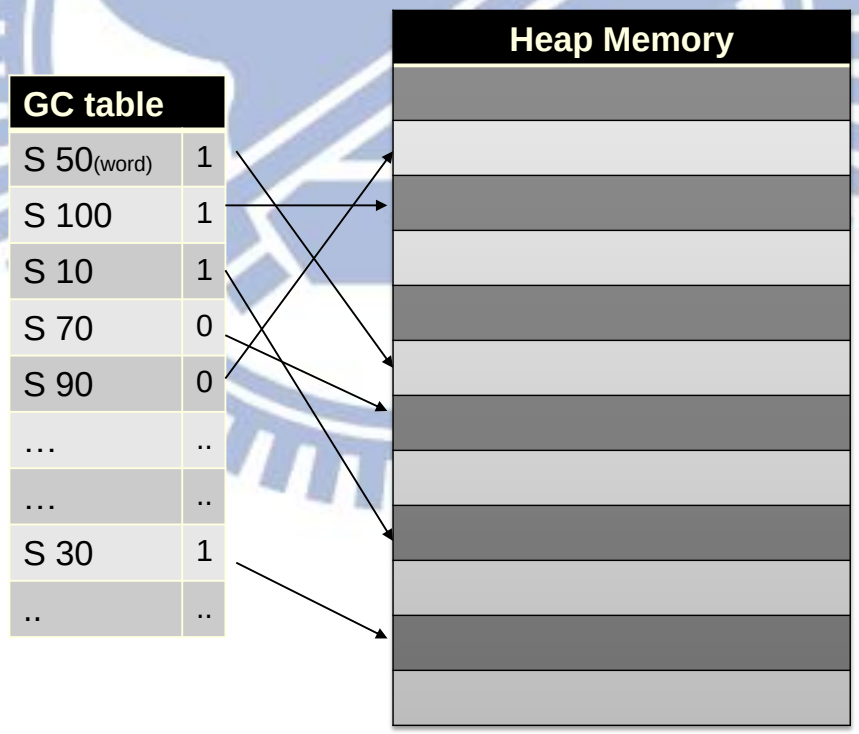


Figure 16. 分配物件時 GC table 的狀況

當每次完成分配空間，另一方面在同時執行的就是回收物件的控制器，只要判斷出這是在一個可以回收的 method 內所產生的物件就會紀錄此物件在另外一

張表格中，等 method 返回即可立刻回收，如果不可回收就不會有記錄，不可回收的就是返回還會用到此物件的。以下將用流程圖完整呈現整體設計的概念，包括分配、回收的情況都會一起討論。

下圖的 Heap Allocation request，就是收到別的元件傳來的創立物件的要求，之後 Garbage Collection Unit 會做出相對應的動作然後到了完成時，就會給出一個位址告訴元件，這個物件可以在 Heap 上做使用，所有的物件控管都是經由 Garbage Collection Unit，除了在本節最後所述的 RISC core，不會經過此 Unit，其餘都會。

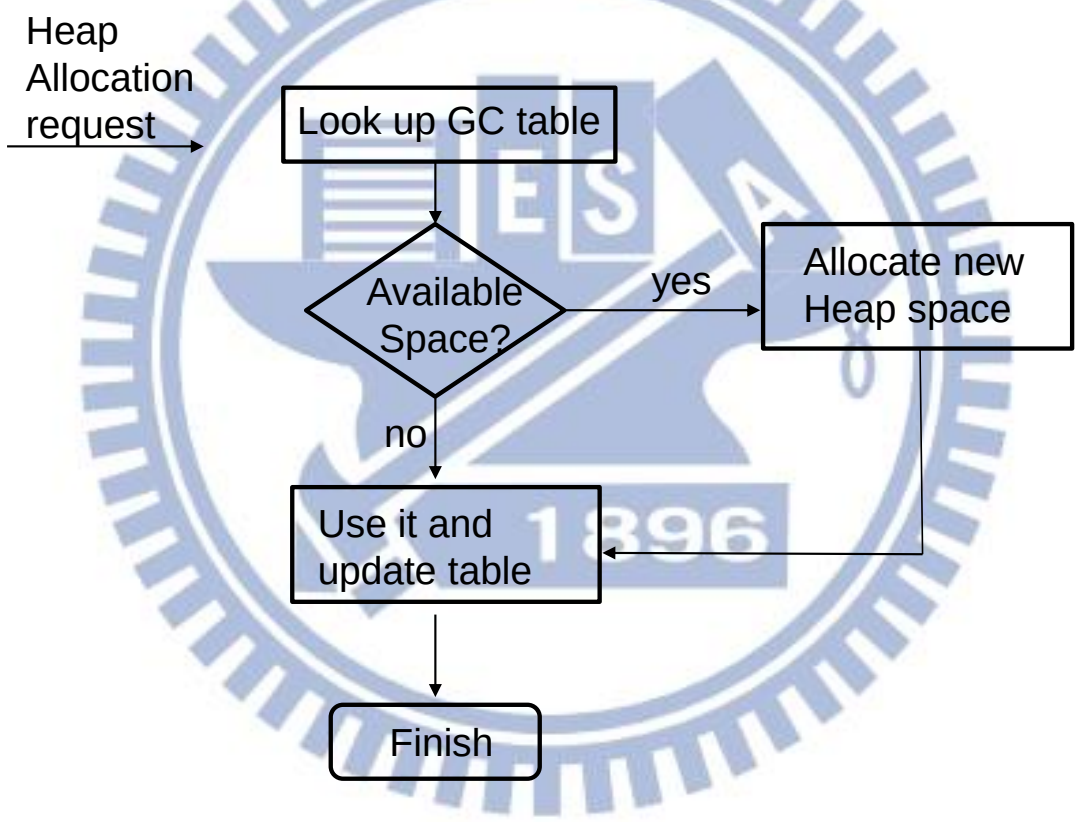


Figure 17. 分配物件的流程圖

收到 Request 後會先去檢查 GC table 是否有可用的回收空間可以用，如果沒有的話就直接出一個新的位址使用，這個位址就是目前 Heap 使用到哪裡的位址，如果有找到回收空間可以用，也就代表所需大小大於此找到的回收空間，就可以使用此空間，使用後也要在更新 GC table 中的欄位，從原本的可回收使用，標示成有物件使用中，避免讓其他物件誤以為此空間還可以使用，這樣就會造成錯誤，

上述流程圖 update table 還有一層意義，就是當所分配的大小大於所要求的大小，就要進行分裂的動作，以便另一塊空間還可以讓其他物件使用。分配物件的流程相對簡單，我們重點是著重於要如何判斷和回收不使用的物件，才可以達到最好的記憶體使用率，接下來我們繼續探討回收的流程圖。

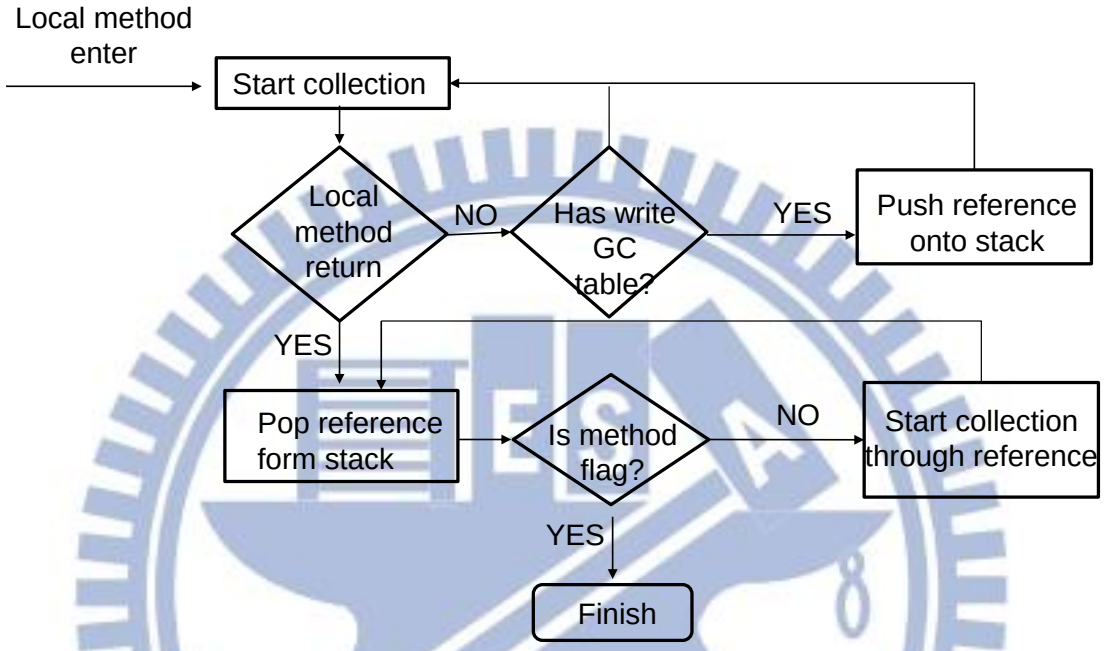


Figure 18. 回收物件的流程圖

開始收集時，當有物件被創立就要被收集起來，也就是有對 GC table 做寫入的動作即物件生成，我們就要把此 reference 推到堆疊上，記綠此物件是在此 method 裡面生成的，當遇到這個 method 返回時這些物件就要一併被回收，因為 method 中不一定只會有一個物件生成，因此不須不斷的檢查是否有物件生成，離開的條件就是當遇到 return 的 bytecode 就可以離開此次的收集，離開後就會一直把堆疊中的資料 pop 出來，這些 reference 是要輸入到 GC table 中，藉由這些可以更新把 GC table 中所記錄的物件標示成可以回收，別的物件就可以重新使用此 Heap 空間，最後如果 pop 出來的 reference 判定為 method flag 就是代表這是這個 method 的結束，就完成此次的收集動作，接下來可以重新回到閒置狀態，等待下一次收集的要求進來。

在收集的過程，因為會有很多 return 的 bytecode 會出現，因此我們要如何判斷這個 return 是屬於某個的 invoke method 的 bytecode 也是值得討論的地方，在這邊的做法是，我們會設計一個累加器，當收到 invoke method 會加一，收到 return 會減一，當進入一個 method 開始收集的時候就必須記錄此累加器的值，因為進入 method 過程中也會有很多 invoke method 的 bytecode 和 return 的 bytecode，我們就可以藉由此累加器，判斷這個 return 的 bytecode 就是屬於當初我們引發收集的 bytecode，也就是收到 return 時而且當初記錄的值就是等於現在累加器的值，就是代表結束此次的收集。

由以上的運作我們可以知道，每當要創建物件就必須要等待一些 cycle 因此在之前的設計不用等待即可直接再 Heap 找到空間分配給此物件，因此我們在原先的設計上必須修改，再 Dynamic Resolution，準備要進入 Heap allocation 狀態時，必須多加入一個狀態，就是 wait for garbage collection state，意義再於等待 garbage collection 回傳位址，回傳之後才可以進行後續的動作，另一個需要修改的地方是再創立陣列的物件，再創立陣列物件有其他的 FSM，因此我們也需要修改，作法如同上述所講，也是新增一個狀態，去等 garbage collection controller 回傳位址之後，才可以進行後續的動作，其餘的狀態皆無改變。

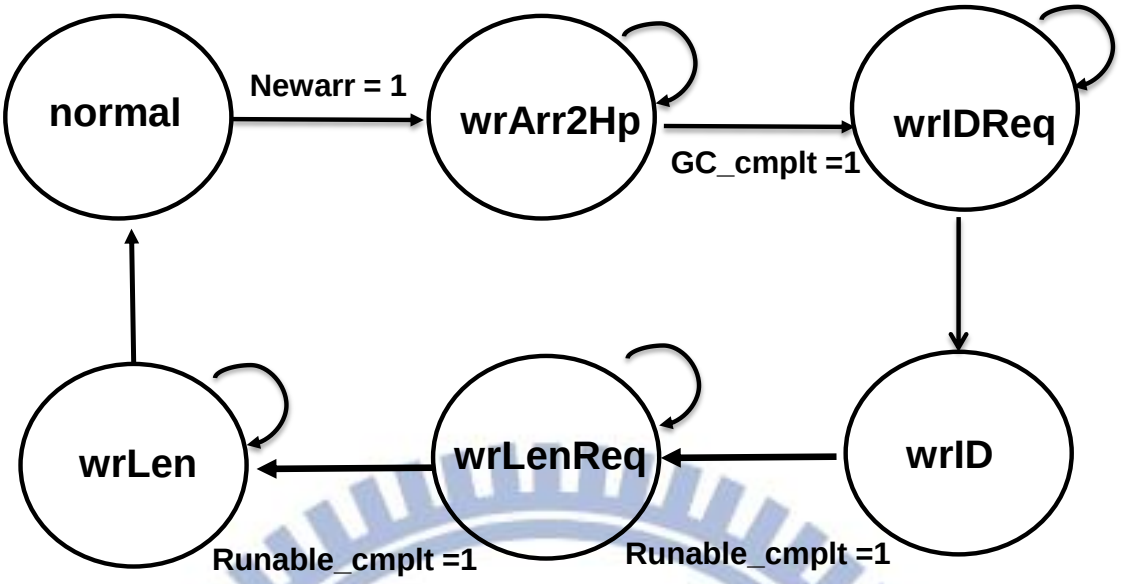


Figure 19. 修改後的 AASM

上圖就是多新增一個狀態 wrArr2Hp 另外剛所提到的 Dynamic Resolution 中要進入 Heap Allocation 時也是如上圖所示一樣的修改，因為所花的 cycle 長短不一，會有像迴圈一樣等待 completion 信號線才會進入到 wrIDReq 的狀態。每個狀態的意義就如命名一般，wr 就代表 write，WrLenReq 就是會先發送寫長度的請求，在來才會進入到下個 state 因此 wrLen 就代表要寫此次陣列的長度到 Heap 中。

還有另外一個修改是去停住整個 JAIP 的運作也就是，stall all 的信號線，由上述的加入另外一個狀態，就是要來等待 garbage collection controller 回傳位址，此時就也必須傳入一個信號告訴 JAIP 說現在整個系統也要停住，這樣才可以正確的執行。

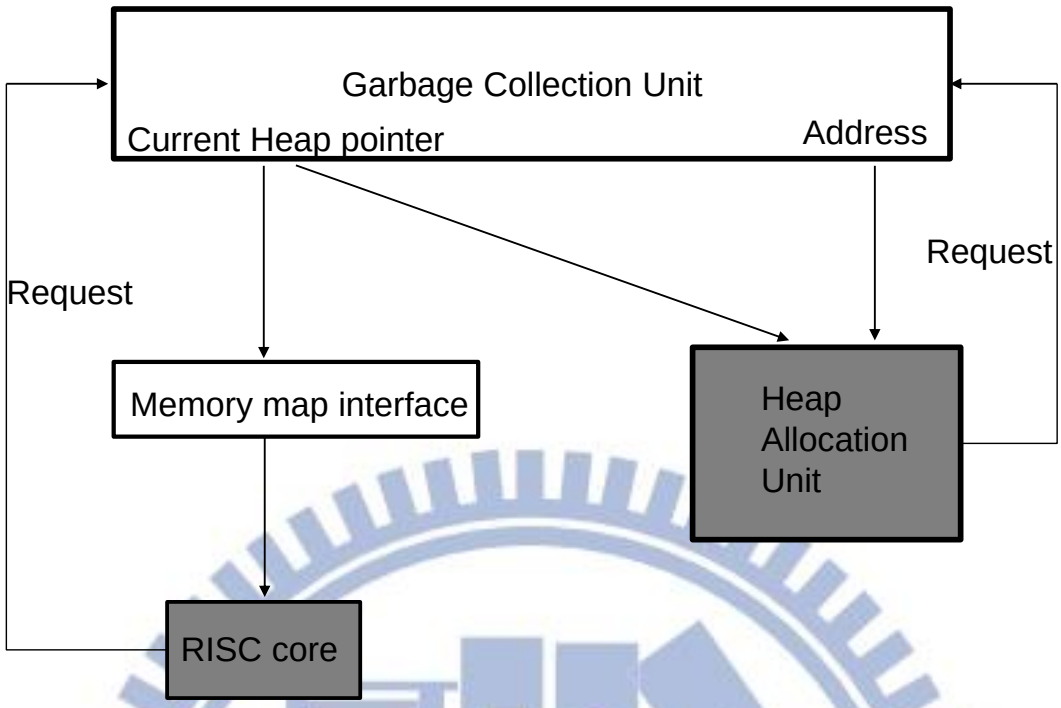


Figure 20. Garbage Collection 處理 RISC 方式

在上圖中，因為我們有兩個核心，除了 JAIP 外另外一個是 RISC core，但是我們實作主要再硬體實作，因此軟體部份我們沒有處理，也就是 RISC core 發送 request 要分配 Heap 空間時，Garbage Collection Unit 就直接給予目前 Heap 使用到哪裡的位址，所以我們只會分配 Current Heap pointer 給它，讓 RISC core 繼續寫下去，不會給予我們搜尋到可回收的空間使用，因為沒有做任何處理，我們也不知道此給予的區塊甚麼時候可以回收，我們會這麼做的原因是，我們開發此 JAIP 最終目的要全部的硬體化，所以日後在 RISC 將不會有要求 Heap 空間的需求。當 JAIP 要求 Heap Allocation Unit 分配空間我們就會處理，因此還會多拉一個信號，也就是 Address，表示這是目前搜尋到可以重複使用的空間。和 RISC 溝通時我們採用 memory map register 的方式，透過寫 register 的值，這些值就會傳送到 BUS 上，而 RISC 就會知道哪些值是屬於它所要接收，就可以達溝通的目的，實作上我們是採用 BRAM，這些 BRAM 的值是可以和 RISC 所互通的，在使用 xilinx 工具創建 IP 時，可以勾選是否要使用此 BRAM。

第四章 Object Cache Controller

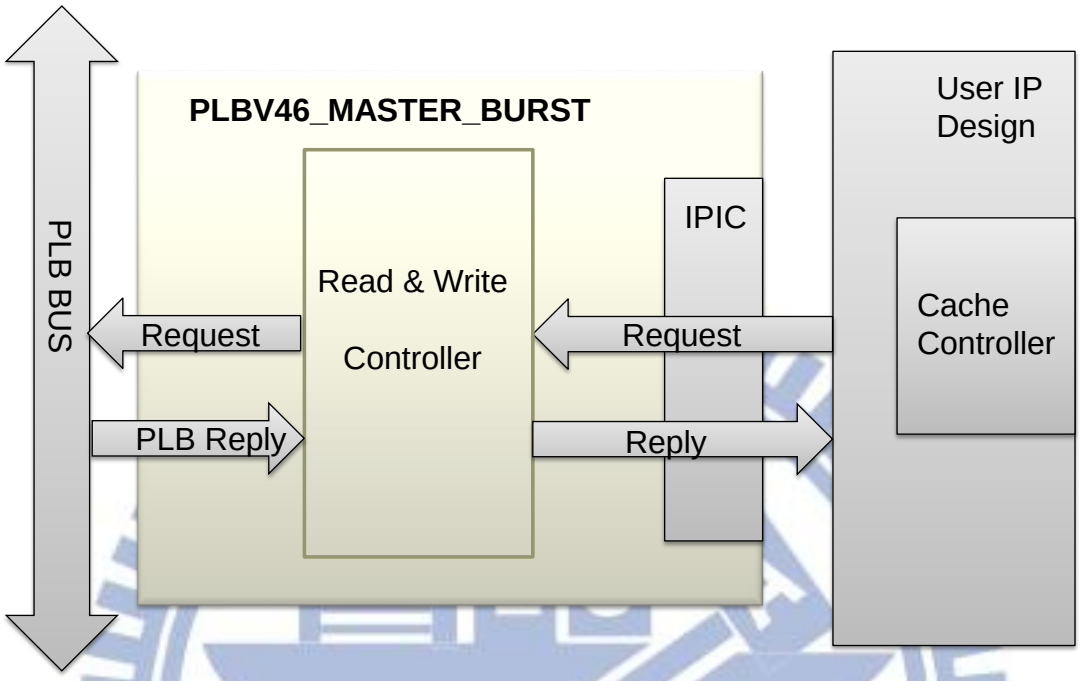


Figure 21. Object Cache Controller 與 PLB BUS 的溝通

在進入本章節之前本論文會先介紹上圖，Object Cache Controller 是在我們 JAIP 上的一個元件，之後我們不會直接跟 PLB BUS 直接溝通，而是透過 IP Interconnect(IPIC)，IPIC 是一個較為簡單的協定，可以讓客製化的 IP 與其 IP Interface(IPIF)溝通，我們無須知道 IPIF 如何設計，只要知道如何把信號線接到 IPIC 即可，就可完成和 PLB BUS 溝通。

4.1 Object Cache Controller 架構圖

本論文所使用的 Object Cache Controller 是採 2-way set associative cache，寫回記憶體的機制 cache 中有分 write through 就是寫到 cache block 也會寫到記憶體，另一個是 write back 也就是指有寫到 cache block 當條件觸發才會引起寫回記憶體中，我們主要採用 write back 機制，畢竟效能是最好的，不過由於設計 cache 的完整性我們兩個機制都有設計且皆會在本論文著墨，當其他程式設計師想要改變機制時只需要改參數即可。

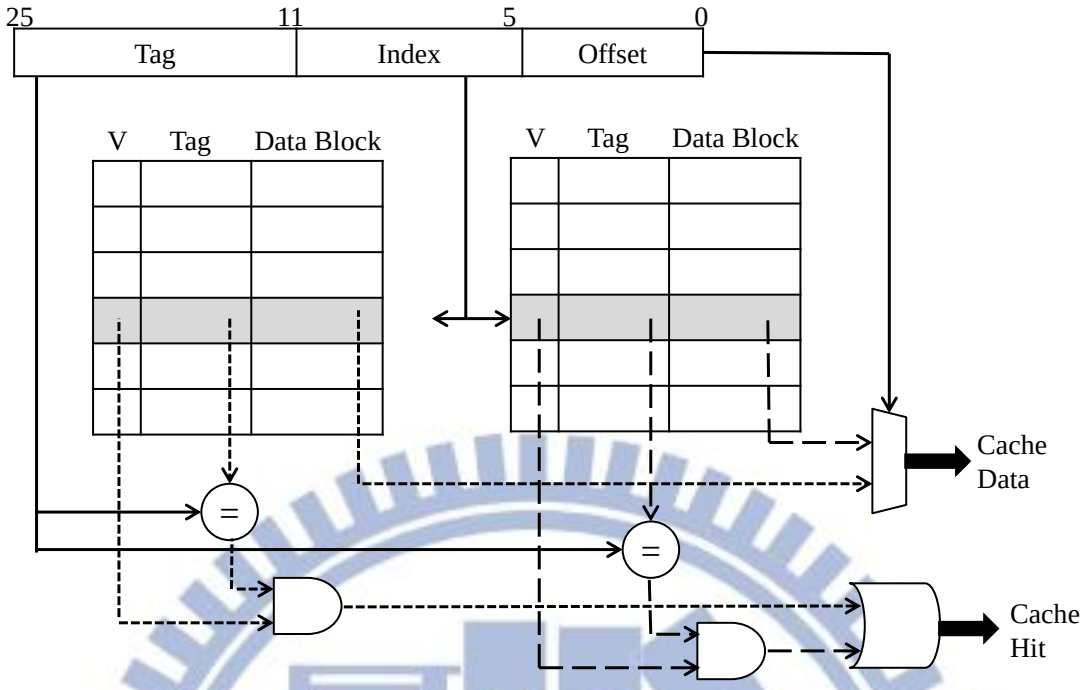


Figure 22. Object Cache Controller

Figure 22 就是 Object Cache Controller 如何取得 Cache Data 的示意圖，Index 是用來選擇區塊，選好之後就會把那一欄位取出來比較 Tag 是否相同，如果是一樣的就代表此資料存在 Cache 中不需花費長時間去存取記憶體拿資料， 反之，如果不同就會去記憶體拿資料，拿完後會把資料寫到 Cache 中，當下次要存取的時候就可以直接在 Cache 中取用。

Figure 23 是 Object Cache Controller 的實作結構，此 Controller 會接收一切要對記憶體做存取的信號當作輸入，之後會開始分析這些輸入，會產生相對應的 Tag、Index、Reference、藉由這些資料判斷此次的 Request 是否已經存在 Cache Storage 中，存在的話就不需要發送 Request 到記憶體中，直接從 Cache Storage 把所要的資料讀出然後完成。當不存在記憶體，Object Cache Controller 會發送信號到 PLB BUS 接下來等待 PLB BUS 回應，取得所要的資料才能完成此次的要求。Figure 23 中開頭以 Memory 開頭的命名表示是要傳送到記憶體的信號線，如果是以 Cache 開頭的命名就是給 Object Cache Controller 的信號線，和輸出資料。

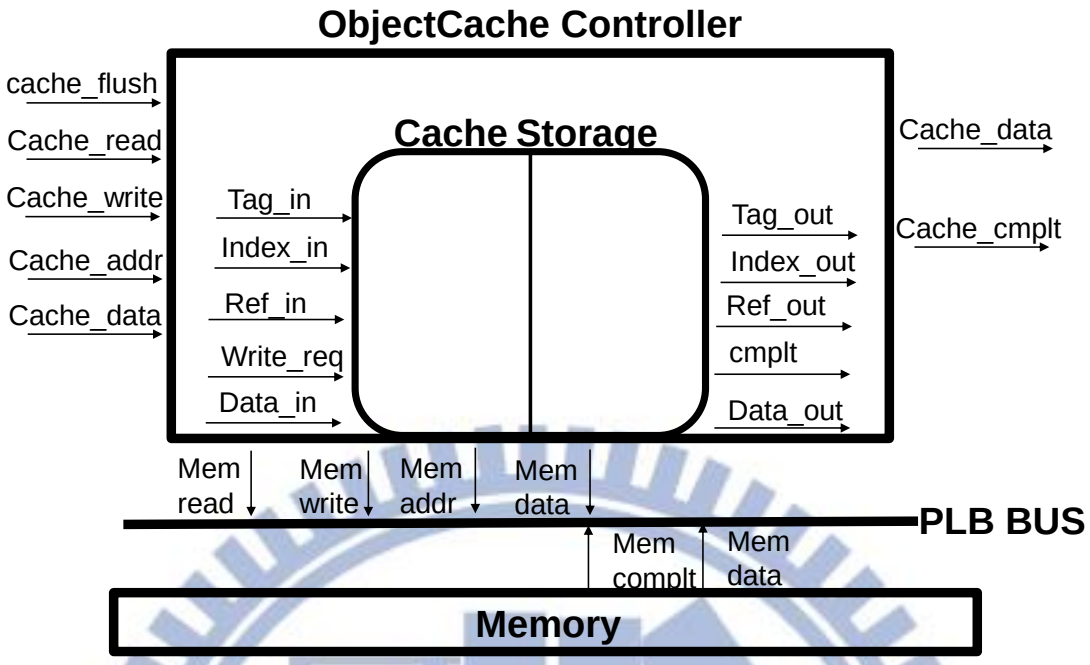


Figure 23. Object Cache Controller 內部信號

4.2.1 The Detail Design of Write through Policy

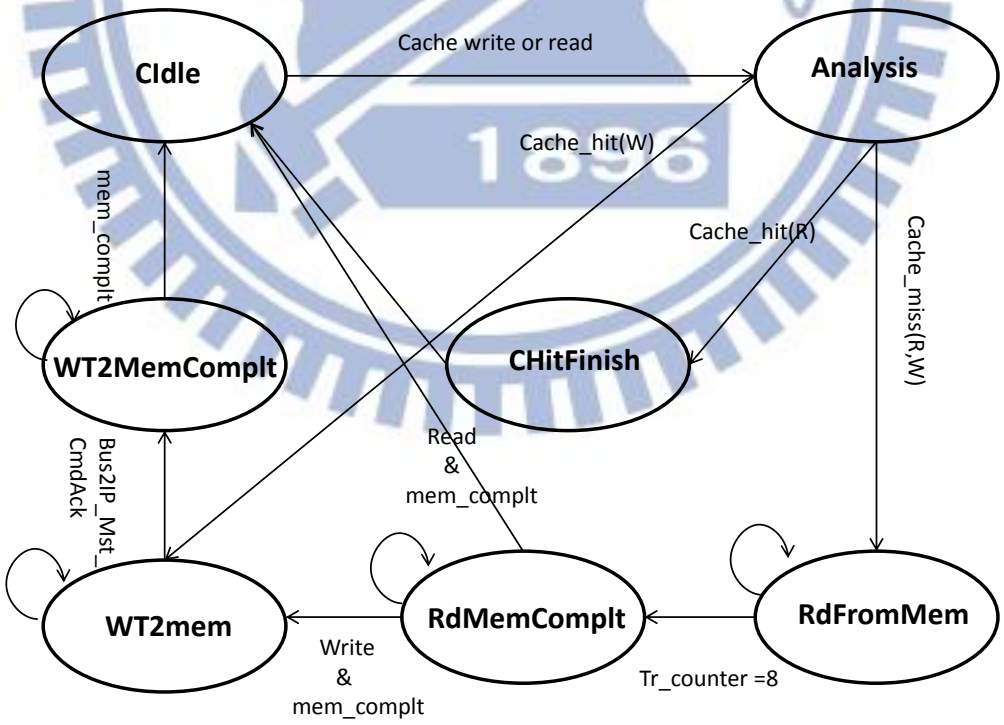


Figure 24. Write Through Policy

Figure 24 是 write through policy 完整的 finite state machine，接下來將會介紹

每個 state 所要做的事情，與所代表的意義。首先，當這此的 Request 資料是在 Cache Storage 中的話，而且又是讀取的 Request 就會從 CIdle→Analysis→ CHitFinish 又回到 CIdle，如果是寫入的 Request 就會從 CIdle→WT2mem→ WT2MemComplt→CIdle。當資料不在 Cache Storage 中且是讀取的 Request 會從 CIdle→RdFromMem→RdMemComplt→CIdle，如果是寫入的 Request 就從 CIdle →RdFromMem→RdMemComplt→WT2mem→WT2MemComplt→CIdle。有了以上的基本認知就可以對每個 state 做更詳細的了解。

(1) Analysis，此狀態是用來分析此次的 Request 是否在 Cache Storage 中，根據分析才決定下一個狀態要往哪邊。

(2) CHitFinish，此狀態代表，此次的 Request 是讀取，而且可以在 Cache Storage 中找到符合的資料，直接把資料傳出，不需要透過記憶體，從發 Request 到資料輸出只需要三個 cycle。

(3) RdFromMem，代表此次的存取是 miss 都會到此 state，之後會產生相對應的信號線傳給記憶體，說明此次要讀取哪部分的資料，由於我們一個 cache block 是八個 word 因此會用 burst mode 一次拿八個 word 回 Cache Storage 然後輸出其中的一個 word。

(4) RdMemComplt，此狀態是代表 RdFromMem 的中繼點，因為當把八個 word 讀取完之後，必須要去判斷是否要寫到記憶體中，或是可以回到 CIdle 代表完成了此次的要求。

(5) WT2mem，代表接下來要對記憶體執行寫的動作，因此這邊要產生相對應的信號線送到記憶體中，像是位址或要寫入的資料，在此同時我們這邊的設計也會順便寫入 Cache Storage 中，利用 overlapping execution 的技巧，同時寫記憶體又同時寫 Cache Storage 我們可以省更多 cycle。

(6) WT2MemComplt，是為了傳送對寫記憶體的 Request 已經完成所以可以回

到 CIdle stage 接受下一次的 Request。

4.2.2 The Detail Design of Write back Policy

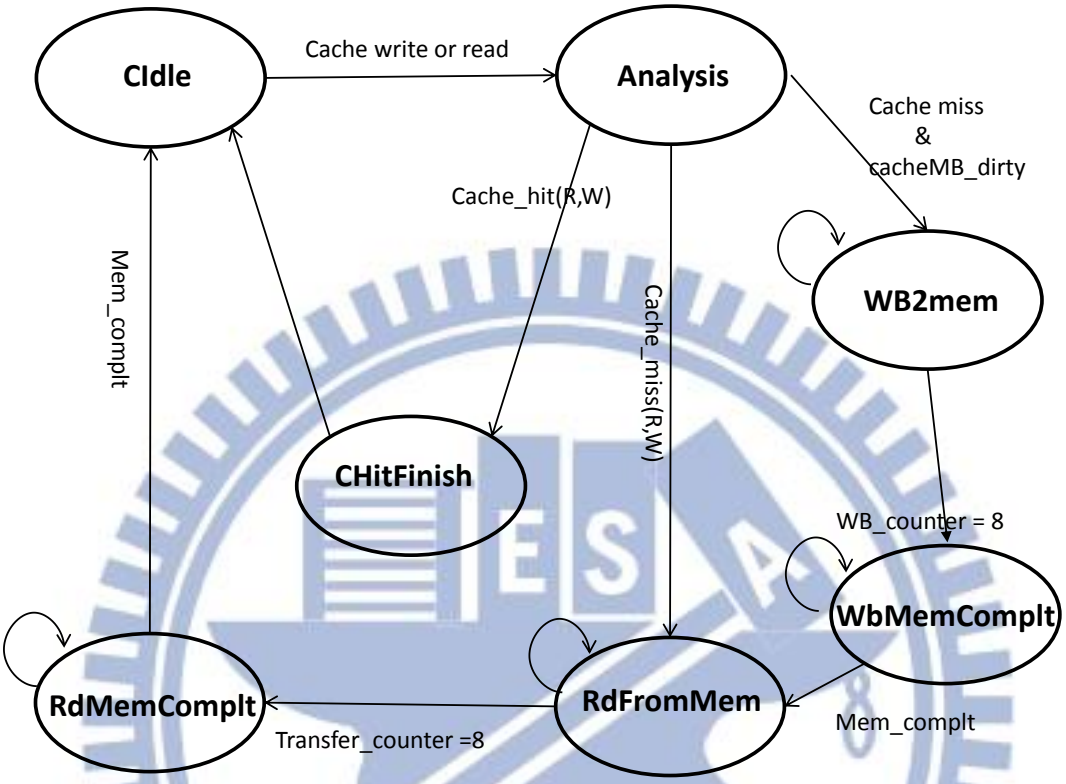


Figure 25. Write Back Policy

此機制不像 Write Through 不會每次都寫回到記憶體中，而是當要被置換掉的 Cache Block 是 Dirty 才會寫回到記憶體中，對於記憶體的存取變少了因此可以把效能大大提升，但實作上比較複雜，接下來本論文會再對每一個狀態作詳細的描述。部分有跟 Write Through 重疊的狀態就不會在此作介紹，但有些需要再重新解釋才會提出，當初本來是設計出兩個獨立的 FSM 當設計完後發現，整個 FSM 很難讀懂，也怕將來要作修改不易我們也發現有很多重複的功能可以整合在一起，一方面可以讓整個 FSM 變得相當簡單，二方面可以更容易理解也更容易去實作，所以我們決定把兩個 FSM 重疊。

(1) WB2mem，會到此狀態代表此次的存取是 miss 所以需要去記憶體把資料引進 Cache Storage 中，且發現用 Index 所指到的 Cache Storage 皆是滿的，

而且 Victim Block 也就是要被置換的 Block，此時是 Dirty，代表此值和記憶體中的值是不一致的，因此必須把這些值寫回到記憶體中，保持記憶體的一致性。在此也會產生相對應的信號線這些是屬於 Master Burst Write 的信號，送到記憶體中，要寫回到記憶體中的資料也是八個 word，因此我們有用一個計數器去計算是否完成等待完成後就會進入到 WbMemComplt 狀態。

(2) WbMemComplt，此狀態代表寫八個 word 資料到記憶體中已經完成，此時會進入到 RdFromMem。

(3) RdFromMem，此狀態代表將會從記憶體把資料讀出來，這邊我們也有採用 overlapping execution 技巧，當一邊讀記憶體資料，一方面就可以直接寫入到記憶體中，不需要等到八個 Word 讀取完成，只要記憶體發出信號說已經把八個 Word 送完，也代表我們的 Cache Storage 也把資料更新完畢，可以進入到下一個狀態中。

因為我們的處理器是雙核心，因此採用 Write Back 就必須要確保每顆核心所看到的資料是一致的，但因為我們的 RISC core 會去對記憶體作讀寫次數比較少，而且大部分的執行皆在 JAIP 中，因此這邊我們設計的演算法會盡可能的簡單，而且不會去影響到效率，這邊我們針對 RISC core 會提出寫和讀得請求一一作探討。第一當處理器要作寫入時，我們會給出一個位址，而且此位址不會在 Cache Storage 中，這樣就確保當 JAIP core 要讀剛剛 RISC core 寫的資料就必須去記憶體中讀出再來就是要作一個對齊的動作，主要原因是我們使用 Burst Mode 每次八個 word，假如沒有作對齊的話，而且此八個 word 中後面四個都是還未使用的位址，因此會丟給 RISC 作寫入，此時換到 JAIP 要執行時，就會以為資料皆是在 Cache Storage 中就直接拿資料出來，殊不知此時資料是在記憶體中，因此每當要寫入我們會對齊八個 word 才會進行寫入。第二點當 RISC core 需要作讀取時，這邊我們就會啟動 flush 功能，此功能會把所有 Dirty block 全部寫回到記憶體中，這就可以確保

RISC core 和 JAIP core 所看到的資料皆是一致的，此法相對其他方法來說比較簡單但也較沒有效率，但是因為我們只有少部分的功能才會使 RISC core 去讀記憶體中的資料，因為大部份我們都是在硬體方面實作，未來的設計我們會朝向把所有 RISC core 會作到的事情都移到 JAIP core 去執行，也就是實現全面的硬體化，接下來我們會介紹 flush 的功能。

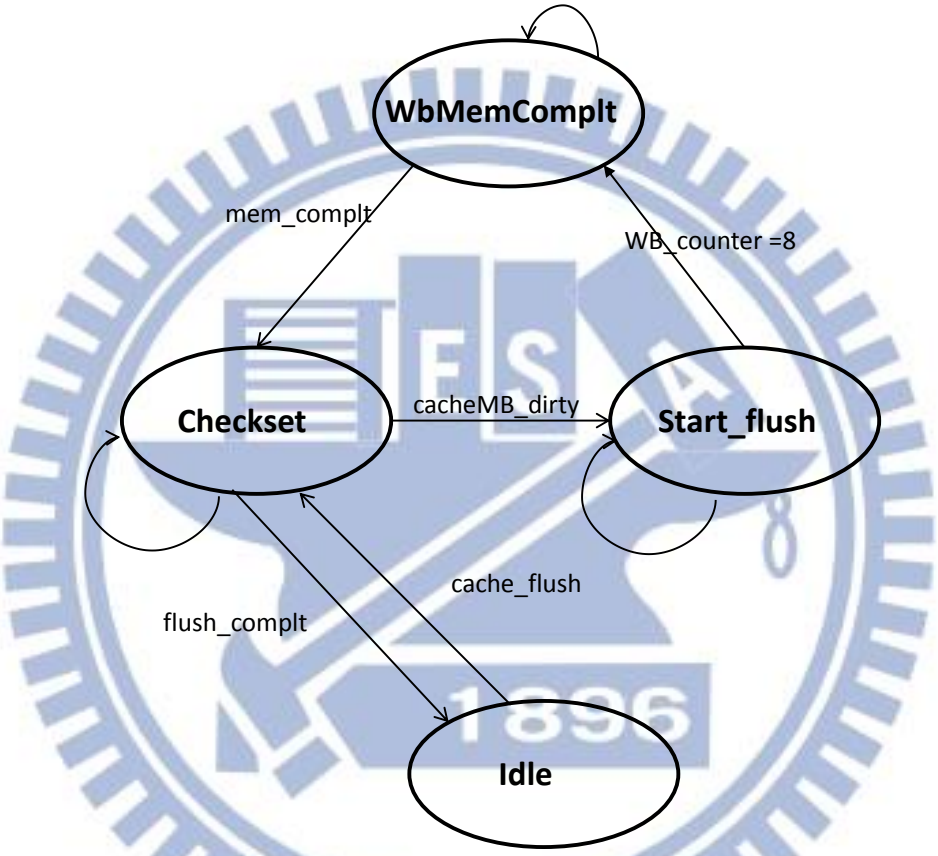


Figure 26. Flush Policy

當 cache_flush 傳送進來就會進入到 Checkset 的狀態，皆下來回從第零個集合檢查到第最後一個集合，如果途中有發現任何 cache block 是 Dirty 的就會進入到 Start_flush 開始把資料寫回到記憶體中，當寫完後就會進入到 WbMemComplt 就代表已經完成寫入到記憶體，接下來在回到 Checkset 中，在一直循環檢查哪一個集合該寫回到記憶體中，當所有的集合都檢查完畢後，此時就會拉起 flush_complt 代表現在的 Cache Storage 皆無 Dirty Block 也就意謂記憶體和 Cache 的資料是相同的，就能確保資料的一致性。

4.3 PLB Master Burst Read and Write

本章節將會解釋我們 Object Cache Controller 對記憶體的存取皆是以八個 word 為單位，首先看下表 1。

Table 2. PLB Master Burst Read

PLB Master Burst Read						
Data Length	1	2	4	8	16	32
# cycle	36	37	39	43	51	108

我們可以清楚知道當每次傳輸一個 word 實所需要的 cycle 是 36 個 word，二個則需要 37 個 word，當 Length 是八個 word 時，平均每傳輸一個 word 只需要 5 個 cycle，當傳輸 Length 是十六個 Word 時，平均傳輸一個 word 需要 3 個 cycle，因此從此表格我們知道 Length 是十六以上時是最划算了。

Table 3. PLB Master Burst Write

PLB Master Burst Write						
Data Length	1	2	4	8	16	32
# cycle	17	16	18	22	30	102

上表也告訴我們當 Length 是八或是十六時對記憶體 Burst Write 是最有效率的。綜合以上兩個表格的分析，我們決定使用每次 Burst 的長度是八個 word，一方面可以節省空間，二方面可以兼顧空間區域性，因為當一個項目被存取到，附近的項目也很快會被存取到。

在 Object cache controller 使用命名信號時我們這邊也使採取跟 xilinx 規則一樣，當開頭以 BUS2IP 就表示從 PLB BUS 傳給 JAIP 的信號，反過來用 IP2BUS 就是代表從 JAIP 要送給 PLB BUS 的信號，這樣做法是要達到和 IPIC 命名的一致，因此可增加程式的可讀性，下表我們標示一些使用 Master 介面時一些信號線的使用，和注意事項，在此我們只列出部分需要注意的，一些比較不重要將不在此列出。

Table 4. 使用 PLB 信號注意事項

Signal name	note
Bus2IP_MstRd_eof_n	當為 0 時代表此次的讀取結束
Bus2IP_MstRd_src_rdy_n	當為 0 時表可以讀取 BUS 傳來的資料
IP2Bus_MstRd_dst_dsc_n	必須永遠保持 1
IP2Bus_MstWr_sof_n	告訴 PLB，這是我第一個要寫的資料
IP2Bus_MstWr_eof_n	告訴 PLB，這是我最後一個要寫的資料
IP2Bus_MstWr_src_rdy_n	告訴 PLB，可以準備開始寫
IP2Bus_MstWr_src_dsc_n	必須永遠保持 1

還有要注意的是，如果要使用 burst mode 必須在創建 IP 時勾選 Master Performance 選項中的 burst support 否則將會無法使用 burst 功能，PLB Bus 只會回傳錯誤訊息到 Bus 上，如果當時創建 IP 時沒有勾選，還可以利用手動的方式修改，就是可以修改在 pcores 資料夾中的 mpd 檔案 ，手動加入 GENERATE BURSTS = TRUE 如 Figure 27 所示，修改此檔要特別注意，如果修改不慎會導致在產生 bitstream 時無法順利產生。

```
## Bus Interfaces
BUS_INTERFACE BUS = MPLB, BUS_STD = PLBV46, BUS_TYPE = MASTER, GENERATE BURSTS = TRUE
BUS_INTERFACE BUS = SPLB, BUS_STD = PLBV46, BUS_TYPE = SLAVE
```

Figure 27. 要修改 mpd 檔的部分

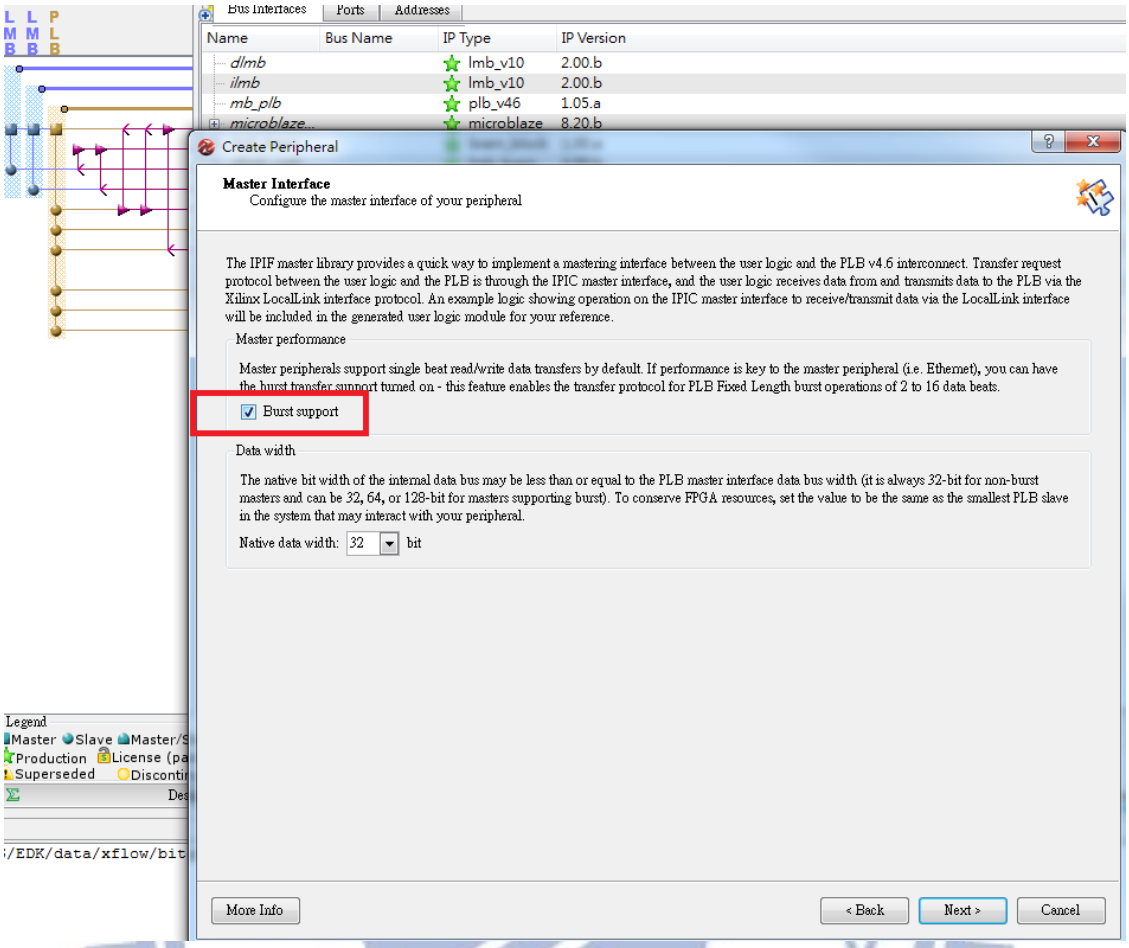


Figure 28. burst 使用注意事項

4.4 Object Cache Controller 運作和設計重點

此章節會介紹 Object cache controller 的運作方式，會區分為兩類的流程圖介紹，第一類是 write through，第二類為 write back。當一個對記憶體存取的要求傳送到 Object cache controller 此時就會停住整個 JAIP 系統，也就是管線就會 Stall 住，直到 Object cache controller 把資料回傳，才會恢復 JAIP 系統的運作，因此我們整體的設計導向是以效率為準則，當停住 JAIP 時間越短，效率也就越高。像採用 write back 時，此時選擇 victim block 就變得相當重要，因為當替換掉的 block 是 dirty 就必須耗費大量的時間寫到記憶體中，因此我們的演算法一定會先選要 dirty 不為 1 的先替換掉，再來才考慮其他因素。Write Through 就沒有選 victim block 的問題，因為每次的寫入都必須寫回記憶體，因此效率也較 write back 差，但如

果在要求資料與記憶體需永遠保持一致性下時，就可選用此法。

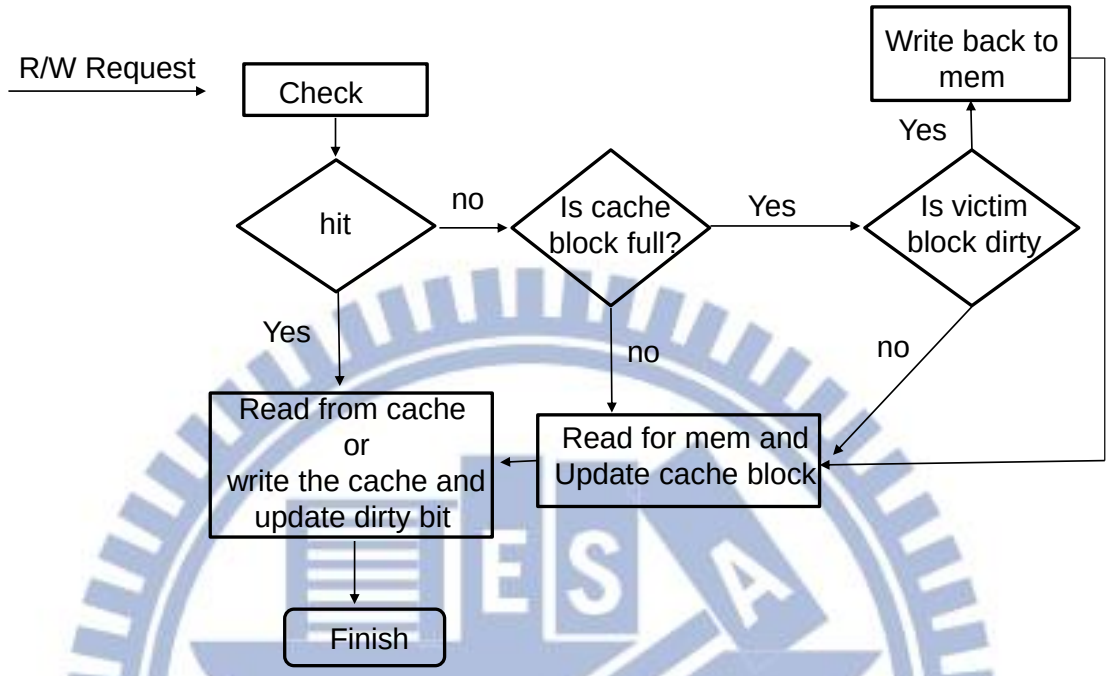


Figure 29. Write back 流程圖

一開始收到 Request 就會開始去檢查 cache block 中有無把此次的資料快取住，有的話就可以不用到記憶體讀取資料，如果是讀取的要求，就可以直接回傳資料完成此次的任務，如果是寫入的要求，因為採取 write back 機制就只需要更新把 dirty bit 設為 1 即可，不用在寫到記憶體中，以上是 hit 的狀況，如果沒有 hit 就會先去先檢查所對應的 cache block 是否已經滿了，如果滿了而且 dirty bity 為 1 就必須把資料寫回記憶體中，代表 cache block 中的資料是最新的，記憶體中的資料是舊的反之 dirty bit 為 0 就不必寫學記憶體中，如果所對應的 cache block 不是滿的就可以直接把記憶體資料寫進去即可，就不必去檢查 dirty bit，下面將繼續講解 write through 的流程。

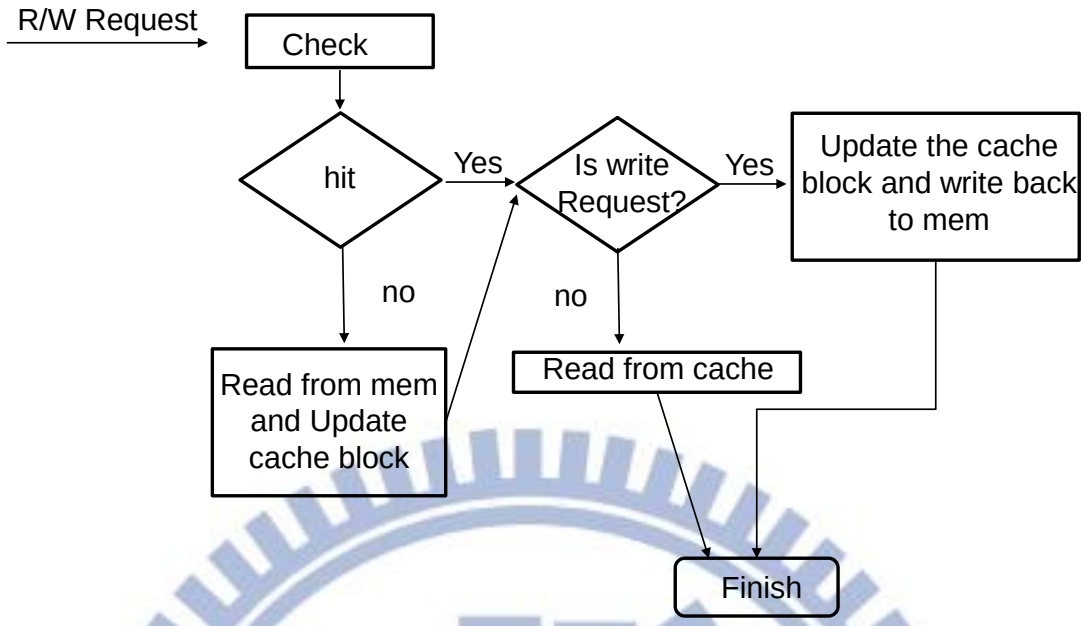
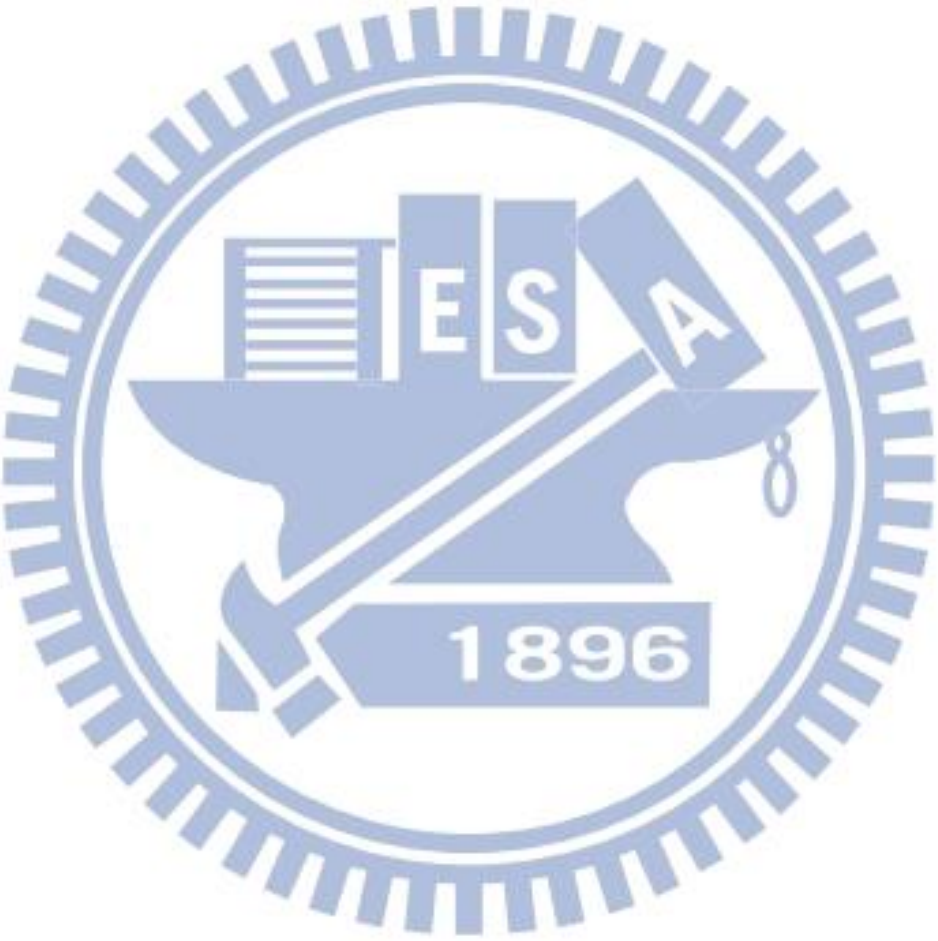


Figure 30. Write through 流程圖

收到 Request 也是先去檢查 cache block 中有無把資料快取住，如果有快取的話就在檢查此次是否為寫入記憶體的要求，如果是就必須更新 cache block 中的資料另一方面也要更新記憶體中的資料，因為採取 write through 的機制，記憶體和快取的資料永遠都會保持一致性，如果沒有命中就會先去從記憶體中把資料寫到 cache block 中，之後就是採取跟有命中的機制一樣。至於讀取的時候在此機制下就很簡單，有命中就直接把資料回傳就可以結束，如果沒有命中就多一個步驟就是去到記憶體中把資料讀到 Cache block 中，接下來就把資料讀出來完成此次的 Request。此兩個流程圖沒有甚麼太大差異，但是實作 write back 要寫的邏輯比較多，需要注意的地方比較多，會有 coherence 問題也是在 write back 中會出現，因為不像 write through 永遠會保持與記憶體資料的一致性。

不管是 write through 或是 write back 一開始會先檢查是否在 cache block 中，也就是有無 hit，接下來就是再根據此次的要求是要寫記憶體還是要讀記憶體，才會有不一樣的 FSM 去處理。觀察以上兩個流程圖可以知道，兩個策略的行為很像，因此在設計 FSM 上面可以分開設計，也可以選擇一起設計，前者的優點是，可以

較容易閱讀，但所使用的資源會比較多，因為需要更多 state 去記錄狀態，後者優點是可使用資源比較少，因為功能很多可以重疊，但是設計上較分開設計複雜，我們因此選用一起設計，畢竟資源在嵌入式系統也是必要的考量。



第五章 實驗結果

5.1 實驗環境

本論文所用的實驗開發板是 Xilinx Virtex-5 ML507，處理器則選用 Microblaze 是一顆 Soft Processor，工作頻率為 83.3MHz 開發工具也是 Xilinx 所提供的 XPS 和 SDK 前者則是用於硬體設計，後者則是用於軟體設計，下圖則是我們完整的實驗環境。如圖所示本論文的 Cache(16KB) 和 Heap Management 與 JAIP 整合成為一個 IP，程式碼是以 VHDL 語言為主。Table 5 中是我們本設計所用資源表。

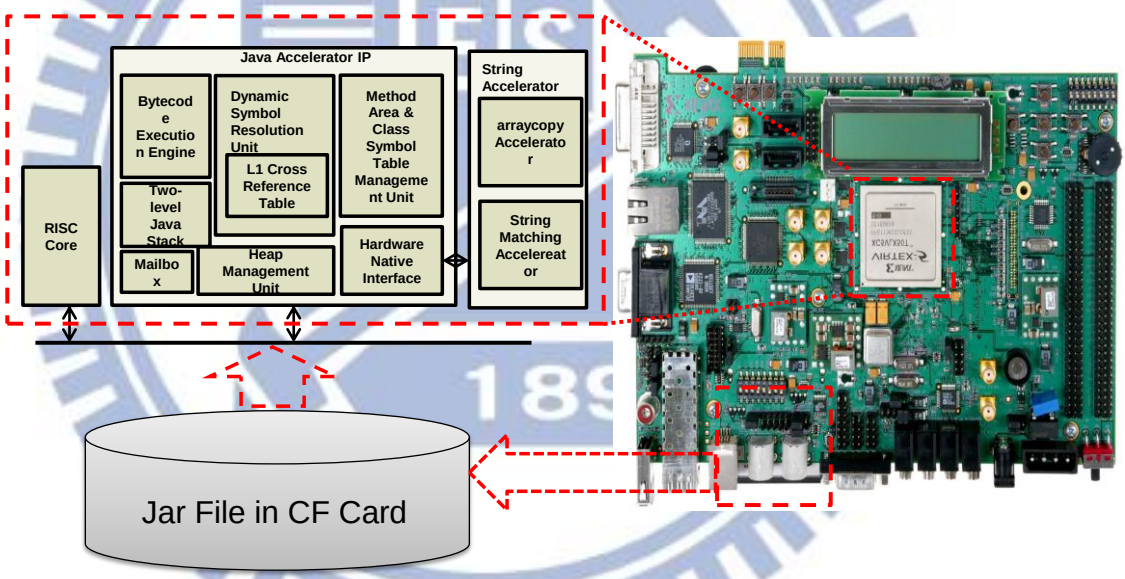


Figure 31. 實驗系統架構圖

Table 5. jaip 合成資訊

Device : vertex-5 ML507	
Number of LUTs	13238
Number of Flip-flops	6747
Number of 2K BRAM	26
Maximum frequency	83.3 MHz

5.2 benchmark 分析

第一個我們使用 Embedded java Benchmark suite Jembench[3]是專門用來測試 java 處理器的效能，裡面包括很多 Benchmark 我們取用 Kernel Benchmarks 和 Application Benchmarks ，其中前者有 sieve 和 bubble sort，後者有 Kfl、Lift 和 Udpip，不同的 benchmark 就有不同的特性，kernel benchmark 就是使用簡單的演算法和迴圈去設計，但缺少物件導像的設計，相反的 Application Benchmark 就比較偏向實際的應用程式，設計也是以物件導像為主，而且大多避免產生垃圾物件。

第二個使用 Embedded CaffeineMark(ECM)來進行測試，Jembench 雖然是物件導向的設計，它在設計上避免產生垃圾物件，但 ECM 中的 StringAtom 會使用大量的物件，而物件就是放在 Heap 中，因此如果要測試我們 Heap 改進的效能就可以在此上面測試。

在分析上我們著重於我們是否有改善目前平台的效率，在來就是用同樣的 benchmark 也跑在 sun's CVM 上做比較，CVM 是嵌入式的 java 虛擬處理器執行於嵌入式的 Linux 上，我們使用 Xilinx ML-405 的 PowerPC 處理器，工作頻率設為 83.3MHz。並且在 CVM 上有支援 Just-In-Time(JIT)的軟體加入功能，接下來將會把 Object cache controller 的測試和 Garbage Collection 分開討論，在 Object cache controller 上我們不會裝載 Garbage Collection 單純只看 cache 所帶來的影響，皆下來討論 GC 的時候會把 Object cache controller 引入一起討論，因為著重在整體 Heap 使用率的探討，因此如果 Object cache controller 有影響的部分也要一起加入探討，才可以知道是否有相互影響的效果。

5.2 cache 校能測試

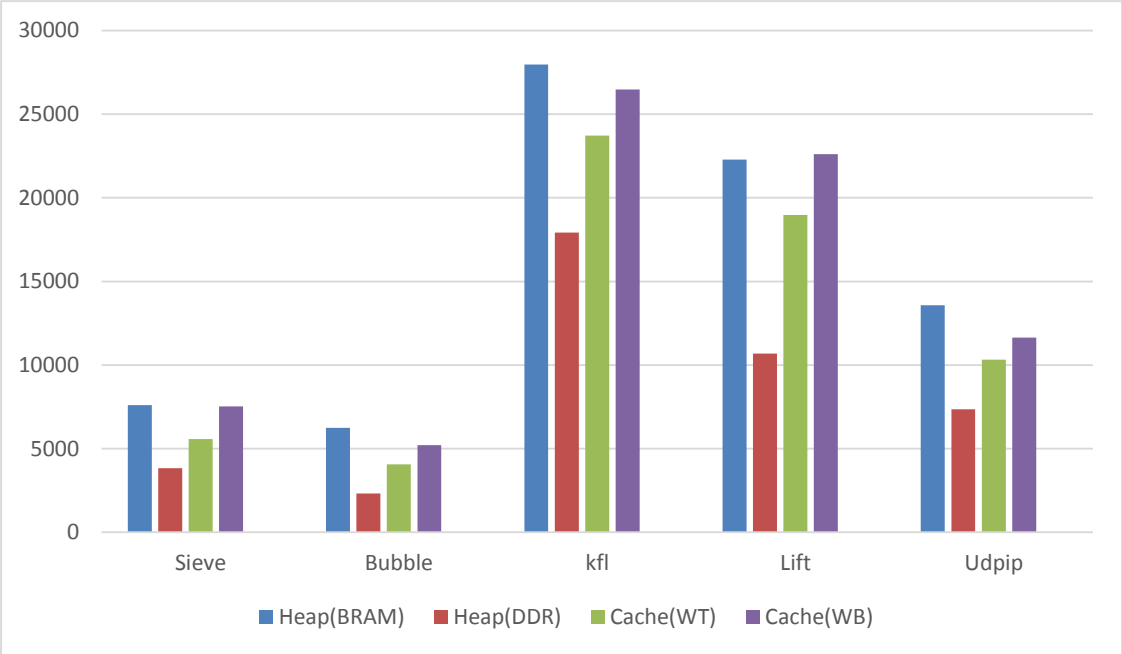


Figure 32.cache 測試圖

原本的 Heap 空間是放在 on-chip memory 也就是 BRAM 每次的存取只需要兩個 cycle，因此這個效能是最好的，所以是上限值，當我們 Heap 空間放到 DDR2SDRM 時，我們可看到所有的 benchmark 除了 Kfl 外都掉了兩倍以上，但是引進 cache 後可以看到效能就有大幅的提升，幾乎可以達到跟使用 BRAM 時候的效果，圖中我們也又放入 write through cache 做比較，還是可以清楚看出效能都比 write back 差，但 write through cache 如果用在處理 coherence 問題上可以比較好解決，write back 所要設計的部分就相對複雜。圖中的括號裡面的字代表用甚麼方式去實作，比如說 Heap(BRAM)就是表示 Heap 放在 BRAM 上，然而綠色的長條標示 cache(WT)代表 heap 放在 DDR 上面，是有加入 cache 而且寫回機制是使用 write through。我們原本會期待加入 cache 後效能是會更好，但沒達到我們的期望，因為我們就算把 Heap 搬到 DDR2SDRAM 時，中間還是有透過 MPMC 的機制，此機制有 caching 的功能，來提升對記憶體存取的效率。

Table 6. Application execution time (milliseconds) of JAIP

Test\#Iter	1	10	50	100	500
Logic	1	16	81	163	816
Pi	170	1705	8528	17057	85285
Kfl	0	1	1	4	19
Lift	23	23	25	27	45
UdpIp	0	1	5	8	40

Table 7.Application execution time (in milliseconds) of CVM-JIT

Test\#Iter	1	10	50	100	500
Logic	55	72	144	235	962
Pi	213	1884	8561	16943	84118
Kfl	1	4	19	46	172
Lift	11	13	25	41	120
UdpIp	2	8	39	58	218

在 Table 7 和 Table 8，我們把 Jembench 中的 Kfl、Lift 和 UdpIp 的程式單獨拿出來測試數據，因此可以調控跑的次數，PI 這支測試程式會去計算 PI 到小數點五百位，由這些資料顯示，一開始 JAIP 的分數都是勝過 CVM-JIT 的，但時隨著跑的次數越多分數開始被超越，因為 CVM-JIT 會去快取之前跑過的資料，因此開始展現出優勢，但是到了實際上的應用程式，不會有程式會一直以一樣的程式在跑。

Table 8. Lift execution time

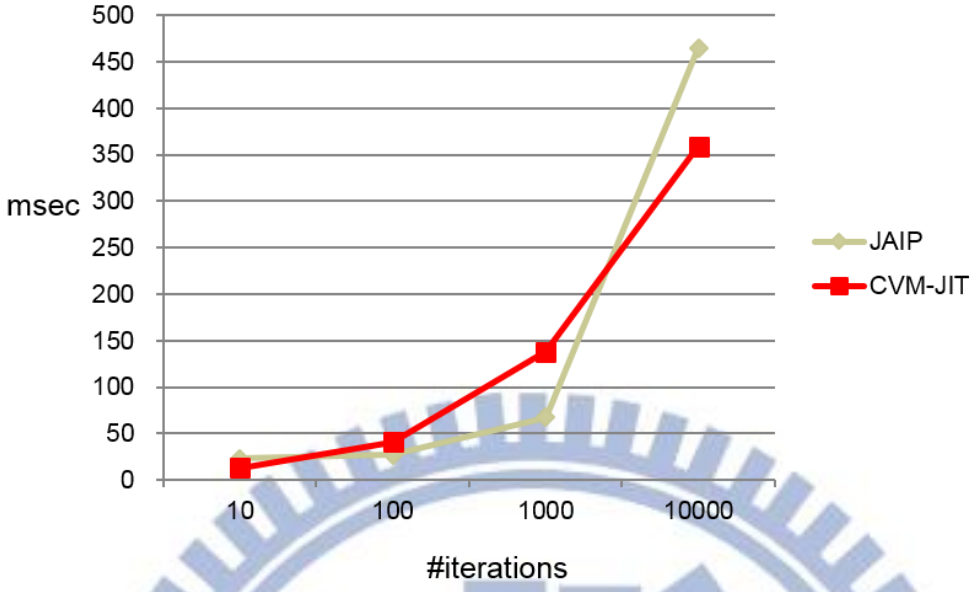


Table 9. Kfl execution time

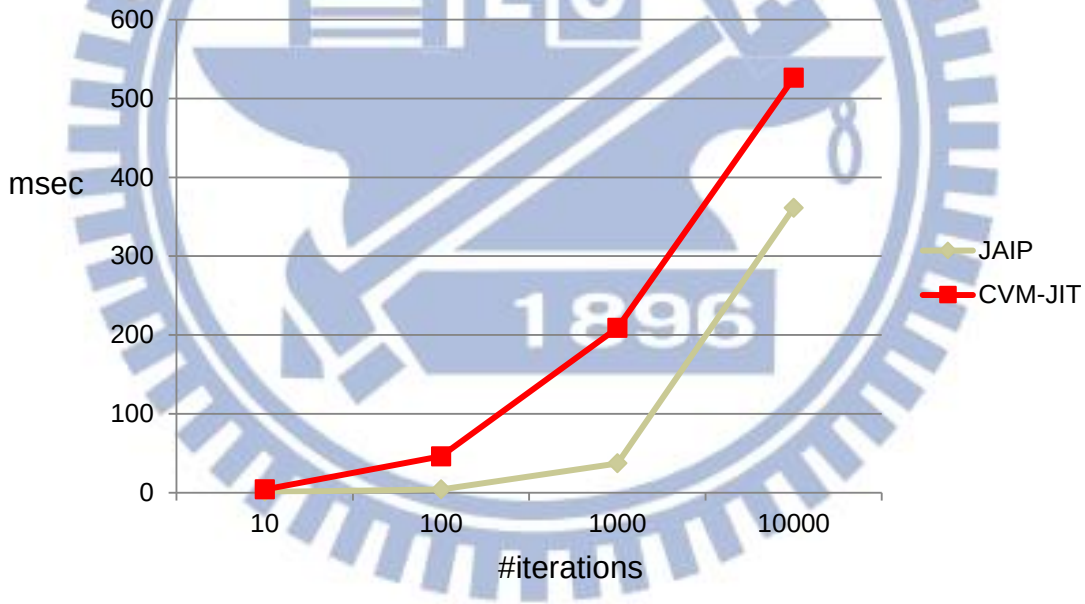
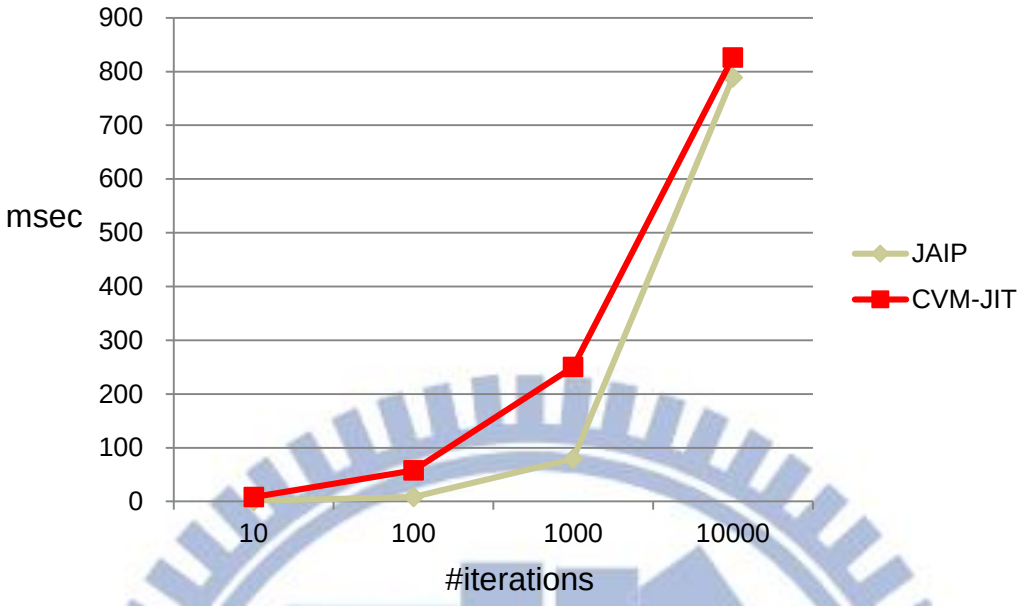


Table 10. UdpIp execution time



因為在 table7 和 table8 看不出來差異性，在 table9、table10、table11 中我使用更多的執行次數去觀察，JAIP 的效能是可以贏過 CVM-JIT 即使執行重複程式的次數已經超過一萬次，但除了 LIFT 此 benchmark 除外。

下面的圖我們使用 ECM 中的 StringAtom 來測試效能，因為此 benchmark 會產生很多的物件，因此對 Heap 的存取負擔很重，更能看出加入 cache 後所帶來的影響，整體的分數提升至少了兩倍。

Table 11.JAIP 執行 ECM StringAtom benchmark 分數

Platform	StringAtom score
JAIP	390
JAIP (with cache)	807

5.2 Garbage Collection 校能測試

Table 12. GC 資源使用圖

Device : vertex-5 ML507	
Number of LUTs	650
Number of Flip-flops	720
Number of 2K BRAM	5

Table 13. GC 在 Jembench 上校能測試

	Kernel Benchmark	
Version	Original version	With GC Version
Sieve	7522	7522
Bubble sort	5207	5207
	Application Benchmark	
Kfl	27745	27745
Lift	22614	22614
Udpip	12671	12671
Heap Usage	15.48KB	14.66KB

在 Table 12 是 garbage collection 的資源使用圖，Table 13 則是加入 garbage collection 後整體系統的測試,我們可以知道整體效能沒有因為加入GC後而下降，反而與之前差不多，但我們先前設計有提過加入 GC 是一種負擔因為每次在建立物件的同時我們都會去搜尋表格然後在做後續處理動作以便記錄物件狀態，而且

隨著物件產生越多負擔會用重,後來我們發現因為我們使用 Object cache controller 去管理 Heap 空間而且寫回記憶體的機制是採用 write back，加上裝載 GC 後會去覆寫已經可以回收的空間，而不是去存取記憶體把資料拿到 cache block 中，因此會節省去存取記憶體的時間，這些時間剛好可以跟上述所說的負擔打平。

下面我們使用 ECM 中的 String Atom 來測試，可以看到原本為裝載 Garbage Collection 前使用了 29.42MB，這對一個嵌入式系統是一個很重的 Heap 使用量，裝載 Garbage Collection 後可以降至 14.25MB,而且在整體效能上幾乎持平還些許超過，空間可以節省 47%的資源。

Table 14. 裝載 GC 後的 ECM 測試數據

Version	Score	Usage
Original version	2095	29.42MB
With GC version	1986	14.25MB

第六章 結論與未來展望

本論文設計兩個可以增進 Heap 效能的實作，一個是使用了 Object cache controller 使得資料之間可以更快速的傳遞，而不用浪費大量 cycle 在傳遞資料上，另一個我們設計了 Garbage Collection 增進 Heap memory 的使用率，把無用的物件可以回收再利用，而且意外的發現 cache 和 Garbage Collection 一起使用效能會更好，因為可以少掉去讀取記憶體裡面的資料，直接覆寫現在的資料即可。

Object Cache Controller 目前是 2-way set associative cache 在未來上可以設計更複雜的 4-way 或者是 8-way，來比較效能是否可以更上一層，但相對的硬體成本就會提高，因此如何在效能和成本間取得平衡也是另一個議題。此外也可以設計好的 pre-fetch 機制，可以判斷在沒有長時間沒有使用 Heap 時，可以由 Controller 本身去抓預測未來會執行的資料放進 Cache storage 中，或者是當一開始 cache Storage 都沒有資料的時候，就可以開始從頭把資料一一的放進來，但必須注意 priority 的設定，畢竟要以 JAIP 所發出的要求為最高 priority，pre-fetch 絕對不可以高於 JAIP。日後如果有更多關於判斷 victim 的演算法引入，也可以設計動態的切換演算法來達到更好的效率

Garbage Collection 方面我們是提供兼顧效能又可以回收記憶體的方法，主要原因我們有採用 two-port BRAM 和兩個控制器來處理，使得分配物件和回收物件兩件事可以同時進行，不會互相影響效能，在處理記憶體碎裂的問題，我們使用表格和指標的資料結構去記錄物件而不是去搬動記憶體，雖然效果沒有移動記憶體來的好，但使用此方法對校能影響是最小的，因此往後如果更注重在記憶體碎裂的問題，可以使用 memory compaction 相關的演算法，有些就是把以使用的物件會分一塊，為使用的區域分一塊，固定時間去把物件排列好，但此分成兩塊的缺點就是記憶體的使用就會少掉一半，原本有 32MB 的容量，分一半就會變成 16MB 的容量，或者可以選用指標法透過修改指標達到記憶體移動的效果，但都

會影響性能，因此必須在效能和記憶體使用率做一個判斷，因為目前 BRAM 使用量偏多，光是 reference 的欄位就必須耗費 22 個 bit 去記綠，未來可以使用對應的方法來減少此 reference bit 的使用，比如是以 2 的次方去存 Heap 中的資料，但缺點就是會有更多的內部碎裂。而且因為現在每次產生出來物件就會表格中搜尋，是否有可用的物件可以用或使要分配新的物件，演算法我們採用 First-fit，所以接下來可以測試不一樣的演算法是否可以達到更有效率的搜尋，因為目前 GC 大部分的時間都是消耗在搜尋上面，如果這部分可以提出更有效率的改善方法就可以再進一步改善效能。

參考文獻

[1] Bacon, David F., Perry Cheng, and David Grove. "Garbage collection for embedded systems." Proceedings of the 4th ACM international conference on Embedded software. ACM, 2004.

[2] Chang, Yang, and Andy Wellings. "Garbage collection for flexible hard real-time systems." Computers, IEEE Transactions on 59.8 (2010): 1063-1075.

[3] Schoeberl, Martin, Thomas B. Preusser, and Sascha Uhrig. "The embedded Java benchmark suite JemBench." Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems. ACM, 2010.

[4] Chun-Jen Tsai, Han-Wen Kuo, Zigang Lin, Zi-Jing Guo, Jun-Fu Wang,"A Java Processor IP Design for Embedded SoC," Accepted by ACM-TECS, Jan, 2014.

[5] H.-J. Ko, "A Double-issue Java Processor Design for Embedded Application," Master, Computer Science, NCTU, 2007.

[6] H.-J. Ko and C.-J. Tsai, "A Double-issue Java Processor Design for Embedded Application," in Proc. of IEEE Int. Symp. on Circuits and Systems, Seattle, 2007.

[7] C. C. Hsu, "Performance Evaluation and Optimization of String Manipulation on a Java Processor," Master, Computer Science, NCTU, 2012.

[8] J.-L. Brelet, "Using block RAM for high performance read/write CAMs," Xilinx Application Note xapp204, 2000.

[9] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, andE. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. Communications of the ACM,21(11):965–975, Nov. 1978.

[10] Z. G. Lin, "Design of Stack Memory Device and System Software for Java Accelerator IP," Master, Computer Science, NCTU, 2011.

[11] BAKER, H. G. The Treadmill, real-time garbage collection without motion sickness. SIGPLAN Notices 27, 3 (Mar. 1992), 66–70.

[12] C HENEY, C. J. A nonrecursive list compacting algorithm. Commun. ACM13, 11(1970), 677–678.

[13] F ENICHEL, R. R., AND YOCHELSON, J. C. A LISP garbage collector for virtual-memory computer systems. Commun. ACM 12, 11 (Nov. 1969),611-612

[14] S IEBERT, F. Eliminating external fragmentation in a non-moving garbage collector for Java. In International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (San Jose, California, Nov. 2000), pp. 9–17.

[15] J. M. O'connor and M. Tremblay, "picoJava-I: The Java virtual machine in hardware," *Micro, IEEE,* vol. 17, pp. 45-53, 1997.

[16] M. Schoeberl, "A Java processor architecture for embedded real-time systems, " Journal of Systems Architecture, vol. 54, pp. 265-286, 2008.

[17] H.-J. Ko, "A Double-issue Java Processor Design for Embedded Application," Master, Computer Science, NCTU, 2007.

[18] A. Inc. (2004) Jazelle technology: ARM acceleration technology for the Java Platform.

[19] Z.-G. Lin, H.-W. Kuo, Z.-J. Guo, and C.-J. Tsai, "Stack memory design for a low-cost instruction folding Java processor," in Circuits and Systems (ISCAS), 2012 IEEE International Symposium on, 2012, pp. 3226-3229.

[20] N. C. inc. JSTAR-Java Coprocessor for ARM Microprocessors.

[21] Myreen, Magnus O. "Reusable verification of a copying collector." Verified Software: Theories, Tools, Experiments. Springer Berlin Heidelberg, 2010. 142-156.

[22] BACON, D. F., C HENG, P., AND RAJAN, V. T. A real-time garbage collector with low overhead and consistent utilization. In Proceedings of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (New Orleans, Louisiana, Jan. 2003). SIGPLAN Notices, 38, 1, 285–298.

[23] C OHEN, J. Garbage collection of linked data structures. ACM Comput. Surv. 13, 3 (1981), 341–367.

[24] Griffin, Paul, Witawas Srisa-An, and J. Morris Chang. "An energy efficient garbage collector for java embedded devices." ACM SIGPLAN Notices. Vol. 40. No. 7. ACM, 2005.

[25] R. Jones and R. Lins. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. John Wiley and Sons, 1998.