國立交通大學

資訊科學與工程研究所

碩士論文

在 Java 處理器上字串操作之效能分析與優化

Performance Evaluation and Optimization of String Manipulation on a Java Processor

研究生：許嘉哲

指導教授：蔡淳仁教授

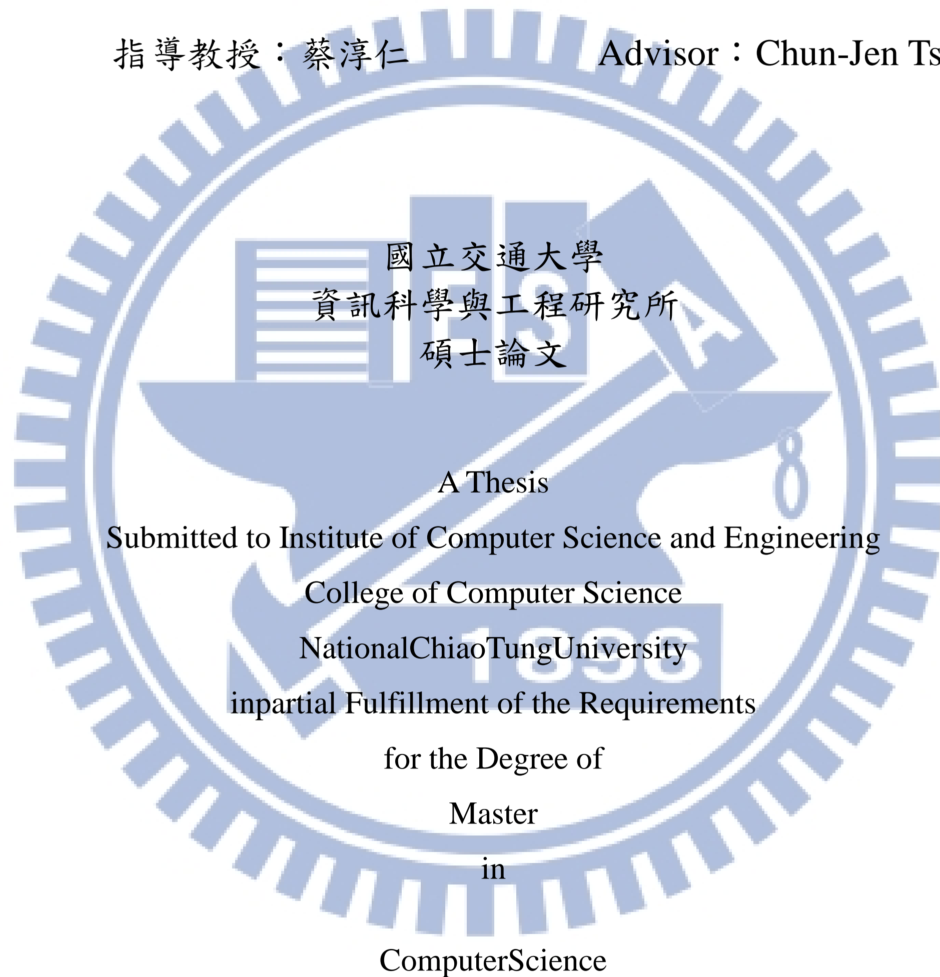
中華民國 102年 7 月

在爪哇處理器上字串操作之效能分析與優化

Performance Evaluation and Optimization of String Manipulation on a Java Processor

研究生：許嘉哲 Student：Chia-Che Hsu

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 2013

Hsinchu, Taiwan, Republic of China

中華民國 102 年 7 月

摘要

字串處理在 Java 和其他高階語言中是不可或缺的功能，例如 web-based 之應用程式裡，需要解析許多腳本語言檔案(scripts)。在本論文裡，我們試著量測與分析 JAIP，一個具可重複使用性的異質雙核心 Java 處理器，處理字串操作之性能，並設計加速電路改進其性能。在 Java 語言中，字串之資料結構是以 String 和 StringBuffer 類別來包裝字元陣列，並且提供 methods 進行字串串接、比較與 subString 等等操作。這些字串操作重複地使用到 heap memory access，其中包括了字元陣列的存取和物件 field 存取。為了加速這些大量迴圈結構的 heap memory access 程式碼，我們提出了一個 String Accelerator Architecture，此架構中包含了兩個字串操作之加速器，分別為 arraycopy 與 indexOf，與一個 Hardware Native Interface。除此之外，為了增加 heap memory 之空間分配更加有效率，我們的 heap 使用了 Heap Management Unit 設計。經過 Embedded CaffeineMark(ECM) benchmark 的 StringAtom 分數測量，JAIP 使用本研究提出的字串加速器再配合 heap access 之優化，比起 heap 未優化且未使用字串加速器之 JAIP，可以得到 4.85 倍的效能提升；而比起使用 JIT 模式的 CVM，我們的效能也高出 1.31 倍。

致謝

這篇論文的完成，得歸功於我的指導教授—蔡淳仁教授的細心指導。在我進行研究的這段時間，老師提供了良好的學習環境，包括了軟體、硬體和其它研究資源。在我遇到困難時，老師總是給予許多解決方案給學生參考。經過在 MMES LAB 的磨練，我的實作能力精進不少，這也得歸功於指導教授的紮實訓練以及良好的研究環境。另外，感謝所有 MMES LAB 以及 CSL LAB 之成員，以及其他所有幫助、鼓勵和支持我的良師益友。

目錄

第一章 前言 1

1.1 研究動機.. 1
1.2 研究貢獻及目的.. 2
1.3 論文架構.. 3

第二章 相關研究 5

2.1 加速 Java 程式的方法 5
2.2 Previous Work on JAIP 5
2.3 Java 之軟硬體界面設計 7
2.4 String Matching 與硬體加速 9

第三章 Java 字串處理加速器設計 11

3.1 字串資料結構與方法介紹.. 11
3.2 類別初始化之實作.. 13
3.3 字串操作之剖析.. 17
3.4 Heap Space Allocation 之設計 19
3.5 Hardware Native Interface 24
3.5.1 Hardware Native Interface 之實作 25
3.5.2 字串操作加速目標分析.. 29
3.5.3 arraycopy 硬體加速器設計.. 32
3.5.4 indexOf 硬體加速器設計 34
3.5.5 其它使用 arraycopy、indexOf 加速器之 methods.. 41
3.5.6 其它使用 Hardware Native Interface 之 methods 44

第四章 實驗結果 45

4.1 實驗環境.. 45
4.2 字串操作平均長度分析.. 46
4.3 JAIP 性能實驗.. 49
4.3.1 StringAtom benchmark 之 method profile 及 bytecode profile 51
4.4 使用 XML Parser 作為 String 操作性能之測試.. 54

第五章 結論 56

附錄 一： Method Profiler 之設計.......... 58

附錄 二： Bytecode Profiler 之設計........61

參考文獻 63

圖目錄

圖 1 異質雙核心爪哇處理器與字串加速器 6
圖 2 動態解析器對 Cross Reference Table 與 Class Info Table 的查詢過程 15
圖 3 類別初始化方法呼叫過程 16
圖 4 動態解析器與類別初始化方法呼叫過程 17
圖 5 JAIP heap 空間分配管理 19
圖 6 new 指令之動態解析過程 21
圖 7 new 指令之動態解析器狀態變化 22
圖 8 newarray 及 anewarray 空間分配過程 23
圖 9 JAIP 的三種調用介面： (i)Java bytecode(ii)RISC 核心(iii)硬體加速器 25
圖 10 method info 訪問過程與 method info 格式 26
圖 11 Hardware native method 在動態解析器內之狀態圖 27
圖 12 Hardware Native Interface(i) 28
圖 13 Hardware Native Interface(ii) 28
圖 14 String 類別裡 arraycopy 直接或間接呼叫關係圖 30
圖 15 StringBuffer 類別裡 arraycopy 直接或間接呼叫關係圖 31
圖 16 arraycopy 硬體加速器 33
圖 17 indexOf 硬體加速器 35
圖 18 Object Resolution Unit 狀態機 36
圖 19 pattern comparator 38
圖 20 Double-Loop State Transition Diagram 39
圖 21 利用 pattern comparator 結果判斷 outer_finish 信號 40
圖 22 利用 pattern comparator 結果判斷 outer_match 信號 41
圖 23 使用 indexOf 加速器實作 String.equals(String str)之流程圖 42
圖 24 使用 indexOf 與 arraycopy 加速器實作 string.replace()之流程圖 43
圖 25 JAIP 與 CVM 執行 Embedded CaffeineMark 之分數比較 49
圖 26 NXML bench 執行時間測量。橫軸為執行次數。縱軸為執行時間(以 log-scale 展示)，單位為毫秒 55
圖 27 Profile Stack 58
圖 28 Method Profiler 內之 Profile Table 59
圖 29 Method Profiler 60
圖 30 Profile Table in Bytecode Profiler 61
圖 31 Bytecode Profiler 62

表目錄

表 1 JAIP 支援之 String methods 13
表 2 計時器控制信號與計時器計數情況 19
表 3 透過 Hardware Native Interface 使用 indexOf accelerator 之 Java methods.. 32
表 4 其它使用 Hardware Native Interface 之 methods 44
表 5 字串加速器在 Virtex-5 FPGA device 上所使用之邏輯元件 46
表 6 XML parser 之字串操作平均長度分析 48
表 7 JAIP 與 CVM 執行 ECM StringAtom benchmark 之分數比較 51
表 8 無字串加速器之 JAIP 執行 2000 次 StringAtom 的 execute()後，蒐集到之 method profile。前 5 欄之單位為 million cycles 51
表 9 含字串加速器之 JAIP 執行 2000 次 StringAtom 的 execute()後，蒐集到之 method profile。前 5 欄之單位為 million cycles 52
表 10 含字串加速器之 JAIP 執行 StringAtom 後所蒐集到之 bytecode profile... 53

第一章 前言

1.1 研究動機

String 在程式語言中是很基本的資料結構，無論是各種字串演算法之應用[1-3]，或是 text 檔案的 parsing，String API 無疑地有很重要的地位。字串是一個很原始的人機互動管道。在 GUI 未流行以前，程式設計師便是以 commands 對機器下指令，或乾脆把許多 command 寫成一份 batch file，機器解析這些 command 並執行對應動作；相似地，當今許多 web 和 script 語言結合，網頁的執行或 script 的 interpreting ，也都是屬於 String manipulation 的應用。在網路安全領域中的研究，許多網路入侵偵測和封包檢測系統使用硬體加速器來加速 String matching 之速度[4-6]。在 Java Processor 領域中，雖然在許多案例中有許多 instrution-level parallelism、object cache、method invocation 方案等等研究 [7-9]。然而，對於字串處理的能力尚未有深入的分析和研究。

過去，有許多 Java 處理器的研究[7-13]，但都沒有針對字串處理進行優化。其中一個對 Java 語言支援較完整的可合成的 Java 處理器是 JAIP(Java Accelerator IP)，根據郭等人的研究指出[9, 14]， JAIP 對於字串處理方面，與同樣針對嵌入式系統之 JVM—CVM(由 Sun Microsystems 所開發)相比，效能有著明顯的落後。之所以 JAIP 對於字串處理能力這麼不足的原因是因為它字串處理的實作方式，是由 Java core 透過 IPC 介面來對 RISC core 做字串操作、object instantiation 和 array creation，而 IPC 介面是由 ISRs 來實作，兩個 core 之間的溝通成本相當的大。本研究除了透過 Heap Management Unit 改善物件與陣列創建的 IPC overhead 之外，我們更設計與實作一個 method profiler 與 bytecode profiler 深入分析 JAIP 之字串處理效能資訊，並根據此分析結果，我們提出一個 String Acceleration Architecture。此架構內包含一個針對 JAIP 處理器設計的 Hardware Native Interface 以及兩個字

串操作加速器。

字串操作與其他 data type 之操作有明顯的不同，數值型態的運算主要是以 arithmetic 和 logic，而字串除了 comparison 和大小寫轉換之外，稍微較少使用到 ALU。字串操作主要有 concatenation，compare 及 subString 之 matching 等等操作，特點就是用到許多記憶體存取指令。除此之外，字串 concatenation，subString 之 matching 或是 StringBuffer 容量之擴充等操作時，程式至少都是由一層以上之迴圈控制連續字元存取，處理器對迴圈的效力也會影響字串操作。因此，一個對連續記憶體存取具高效能處理器架構是必要的。

1.2 研究貢獻及目的

本論文在一個異質雙核心 JAVA 應用處理器的嵌入式系統環境裡進行整體效能的評估與改進。JAIP 一個高度可移植性 IP core。在這個應用處理器中，JAIP 是一個具高度可疑質性的 Java IP core，負責處理所有 Java bytecode 的執行。而在 JAIP 裡，有許多效能瓶頸未解決，例如在進行物件解析的時候，動態符號解析器(Dynamic Symbol Resolution Unit)花費相當多的時間，整個 JAIP 處理器整個管線必須暫停等待或者是插入nop指令讓沒有工作的管現階段暫停。礙於架構的限制，當程式大量使用原生方法時，例如字串類別，也對我們的效能產生極大的障礙。本篇論文研究重點是找出這些效能瓶頸，並且提出有效的解決方案。

為了讓 JAIP 完整支援字串操作，我們對 JAIP 增加了類別初始化方法(Class InitializationMethod)呼叫的功能。我們盡可能地降低JAIP與RISC-core溝通次數，我們在 JAIP 中建立了 Heap Management Unit，使得 JAIP 在不需要使用 IPC 介面來完成物件與陣列之創建動作。此外，為了讓 JAIP 支援更多字串操作與系統類別操作，我們完成了每個類別之靜態初始化函式(clinit)之呼叫動作。這個函式的呼叫，可以讓每個類別的靜態變數做初始化，使得 JAIP 可以支援更多 Java 語言特性與 API，其中包括許多字串操作與字串打印。目前我們支援將近 31 個標準 Java

系統程式庫的字串運算程序，而剩餘的方法是受限於資料型態的因素而暫時不支援。

本論文第一部分是針對 Java 程式使用字串類別的案例來做分析及優化。分析方法也是使用硬體剖析器。剖析字串操作的剖析器—method profiler 剖析了兩類資訊：1.各類別之方法被呼叫次數 2.各類別之方法硬體使用時間分布。剖析 bytecode 的剖析器—bytecode profiler 同樣剖析兩類資訊：1.各 bytecode 之使用次數 2.各類別之方法硬體使用時間分布。硬體使用時間分布是將程式執行時間切割為四種時間：1.管線時間 2.記憶體存取時間 3.中斷時間 4.動態解析時間。

第二部分是根據剖析結果，我們設計並實作了兩個字串加速器，並為了讓 Java 支援對底層硬體操作，我們也設計了一個讓 JAIP 裡面的 byetcode ececution engine 直接透過 dynamic resolution 機制來操控硬體設備或電路之介面—Hardware Native Interface。透過這個介面，Java 程式語言可以使用兩個特殊用途之硬體加速器來實作數個 string method。特殊用途之硬體加速器有：1.一個 arraycopy 硬體加速電路來實作 Java 中 arraycopy 的方法。此電路能夠透過 Processor Local Bus(PLB Bus)來控制記憶體控制器(Memory Control Unit)進行大量的連續記憶體存取來實現陣列的複製操作。2.第二個字串加速器為 indexOf。indexOf 方法在 Java 語言中是進行字串搜尋：給定一個正文及一個字串，搜尋此正文裡面是否存在這個字串。在這個 indexOf 硬體加速電路中，我們使用創新的字串比較概念—每次進行 3 個字元的比對，這個方法可以減少將近兩倍的迴圈數目。

1.3 論文架構

本論文一共包含 5 個章節。第一個章節是一個蓋括性的導論，包含了研究動機、背景及目的。第二章介紹相關研究與 JAIP 之先前研究，相關研究部分，分成三個小節，分別是 1.Java processor，2.Java 之 hardware interface，3.hardware string matching。第三章為詳細的設計與實作介紹，包含了字串操作實作、profiler、

hardware interface 與字串加速器。第四章為幾種字串加速器之效能評估實驗與測量。最後，第五章為本篇論文的結論與未來研究方向。

第二章 相關研究

2.1 加速 Java 程式的方法

在一個沒有經過任何優化的 Java runtime system 上，Java 程式的性能是眾所周知的差，因其 bytecode 需要被順序地被 interpreter 解譯。經典的優化方法便是 just-in-time compilation(JIT)，或稱 dynamic translation。雖然 JIT 能夠以 binary-level optimization 產生出品質不錯的 binary code，但它必須犧牲較大的 memory footprint[15]。除此之外，startup delay[16]。另一種增進 Java 程式性能的方式便是 Java processor/co-processor。Java 加速器之設計分為 standalone 的 Java processor 及協同式的 Java co-processor。前者為在單獨的 Java processor 內完成大部分的 bytecode interpreting 及執行，協同式則是需要與 genral-purpose 之處理器進行 co-processing。採用 standalone 之設計的 Java processor 主要有 sun 的 pico-Java[7]，aJile 的 JEMCore[10]，Komodo[11]及 JOP[8]等等。使用協同式的 Java co-processor 有 ARM 的 Jazelle[12]及 Nazomi 的 JSTAR[13]。JAIP 是採用易於與其他處理器整合的弱協同式設計(weakly-coupled co-design)，只要目標處理器支援 interrupt-driven 之 communication 便可與 JAIP 整合。JAIP 可以獨力完成幾乎所有的 Java 程式計算，只有程式需要低階 I/O 裝置控制例如 RS-232 controller，或 class file 的解析，才會需要使用 general-purpose 處理器計算。

2.2 Previous Work on JAIP

本論文所採用的處理器架構是由柯等人所提出的異質雙核心 Java 處理器[17, 18]。在這個架構當中，由兩個異質之處理器所組成。一個核心為 Java-core，內部含有 Bytecode Execution Engine(BEE)，負責 Java bytecode 之運行，Java bytecode 所需之符號解析工作是由 Dynamic Symbol Resolution Unit(DSRU)負責，而每次所解析到的新的 Class 則由 Class Symbol Table Management Unit(CSTMU)與 Method

Area Management Unit(MAMU)來負責做 class loading。另一個核心為 RISC-core，主要負責記憶體管理，class parsing 和 physical resource 管理等工作。

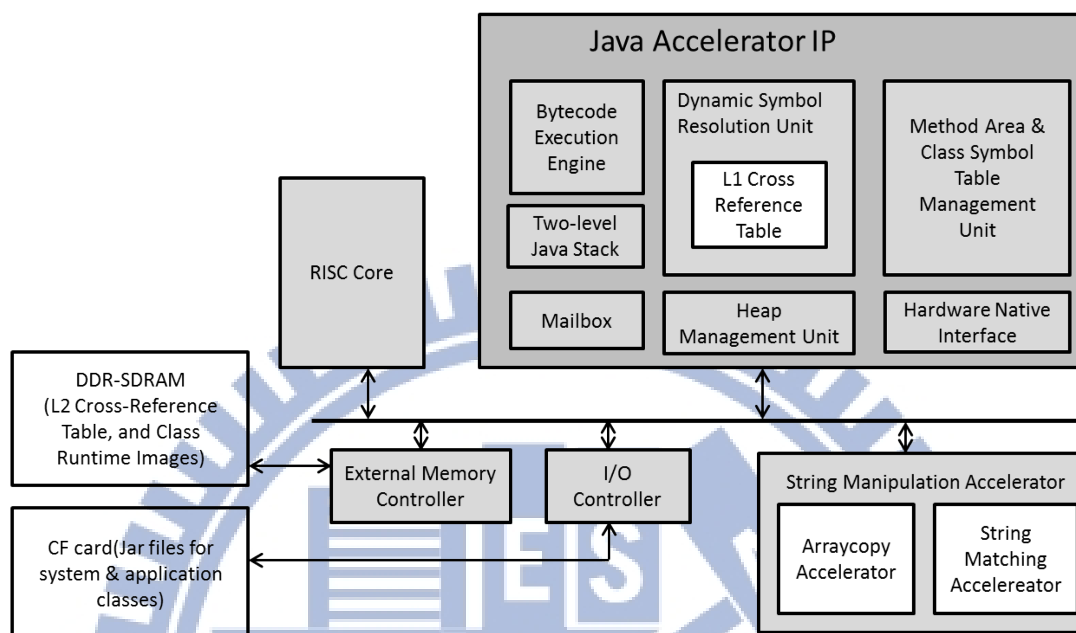


圖 1 異質雙核心爪哇處理器與字串加速器

BEE 是由柯等人提出的 double-issued 之 Java Bytecode Execution Egine[17, 18]所改良而來的 bytecode ececution pipeline。BEE 是一個獨立的 IP。也就是說，它可以整合到任何的 host 端處理器，只要 host 端支援 interrupt-driven 之 inter-processor communication 機制。BEE 是一個四個 stage 之 pipeline，包括 translate、fetch、decode 及 execution stage。Translate stage 透過 jpc 將 Method Area 內的 Java Bytecode 讀出，每次盡可能地讀出兩個 byte 之 Method Area 資料，每個 OP code 皆會經過 translation ROM 轉換為一組 j-code info.。j-code info.被送進 fetch stage 後，被分類成為 simple 及 complex 兩種類型 bytecode，若為 simple bytecode，會被轉換成一個 BEE 指令—j-code，兩個無 hazard 之 jocde 可以被 double-issued；若為 complex bytecode，會被轉換為一組 j-code-pair sequence。每個 j-code pair 會經由後面的 decode stage 和 execution stage 執行。為了讓指令平行度增加，林等人提出了 2-level stack memory 架構[19, 20]。

DSRU 是由郭等人提出之符號解析器[9, 14]。負責在 runtime 時解析 Class Symbol Table(CST)中的資訊、協助引發 class loading 請求以及傳遞 class ID 和 method ID 到 CSTMU 和 MAMU 中。Symbol table 在一般程式語言中，是一個專門儲存 identifier 資訊之資料結構，可供 compiler、interpreter 或 linker 查詢變數型態、scope 或函式進入點。在我們的 JAIP 平台中，class symbol table 存放著 ldc bytecode 所用的 constant 資訊、new bytecode 需要的 class ID、field 操作和 invocation 時都需要的 Cross Reference Table 之 entry reference。Cross referece table 內的結構則是由一個 entry 表示的 field info.，或是由 3 個 entry 組成的 method info.。DSRU 在 BEE 遇到 field 操作、load constant 或 invocation 類型 bytecode 時被啟動後，便會啟動 DSRU 來解析 CST 與 Cross Reference Table 之內容。

2.3 Java 之軟硬體界面設計

不同 JVM 平台有不同硬體控制之方式。為了讓 JAIP 能夠以 Java 固有之語言特性啟動字串硬體加速器，我們對於各種 Java 平台之軟硬體協同設計方式做了研讀。研讀的對象包括完全軟體實作的 JVM 與透過 Java co-processor 實作的平台，希望可以找出適合 JAIP 之方法。由於 Java 語言在對底層設備操作的方法是很抽象的，而且 Java 之保護機制，使得 Java 無法直接對特定的記憶體位址或 I/O port 直接存取。若平台是屬於 memory-mapped I/O，使用 C 或 C++這種支援指標的程式語言可以透過 I/O port 之位址來對 hardware device 做操作；若平台是屬於 port-mapped I/O，則程式語言必須使用作業系統提供的 API 或者是直接 inline assembly code 來使用處理器之 I/O 存取專用指令。然而，Java 程式語言並不支援這種直接記憶體位址的存取或 I/O port 存取之功能。

一些研究制定與實作 Java 層面的硬體介面，例如建立硬體物件的 class packages 來供 Java 應用程式操作使用。Schoeberl 等人[21]提出了 Hardware Abstraction Layer 之標準，概念是將 hardware device 映射到一個 Java 物件，其 fields 代表 device registers，透過 static fields 的存取，Java program 可以實現對 device 之

操控。Whitham 等人[22]設計的系統中，一個 Interface Generator 可以將 Interface Description Language(IDL)檔案轉換成軟體和硬體介面，分別是 class file 和 HDL file，並且以 JOP[8]為基礎，利用 JOP 內部之 communication logic 實現 hardware method 功能。Borg 等人[23]提出兩種不同 JVM 硬體支援方法，分別是 method-based 及 object-based，並採用 method-based 方法實作出 JavaMen 類別庫。Ha 等人設計的 Hard-HotSpot[24]利用 JBit API[25]來實現 runtime-reconfiguration 機制，並且以 hardware 加速 Java 程式。然而，Java 不支援任何對硬體的直接操作，這些設計都是仰賴更底層的 hardware interface，例如 native method 或使用自定義 bytecode，才能夠真正實現 Java 對硬體操控之功能。下一段將會介紹 Java 平台更底層的 hardware interface 設計方法。

所有的 JVM 軟硬體協同設計之實作方式不外乎兩種：1.透過修改 JVM 來擴充其硬體支援以及 2.透過 native library 之介面對應體操控[20, 26, 27]。第一種是透過修改 JVM 實作方式來分割 Java method、bytecode[8, 26]，甚至是 object-oriented 才有的 object 及 field[8, 23]，並分別以軟體和硬體來實作這些被分割的語意。修改的地方可能是 class loader、interpreter，execution engine 甚至是 class image[10, 28]。aJile[10]使用自行定義的 extended Java bytecode 作為與底層硬體溝通之介面，例如 thread 的 concurrent controll 及 multiple JVM context 之支援，而 extended bytecode 是使用 application tool—JEMBuilder 修改 class image 內容，讓特定 method 內部含有自定義的 bytecode。JOP[8]使用 micro code 來對應體觸發及參數傳遞，完成花費較昂貴的 bytecode，例如用 HDL 程式碼實作的電路來進行 imul。第二種類型則是使用 native library 之 interface 對 device 做操作[29, 30]。這種方式的好處是 Java application 只需要移植 native library 到其它 JVM 平台，便可使用原有的 Java program。但是對於工作幾乎獨立於自身能力的 Java processor 來說，要讓系統可以使用 native library，則必須要透過 bus 實作 polling 或 interrupt 機制，來與 general-purposed 之處理器做溝通，而其溝通成本會直接影響到 Java program 之效

能。

為了讓 Java 字串之處理程式能夠使用字串加速器，並在 Java application 不需修改的情況之下增進效能，我們修改 Java 標準類別庫中的 String.class、StringBuffer.class 和 System.class 等等，將其時間花費較多的 method 以功能等價的硬體方式實作。為了讓 JAIP 和硬體加速器能夠溝通，我們實作了 Hardware Native Interface，此介面可以經由 Dynamic Symbol Resolution Unit 進行觸發，因此 JAIP 與 RISC-core 之間，不需要昂貴的溝通成本便可以操控硬體加速器或其它電路邏輯。Java Language 之介面方面，developer 僅需將想使用等價電路實作的 method 宣告為 native，並修改 class parser 即可。這些 native method 未來也可以透過更有系統的 class package 包裝，避免 Java application level 的誤用。例如本篇論文在第三章將會介紹的 hardware profiler，Java 程式即是以一個擁有數個 static native method 之 class 來啟動或關閉。

2.4 String Matching 與硬體加速

Pattern Matching 問題可以應用到許多不同的科學領域之計算上。常見的應用有資料庫的字串搜尋、DNA 分析、網路入侵偵測或影像辨識等。在嵌入式 Java 平台上，我們希望以簡單且有效，且字串大小可參數化之硬體加速器，來輔助多核心 Java 處理器之 String class 中與字串搜尋相關之操作。

有許多適合軟體的經典字串搜尋演算法，例如 KMP、Boyer-Moore 等等[1, 2]，大部分的演算法都需要額外的線性 preprocessing 時間和空間，意味著如果將這些演算法以硬體實作的話，需要額外的記憶體空間以及電路邏輯來處理 preprocessing，且字串比對的長度會受到硬體之設計而限制。一些硬體加速字串的方法使用許多 processing elements 來進行平行的比較[31, 32]。一個很受歡迎的 pattern matching 方法，便是使用基於 Content-Addressable Memory(CAM)的比對系統。CAM 是適合使用在需要快速搜尋資料的系統之記憶體結構，透過將比對資

料輸入至 CAM blocks，資料所在之位址便會從輸出端送出。CAM 為基礎的比對系統通常會使用到龐大的記憶體空間。不論是使用 ASIC 技術或 FPGA，都會使用許多的硬體資源。以 Le 等人的做法為例[33]，一個大小為 1kB 的 Altera M9K 記憶體結構可以作為一個 32-word x 8-bit 的 CAM unit。若需要實作可以容納 1024 個 UTF-16 字元的 CAMs，則需要 1024-word x 16-bit，也就是需要 64 個 CAM unit，可換算成大小為 64kB 的 Altera M9K 記憶體。參考 Xilinx 之 BlockRAM 實作 CAMs 的 Reference Design[34, 35]，在 XCV50 的 device 上欲實作一個能儲存 1024 個 UTF-16 字元之 CAM，需要 128 個 BlockRAM，也就是 256kB 的記憶體空間。此外，我們尚需要額外的 shift register 及 logic element，來完成一連串的字元比對以及記憶體位址之 encoder。

這些字串比對系統都可以很快速地進行比對，特別是使用 CAM。然而，幾乎每種系統都需要 preprocessing，換句話說，每次新的 matching system 中的 database 或字串文本更新時，都需要重新將字串寫入比對系統以進行 preprocessing。這些優異的系統非常適合在 database 或字串文本鮮少更動的系統中。但在 Java 平台中，字串比對操作無法保證每次使用操作的字串物件都是相同的，字串比對系統的 preprocessing 時間是無法避免的。若為了節省 matching system 的 preprocessing 時間而將字串物件直接儲存於像 CAM 的特殊儲存體中，那麼當程式要透過索引值將對應位置的字元讀出時，會是一個很大的麻煩。因此將字串儲存於 RAM 中是比較合理的作法。儘管將字元寫入比對系統的動作與字元從字串物件中讀出的動作可以被重疊，一般情況下整個字串比對時間仍然被字串文本之讀出時間支配著。因此，以字串文本經常變動的前提之下，只要比對動作能夠與字元從字串物件讀出的動作可以被重疊，並且盡可能以 word 為單位進行比對，迴圈次數以及 bytecode 執行時間的節省，足以使許多字串比對的情況得到不錯的效能了。本論文採用 brute-force 的字串比對方法，是一個不需要太多 logic element、字串大小有彈性且完全不需要額外的記憶體空間之設計。

第三章 Java 字串處理加速器設計

3.1 字串資料結構與方法介紹

在 Java 字串類別中，每一個根據字串類別所創建的物件有三個 field，分別為 value，offset 及 count。其中 value 為一個字元陣列的參考指標，也是此字串物件裡最重要的 field。字元陣列裡包含一個字元陣列長度，以及一個存放此陣列之字元的連續空間，其中字元陣列長度與參考此陣列的字串物件之字串長度並不一定相等。offset 是一個整數數值，用來指示字串的開頭。由於字串物件裡的 value 值是有可能共享的，也就是說，當兩個字串物件擁有一樣的字串值，或者，某個字串為另一個字串的子字串時，兩個字串物件所放的 value 值是有可能一樣的，而 offset 則用來指示個別字串物件的字元開頭。當 Java 使用到任何操作 value 的方法時，offset 即可用來指引這些方法使之參考到正確的字元開頭。最後一個 field 為 count，此數值用來指示字串有效的字元數。與上述例子相似，當兩個字串物件擁有相同 value 值時，個別字串物件之 count 並不一定相同。

字串存放在類別檔裡面時，是由一個 CONSTANT_String 資料結構儲存。CONSTANT_String 資訊包含一個 Tag 標記以及一個常數池(constant pool)的指標，它指向常數池裡另一個常數池資料結構—CONSTANT_Utf8。由於空間的考量，類別檔裡 CONSTANT_Utf8 格式就如同他的名字一樣，是使用 UTF8 的格式。而 JVM 裡面 char 型態是使用較大的 UTF16，每一個字元皆固定使用 16 位元來表示。之所以會使用較大的空間來表示，是因為當程式對固定大小的字元作操作的時候，可以節省像使用 UTF8 這種不定長度的字元大小時，較為複雜的解碼方式。

字串物件的創見主要分成兩種情況。第一種為靜態產生的字串物件。第二種為動態所創建的字串。靜態的字串物件是由類別解析器根據常數池裡面 CONSTANT_String 的資訊建立。當一個類別被解析的時候，解析器會根據常數池

裡面 CONSTANT_String 資訊，替字串物件以及字串陣列分配適當的堆積(heap)空間，並且填入初始值。而動態字串的建立則是透過 Java Bytecode—new 及 invokestatic 分別為字串物件進行堆積的空間分配，以及建構子的呼叫。

在舊的 JAIP 版本裡，不論是動態或靜態的字串，字元型態皆是使用 1 位元來表示。這樣雖然有助於節省堆空間，但是這樣將有可能隨時產生一個嚴重的錯誤：由於 UTF8 所使用的是可變長度字元，當類別解析器直接把轉解碼的 UTF8 拷貝到字元陣列上時，那些非得使用多重位元組的字元將會被誤判成為多個字元。這個缺陷使得舊版 JAIP 只能表示 ASCII 字元，任何非 ASCII 字元將產生多個不相干的字元。因此，新版的 JAIP 之類別解析器中，每個除數池裡的 CONSTANT_UTF8 字串皆會被轉碼成為 UTF16，這樣不但所有字元表示法正確，而且也符合 Java 規格書中定義的字元型態。

下表為目前 JAIP 所支援的所有字串方法。其中 getBytes()及 valueOf(int)方法的執行過程中，皆會呼叫類別初始化方法(clinit method)。而在舊版的 JAIP 中，是不支援類別初始化方法的。為了支援字串的這兩個方法，下一小節將會介紹如何增加呼叫類別初始化方法的機制到 JAIP。

表 1 JAIP 支援之 String methods

String methods
chatAt(int index)
compareTo
equals
equalsIgnoreCase
regionMatches
startsWith(String prefix, int toffset)
startsWith(String prefix)
indexOf(int ch)
indexOf(int ch, int fromIndex)
lastIndexOf(int ch)
indexOf(String str)
indexOf(String str, int fromIndex)
hashCode
substring(int beginIndex)
substring(int beginIndex, int endIndex)
concat
replace
toLowerCase
toUpperCase
trim
length
getChars
getBytes(String enc)
getBytes()
toString
toCharArray
valueOf(Object obj)
valueOf(char data[])
valueOf(char data[], int offset, int count)
valueOf(boolean b)
valueOf(char c)
valueOf(int i)

3.2 類別初始化之實作

Java 初始化的類別方法有兩種。第一種為物件的初始化，此種類別方法即為

建構子的呼叫，負責 non-static field 的初始化。第二種為類別的初始化，負責 static field 的初始化。物件初始化方法必須每個物件個別呼叫，因為每個物件的 non-static field 是私有的，所以必須分開初始化。而由於 static field 在相同類別或相同父類別的物件中是共享的，類別初始化的方法在整個 Java 程式執行過程中只需要呼叫一次。在 Java 規格書中，每個類別在被解析的時候，JVM 內部都會呼叫此類別的類別初始化方法，而且這些方法也只能讓 JVM 內部呼叫，沒有任何語法或 Java bytecode 可以讓程式開發者直接呼叫。

每一個宣告 static field 之類別都存在一個類別初始化方法。若一個類別不存在任何自身宣告的 static field，則此類別不存在類別初始化方法。在 Java 程式語言中，子類別若繼承父類別之 static field，那麼父類別與子類別會共享此 static field 的值。一個從父類別繼承來的 static field 並不需要初始化，因為在父類別呼叫過類別初始化方法時，就已經將此 static field 初始化過，子類別不過是與父類別共享此數值。

在我們的實作當中，類別初始化的時機是在每個類別的 static field 第一次被讀取的時候進行，也就是說，在我們的 JAIP 中，並非與一般 JVM 類別初始化的時間一樣在類別解析的時候，而是在每個類別 static field 第一次被讀取的時候初始化。這種設計主要是考量有些類別的 static field 並沒有經常被使用，當某些類別被解析過後，它們的 static field 從來沒被讀取過。因此，在我們的設計中，每次進行 static field 讀取時，都要檢查此類別是否被初始化過，若尚未初始化則讓 JAIP 進入類別初始化方法的進入點。

Static filed 的讀取是透過 Java Bytecode—getstatic 來進行。此指令屬於類別操作指令，這種指令在 decode stage 被解碼完畢後都會啟動動態解析器進行類別資訊的解析。當 getstatic 指令被執行時，動態解析器都必須檢查使類別是否已被初始化。每個 getstatic 指令後面都會接續一個運算元，此運算元用來表示 static field 在類別符號表中的條目編號，透過此編號，動態解析器可以取得一個 32 bit 的符

號資訊，如圖 2 前 16 bit 代表此 static field 所屬之類別 ID；後 16 bit 代表一個 Cross Reference Table 的 offset。動態解析器分別利用這 2 個 half-word 資訊進行 Cross Reference Table 與 Class Info Table 的存取，這時兩個 table 的讀取是並行的。

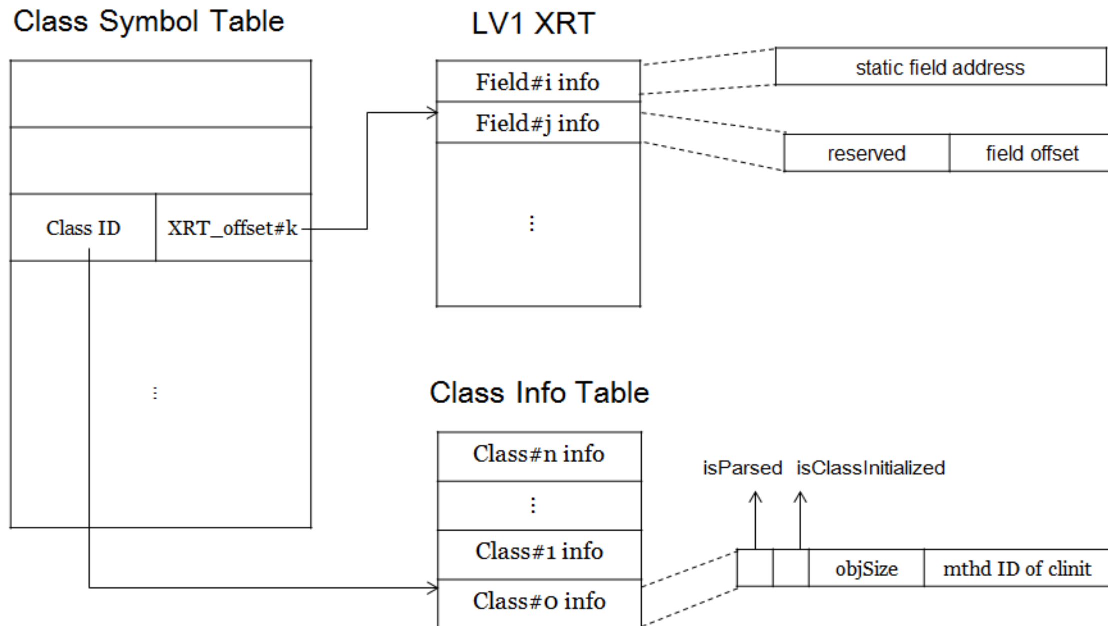


圖 2 動態解析器對 Cross Reference Table 與 Class Info Table 的查詢過程

一旦 isClassInitialized 旗號指示出類別尚未初始化，那麼 JAIP 忽略從 Cross Reference Table 中讀到的位址，並準備進行類別初始化方法的呼叫，過程如圖 2。由於此時管線處於等待動態解析器的狀態，如果類別初始化方法的返回點想要記錄在動態解析器中的某個狀態，那麼就會必須耗費更多的堆疊空間來儲存 return frame。為了不增加方法呼叫時 return frame 的大小，在本設計的類別初始化方法返回點是設定在 getstatic 此指令本身。也就是說，類別初始化方法是在 getstatic 指令執行到一半時呼叫，而返回時，管線必須重新執行 getstatic 指令，動態解析器也要重新解析。類別初始化方法底下都會有一到返回指令—return，return 指令在管線中會映射到一組預先設置好的 j-code 序列，j-code 序列會透過 return frame 的資訊進行返回的操作。類別初始化方法呼叫過程如圖 3 返回方式是透過類別初始化方法本身的 return 指令，返回位址是 getstatic 指令本身，返回後 getstatic 會重新進入管線。

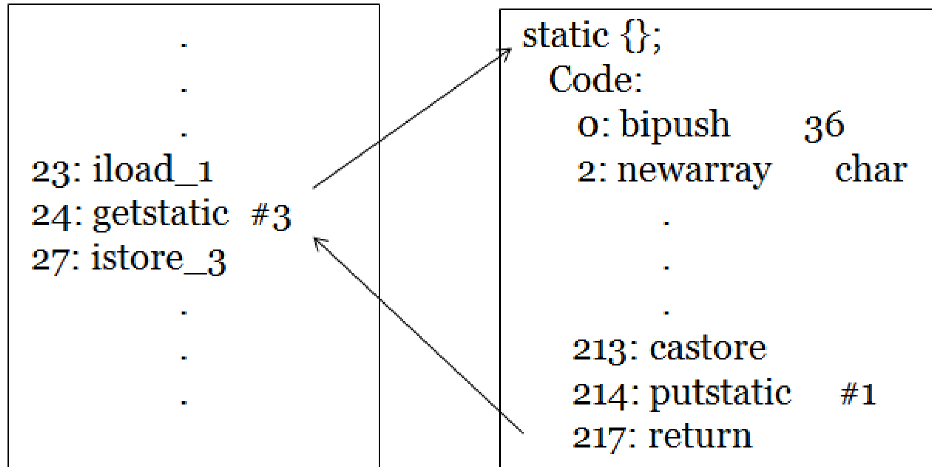


圖 3 類別初始化方法呼叫過程

解析過程如圖 4 所示。Get_L1_XRT_Ref 為 Class Symbol Table 的讀取，讀到 Cross Reference Table 的 offset 與類別 ID 後，進入 Offset_access 狀態同時進行 Cross Reference Table 與 Class Info Table 的讀取，若從 Cross Reference Table 讀到 0 的值，代表此類別尚未解析，動態解析器將進入狀態 Illegal Offset 並以 interrupt 信號觸發 RISC 核心的解析動作；若從 Cross Reference Table 讀到的值不為 0，此值代表 field 在堆中的位址。同一時間檢查 clinit_flag 值來判斷是否呼叫類別初始化方法，若為 0，則代表此類別已經初始化過，直接進入 field_load 狀態進行 field 的讀取；若 clinit_flag 為 1，進入 clinit_ret_frame 設置堆疊的 return frame。En_MA_mgt 及 Class_Loading 為觸發 class loader 對類別的讀取。Method_entry、Method_flag、Max_stack 與 Max_local 分別進行 JPC 與堆疊的調整。

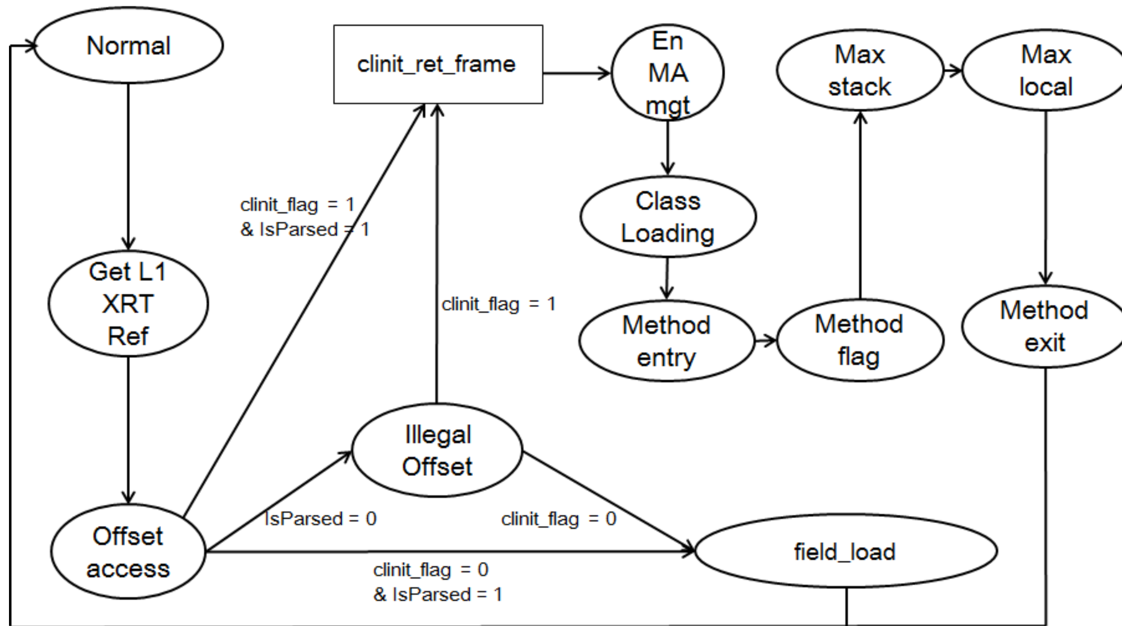


圖 4 動態解析器與類別初始化方法呼叫過程

3.3 字串操作之剖析

字串操作包含大量之記憶體搬移動作，及迴圈結構指令，例如 goto 和 iinc。我們推測這些時間會是字串處理程式佔最多數之地方。除此之外，JAIP 在類別解析上級部分 native method 需仰賴 RISC-core 的協助。一旦 Java 程式含有這種句子，JAIP 核心和 RISC 核心之溝通成本相當龐大。雖然我們已經預測 new 及 newarray 兩個 bytecode 在 RISC-core 之溝通時間佔相當多數，我們仍然希望有一個方便由 Java 語言控制的 profilig 機制，可以發現字串處理程式的瓶頸，包括在核心之間的溝通時間、記憶體時間、迴圈控制指令和其他未預測到之 bytecode 時間。因此，本篇論文設計了兩個適合 JAIP 之 profiler—method profiler 及 bytecode profiler。這兩個 profiler 可以剖析各種 Java 程式，在本篇論文中，我們使用這兩個 profiler 來剖析字串處理程式；在未來，兩個剖析器可以為任何 Java 程式進行效能分析。

為了要證實上述之推測，以及找出先前未預測到之程式耗費時間，我們設計並且實作了針對 JAIP 的硬體剖析器。有兩個不同型態之 hardware profiler，分別蒐集兩種型態之 profile： 1.Method Profiler，蒐集每個 method 之被呼叫次數與總執行 cycles 數；2.bytecode profiler，蒐集每個 bytecode 之被使用次數與總執行 cycles

數。其中總執行 cycles 數被分成四個 counter 計數值，每個 counter 計數值分別代表此 method 或 bytecode 之四種時間分布，下段將會介紹這四個 counters 和其代表意義。此硬體剖析器無須進行指令碼的插入，也就是說，Java 程式無須重新編譯即可進行剖析。這個硬體剖析器利用數個互斥的硬體計時器來統計各個方法的執行時間分布，並將統計時間儲存於 on-chip 的 BRAM 上。為了在 Java language 上提供一個讓 Java application 端，方便操控 profiler 之介面，我們透過一個 MMESProfiler.class 類別來開啟、關閉 profiler 並印出其 profile。由於兩種 hardware profiler 在 JAIP 中僅有一組，MMESProfiler 類別之使用不需要 instantiate object instance，直接使用 static method 來操作 profiler，而操作之介面會透過一個本篇論文設計之介面—hardware native interface，來與電路做溝通。3.5 小節將會詳細介紹此介面。

整個程式的執行時間被分成四個部分：1.管線執行時間 2.動態解析時間 3.堆的存取時間 4.等待中斷服務時間。由於字串類別方法的實作，使用了不少系統方法，而在舊版本 JAIP 實作中，有許多系統方法都是以原生方法完成，這些原生方法會使得 JAIP 發送中斷信號到 RISC 進行對應的 ISR，我們推測字串處理程式有很大的效能瓶頸是落在等待 RISC 的中斷服務。除此之外，字串物件裡面的核心資料—字串陣列也經常被搬移、複製等操作，這些物件與陣列的資料都是屬於堆的空間，堆的記憶體存取也會造成 JAIP 的等待。而動態解析時間也是一個造成管線等待因素，因此我們也把它的運行時間獨立出來。

一共有 4 個計時器協助此硬體剖析器，分別為 HW_timer、Intrpt_timer、DSRU_timer 及 heap_timer。此四個計時器計數的時間是完全互斥的，也就是說同一個時間點，不會有超過 1 個計時器計數。四個計時器由三個信號控制，分別是 heap_access_on、Intrpt_on 及 DSRU_on，表 2 描述了所有控制信號組合的計數情況。

表 2 計時器控制信號與計時器計數情況

	Intrpt_on = 0		Intrpt_on = 1	
heap_access_on = 0	DSRU_on = 0	DSRU_on = 1	DSRU_on = 0	DSRU_on = 1
	HW_timer	DSRU_timer	Intrpt_timer	
heap_access_on = 1	heap_timer			

3.4 Heap Space Allocation 之設計

堆積空間之管理我們用了一個 32 位元之暫存器來儲存目前可用堆積空間之位址，每次要分配新的堆積空間時，將 current_heap_ptr 推到堆疊頂端或 RISC 核心，並使用 heapAllocSize 及 heapAllocEn 兩個信號將 current_heap_ptr 增加。RISC 核心在某些服務中也可能會需要分配堆積空間，例如執行一個以 RISC 核心來實作的原生方法時，如果此原生方法產生新的物件或陣列時，此時 RISC 核心就必須對此暫存器讀寫。另外，類別解析器在產生靜態的 ldc 物件或陣列時，也會分配堆積空間。當 RISC 核心需要分配堆積空間時，只需透過 PLB Bus 修改 current_heap_ptr 即可。

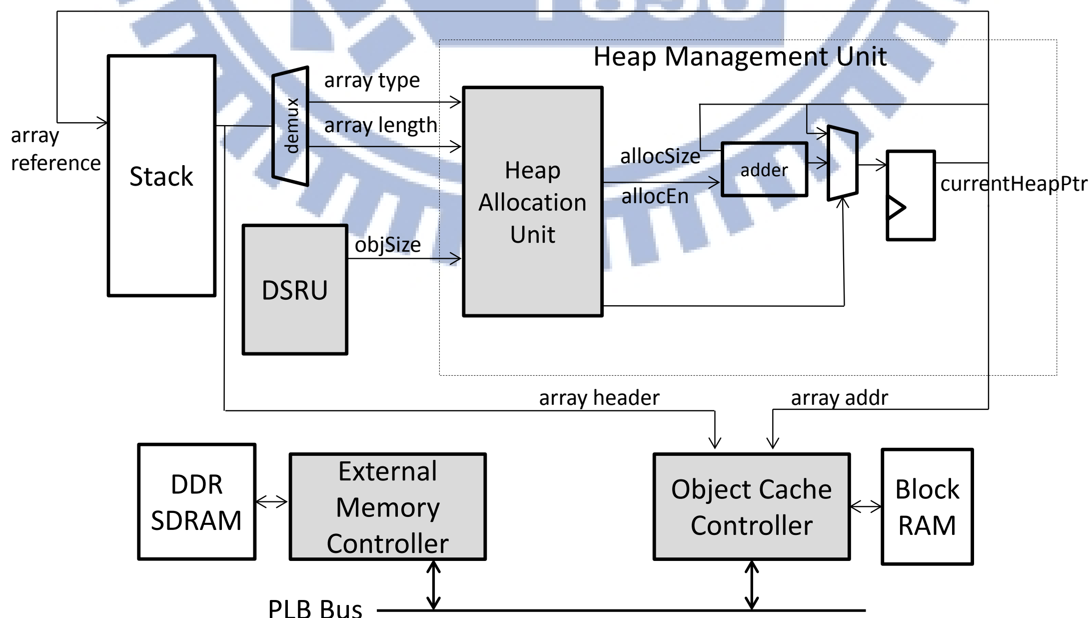


圖 5 JAIP heap 空間分配管理

在舊版的 JAIP 中，物件創建指令與陣列創建指令是經由 IPC 介面發送一個 ISR 信號給 RISC 核心來完成[20]。new 指令是經由動態解析器經由類別符號表的查詢放到堆疊中，之後產生一個中斷請求，由 RISC 核心檢查此類別 ID 是否已被解析過，若尚未解析，則解析此類別。之後根據 L2 的 Cross Reference Table 中的物件大小分配對應的堆積空間，再將物件位址推到堆疊頂端。newarray 的指令通常需要搭配另一個將陣列大小推到堆疊中的指令，newarray 指令執行之前，陣列大小就必須存在堆疊頂端中。JAIP 執行 newarray 指令時，會再將陣列元素型態推到堆疊頂端，之後引發一個中斷服務請求，RISC 核心則根據堆疊頂端兩個項目來決定要分配多少堆空間，分配完後再將陣列位址推到堆疊頂端。

上述設計一旦遇到大量堆空間分配指令時，就會產生極大的效能瓶頸，原因是因為 RISC 核心的中斷服務請求耗費大量時間。因此，新的設計不需仰賴 RISC 核心來輔助堆積空間的分配，獨立以硬體方式進行分配。新的設計承襲舊有的方式，當 decode stage 解碼到 new 指令時，啟動動態解析器，並且 stall 住管線。動態解析器解析步驟如圖 6。new 指令經由 decode stage 解碼後，將指令碼後面之運算元傳遞給動態解析器，動態解析器根據後面運算元取得對應的類別符號表之內容，獲取欲創建的物件類別 ID，根據此 ID 讀取 Class Info Table 中的物件大小—objSize，以及檢查旗標—isParsed 判斷類別是否被解析過，若尚未解析，則發送中斷服務請求給 RISC 核心進行類別解析。

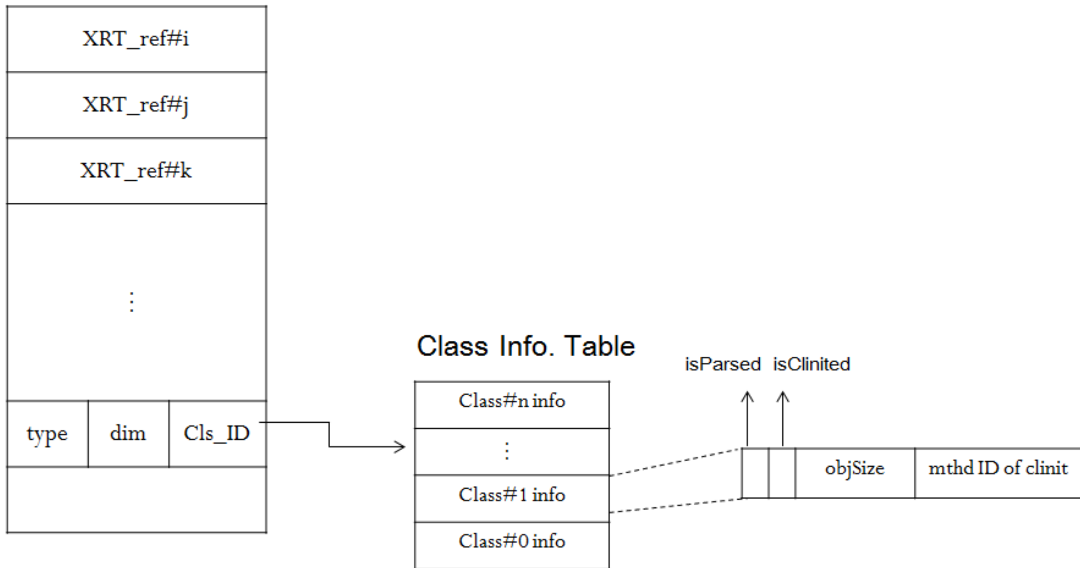


圖 6 new 指令之動態解析過程

圖 7 為動態解析器處理 new 指令的過程。normal 狀態為等待 decode stage 傳送過來的啟動信號，動態解析器一經啟動後，進入 CST_Access 狀態。CST_Access 是根據 new 指令碼後面之運算元取得 Class Symbol Table 之條目內容。CLSI_Access 是透過 Class Symbol Table 取出的類別 ID 進行 Class Info Table 的讀取，此時會讀出 objsize 與 isParsed 之旗號，並根據 isParsed 旗號來決定下一個狀態是 Class_Parsing 或 HeapAlloc。Class_Parsing 狀態目的是對 RISC 核心發出類別解析的請求。當進入到 HeapAlloc 狀態後，對目前分配到的物件開頭寫上類別 ID 作為標頭檔，完成後把分配到的物件位址推到堆疊上並將 objsize 與 current_heap_ptr 相加。

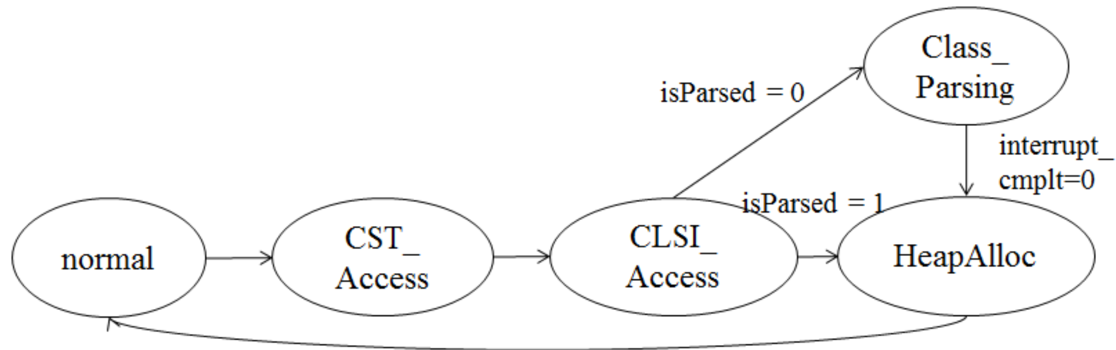


圖 7 new 指令之動態解析器狀態變化

在 Java 裡面，有三種指令可以實現陣列宣告的操作，分別是 newarray、anewarray 及 multianewarray。這三種指令分別對三種不同型態的陣列作堆積空間的分配，newarray 是對 1 維 primitive type 陣列的空間分配指令；anewarray 是對 1 維 reference type 陣列的空間分配指令；最後，multianewarray 是對多維度的 primitive type 及 reference type 陣列的空間分配指令。目前的 JAIP 能夠在硬體上單獨完成 newarray 及 anewarray 指令，multianewarray 則須倚靠 RISC 核心的協助。以下分別介紹 newarray 及 anewarray 的實作。

newarray 指令的陣列長度是透過堆疊來傳遞，陣列長度必須在使用 newarray 之前由其他 bytecode 將之推上去。而陣列之型態是經由 Java compiler 編譯完成後，已被串接在 newarray 指令碼之後方 1 byte。在舊版的 JAIP 中，newarray 也像 new 指令一樣，是透過 IPC 介面將堆的配置請求傳給 RISC 核心，並由 RISC 執行對應的 ISR 來完成堆空間分配[20]。newarray 指令在我們的實作中，採取硬體途徑來分配堆積空間，實作方式與 new 指令類似，不一樣的地方是 newarray 不需要經由動態解析器來取得欲分配的空間，因為根據 Java bytecode 定義，newarray 指令執行前就必須將陣列長度推到堆疊頂端。另外，newarray 宣告的陣列元素類型皆為 primitive type，陣列元素為 reference type 之宣告為 anewarray 指令，而多維度陣列是用 multianewarray 指令。

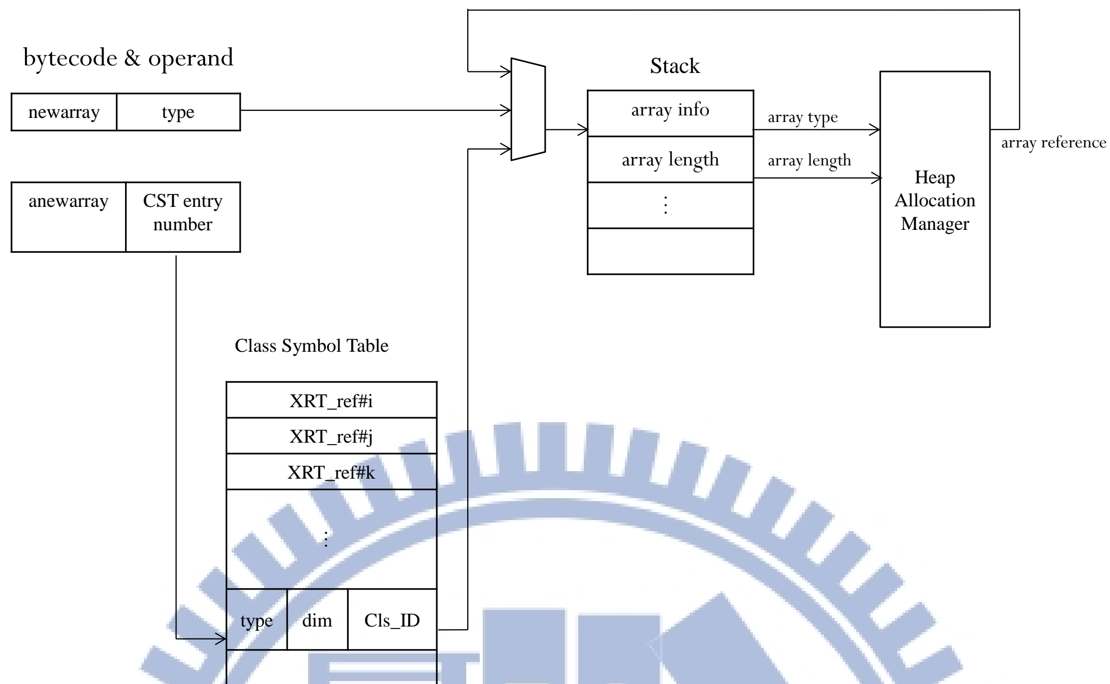


圖 8 newarray 及 anewarray 空間分配過程

由於 newarray 指令在 JAIP 中無法在 1 個 cycle 內完成執行，translation stage 會判斷此指令是屬於 complex instruction，因此在 fetch stage 會被轉換成為兩個 j-code，第 1 個 j-code 是將 newarray 指令碼後方的運算元推到堆疊頂端中，這樣做是為了利用堆疊來傳遞引數給後面的 j-code。此時陣列的形態及長度皆在堆疊上。第 2 個 j-code 是啟動 heap allocation unit。Heap allocation unit 被啟動後，根據堆疊上的型態、長度計算出對齊 4 byte 之大小，再將陣列型態及長度寫到目前可用堆積空間，完成後將 array reference 推到堆疊頂端，再將可用堆空間位址加上對應的大小。

anewarray 與 newarray 指令一樣，需要兩個參數，分別是陣列型態和長度。陣列長度需要搭配其他指令將陣列長度推到堆疊頂端，之後在執行 anewarray 指令。跟著 anewarray 後面的運算子代表著一個常數池項目的編號，這個常數池存放陣列的資訊，包含了型態、維度與類別編號，anewarray 只需要類別編號，維度和型態是供 multianewarray 指令參考。與 newarray 指令一樣，anewarray 指令被管線轉換成為數個 j-code 指令，第一個 j-code 設置適當的旗號並啟動動態解析器，

動態解析器根據 newarray 之運算子取得陣列資訊，第二個 jocde 將陣列資訊推到堆疊頂端後，第三個 jocode 啟動 heap allocation unit，heap allocation unit 被啟動後，與 newarray 一樣，根據堆疊最上面的兩個元素，分別是陣列資訊與長度來做 tag 與陣列長度寫入陣列頂端動作，並分配適當空間。

3.5 Hardware Native Interface

在 Java 語言中，native method 是指透過 native code 所實作出的 class methods。透過 native code，可以使用其他語言或組合語言所實作之 functions 或 libraries。一些與作業系統或機器相依之 method，JVM 皆是以 native method 完成的。此外，一些經常性使用的簡單且與效能相關的 methods 也會以 native method 來實作，以節省 bytecode 之 interpreting 時間。

在 JAIP 之中，native method 是由 RISC-core 之指令或 library 完成。一些 I/O 資源的存取 method，例如 consoleOutputStream.wrtie()，或是 Java bytecode level 無法完成之操作，例如 Class.forName()，皆是以 8 個暫存器作為 Mailbox，並利用 RISC-core 所提供之中斷服務作為 native method 之呼叫。然而，由先前的研究所指出[9]，這種介面的 method 呼叫過程產生了極大的 overhead。因此，在 CLDC 1.1 版中的 String.charAt()或 String.euqals()這種為了效能而使用 native function 完成的 method，在我們的系統中皆是直接以 java bytecode 實作之。而 java bytecode 無法完成之操作則仍然使用 RISC-core 來完成。此外，本篇論文中，我們提出了 Hardware Native Interface，此介面的擴充讓 JAIP 多了一種 method invocation 方式。如圖 9 所示，本篇論文的設計，使得 JAIP 擴充到 3 種方式來完成 invocation。一般的 Java method 若直接由 Java 語言撰寫的話，DSRU 透過解析出來的 method ID 與 Class ID 傳遞給 Method Area Circular Buffer Controller，再將 Java bytecode 讀出。Native method 之 invocation 有兩種方法，第一種為 IPC，這個介面將參數與 ISR ID 傳遞給 RISC-core，RISC-core 提供對應的中斷服務完成 native method。第二種 native method invocation 則為 native hardware interface，透過此介面，JAIP 可以使

用 hardwired 方式，或者是其他硬體加速器模組來完成 native method。目前 JAIP 提供 4 個使用 native hardware interface 之 method，本章節將會介紹其設計。

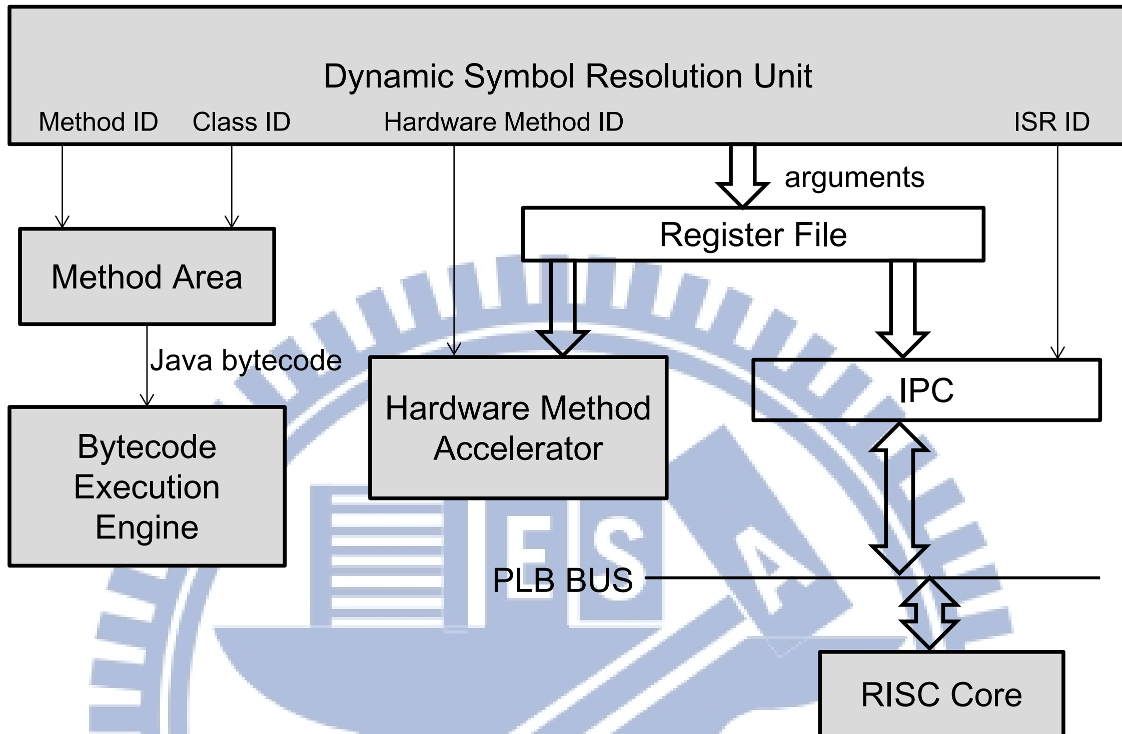


圖 9 JAIP 的三種調用介面： (i)Java bytecode(ii)RISC 核心(iii)硬體加速器

3.5.1 Hardware Native Interface 之實作

類別方法的呼叫是透過 invokevirtual 的指令碼，若是靜態方法的呼叫則使用 invokestatic。invoke 類型指令會啟動動態解析器，一開始動態解析器停留在 normal 狀態等待管線下的命令。一旦管線的 decode stage 解碼發現有一 j-code 需要動態解析，將運算元及旗號傳遞給動態解析器，動態解析器根據 decode stage 給的旗號來決定工作內容。一旦啟動後，動態解析器便開始解析 Class Symbol Table 與 Cross Reference Table 中的內容。Class Symbol Table 之 entry 編號是根據 bytecode 之 operand。Class Symbol Table 中的 entry 為一個 Cross Reference Table 的索引，透過這個索引，動態解析器可以取得每個 method 的 method info。如圖 10 所示，method info 之格式包含了兩個 word。non-native method 之 method info 包含了類別 ID，實作類別 ID，參數個數，method ID 與 method list 之下一個 node 索引。

而 native method 之 method info 包含了類別 ID，參數個數，回傳值大小與 native method ID。8 位元的 native method ID 是根據最高有效位來判斷此 native method 是由 RISC-core 實作與否。也就是說，如果 native method ID 小於 128，此 method 便是 RISC-core 實作；若大於等於 128，此 method 是由 hardwired 電路邏輯或加速器實作。此外，若 native method 為 RISC-core 實作，native method ID 為 ISR routine 之 ID；若為 hardware native，此 ID 代表 hardware native 之編號。

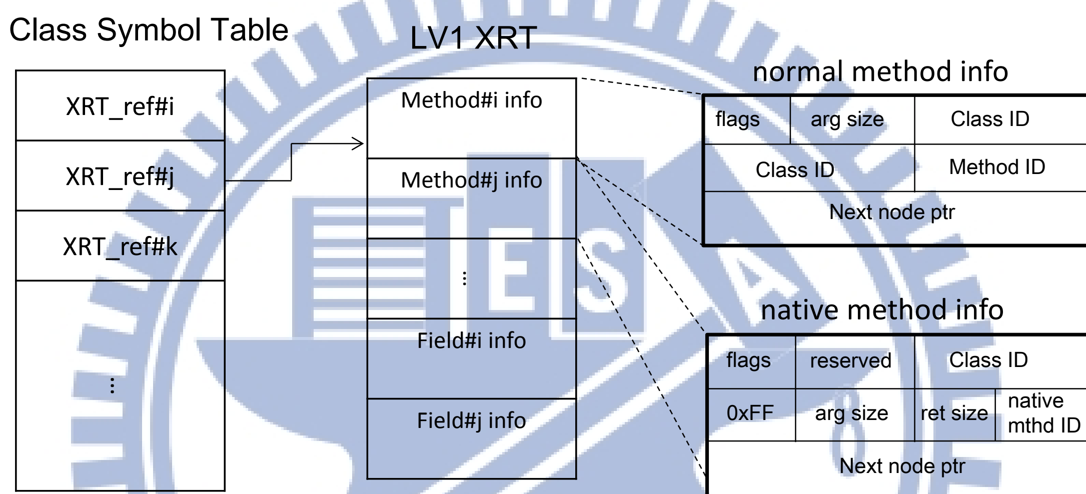


圖 10 method info 訪問過程與 method info 格式

動態解析器過程如圖 11 所示。Get_L1_XRT_ref 狀態是根據 decode stage 傳送過來的運算原來決定讀取哪一個 Class Symbol Table 項目，而從 Class Symbol Table 讀到的內容會是一個 Cross Reference Table 的位址。L1_XRT_Access 狀態是從第 1 級的 Cross Reference Table 中讀取方法資訊。Class_Parsing 狀態是根據 Cross Reference Table 讀到的方法資訊來判斷此類別是否被解析過，若無則進入此狀態發送 interrupt 及 ISR ID 給 RISC 核心，請求類別的解析服務。Native_start 狀態根據方法資訊來決定要如何調整堆疊上的引數。Native_HW 狀態觸發及等待硬體方法加速器的完成。最後根據方法的回傳值大小來調整堆疊。

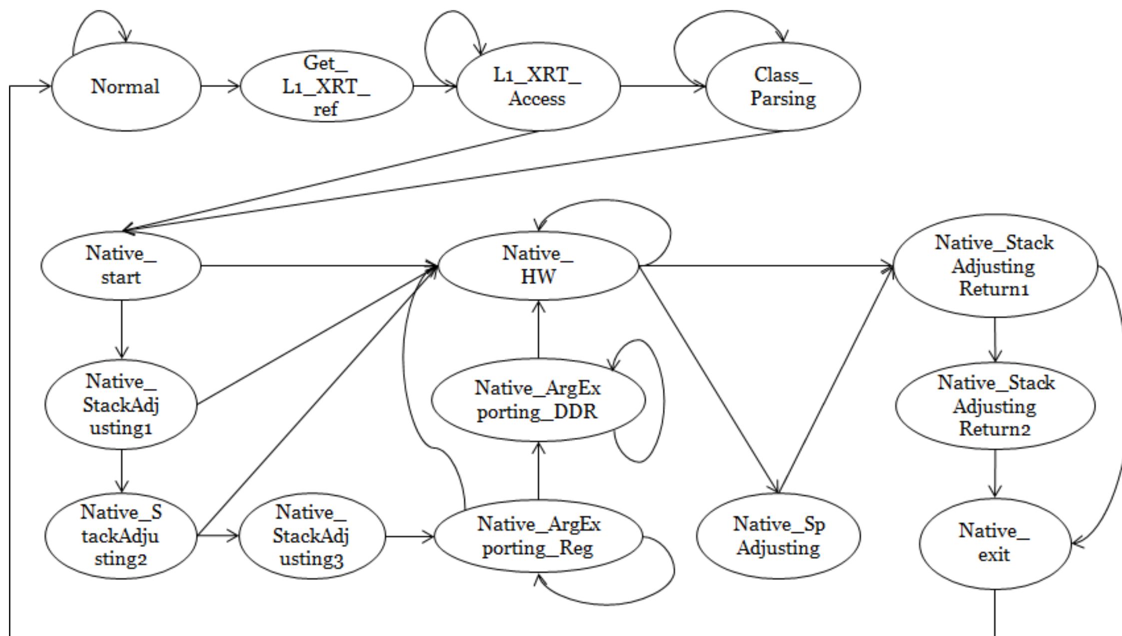


圖 11 Hardware native method 在動態解析器內之狀態圖

承襲 IPC 介面實作原生方法之方式，我們建立了一個提供 JAIP 使用硬體方法加速器介面，如下圖 12。其中 IPC 介面和硬體加速器的引數、回傳值的傳遞是透過專用暫存器和第一級堆疊。由於第二級堆疊是以 BRAM 方式實作，若利用第二級堆疊傳遞引數會使得每一個引數之讀取都得花費 1 個 cycle 時間。而第一級堆疊目前只提供 3 個 32 bit 之暫存器，所以在林的研究中[20]，提出以 5 個專用的暫存器來傳遞引數。Exception 方面，hardware native 產生的 exception 是由 JAIP 上之 Exception Handling Unit 來處理。此電路邏輯是透過 exception ID 找尋 exception lookup table 上對應的 exception handler，並將 exception handler 載入 Method Area 並在 BEE 上執行。其 exception 物件產生過程不像 bytecode athrow 一樣由 new 指令產生，而是給 Exception Handling Unit 一個 exception ID，由 Exception Handling Unit 自行產生例外物件。回傳值方面，部分 hardware native method 是有回傳值的，hardware native 透過 JAIP 內部 memory access logic 對 first-level stack 進行 push 動作。也就是說，在不需要使用 bus 情況下，透過 local address 對 stack 進行 read-write 動作。而 second-level stack 上之參數之 pop 動作，是由動態解析器根據解析出的參數個數及回傳值大小來進行的。

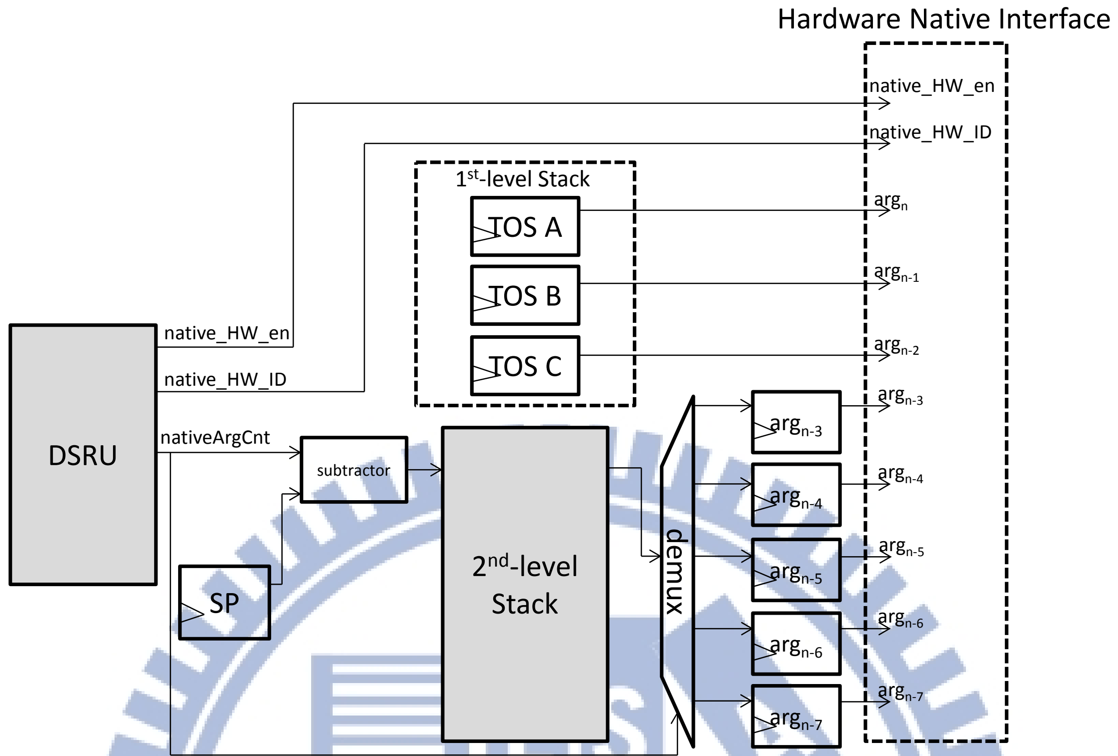


圖 12 Hardware Native Interface(i)

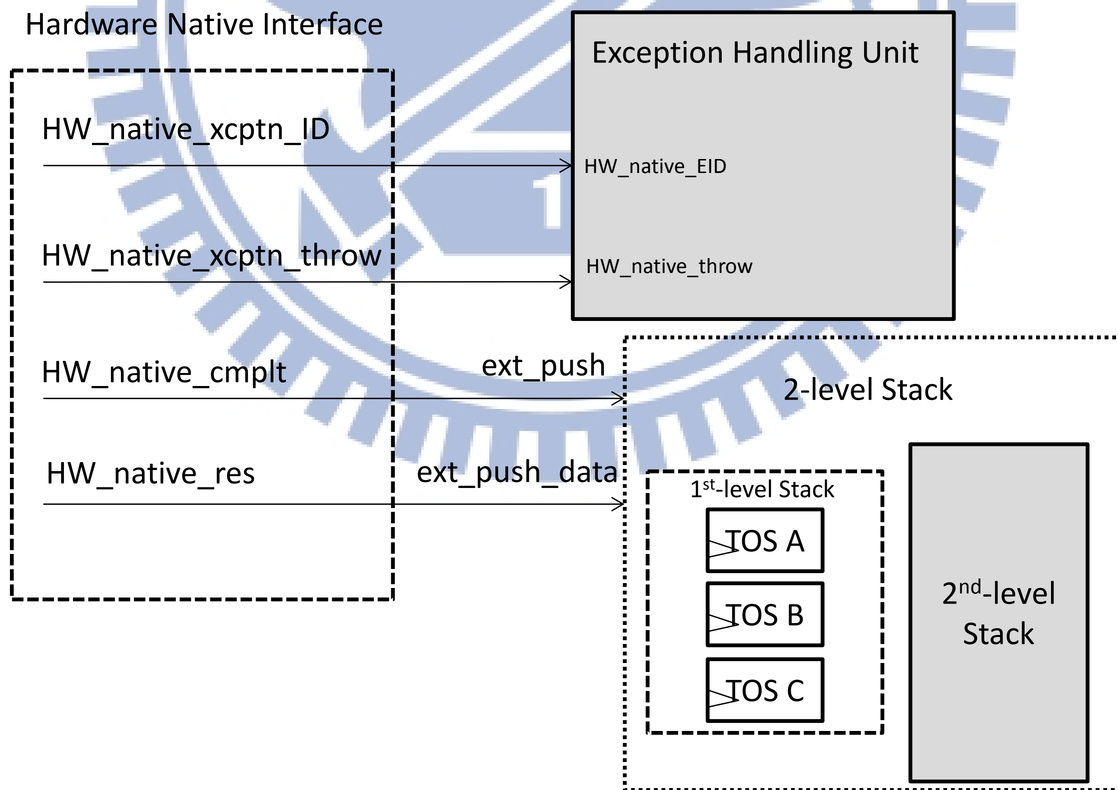


圖 13 Hardware Native Interface(ii)

3.5.2 字串操作加速目標分析

為了加速 JAIP 之字串操作性能，我們針對 Java 標準字串類別庫中的 String 以及 StringBuffer 之中的基本操作進行分析。本小節是詳細描述 arraycopy 及 indexOf 兩個 Java method 在字串操作之中之重要性。此分析是與後文 4.3.1 小節中，字串處理程式之 method profile 是吻合的。

多數修改字串的方法會使用到 arraycopy，arraycopy 執行的效能若太差，很容易影響整個字串操作程式。在過去 JAIP 對 arraycopy 的實作中[20]，是使用 IPC 介面來觸發 RISC 核心上定義好的 arraycopy 原生方法，而這樣的設計讓 JAIP 耗費相當可觀的執行時間開銷。本小節開頭先介紹 arraycopy 使用時機，而後介紹本論文的 arraycopy 方法使用介面及設計。

arraycopy 方法在 String 類別裡面是相當重要的一個方法，它被直接或間接呼叫的 call graph 如下圖 14 所示，此圖畫出了 CLDC 版本之 String 類別方法對 arraycopy 使用情況。由於字串的存放空間在 JVM 屬於 Permanent Generation，此種空間永久不被 Garbage Collection 回收，直到 Java 程式結束。因此每當字串內容被更改時，都必須創建新的字串物件來存放被修過的字串內容。而每次將新字串內容存放到新的字串物件時，都會直接或間接用到 arraycopy。例如當我們使用 concat(String)將 B 字串接到 A 字串之後端時，一種實作方法便是先創建新的字串物件，將 A 字串利用 arraycopy 拷貝到新物件，再將 B 字串利用 arraycopy 拷貝到新字串物件的尾端。

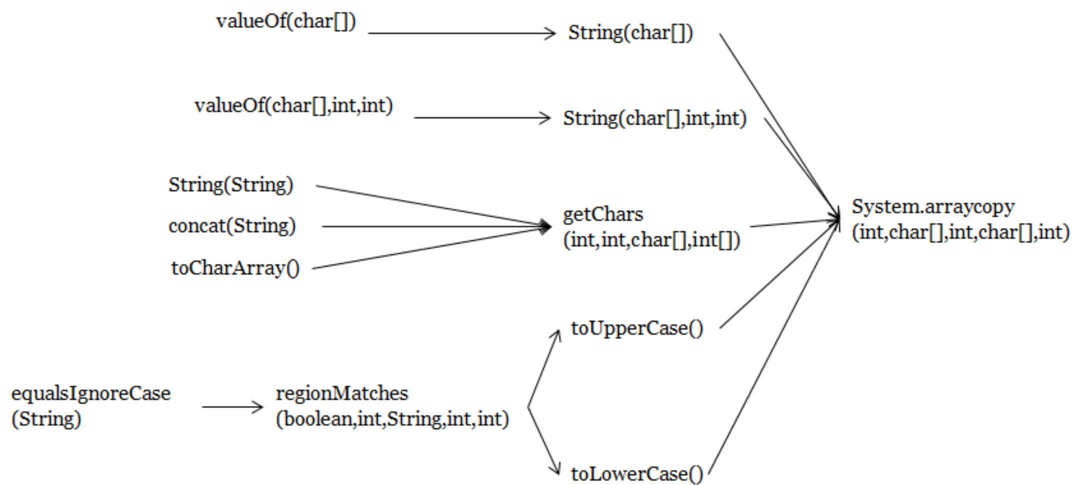


圖 14 String 類別裡 arraycopy 直接或間接呼叫關係圖

字串處理程式常用的類別有 StringBuilder 及 StringBuffer，兩種類別都是提供一個字串緩衝空間，這個緩衝空間大小是預先設定好的，任何需要大量被操作的字串在每次操作結束都可以先暫存在緩衝空間中，等到所有操作結束後，再利用 toString 方法將緩衝空間內的字串內容轉換成為真正的 String 類別，操作過程中，如遇到空間不足時，利用 expandCapacity 方法增加空間。StringBuilder 及 StringBuffer 的差別在於 StringBuffer 是 thread-safe 的，而 StringBuilder 只適合在 single-thread 程式中使用。StringBuffer 裡面常用的方法有 append、insert 等等，這兩個除了本身就會呼叫 arraycopy 之外，有時候還會透過兩個 private 的方法，expandCapacity 及 copy 間接使用 arraycopy。expandCapacity 是當 StringBuffer 空間不足的時候，expandCapacity 會宣告新的字元陣列空間，而後會將字串值透過 arraycopy 從舊的空間拷貝到新的空間。copy 則是為了支援快速的 toString 方法，toString 時僅將字元陣列的位址賦予給字串物件，任何會修改到 StringBuffer 字元陣列內容時，採取 copy-on-write 策略，而 copy 這個 private 方法就是在這個時候被使用，copy 的實作也是使用 arraycopy。

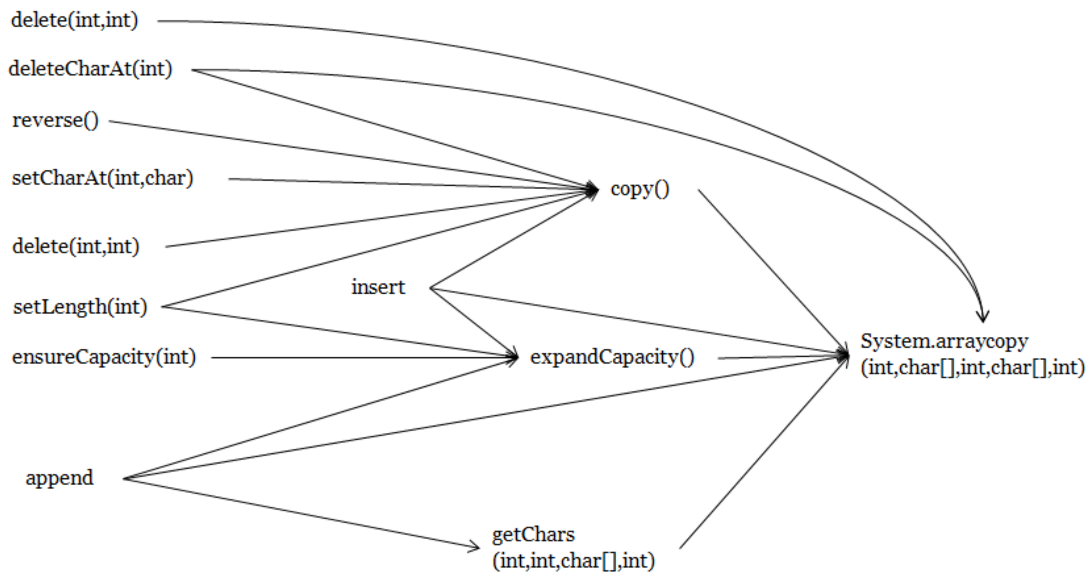


圖 15 StringBuffer 類別裡 arraycopy 直接或間接呼叫關係圖

在 Java String 類別中，indexOf 功能是進行子字串或字元搜尋：在一個本文 t 中，找尋是否存在一個 string pattern s 或字元 c，並回傳其最小索引值 k 使得 t[k:k+nc] = s 或 t[k] = c。此方法應用很廣泛，例如 DNA 序列檢驗，網路封包檢查。而與字串處理本身相關性較強的應用，存在於許多腳本語言解析器或 xml 解析器。例如某個語言的 parser 要檢查一個 identifier 的字元是否為合法的字元，這時就可以使用 indexOf()來搜尋 identifier 的字元是否存在於一個由所有合法的字元組合起來的文本中。indexOf 也可以做為 substring 的輔助工具。例如 xml parser 要切開 xml 標籤中的 namespace 與 prefix 的時候，可以使用 indexOf 找尋其”:”字元之索引值，在透過其索引值使用 substring 之 method。目前在我們的系統中，共有 3 個 Java methods 會直接使用 Hardware Native Interface 來呼叫 indexOf accelerator，如表 3 所示。3 個 method 會分別使用 3 種不同的信號線觸發其加速器，因此加速器可以很容易地知道參數型態和工作項目。其中 String.indexOf(String str, int fromIndex)會透過加速器內部的 Object Resolution Unit 解析其物件的 field 以取得更細部的參數。因此，不需要額外的程式碼來傳遞或轉換參數。String.indexOf(String str, int fromIndex)是典型的 string matching 操作，呼叫它的 object instance 及第一個 String object 參數分別代表 text 及 string pattern。

String.indexOf(int char, int fromIndex)除了 string pattern 參數是直接使用字元以外，與上者之操作內容相同。

表 3 透過 Hardware Native Interface 使用 indexOf accelerator 之 Java methods

Methods using the indexOf Accelerator
String.indexOf(String str, int fromIndex)
String.indexOf(int char, int fromIndex)
String.lastIndexOf(int char, int fromIndex)

3.5.3 arraycopy 硬體加速器設計

String 及 StringBuffer 裡面，arraycopy 算是很關鍵的原生方法，故執行效能不容小覷。這我們的實作中，我們使用專用硬體來處理 arraycopy 方法。arraycopy 硬體加速器的架構如圖 16。主要有 4 個部分組成，控制整理運作的 Arraycopy Controller、讀取陣列來源的 Reader、寫入目標陣列的 Writer 以及一個暫存 Reader 讀到的元素之 Array Circular Buffer。arraycopy 方法剛被呼叫時，arraycopy 硬體加速器便啟動。Arraycopy Controller 一開始會檢查來源陣列的位址及目標陣列的位址是否為 0，是的話即停止 arraycopy 並且傳送 NullPointerException 信號給 JAIP 上的 exception handler。假如沒有 NullPointerException，依據傳送進來的參數讀取來源陣列之 tag 及長度以及目標陣列之 tag 及長度，檢查來源陣列及目標陣列之 tag 是否相同，若不相同則發送 ArrayStoreException 給 exception handler。之後再檢查是否有 IndexOutOfBoundsException，檢查方式是根據參數—srcPos、destPos、length 及兩個陣列的長度計算出運算過程中有沒有可能超出陣列邊界範圍。Reader 每次都會讀取 1 個 word 大小的陣列元素，而 1 個 word 的資料有幾個陣列元素則要依據陣列是哪種型態，例如陣列型態是 UTF-16 的 character，每個 word 包含兩個元素；若陣列是 32-bit integer，1 個 word 則只有一個元素。每次讀取到的陣列元素

都暫存在 Array Circular Buffer，Reader 讀取完 1 word 資料後，進入等待狀態，等待 writer 將 buffer 裡面的元素寫到目標陣列，在進行下一次的讀取。Reader 的任何讀取案例皆可以完成，即使有 non-alignment 的情形，也只要忽略那些未對齊的資料，不要將之儲存到 buffer 裡即可。Writer 在進行寫入動作之時，需使用 byte-enable 功能，writer 內部有控制 byte-enable 之電路邏輯。所謂 byte-enable 功能即是在每次讀寫一個 word 資料時，分別去控制讀寫單一 word 裡面的每一個 byte 有效與否。Writer 每次在 Reader 結束一次的讀取，都會把 buffer 中的陣列元素寫到目標陣列。Non-alignment 情形可能會出現在陣列的頭和尾，因此陣列的頭和尾都要檢查位址是否對齊。開頭的對齊與否只需檢查目標位址，而尾端的對齊則依據目標位址和拷貝的長度來算出。為了避免每兩次寫入動作之間產生沒有對齊的情況，除了陣列開頭和尾端之外，每次的寫入動作以單個 word 為基本單位，不足一個 word 之 bytes 只需存放在 circular buffer 中，併入下一次的寫入資料。

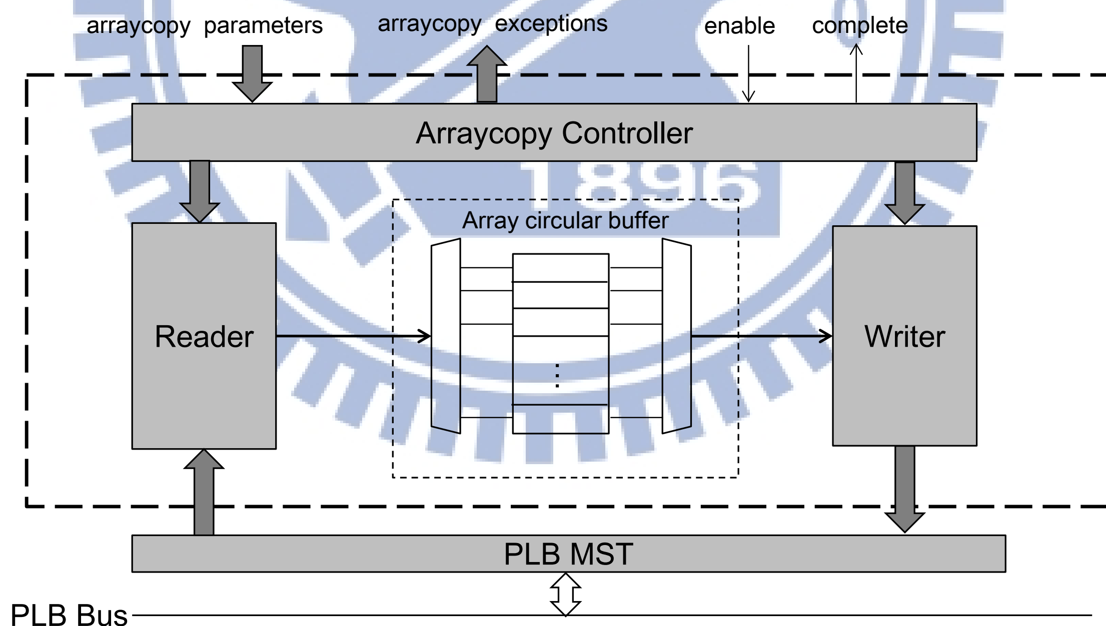


圖 16 arraycopy 硬體加速器

arraycopy 在 Java 規格書中定義了三個 exception，分別是 NullPointerException、ArrayStoreException 及 IndexOutOfBoundsException。NullPointerException。NullPointerException 是發生在 source array 或 destination array 其中一個為 null 的

時候丟出；ArrayStoreException 是發生在 source array 和 destination array 型態不一致的時候丟出；IndexOutOfBoundsException 是發生在程式被檢查出會存取超過陣列邊界時丟出。在我們的實作中，採用了郭所提出來的適合 Java 硬體處理器之例外處理器[36]。在硬體 arraycopy 加速器一啟動時就檢查 source array 和 destination array 是否為 null; ArrayStoreException 及 IndexOutOfBoundsException 是等到 arraycopy controller 讀取完 source array 和 destination array 的 tag、長度後進行檢查。一旦例外條件成立，將例外觸發信號和它的 Exception ID 傳送給硬體例外處理器，並且立即離開 arracopy 方法，交由例外處理器找尋適合的例外處理常式。

3.5.4 indexOf 硬體加速器設計

為了以低成本的硬體實作 indexOf 加速器，我們採取 Brute Force 作為字串比較演算法。它不像 Knuth-Morris-Pratt 或者 Boyer-Moore 等演算法需要額外的記憶體空間及 preprocessing，或者是 CAM 的龐大記憶體量需求。本加速器不需要額外的 RAM 空間，僅需對 heap 連接 data read/write ports 即可。然而，雖然採取最簡易的演算法，不代表著演算法沒經過優化，我們針對 data width 為 32-bit 之 bus 做演算法上的改進。在 JVM 中，字元採用 UTF-16，因此每次 memory read 最多可以讀進兩個字元。因此，我們可以在 String matching 之外層迴圈在 single clock cycle 內進行 3 個 character comparison，以下例子解釋為何進行三次 comparison。假設 text 字串存在陣列 t，string pattern 存在陣列 s，且目前正進行第 i 個 iteration，一次記憶體讀取可以得到 t(i, i + 1)，這兩個字元現在需要跟已經暫存在正反器中的 s(0, 1)比較，這四個字元最多可以進行 3 個 comparison，分別是 t(i)和 s(0)，t(i + 1)和 s(1)以及 t(i + 1)和 s(0)，若比較失敗，i 值可以一次累加 2。而內層迴圈只需在 single clock cycle 內完成 2 次 comparison 和兩個字元的 shift。

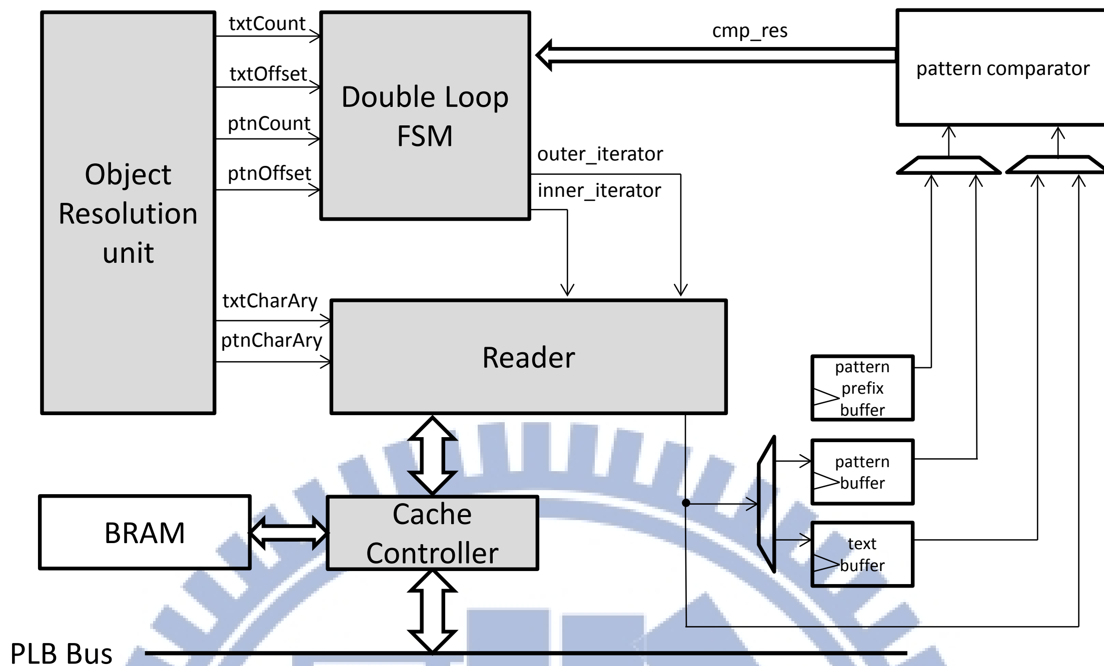


圖 17 indexOf 硬體加速器

如圖 17 所示，indexOf 硬體加速器內部主要是由 Object Resolution Unit，Double Loop FSM，Reader，pattern match unit 及兩個 32-bit 之 buffer。Object Resolution 負責解析字串物件的 field。為了要計算迴圈的起始點和中指點，以及 text 和 string pattern 的記憶體位址起始點和終止點，object resolution unit 會將解析後的資訊傳給 reader 和 double-loop FSM。double-loop FSM 根據 object resolution unit 解析的結果和 indexOf 參數初始化在第一層迴圈的 iterator，之後根據 pattern comparator 結果來決定是否進入第二層迴圈。pattern comparator 主要是由四個 16-bit 之比較器組成，比較結果是一個 4-bit 的 vector，每個 bit 都代表著一組 buffer 裡的 text 字元與 string pattern 字元之比較結果。pattern prefix buffer 是 32-bit flip-flop，固定存放著 string pattern 之前兩個字元，indexOf method 一開始執行後，這個 buffer 的內容不會被改變。由於大部分搜尋時間都會在 text 上找尋 string pattern 之 prefix，將 string pattern 之 prefix 存放起來可以避免每次外部迴圈的記憶體讀取時間。pattern buffer 是 48-bit flip-flop。受限於 cache 及 bus 的 read port 都只有一個，在進入第二層迴圈時必須將讀進之 32-bit string pattern 之片段暫存起來，

而另外 16-bit 是考量 string pattern 片段與 text 片段的兩個索引沒對其的情況下，string pattern 最多會有 3 個字元在加速器中等待比較。Text buffer 為 16-bit flip-flop，此設計也是考量到 string pattern 片段與 text 片段的索引沒對齊。

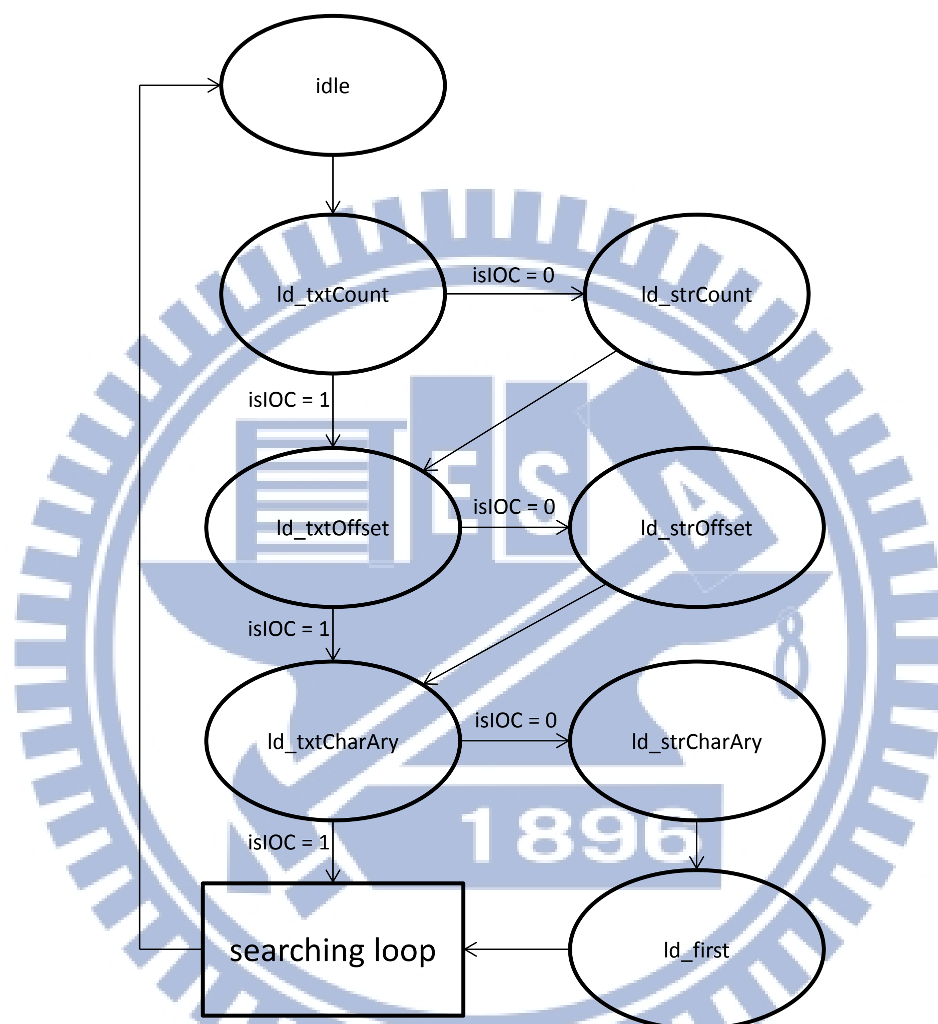
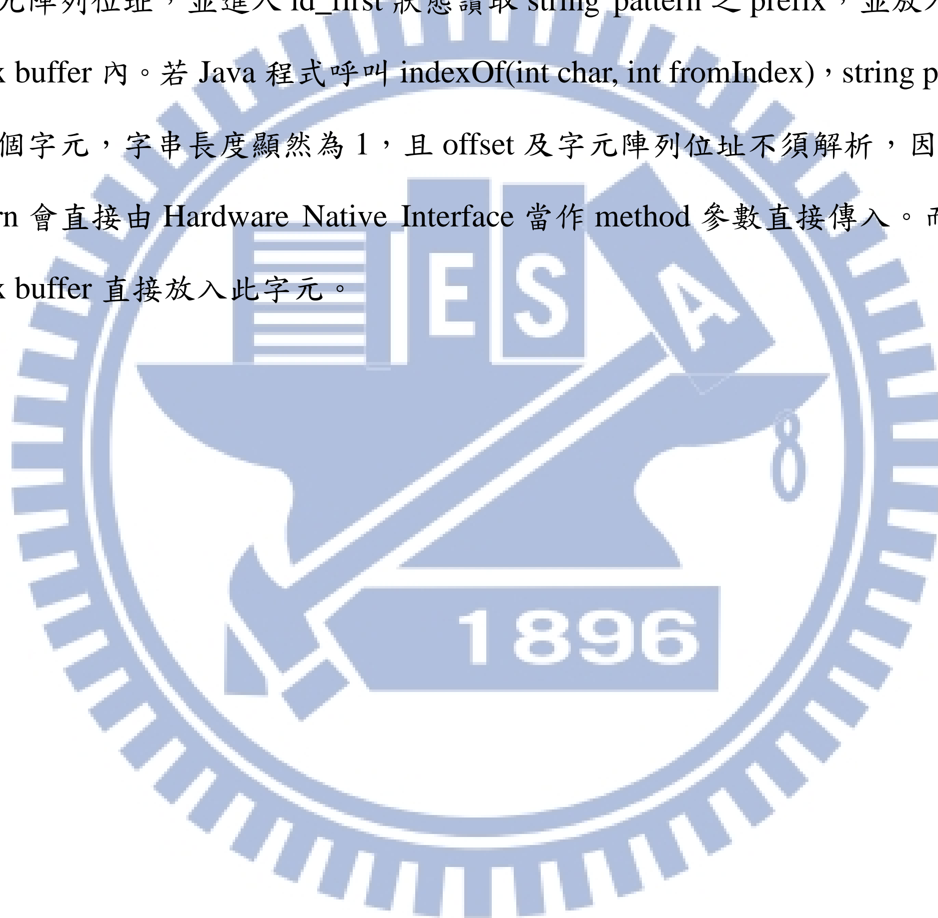


圖 18 Object Resolution Unit 狀態機

Object Resolution Unit 主要在進行物件的 field 之解析動作，其中物件之 field 包括字串長度，offset 或字元陣列位址等。Object Resolution 過程如圖 18 所示。Object Resolution Unit 分別解析由 Hardware Native Interface 傳進來的其中兩個參數，分別是代表 text 的 String Object 及 string pattern 的 string object。由於 indexOf 在字串類別中有重載(overloading)，分別是 indexOf(String strPtn, int fromIndex)及 indexOf(int char, int fromIndex)，class parser 在解析過程中將此兩種型態的參數輸

入賦予不同的 method ID 及 native method ID，因此 Hardware Native Interface 只須根據 native method ID 來設置不同的啟動信號，分別是 IOEn 及 IOCEn。若 idexOf 加速器是由 IOEn 啟動，isIOC 旗號設置為 0;若由 IOCEn 啟動則 isIOC 旗號為 1。若 isIOC 旗號為 1。Object Resolution 則需另外進行 string pattern 之 field 的解析。也就是說，若 Java 程式呼叫 indexOf(String strPtn, int fromIndex)，string pattern 是由一個 string object 包裝，indexOf 加速器必須讀取此 string object 的長度、offset 及字元陣列位址，並進入 ld_first 狀態讀取 string pattern 之 prefix，並放入 pattern prefix buffer 內。若 Java 程式呼叫 indexOf(int char, int fromIndex)，string pattern 只是一個字元，字串長度顯然為 1，且 offset 及字元陣列位址不須解析，因為 string pattern 會直接由 Hardware Native Interface 當作 method 參數直接傳入。而 pattern prefix buffer 直接放入此字元。



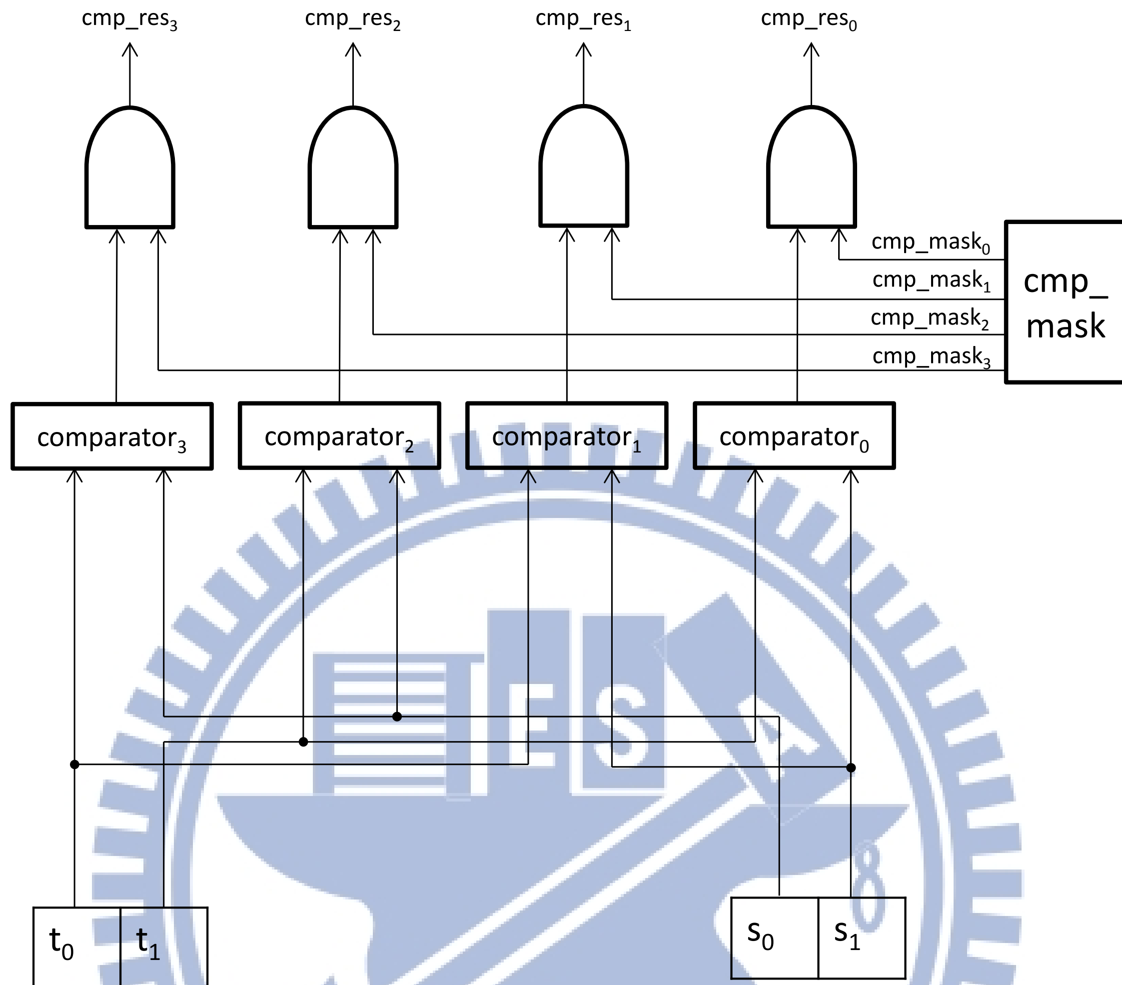


圖 19 pattern comparator

Pattern comparator 主要是由四個 16-bit 比較器組成。如圖 19 所示。由於 object cache 和 PLB bus 資料寬度都為 32-bit，等於兩個字元的大小，因此 indexOf 硬體加速器之設計是每次都以兩個字元作為比較單位。又因為每次比較的兩個字元中，兩個字元都有可能是 string pattern 之開頭，例如當 text 為 “supply”，string pattern 為 “up”，第一次迴圈便會比較 “su” 與 “up” 。若我們直接使用單一 32-bit 來進行比較，第一次迴圈便會有 false negative，第二次迴圈也會直接進行 “pp” 與 “up” ，那麼這次字串搜尋便找不到 text 中對應的 string pattern。因此，pattern comparator 使用 4 個 16-bit comparator 對所有 text 和 string pattern 的字元配對做比較，分別是 t0 和 s0，t1 和 s0，t0 和 s1 以及 t1 和 s1。Double Loop FSM 根據 pattern comparator 計算出來的 4-bit 向量來決定是否進入第二層迴圈。當 reader 讀到字串

邊界時且邊界是沒有 word-aligned 情形下，cmp_mask 可以消除那些無效字元之比較結果。cmp_mask 是由 text buffer 之 valid bit 及

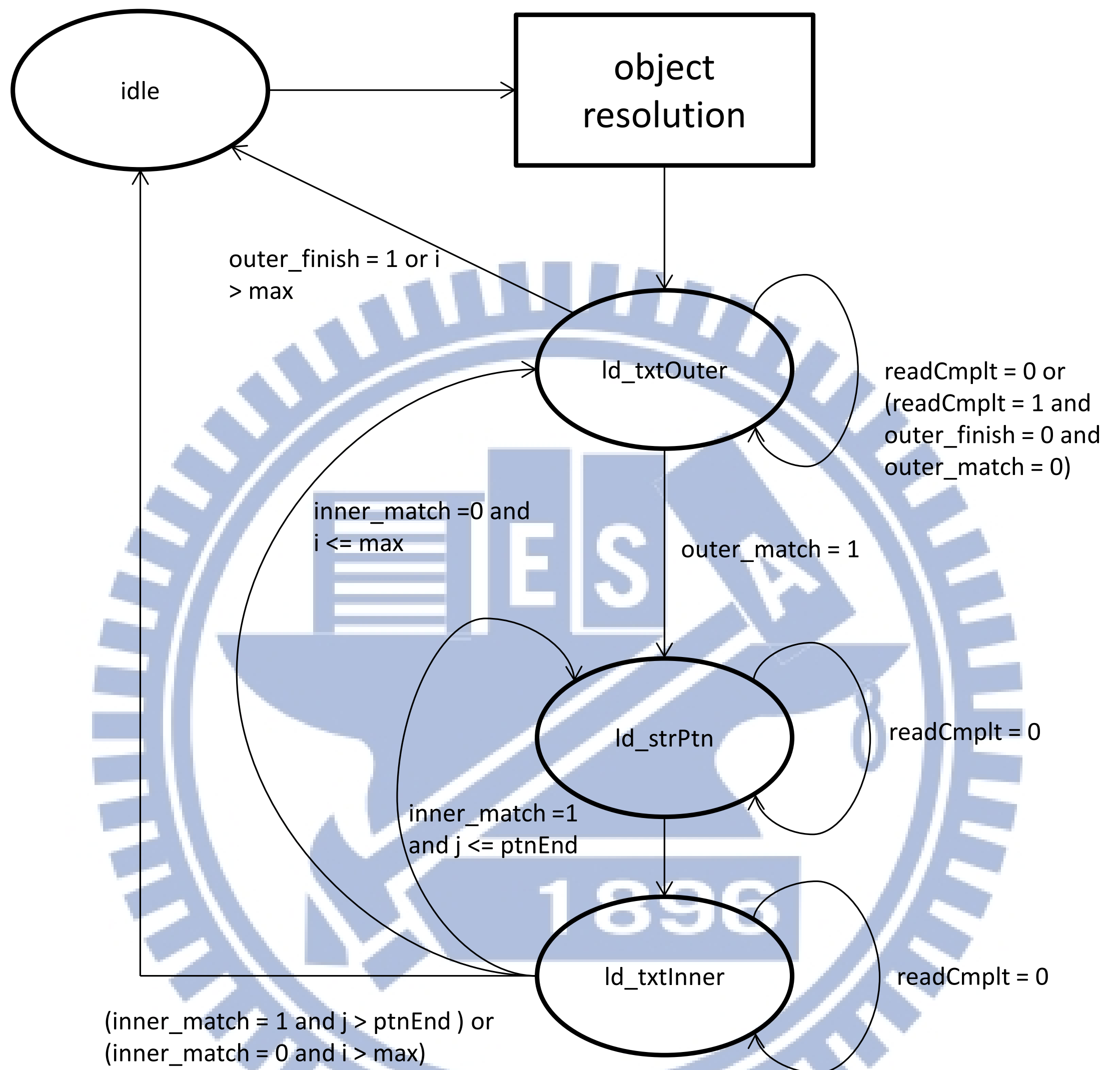


圖 20 Double-Loop State Transition Diagram

Double-Loop FSM 根據 pattern comparator 計算的結果進行迴圈，且每個狀態都會分別對 text 及 string pattern 讀取新的字元。此狀態機主要是進行內外兩層迴圈，外層迴圈是找尋 text 中可能會與 string pattern 匹配的起始點，更精確來說，便是找尋 text 中與小於等於兩個字元的 string pattern prefix 匹配的地方；內層迴圈則是從外層迴圈找尋到的起點開始比對，判斷 string pattern 是否可以完全吻合。ld_txtOuter 是讀取 text 新的字元，並且在讀取結束後與 string pattern 的 prefix 進行比較。若無任何匹配的情形，停留在 ld_txtOuter 狀態並讀取新的 text 字元。若

有字元匹配的情況，分成兩大類：outer_finish 及 outer_match，案例解說請參考圖 21 和圖 22。outer_finish 信號被拉起代表 string pattern 在外層迴圈已經完整的比較過並且匹配。這種情形只有在字元數目小於等於 2 個之時才有可能發生。String.indexOf(int char, int fromIndex)及 String.lastIndexOf(int char, int fromIndex)皆會符合此案例，而 String.indexOf(String str, int fromIndex)則要視其 string pattern 長度和是否 word-aligned 來決定此案例是否成立，若 string pattern 之大小為 2，但是沒有 word-aligned 之案例，double-loop FSM 仍然需要進入第 2 層迴圈才能完成 string matching。若外層迴圈之 outer_match 信號拉起，代表找到 text 與 string pattern 之 prefix 匹配的地方，此時便可進入第 2 層迴圈比對是否 prefix 後方的字元也完全吻合。一旦 outer_match 信號拉起，進入 ld_strPtn 與 ld_txtInner，代表 double-loop FSM 此時正在進行第二層迴圈並讀取 string pattern 與 text 之剩餘字元，兩個字串每次皆讀取 32-bit 進行比對。inner_match 信號為內層迴圈比較結果，由於內層迴圈必須檢查字串是否完整匹配，因此 inner_match 只有當 32-bit 內的字元完全吻合才會拉起。

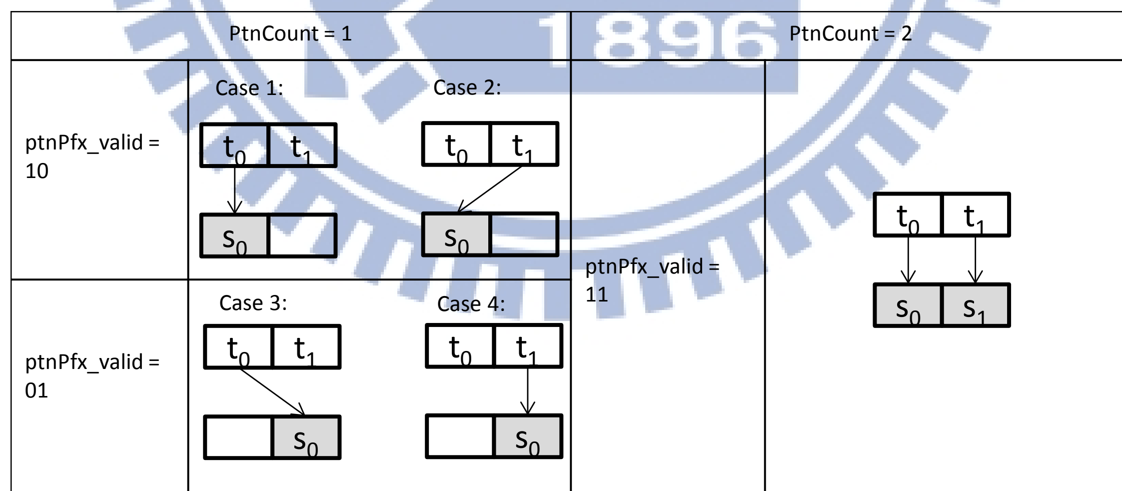


圖 21 利用 pattern comparator 結果判斷 outer_finish 信號

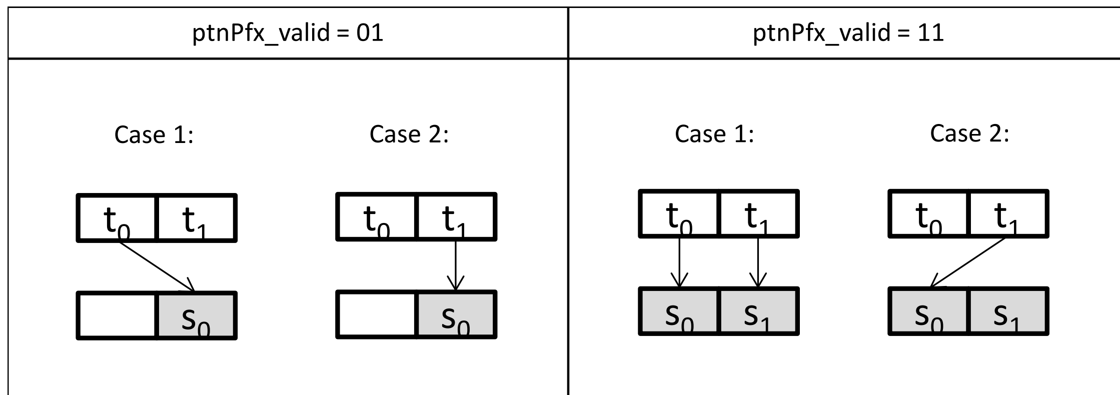


圖 22 利用 pattern comparator 結果判斷 outer_match 信號

3.5.5 其它使用 arraycopy、indexOf 加速器之 methods

雖然兩個字串加速器之命名是以兩個高度相關的 method 名稱來命名，但不代表這兩個加速器只能使用在這兩個 method 或 overloading 之 methods 上。由於 indexOf 加速器具有字元搜尋與字元比較之操作，陣列操作只要有搜尋或是比較操作，且元素大小是大於等於 2-byte，都有機會使用 indexOf 加速器增進效能。而本小節將介紹如何使用兩個加速器來加速 Java String Class 中的 equals()和 replace()兩個 methods。

在 JAIP 平台中實作 String.equals(String str) method 有三種方法，第一種是使用 c function 實作之 native method，此介面雖然廣泛在各種 JVM 上來加速，但由於 JAIP 若要使用 c function 定義的 native binary，需透過 IPC 介面傳送 interrupt 訊號到 RISC-core 上，此 overhead 不但沒有達成加速目的，反而降低程式效能。第二種方法便是直接使用 Java 程式碼實作。透過 Java 程式碼撰寫迴圈程式，每個 iteration 都被 javac compiler 編譯成為 caload 與 ifeq 等 Java bytecode，逐一比對字元是否相同。若要達到更有效率的比對操作，可以直接實作一個 equals 專用加速電路並使用 hardware native interface 來呼叫 method。但考量到 indexOf 加速電路已有高度相關操作，我們重複使用此加速電路。String.equals(String str)設計如圖 23 所示，首先檢查兩個字串物件之 reference 是否相等，若是相等，代表兩個字串是同一個 object instance，不需要進行字元陣列的比對，直接 return “true”。

若兩個 string object 不相等，這時候可能需要比對字元陣列才會知道是否 equals。比對字元陣列之前，先檢查兩個字串長度是否一致，若不一致，我們可以肯定兩個字串一定不相等。此 method 最為關鍵的地方在於字元陣列的比對。若兩個長度相等的字元陣列相符，使用 indexOf method 一定會回傳 0，因為兩個字串值在 index 等於 0 的地方互為子字串。由於 indexOf 加速器在字串比對上使用一個 word 當作單位，而第二種方法是使用單一字元為比較單位。因此，理想上此方法會比第二種方法快了兩倍。

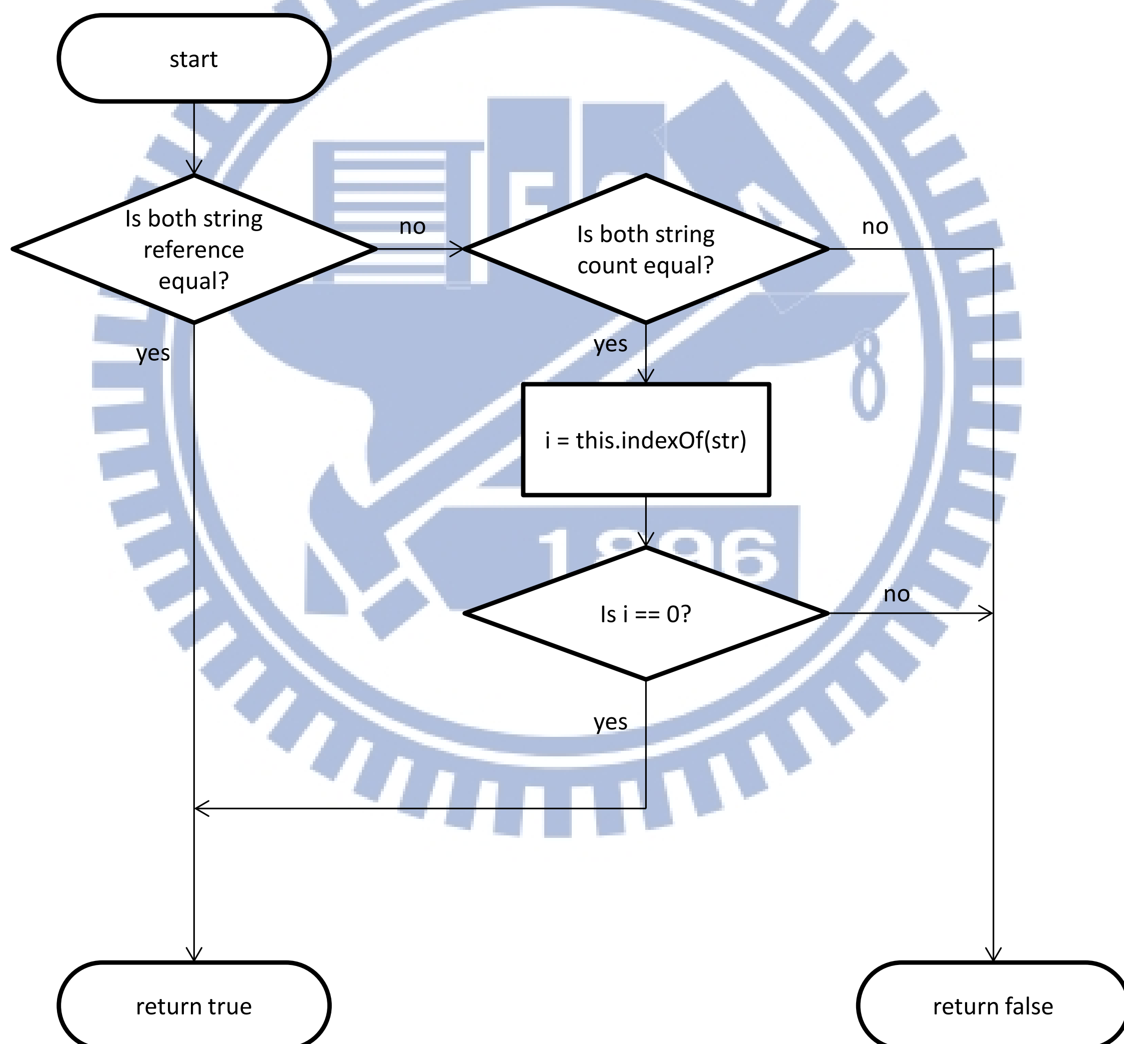


圖 23 使用 indexOf 加速器實作 String.equals(String str)之流程圖

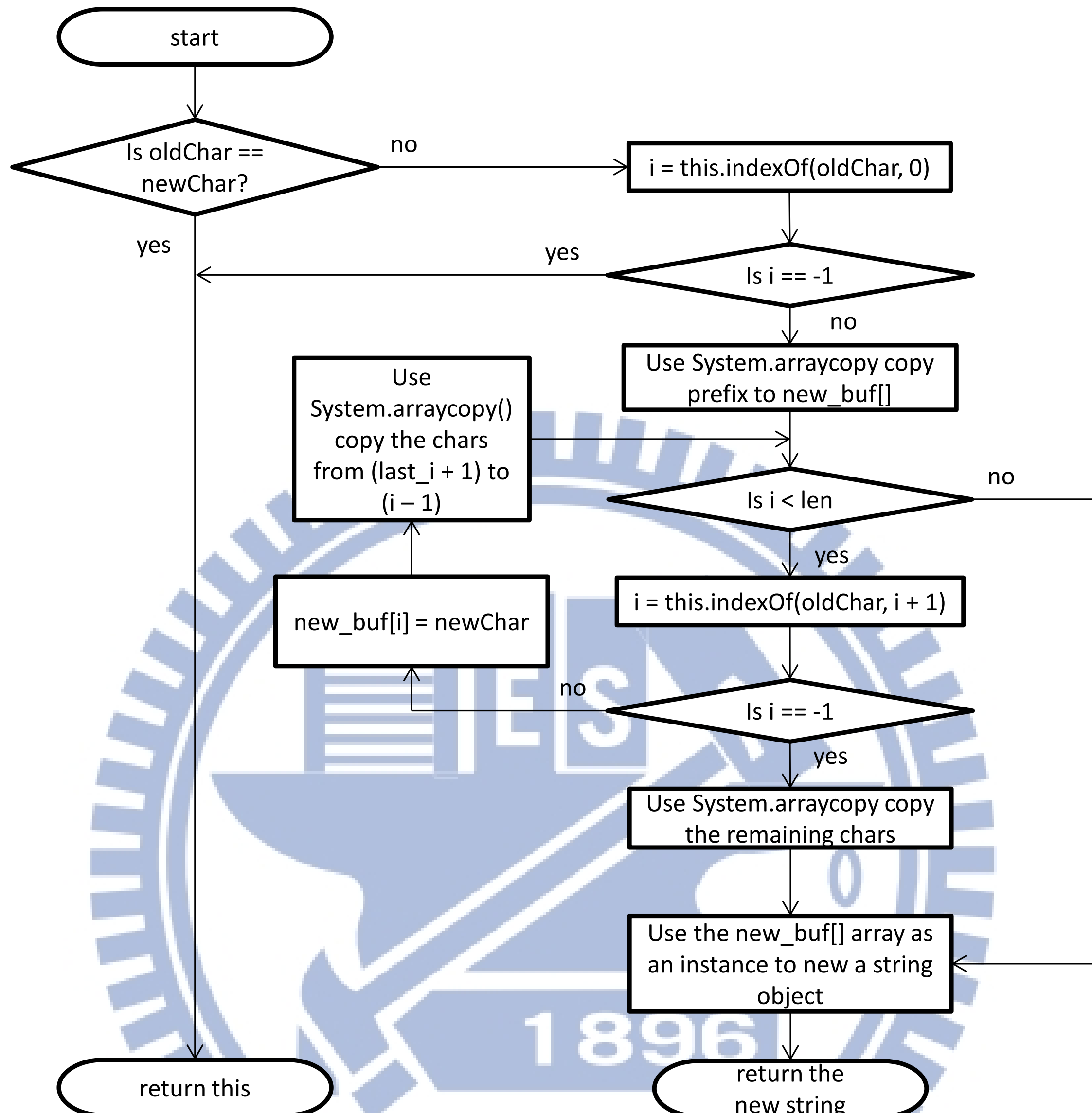


圖 24 使用 indexOf 與 arraycopy 加速器實作 string.replace()之流程圖

另一個使用字串加速器的 method 為 string.replace(char oldChar, char newChar)。此 method 是在一個 string object 內搜尋由 programmer 輸入的一個字元—oldChar，所有出現位置，並將之取代為另一個輸入的新字元—newChar。若位搜尋到符合的字元，直接回傳 string 本身；若至少一個 oldChar 被找到，宣告新的 string object 存放取代後的字元陣列，並回傳之。應用之案例如：當程式解析使用者輸入的檔案路徑、class path 或 URL 時，replace method 可以找尋其路徑的 separator，並取代成為系統內部使用的正規 separator，例如 Java class loader 可能會使用 className.replace('.', '/')。傳統作法即為撰寫一個單層迴圈，逐一搜尋。一旦欲被

取代的字元出現，直接將 newChar 寫進原本之位置。兩個加速器可以應用的地方在於 oldChar 之搜尋，以及 oldChar 未出現之字串片段複製動作。其完整流程如圖 24 所示。

3.5.6 其它使用 Hardware Native Interface 之 methods

表 4 其它使用 Hardware Native Interface 之 methods

MMESProfiler.profileOn()
MMESProfiler.profileOff()
System.currentCycles()

表 4 列出了除了 string 與 method 相關之外的其他 4 個使用 Hardware Native Interface 之 method。Hardware Native Interface 除了使用硬體加速器外，也可以用來直接對硬體電路之訊號線設定及讀取。MMESProfiler 是一個提供 profileOn 及 profileOff 兩個 method 的 class。在使用 method profiler 或 bytecode profiler 中，並非整個 Java 程式碼都是我們想剖析的。例如我們不會想要知道程式在初始化時或是打印結果時用了哪些 method 和 bytecode。profileOn()和 profileOff()分別是啟動和關閉 hardware profiler 之 methods。System.currentCycles()是取得 JAIP 之 hardware counter，此 counter 是紀錄系統從開機到呼叫這個 method 當下所總共經過 clock cycles 數目。此做法在不需要 RISC-core 或作業系統的協助之下得到程式耗用周期數。這個數目會被 push 到堆疊頂端，之後可以透過程式碼的換算，可以協助我們測量 Java 程式執行時間。

第四章 實驗結果

4.1 實驗環境

本論文所提出之字串加速器，提供了 Java 語言中 arraycopy 與 indexOf 的加速功能，與 JAIP 整合成為一個 IP，並且實作在 Xilinx ML-506 Evaluation Platform。此平台上包含了一顆 Virtex-5 FPGA，我們採用 MicroBlaze soft IP core 作為系統上的 RISC-core。系統頻率設置為 83.3MHz。JAIP 與字串加速器之 RTL 皆是以 VHDL 語言實作並以 Xilinx 之 XST 合成。此外，所有的電路皆透過 Xilinx iSim 模擬器驗證。軟體部分，我們使用 Xilinx SDK 來建立 Board Support Package(BSP)，這是一個輕巧、低階且獨立無須作業系統支援的軟體層，裡面包括了基本的操控各種硬體裝置(例如 UART 或 Memory controller)之函式庫，以及 C 語言的基本函示庫。RISC-core 之 runtime 環境(JAIP 初始化、class parser 或各種 ISR service routine 等軟體)是建立在此 BSP 之上。

在字串加速器使用的電路大小方面，arraycopy 及 indexOf，在此平台上使用之 LUT 及 Flip-flop 數目如表 5 所示。由於電路使用資源，在不同組態之合成器和不同電路之整合上，會有不同的使用量，合成器會根據 FPGA 上整體電路狀況做優化，在 data path 和 resource 使用量之間做權衡。雖然兩個字串加速器皆有迴圈控制之電路邏輯和 reader，但其條件皆不相同，例如迴圈 branch 條件，因此要達到邏輯共用，並不是很直觀的事，必須更深入以 gate-level 分析。然而，部分儲存元件例如 string buffer，若外部使用額外的多工器，是可以達到共用的目的，但是邏輯的增加所產生之 data path 及 LUTs 增加問題，未來得做更深入的研究。影響 FPGA 上硬體資源使用量之因素有很多，本小節僅列出兩個字串加速器在 JAIP 上所增加之 LUTs 及 flip-flops 做為移植其他系統之資源使用量參考，包含字串加速器單獨整合 JAIP 所增加之資源數、兩個字串加速器一起整合 JAIP 之增加資源數

和 JAIP-core 不含字串加速器之資源數。兩個加速器內部皆有使用少量的 Distributed RAM 作為 string buffer，除此之外沒有使用任何其它大型的儲存體，例如 BlockRAM。雖然使用 CAM 的 string matching 做法在尚未考慮字串物件之解析及讀取字元的 logic 條件之下，shift register 及 address encoder 只消耗數百個 LUT[33]，但其使用之 RAM 大小相當龐大，以 1024 個 UTF-16 之 character 為例，在 Virtex-5 之 FPGA device 上就需要 128 個大小為 2kB 之 BlockRAM[34, 35]，而且使用上還有字串大小之限制。我們的做法最多可以支援到 2^{25} 個字元，未來若要擴充也只需擴充其位址線寬度。

表 5 字串加速器在 Virtex-5 FPGA device 上所使用之邏輯元件

	arraycopy	indexOf	arraycopy & indexOf	JAIP
Number of LUTs	1082	1735	2926	10874
Number of Flip-Flops	474	576	1109	4786
Number of 2k BlockRAM	0	0	0	31
Maximum frequency	83.3MHz			

4.2 字串操作平均長度分析

本論文提出之字串加速硬體得需要少許 on-chip 儲存空間作為暫存用途。當加速器使用 burst mode 與 cahce 或其他記憶體架構存取時，此暫存空間或 cahce

block 之大小會影響加速器之效能。理想上，一個 cache block 或加速器暫存空間可以存放之字元數越多越好。在我們的實作中，採用了一個 2-way set-associative cache，為了節省硬體資源，加速電路之暫存空間僅使用最小可以處理 un-aligned 之大小。因此，我們僅需要考量 cache block 大小與字串處理之關聯性。

為了分析 arraycopy 與 indexOf 兩個字串操作平均字串長度，我們使用 NanoXML 1.6.4 做為測試程式[37]，NanoXML 是一個以 Java 實作並遵守 CLDC 規範的 XML parser。測試的 xml 檔案包括：1.從一個 docx 檔案解壓縮所得到之數個 xml 檔。2.Xilinx Platform Studio Project 合成報告中的 xml 檔。3.Xilinx coregen 工具內的 xml 檔。表 6 列出測試結果，表中包括平均 arraycopy 長度、平均 indexOf 之文檔長度、平均 indexOf 之 pattern 長度以及檔案大小。表中最後一列是所有檔案之平均值。在這些檔案中，arraycopy 平均長度是 11 個字元，indexOf 是 14 個字元。由於 Java 字串編碼是使用 UTF-16，因此我們認為大小為 32-bytes 之 cache block 在大部分情況可以涵蓋整個字串。

表 6 XML parser 之字串操作平均長度分析

XML file name	avg. copy length	avg. matching text length	avg. matching pattern length	file size
[Content_Types].xml	16	26	3	3kB
app.xml	7	11	3	1kB
core.xml	11	14	2	1kB
endnotes.xml	12	15	2	2kB
footer1.xml	12	12	2	2kB
footer2.xml	11	11	2	3kB
footnotes.xml	12	14	2	2kB
header.xml	15	19	2	2kB
item1.xml	7	6	1	4kB
itemProps1.xml	21	21	2	1kB
settings.xml	21	23	2	118kB
theme1.xml	5	6	2	7kB
webSettings.xml	4	5	2	6kB
system_usage.xml	4	6	2	200kB
xil_index.xml	8	10	2	1,099kB
avg. of above files	10.8	13.2	2	-

4.3 JAIP 性能實驗

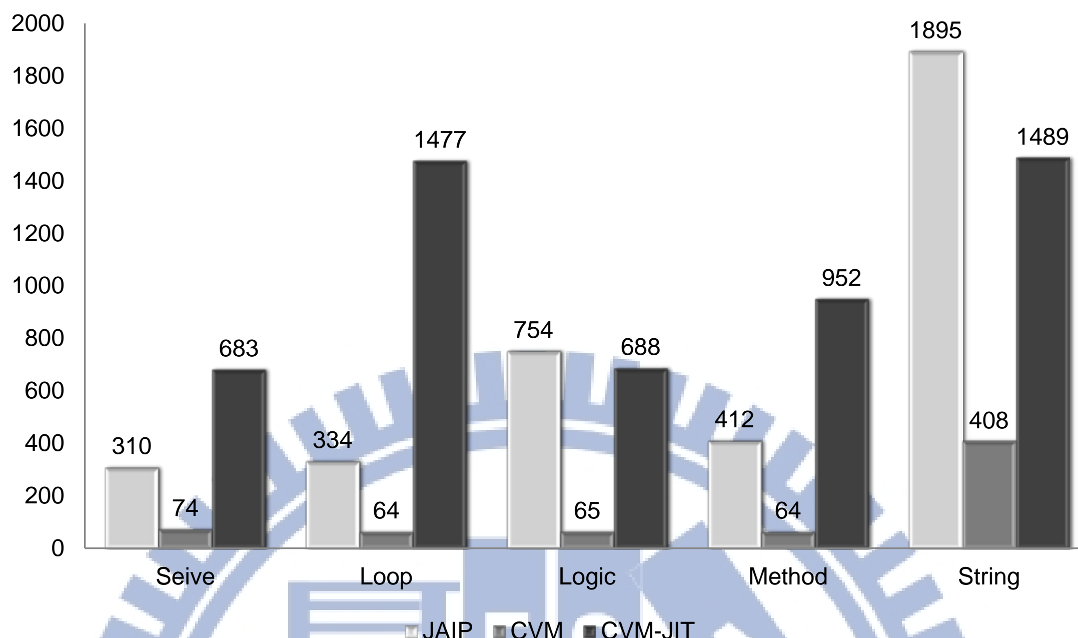


圖 25 JAIP 與 CVM 執行 Embedded CaffeineMark 之分數比較

為了比較 JAIP 之整體性能比較，我們使用 Embedded CaffeineMark(ECM)之 benchmark suit 來進行測試，並以 Sun 公司的 CVM 來進行比較。CVM 是一個支援 JIT 且被設計使用在 CDC configuration 裝置上的一個 Java virtual machine。我們使用 Xilinx ML-403 做為 CVM 測試平台。平台上包含一個 Virtex-4 FPGA 和一顆 PowerPC 405 處理器。時脈頻率與 JAIP 測試環境同樣設置為 83.3MHz。

圖 25 為 ECM 內部 5 個 benchmark 之分數比較圖。在 LogicAtom 及 StringAtom 操作皆有不錯的表現。這些 benchmark 皆是測試 Java 程式在長時間重複動作操作下，系統效能的表現力。而 SeiveAtom、LoopAtom 及 MethodAtom 則由於缺乏 bytecode level 之優化，使得一些 field 操作或 invocation 重複執行；相比之下，JIT 可以將讀取過的 field 直接暫存在處理器的 register 上，以供下次使用。Method 方面，JIT 可以 inline 一個經常使用的小型 function，甚至 unrolling 一個遞迴結構。迴圈方面，JAIP 仍缺乏有效的 branch 指令之優化，例如 branch prediction、target prefetching 等等。

無論如何，本篇論文的研究重點—字串處理，在這個實驗中與 CVM-JIT 有著 1.27 倍的優勢。這是由於 String Accelerator 可以節省字串操作之迴圈數目，並且節省指令之 translate，fetch，decode，甚至是部分指令之 execute 時間。以 arraycopy 加速器來說，reader 讀取 32-bit 資料後，writer 將此 32-bit 資料寫出，而在 Java language 中，32-bit 資料可以存放兩個字元。也就是說，加速器一次迴圈所能完成之工作量，而以 Java bytecode 實作 method 方式則必須要以兩次迴圈才能做完相同份量工作，並且每個 iteration 有將近 23 個 bytecode 需要在 BEE 上運行。indexOf 方面，每次外部迴圈同樣皆可進行 3 個字元之比較與 2 個字元之 shift，內部迴圈則是進行 2 個字元之比較與 2 個字元之 shift；而 Java bytecode 方式實作 method，內部與外部迴圈之 bytecode 共約 83 個 bytecode 需要被運行，且內外迴圈次數皆是加速器的 2 倍。然而，需要更多記憶體空間與更多時間成本做優化的 JIT，相比之下我們的實作慢了 2.1 倍。

為了瞭解字串加速器及 heap cache 對字串處理的影響，我們又單獨使用 ECM 裡面 StringAtom 這個 benchmark 來測試未加字串加速器及 heap cache 的 JAIP，與使用 heap cache 及字串加速器的 JAIP 進行比較。Heap cache 是針對 JVM 的 heap 空間內之資料進行快取，其設計是採用資料總大小為 2kB 之 2-way set-associative cache。如表 7 所示，使用 heap cache 可以增進 StringAtom 效能 2.06 倍。使用 heap cache 及兩個字串加速器後，JAIP 效能更提升 4.85 倍。這樣的效能提升歸功於 heap access 之優化和兩個字串加速器。我們除了能夠更有效讀取和創建字串內容外，字串加速器也可以增加字串相接、字串創建或字串搜尋等操作。而字串搬移和 character 之比較相較於使用 Java bytecode，每次操作之位元數皆高出兩倍，而迴圈操控之 bytecode 執行時間也可以節省。

表 7 JAIP 與 CVM 執行 ECM StringAtom benchmark 之分數比較

Platform	StringAtom score
JAIP	390
JAIP w/ Heap Cache	807
JAIP w/ String Accelerator & Heap Cache	1895
CVM	408
CVM-JIT	1489

4.3.1 StringAtom benchmark 之 method profile 及 bytecode profile

表 8 無字串加速器之 JAIP 執行 2000 次 StringAtom 的 execute()後，蒐集到之 method profile。前 5 欄之單位為 million cycles

	HW	Intrpt	DR	Heap	Total	# of calls
execute	345	8	71	160	584	2000
append	163	2	37	121	323	304000
getChars	69	2	11	39	121	304000
expandCapacity	69	0	0.5	67	137	12000
arraycopy	123	0	0.3	103	226	316000
indexOf	157	0	5	21	183	100000

表 9 含字串加速器之 JAIP 執行 2000 次 StringAtom 的 execute()後，蒐集到之 method profile。前 5 欄之單位為 million cycles

	HW	Intrpt	DR	Heap	Total	# of calls
execute	76	8	65	100	249	2000
append	44	2	37	76	159	304000
getChars	15	2	13	23	51	304000
expandCapacity	3	0	11	38	41	12000
arraycopy	6	0	0.5	58	64	316000
indexOf	9	0	0	7	16	100000

表 8 與表 9 是 ECM 的 StringAtom benchmark 在分別沒有使用字串加速器的 JAIP 與使用字串加速器的 JAIP 上執行，並開啟 method profiler 獲取的 profile。雖然我們的 profiler 能夠蒐集的 method 數目能夠達到 256 個，但礙於篇幅關係，這裡僅將最重要的 method 列出。表中的 HW 是指 method 使用 BEE 及硬體加速器花費多少 million clock cycles；Intrpt 是指 method 透過中斷服務與 RISC-core 溝通之花費 million cycles 數；DR 是指使用 DSRU 進營符號解析之花費 million cycles 數；Heap 是指 heap access 之花費 million cycles 數；total 則是前四項總共累加之數目，也就是 method 執行總 million cycles 數目。execute 是 StringAtom 主要執行之 method，其 profile 幾乎可以代表整個 benchmark 之時間分布；append 為字串相接操作，其 method 裡面主要包含 getChar 和 expandCapacity；getChar 為 character array 之搬移操作，expandCapacity 則是 StrignBuffer 容量之擴充，兩者都會用到 arraycopy；arraycopy 與 indexOf 則直接使用硬體加速器。

根據表 8 的 Profile 來推算，可得知 indexOf 的使用時間占用總時間 31.3%，再加上 indexOf 是很常用的字串操作，因此將它用硬體實作是很值得的。表 8 及

9 可以看出 arraycopy 整體時間從 226 million cycles 降到 64 million cycles；indexOf 從 183 million cyles 降到 16 million cycles;整體時間也從 584 million cycles 降到 249 million cycles。從表 9 的 execute 的 profile 來看，在使用加速器後的 JAIP，動態解析的時間佔了整體的 26.4%，對整理的性能危害甚大。從 expandCapacity 的 profile 來看，使用時間就佔了 append method 的 27%，且它的 heap access 佔了大多數，並且大於 getChars，但使用次數僅為 0.039 倍。從這點可以看出 StringBuffer 之大小，或者是 expandCapacity 之機制影響大量字串 append 之效能甚大，因此值得我們做更深入的研究。

表 10 含字串加速器之 JAIP 執行 StringAtom 後所蒐集到之 bytecode profile

	HW	Intrpt	DR	Heap	Total	#of used
caload	0	0	0	0	0	0
goto	1500	0	0	0	1657	375006
getfield	15803	0	27656	10194	53653	3950944
putfield	5478	0	5478	20952	31908	913096
invokevirtual	7055	0	38193	6963	52211	1763992
invokestatic	1574	0	5903	0	7477	393726
ireturn	4562	0	0	0	4562	507552
areturn	5642	0	0	0	5642	626976
return	6963	0	0	0	6963	773768

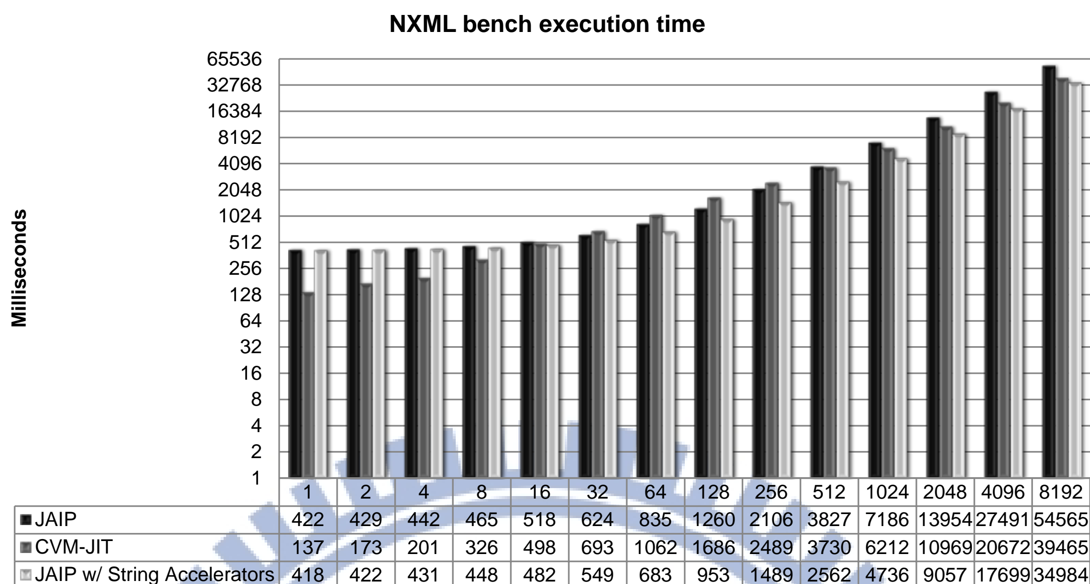
bytecode profiler 之 profile table 擁有 256 個 entry，也就是說 bytecode profiler 可以蒐集所有 bytecode 之 profile。這邊僅列出與字串操作相關或者耗用 cycles 數較有影響力之 bytecode profile。表 10 是幾個重要的 byecode profile。Total 總和即為 HW，Intrpt，DR 及 Heap 之總和，也就是每個 bytecode 之執行總 cycles 數，其單位皆是 kilo cycles。# of used 是指這個 bytecode 總共執行幾次。caload 是 character array read 一個 element 之指令。goto 是無條件 branch 指令。getfield 及 putfield 皆

是 field 存取指令。invokevirtual 及 invokestatic 皆是 method invocation 指令。ireturn，areturn 及 return 皆是 invocation 之返回指令。

從 caload 及 goto 可以看出字串加速器之操作已經大大地取代字串陣列操作指令與迴圈指令。相較其他類型指令，field 操作，invocation 及 return 類型指令幾乎是 StringAtom 在我們 platform 上，時間佔最多的 bytecode。光是從表 10 裡的三種類型所使用之總 cycles 數累加就有 162 million cycles，程式總執行 cycles 數約 306 million cycles，而扣掉 arraycopy 與 indexOf 兩個使用字串加速器 method 之執行時間為 203 million cycles。也就是說，整個程式之時間除了字串加速器使用時間之外，有 79.8%的時間都花費在這三種類型指令上，特別是前兩種需要符號解析的 bytecode。

4.4 使用 XML Parser 作為 String 操作性能之測試

為了能夠更了解 JAIP 與字串加速器在更實際的例子下效能的增進之幅度，本小節是介紹一個自行設計的以 XML parser 為基礎的 benchmark，並展現其執行之效能結果。我們採用 NanoXML 1.6.4 作為測試程式。NanoXML 是將 XML 內部之 tag、atribute 及 content 等內容以 Java 的 hash table 或 vector 等結構儲存。測試的 XML file 大小為 1.2kB，共有 34 行，字元數 1233 個。



	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192
JAIP	422	429	442	465	518	624	835	1260	2106	3827	7186	13954	27491	54565
CVM-JIT	137	173	201	326	498	693	1062	1686	2489	3730	6212	10969	20672	39465
JAIP w/ String Accelerators	418	422	431	448	482	549	683	953	1489	2562	4736	9057	17699	34984

圖 26 NXML bench 執行時間測量。橫軸為執行次數。縱軸為執行時間(以 log-scale 展示)，單位為毫秒

如圖 26 所示，我們比較各種不同執行次數之執行時間比較。其橫軸為執行次數，縱軸為執行時間，單位為毫秒，橫軸與縱軸皆以 2 的冪次方成長。我們選擇 8192 次為最大上限，這是因為 CVM-JIT 在 8192 次時已趨近穩定，平均每個 iteration 消耗 4.8 毫秒。而 JAIP 除了 startup delay 較高之外，每次增加一個 iteration 時間只增加 4.2 毫秒。從這個圖中可以看出兩件事情。第一，JAIP startup delay 較 CVM 高出約 3 倍，這是由於 JAIP 在 class parsing 上消耗的時間比較 CVM 長。第二件事情便是 JAIP 性能的優勢，JAIP 在解析時間較長時，甚至等到 CVM-JIT 的優化在趨近穩定時，執行時間仍然比 JAIP 搭配字串加速器多出 12%。雖然這個實驗與我們原先預期的結果是相反的，我們預期 CVM-JIT 的 startup delay 會比較長，長時間執行下會勝過 JAIP。但根據本次實驗，結果卻是完全相反。無論如何，字串加速器的優勢仍然很明顯，從圖 26 可以看出，在 JAIP 使用字串加速器後效能可以增進約 55%；而與穩定後的 CVM-JIT 相比，仍然有 12%的優勢。

第五章 結論

本論文為了找出 Java 程式中，字串處理時之效能瓶頸之處，我們提出了 method profiler 及 bytecode profiler。這兩個剖析器可以在不改變程式執行行為與執行時間之下，取得 method 之 invocation 與 bytecode 執行次數與執行時間資訊。為了增加 heap 的使用效率，我們設計並且實作 on-chip 的 Java heap management unit 來增進每次 heap 空間之分配時間，並搭配 heap cache 減少記憶體存取時間。為了可以讓 JAIP 可以對硬體加速電路進行操控，我們更設計了 Hardware Native Interface，藉著 native method 之宣告及 cross reference table 等修改，可以讓特定 Java method 以電路邏輯實作。字串操作加速方面，我們找到兩個基本且常用的操作，分別是 arraycopy 與 indexOf，並分別實作加速電路。根據 benchmark 及實際例子測試，硬體加速器的效果十分顯著。

利用本論文設計的 method profiler 及 bytecode profiler，未來 JAIP 可以尋找各種效能的瓶頸。除了可以得知每個 method 和 bytecode 的使用狀況外，更可以從 4 個計數器得知 BEE 執行時間、動態解析時間、中段時間及 heap 存取時間。以 StringAtom 程式來說，從 method profile 觀察頂層 method，動態解析時間佔了約 26.1%整體時間。從 bytecode profile 來看，field 操作、invocation 及 return bytecode 占了 79.8%的整體時間，因此，接下來的優化方向可以從符號解析之流程下手，例如在迴圈內部執行一個 getfield 動作，每次皆重複進行一樣的符號解析流程，若是可以直接將 field 之值暫存在 local variable，可以節省不少重複且冗長的動作；同樣地，invocation 過程中解析的資訊，也可以暫存在特定記憶元件，在尚未執行 return 或下一個 invocation 之前，每次執行同一個 invocation 可以使用自定義的快速版 invocation 之 bytecode。

本篇論文提出的 Hardware Native Interface，可以讓 Java 程式之參數值透過 native method 之呼叫傳送到 hardware device 上，更可以將 hardware device 之計算

結果傳遞到 Java stack 上。雖然本篇論文之字串加速器為了效能考量裝置在 JAIP 之 IP 上，而不是掛載到 bus 上並使用 I/O port 之直接存取，但有了這個介面，memory mapped I/O 之實作已不是什麼難題。例如我們可以將欲操作的記憶體位址、存取請求種類和模式等等 bus 信號，透過 Hardware Native Interface 傳遞到 JAIP 的 external memory access logic，並將此邏輯電路以 Java native method 封裝，如此一來，Java programmer 便可透過 method invocation 使用 memory mapped I/O 之語言特性。所有 I/O device 存取皆可以在不用透過 RISC-core 的 co-processing 條件下，JAIP 皆可以使用 Java 程式碼，搭配 memory mapped I/O 特性直接存取任何硬體 device。除此之外， JAIP 內部邏輯同樣也可以透過此方法來控制。例如我們可以使用 native method 來控制 thread 的 schedule 策略。

附錄 一： Method Profiler 之設計

Method Profiler 主要由兩個 table 組成，分別為 Profile Stack 和 Profile Table。Profile Stack 為一個堆疊結構，堆疊每一個條目為 160bit，分別存放 32bit 的計時器時間 4 項，以及一個 32bit 的方法 ID。Profile Table 也是每一條目皆為 160bit 的表格，存放了每一個方法的 4 種計時器之時間累積，剩餘的 32bit 則存放此方法被呼叫次數。Profile Table 的每一條目分別代表每一個方法的 Profile。每當方法被呼叫的時候將其被呼叫方法 ID 和 4 個計時器之時間分別推進堆疊頂端。當方法返回時，則拋出堆疊頂端之條目，並以目前 4 個計時器時間減去條目中 4 個計時器時間算出時間差，再將時間差根據條目中的方法 ID 存放至對應的 Profile Table 條目，最後將此方法被呼叫次數加上 1。

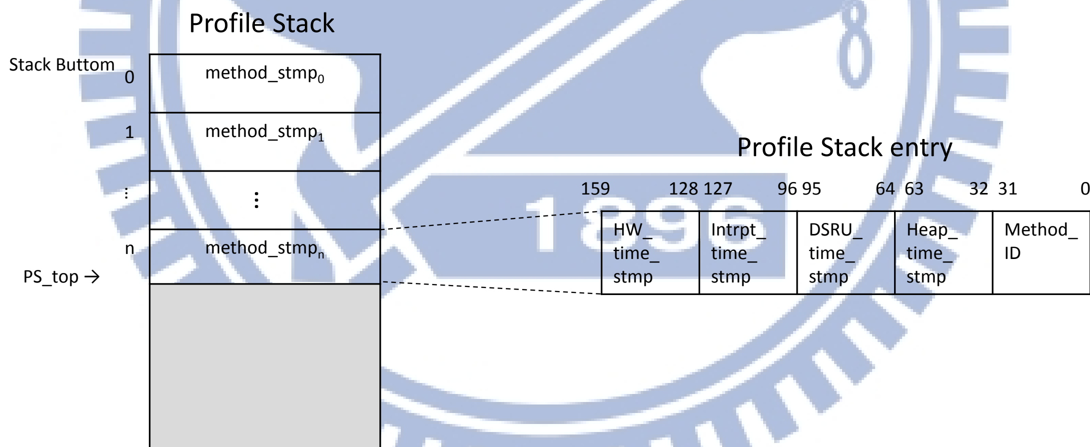


圖 27 Profile Stack

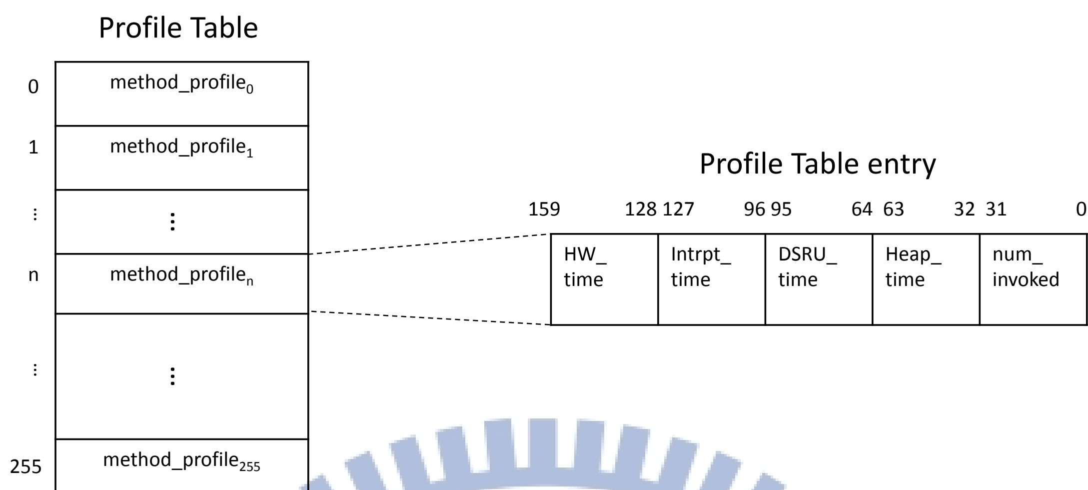


圖 28 Method Profiler 內之 Profile Table

Method Profiler 架構如圖 27 所示。內部電路主要包含由 BRAM 實作的 Profile Stack 及 Profile Table 和一個控制電路的 FSM。Input 信號有四組 counters 值、invoke_flag、return_flag 及 invoke_method_ID。invoke_flag 若升起，直接將 4 個 counters 值和 method_ID push 進 Profile Stack 上。當 return_flag 升起，counters 值會被暫存到 return_time_stmp 之 flip-flops 中，代表執行 return 動作當下的時間戳。除此之外，MP_FSM 會被啟動。MP_FSM 一共有四個狀態，分別是 normal、load_time_stmp、load_time_accm 和 update_prof。load_time_stmp 狀態是將 Profile Stack 頂端之資訊讀出，其資訊除了 invoke 此 method 時之時間戳外，還包含了 method ID，此 ID 為當初因為 invoke_flag 升起所以被 push 進去的 method_ID，那麼當 return_flag 升起時，pop 出的 method_ID 即代表目前執行 return bytecode 之 method_ID。load_time_accm 是根據剛剛 pop 出來的 method ID 讀取 Profile Table 上對應的 method profile，也就是目前此 method 累計的四個 counters 計數值，以及 method 呼叫次數。最後的狀態 update_prof 是根據上一個狀態讀出的累計 counters 計數值，加上此次 method 執行時間和執行次數。而此次 method 執行時間很容易由 return_time_stmp flip-flop 上的時間戳和 profile stack pop 出的時間戳相減求得。

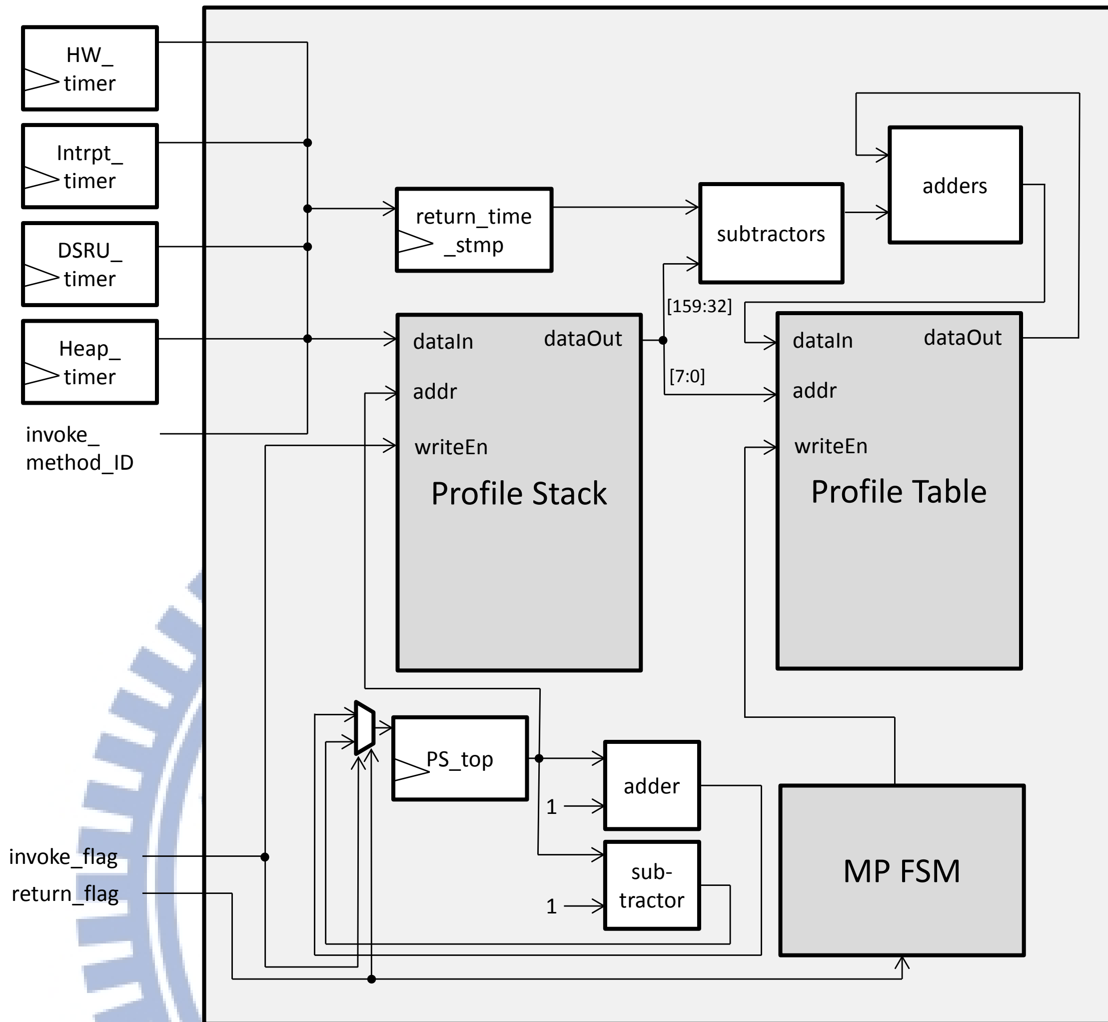


圖 29 Method Profiler

附錄 二： Bytecode Profiler 之設計

為了要量測每個 bytecode 在 decode stage 之後執行之 cycles 數目，除了 hardware counters 之外，bytecode profiler 需要兩項資訊：目前正要執行的 bytecode 及一個 issued 信號。Bytecode profiler 透過 issued 信號每次拉起的時間間隔，來計算每個 bytecode 執行時間，每次 issued 拉起，都同時代表目前 bytecode 之執行開始與上一個 bytecode 之執行結束。由於 decode stage 僅有已轉換好的 j-code，為了要得到 j-code 對應的 bytecode，我們在 pipeline 上建立另外的信號線及正反器，來傳遞 bytecode 到 decode stage。BEE 之 fetch stage 會將每個 bytecode 分類成為 simple 與 complex 兩種類型。Simple 類型之 bytecode 會被轉換成為一個 j-code；complex 類型會被轉換成為一串 j-code sequence，這些 j-code 由 fetch stage 根據 bytecode 查找後送出到 decode stage。Issued 信號是由 fetch stage 拉起。在遇到 simple bytecode 時，直接拉起；而 complex bytecode 時，僅在 j-code sequence 中的第一個 j-code 送出時拉起。Fetch stage 將 issued 信號傳遞到 decode stage，再從 decode stage 送到 bytecode profiler。

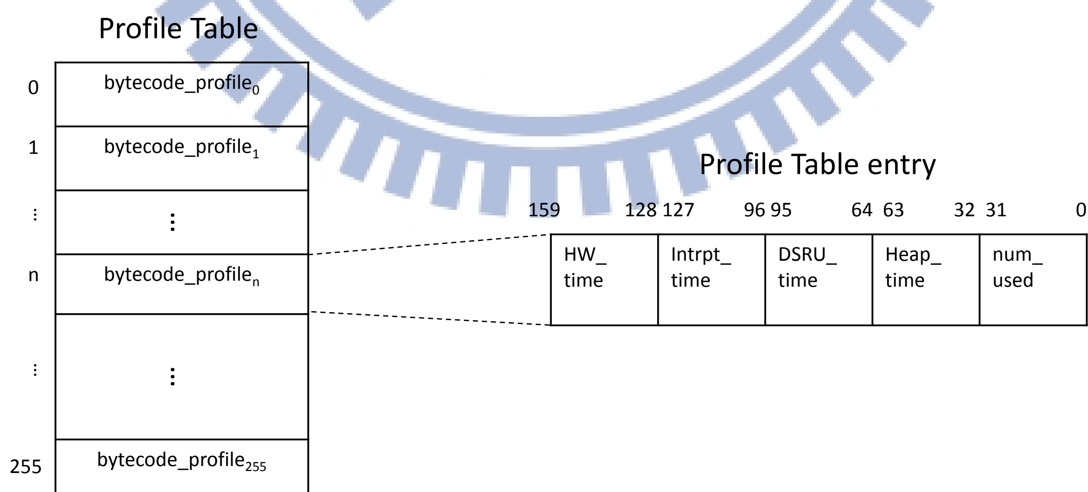


圖 30 Profile Table in Bytecode Profiler

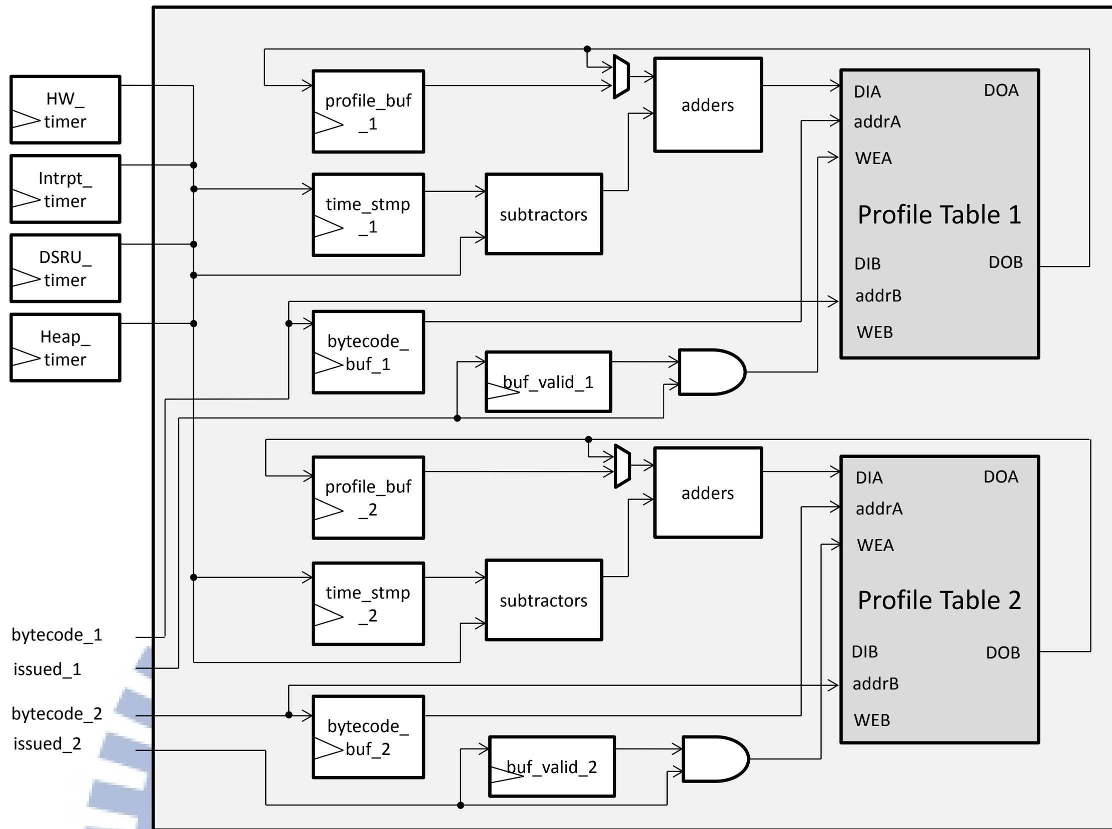


圖 31 Bytecode Profiler

Bytecode profiler 設計如圖 31 所示。與 method profiler 一樣，透過四個 counter 紀錄 HW、Intrpt、DSRU 及 Heap 時間。圖中 bytecode_1、issued_1、bytecode_2 及 issued_2 是由修改後 BEE 之 decode stage 傳送，issued_1 代表一道 bytecode 剛開始執行，其 bytecode 會由 bytecode_1 同時傳入；若兩個 simple bytecode 被 double-issued，其第二個 issued 資訊和 bytecode 會由 bytecode_2 及 issued_2 傳入。由於完整 bytecode 執行時間必須要等 issued 拉起兩次才會得知，每次 issued 都先記錄目前時間戳，等到 issued 再度被拉起時，再將其時間間隔累加到 bytecode profile 中。issued 信號拉起時主要有四件事：1.透過 bytecode 將 profile table 中對應的 profile 讀出，並存到 profile_buf_1 或 profile_buf_2。2.將目前 counters 值放入 time_stmp_1 或 time_stmp_2。3.將目前正執行的 bytecode 放入 bytecode_buf_1 或 bytecode_buf_2，並將 bytecode_buf_valid flag 設定為 1。

參考文獻

[1] D. E. Knuth, J. Morris, James H, and V. R. Pratt, "Fast pattern matching in strings," *SIAM journal on computing,* vol. 6, pp. 323-350, 1977.

[2] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM,* vol. 20, pp. 762-772, 1977.

[3] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of the ACM (JACM),* vol. 21, pp. 168-173, 1974.

[4] M. Aldwairi, T. Conte, and P. Franzon, "Configurable string matching hardware for speeding up intrusion detection," *ACM SIGARCH Computer Architecture News,* vol. 33, pp. 99-107, 2005.

[5] Y. H. Cho and W. H. Mangione-Smith, "A pattern matching coprocessor for network security," in *Proceedings of the 42nd annual Design Automation Conference*, 2005, pp. 234-239.

[6] C. R. Clark and D. E. Schimmel, "Scalable pattern matching for high speed networks," in *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, 2004, pp. 249-257.

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

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

[9] H.-W. Kuo, Z.-G. Lin, Z.-J. Guo, and C.-J. Tsai, "Double-Issue Java Accelerator IP for Embedded SoC," in *Proc. of VLSI Design/CAD*, Keng-Ting, Taiwan, 2011.

[10] D. S. Hardin, "Real-time objects on the bare metal: an efficient hardware realization of the Java< sup> TM</sup> Virtual Machine," in *Object-Oriented Real-Time Distributed Computing, 2001. ISORC-2001. Proceedings. Fourth IEEE International Symposium on*, 2001, pp. 53-59.

[11] U. Brinkschulte, C. Krakowski, J. Kreuzinger, and T. Ungerer, "A multithreaded java microcontroller for thread-oriented real-time event-handling," in *Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on*, 1999, pp. 34-39.

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

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

[14] H.-W. Kuo, "Design of Java Accelerator IP for Embedded Systems," Master, Computer Science, NCTU, 2011.

[15] C.-H. A. Hsieh, J. C. Gyllenhaal, and W.-m. W. Hwu, "Java bytecode to native code translation: The Caffeine prototype and preliminary results," in *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, 1996, pp. 90-99.

[16] S. Meloan, "The Java HotSpot performance engine: An in-depth look," *Sun Microsystems, Jun,* 1999.

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

[18] 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.

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

[20] 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.

[21] M. Schoeberl, S. Korsholm, T. Kalibera, and A. P. Ravn, "A hardware abstraction layer in Java," *ACM Transactions on Embedded Computing Systems (TECS),* vol. 10, p. 42, 2011.

[22] J. Whitham, N. Audsley, and M. Schoeberl, "Using hardware methods to improve time-predictable performance in real-time Java systems," in *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, 2009, pp. 130-139.

[23] A. Borg, R. Gao, and N. Audsley, "A co-design strategy for embedded Java applications based on a hardware interface with invocation semantics," in *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, 2006, pp. 58-67.

[24] Y. Ha, R. Hipik, S. Vernalde, D. Verkest, M. Engels, R. Lauwereins*, et al.*, "Adding hardware support to the hotspot virtual machine for domain specific applications," in *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, ed: Springer, 2002, pp. 1135-1138.

[25] S. Guccione, D. Levi, and P. Sundararajan, "JBits: A Java-based interface for reconfigurable computing," in *2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, 1999.

[26] J. Fleischmann and K. Buchenrieder, "Prototyping networked embedded systems," *Computer,* vol. 32, pp. 116-119, 1999.

[27] 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 *Proc. Of VLSI Design/CAD*,

Keng-Ting, Taiwan, 2011.

[28] E. Lattanzi, A. Gayasen, M. Kandemir, V. Narayanan, L. Benini, and A. Bogliolo, "Improving Java performance using dynamic method migration on FPGAs," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 2004, p. 134.

[29] S. Liang, *The Java TM Native Interface: Programmer's Guide and Specification*: Addison-Wesley Professional, 1999.

[30] T. Fast and T. Wall. (October). *Java Native Access*. Available: https://github.com/twall/jna

[31] Y. Utan, S. i. Wakabayashi, and S. Nagayama, "An FPGA-based text search engine for approximate regular expression matching," in *Field-Programmable Technology (FPT), 2010 International Conference on*, 2010, pp. 184-191.

[32] H.-C. Lee and F. Ercal, "RMESH algorithms for parallel string matching," in *Parallel Architectures, Algorithms, and Networks, 1997.(I-SPAN'97) Proceedings., Third International Symposium on*, 1997, pp. 223-226.

[33] L. Duc-Hung, K. Inoue, S. Masahiro, and P. Cong-Kha, "An FPGA-Based Information Detection Hardware System Employing Multi-Match Content Addressable Memory," *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences,* vol. 95, pp. 1708-1717, 2012.

[34] J.-L. Brelet, "An overview of multiple cam designs in virtex family devices," *Xilinx Inc., Application Notes,* vol. 201, pp. 4-5, 1999.

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

[36] Z.-J. Kuo, "Design of Dual-Core Java Application Processor for Embedded Systems," Master, Computer Science, NCTU, 2012.

[37] M. D. Scheemaecker and E. Giguere. NanoXML 1.6.4 for the KVM/CLDC [Online]. Available: http://nanoxml.sourceforge.net/orig/kvm.html