國立交通大學

資訊科學與工程研究所

碩 士 論 文

重新排列映像檔物件以增進虛擬機器間程式分頁共享可能性

Make Code Page Sharing Possible Between Virtual Machines - Reorder Objects by Binary Rewriting

研 究 生：林志峰

指導教授：張瑞川　教授

中 華 民 國　九 十 五　年 六 月

重新排列映像檔物件以增進虛擬機器間程式分頁共享可能性
Make Code Page Sharing Possible Between Virtual Machines - Reorder Objects by Binary Rewriting

研 究 生：林志峰　　Student：Zhi-Feng Lin

指導教授：張瑞川　　Advisor：Ruei-Chuan Chang

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

June 2006

Hsinchu, Taiwan, Republic of China

中華民國九十五年六月

重新排列映像檔物件以增進虛擬機器間程式分頁共享可能性

學生：林志峰　　指導教授：張瑞川　教授

國立交通大學資訊科學與工程研究所

論文摘要

利用虛擬機器的技術來達成伺服器整合是未來的趨勢。在伺服器整合的環境中，我們可以利用記憶體分頁共享來降低 L2 快取失誤率及節省記憶體用量。但是因爲程式映像檔之間的差異，使得記憶體分頁共享不易達成。本論文提出一套方法來重新調整映像檔內物件，縮小映像檔間差異，讓共享容易達成。

我們的實驗評估映像檔之間共享的部分。實驗顯示映像檔平均有 200 Kbytes 的物件可以共享，這些物件平均佔整個映像檔 17%。

Make Code Page Sharing Possible Between Virtual Machines - Reorder Objects by Binary Rewriting

Student: Zhi-Feng Lin Advisor: Dr. Ruei-Chuan Chang

Computer Science and Engineering College of Computer Science National Chiao Tung University

Abstract

The server consolidation by using virtual machine technology is the trend. In the server consolidation environment, we may reduce the L2 cache miss rate and memory usage by sharing program pages. To share program pages between program images is not easy because difference of program images. We provide a method to reorder objects of the program image and try to reduce difference between program images.

Our evaluations show that there is about 200 kbytes sharable objects between program images.

誌謝

首先感謝我最尊敬的指導老師　張瑞川教授。這兩年在老師費心的教導下，學生方能順利完成這篇論文。於授業期間，老師耐心的指導我正確的研究態度與方法，讓我受益良多。在撰寫論文期間，感謝大緯學長給予的建議及不斷的與我討論。也感謝在實驗室一起努力的同學，學長以及學弟的支持和勉勵。

最後將此篇論文獻給我最親愛的父母，感謝您在我求學期間全心權益的支持，讓我得以專心地完成研究。

目錄

圖目錄

表目錄

一、緒論

虛擬機器是在早期就有的一種技術。利用虛擬機器，使用者可以在同一台機器上模擬出多套的虛擬硬體，並且在每套虛擬硬體上執行各自的工作而不會互相干擾。然而早期受限於硬體，虛擬機器並沒有廣泛的被使用。現今因爲硬體的快速發展，爲了能充分利用硬體資源，虛擬機器這項技術又再度被重視。將多項服務整合到一部強大的硬體，不僅能節省硬體成本，更能充分發揮硬體效能，減少資源浪費。

所謂的伺服器整合，就是將在個別機器上的工作，整合到一部強大的機器上。每個網路服務都在一個虛擬機器上執行，因此不會互相干擾。而且每個網路服務都能充分使用硬體資源，硬體資源不會被閒置。而且硬體數量的減少，不僅方便管理也能減少因硬體故障所造成的服務中斷。

利用虛擬機器來進行伺服器整合，不僅能節省硬體資源，而且還能得到其它的好處。其中一項就是能分享虛擬機器之間的記憶體分頁。分享記憶體分頁不但可以降低記憶體的使用量，而且減低L2快取失誤的機率。L2快取失誤機率的降低意味著系統執行效能的增加。

雖然分享記憶體分頁可以得到不少好處，但是在某些情況下反而會降低系統效能。一個被分享的資料分頁一旦被修改，虛擬機器就必須對此資料分頁進行複製，並修改分頁表。如果這樣的情形常常發生，反而會降低系統執行效能。因此分享不會被修改的程式碼分頁是比較好的選擇，不但可以獲得分享記憶體分頁的好處，而且也可以避免上述的情況發生。

在一般伺服器整合的環境下，要讓程式碼分頁進行分享是不太容易的。因爲只要程式碼一經修改，則編譯出來的映像檔就可能會產生極大的變化。不只修改程式碼會對映像檔造成影響，編譯器也會令映像檔產生極大的變化。然而在伺服器整合的環境下，要求使用者使用相同版本的作業系統及應用程式是不合理且不實際的。在這個前提之下，要讓程式碼分頁能夠被分享是非常困難的。

本論文提出一個方法來減輕原始碼改變及編譯器差異所造成的影響。藉由重新排列映像檔內物件順序來修正映像檔，讓映像檔和映像檔之間的相似度增加。經過此一修正後，即使是不同版本的程式也能有較多的分頁進行共享。這可以讓使用者

不需更換軟體及環境也能享受到共享程式碼分頁的好處。

本論文的第二章爲介紹相關研究，第三章將講解整個系統的架構。在第四章則是設計及實作。第五章會說明實驗的結果。第六章是結論的部份。

二、相關研究

2.1 Denali

Denali**[1]**是一個para-virtualize的虛擬機器。它的目標是要提供一個平台讓不安全的網路服務在機器上執行，並且做到完整的隔離，因此惡意的程式無法攻擊或干擾其它的網路服務。另外一個目標就是希望這個平台的規模能夠容易的延伸，所以在設計上不會去模擬出虛擬機器所有的介面，以節省資源。

Denali的設計者希望它的規模可以延伸到10000部虛擬機器，因此也考慮到如何減少記憶體的浪費。其中一個很好的著眼點就是虛擬機器上所執行的作業系統，故Denali上所有的虛擬機器都分享同一份作業系統。

雖然利用上述的方法可以減少記憶體的使用，但是卻有一個限制，那就是所有的虛擬機器必須使用相同的作業系統。一旦使用者用了不同的作業系統，就無法達成作業系統的分享。

2.2 VMware ESX Server

VMware ESX server **[2]**是一個fully-virtualized的虛擬機器，它完整地模擬出虛擬機器所需的介面。所以在虛擬機器上所執行的程式完全不用修改就可以執行。使用者在進行伺服器整合時能有充分的的彈性，並且不會被限制。VMware ESX server不只提供了相當好的隔離性，而且也讓管理者可以管理硬體資源的分配，容易進行商業上的應用。

在記憶體資源的應用上，ESX server也設計了一套節省記憶體的方法。當系統運作時，ESX server會定期掃描系統內的記憶體分頁。一旦發現有記憶體分頁是相同時，就會讓這些分頁進行共享。這個方法實作上容易，而且可以確實分享所有相同的的記憶體分頁，不管虛擬機器上的配置爲何。

上述的共享機制因爲不需要虛擬機器提供任何資訊，所以也造成了一些問題。當共享的記憶體分頁要被修改時，虛擬機器就必須對此分頁進行複製，再修改分頁表。如果這種情形太常發生的話，則會嚴重影響系統效能。另外的一個問題是系統必須定期掃描記憶體分頁，所以會造成部分的負荷。

三、系統架構

要在虛擬機器之間共享不同版本的映像檔，必須要消除原始碼和編譯器所造成的影響。這些影響使得映像檔之間的差異性變大，這會導致映像檔之間可以分享的部分變少，而最糟的情況會讓兩個映像檔完全沒有分頁可進行共享。圖1的情況是target映像檔多出了物件K，而物件K使得後面的物件全部被改變位置。這將導致物件B、C、D和E無法與參考映像檔中的物件做共享。

爲了要讓兩個不同版本的映像檔能夠被共享，我們必須將來源映像檔內的物件重新排列。一旦移動了映像檔內的物件，其相對應的指標及指令也要被修正。重新排列物件的依據是參考映像檔。系統會將參考映像檔內紀錄的物件位址資訊取出，接下來分析來源映像檔，決定哪些物件是能夠被共享的。能夠被共享的物件會有和來源物件相同的位置，而其它的物件會被移動到新的位置。圖2表示整個重排物件的過程。標示2的來源映像檔中的物件B與參考映像檔中的物件B不同，所以這個物件必須被移動到新的位置。而物件C和物件A則是擺放位置和參考映像檔中的物件不同，故必須被重新修正。在來源映像檔中的物件B被移到新的位置後，會留下空洞。爲了使這部分能被共享，系統會將標示1中的物件B複製到此一空洞中。最後標示3爲修改過後的目的映像檔，其中由物件A(2)、B(1)、C(2)、D(2)所組成的部分是和參考映像檔完全相同的部分，所以這個部分可和參考映像檔作共享。而物件B(2)會被重新排列到新的位置，故不會影響到共享的部分。

參考映像檔和目的映像檔之間能共享物件的多寡完全要看來源映像檔被修改哪些物件。但是有兩種物件被修改會造成較大的影響。第一種爲被廣泛存取的物件。第二種爲在存取路徑中最深的物件。圖3可說明以上的兩種情形。本圖是表示物件之間的存取關係，箭頭所指的物件代表被存取方。物件H會被物件A、B、C和D所存取。所以當物件H被修改後它將不能被共享，而且此一影響會擴散至物件A、B、C和D，導致此四個物件失去共享的可能性。同樣的情形如果發生在物件F則受影響的只有物件C，也就是當物件越被廣泛存取則它所造成的影響也越大。而修改物件L會影響到物件J、E和B，這個例子說明了如果被修改的物件在存取路徑的越深處，那它將影響越多的物件。

當一個物件被移動後需要被修正的部分我們稱它爲非區域性指涉。一個非區域

性指涉通常是一個指標，或者是指令中的位址欄位。存取堆疊內的區域變數是一個區域性指涉，因此當物件被移動時並不需要被修正。相對位址的跳躍指令，只要它的目的位址在物件內，物件的移動也不會對它造成影響。除了以上的情況外，其它有涉及位址的指令或資料都是非區域性指涉。

爲了辨識出這些非區域性的指涉，只靠映像檔內的資訊是不夠的。額外需要的資訊有除錯資訊及映像檔的原始碼。首先程式必須分析出這個映像檔內有哪些物件，這些資訊可從映像檔內的符號表及除錯資訊取得。接下來系統必須分析組成映像檔的原始碼，並紀錄所有可能造成非區域性指涉的區域。再來程式會定位出這些非區域性指涉在映像檔內的位址，並且分析該位址的指令，產生相對應的修正紀錄。

根據上述所產生的修正紀錄，系統可以重新建構出整個映像檔的物件存取關係圖。並且利用關係圖來分別哪些物件可以被兩個映像檔共享。整個辨識的流程如下：先選出所有只在來源映像檔中出現的物件，這些物件不能與參考映像檔共享是顯而易見的。另外要被挑選的物件是大小與參考映像檔內物件不符合，因爲這些物件要被共享必須透過一些特殊的手法。爲了簡化處理的程序，我們不對這些物件進行共享。經過初步篩選的物件我們稱之爲污染源，正如字面所形容的，污染源會將污染傳播給其他的物件。一旦程式物件存取到這些污染源，那麼這個程式物件也會成爲一個污染源，並且將污染傳播給其他存取到它的程式物件。所有成爲污染源的程式物件將不能被共享。

圖4是表示污染是如何在物件中傳遞的。圖中來源映像檔內的物件A其程式碼被修改，所以此物件無法和參考映像檔做共享。於是此物件將被移動到新的位址。而物件B中的call指令目的位址爲物件A，於是這個指令的位址必須被修改成0x50，這會導致物件B的程式碼與參考映像檔中的物件B產生差異，故這個物件將不能被共享。也就是物件A的污染被傳遞到物件B。最終在目的映像檔所呈現的結果就是，只有物件C能夠被共享。

當所有物件都被分類完成後，系統會利用參考映像檔的物件分佈表來重新排列來源映像檔物件。所有可被分享的物件將會依照分配表的位置排列，而不能共享的物件將會被給予新的位址。當所有的物件都分配到位址後，系統會開始修正所有的非區域性指涉。修正完成後程式會辨識隱藏污染源，所有的隱藏污染源也是不可共

享的部分。而系統也會試圖尋找出所有和隱藏污染源相關的物件，並且將這些物件移動到新的位址。最後系統會將映像檔內所有的空洞填補，而填補的資料則是由參考映像檔取得。並輸出目的映像檔。

當一個物件在參考映像檔和來源映像檔有相同的大小，但是卻有不同的程式碼時，我們稱它爲隱藏污染源。這些物件無法在最初的分類被發現。如果系統在物件尚未被重新排列前就先檢查物件的內容，會造成誤判。因爲可以被共享的物件在來源映像檔尚未被重新排列之前，內含的程式碼也會和參考映像檔內的物件不同，所以系統無法在來源映像檔被排列之前，就找出這些隱藏污染源。

Figure 1: Image sharing

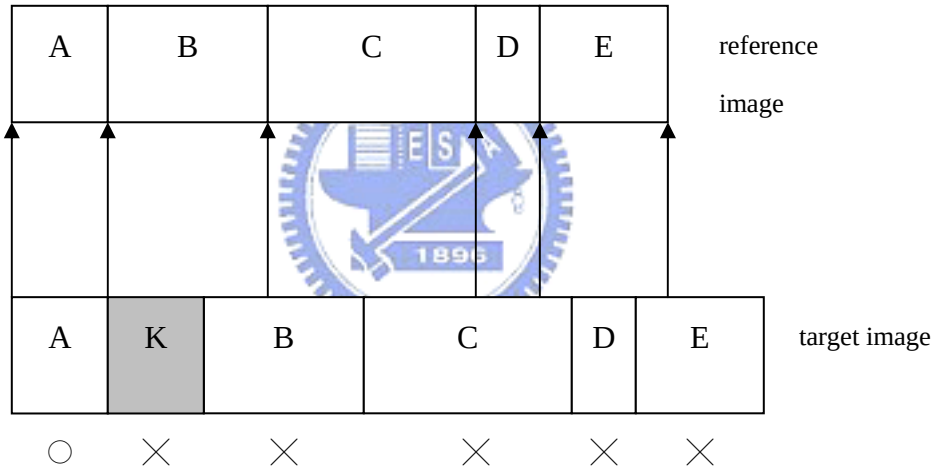


Figure 2: Object reordering

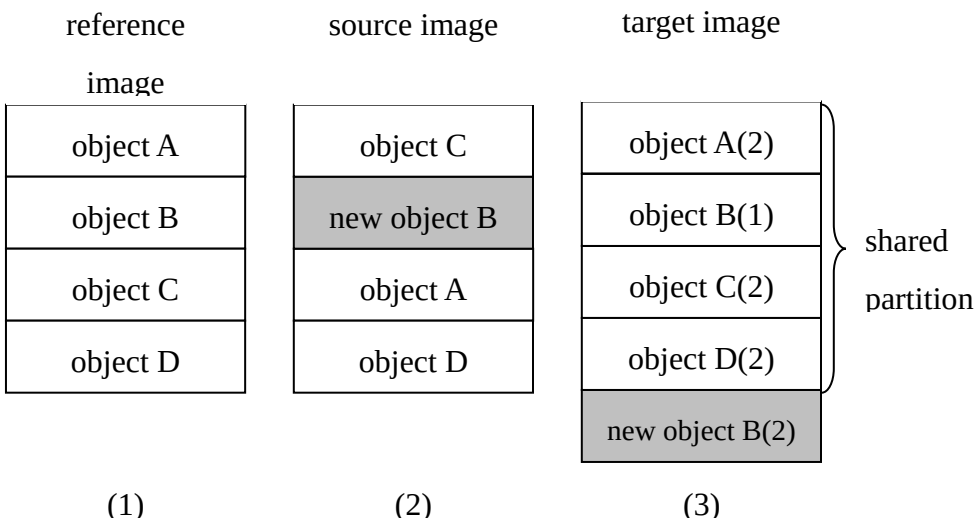


Figure 3: Access map

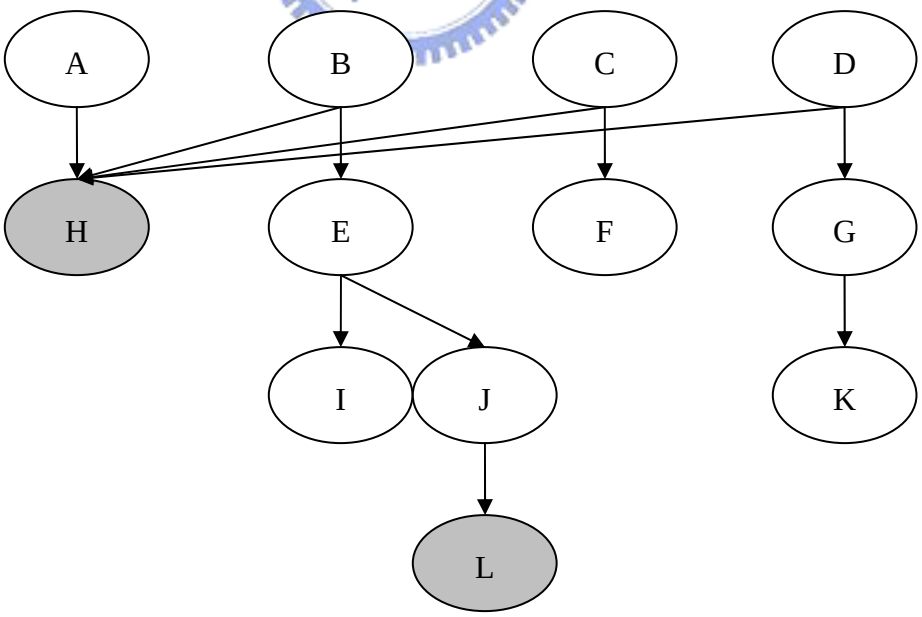
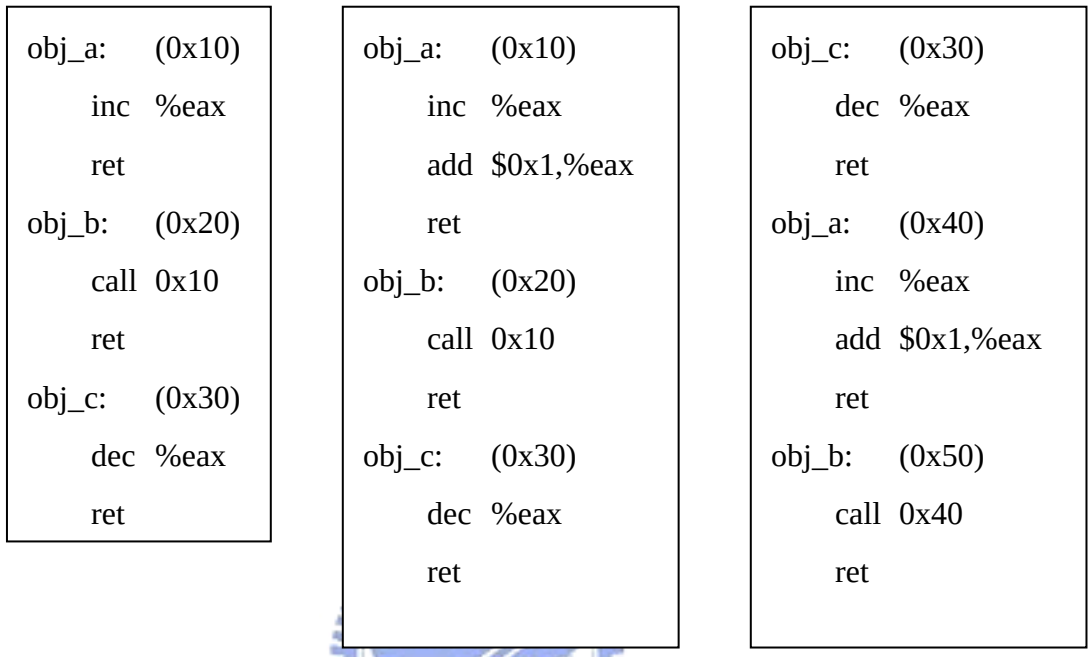


Figure 4: Pollution Propagation



(A) reference image

(B) source image

(C) target image

四、設計與實作

4.1 ELF 格式

ELF [3]是在 Unix 系統上流行的一種執行檔格式。而 Linux 作業系統預設使用的執行檔格式也是 ELF。對一個 ELF 格式檔案可以從兩個觀點來看待它。如圖 5 所示。從 linker 的角度來看，ELF 格式檔案是由一個個 section 所組成，每個 section 有自己的屬性及作用。Linker 可以利用這些資訊來將分別的 section 組成一個 ELF 格式的檔案。從 loader 的角度來看，ELF 格式檔案是由一個個的 segment 組成，loader 可以由 segment 的屬性來決定如何載入這個 segment 到記憶體中。

4.1.1 ELF 檔頭

判斷一個檔案是不是一個 ELF 格式的檔案，可以由檔案開頭的內容得知。從 ELF 檔頭我們可以知道這個 ELF 格式檔案是用在哪種平台上，也可以得知這個 ELF 格式檔案是一個執行檔或是一個動態載入的函式庫。

4.1.2 Section 標頭表

圖 6 可以告訴我們如何找到一個 section。ELF 格式檔案是由一個個的 section 所組成，每一個 section 都有一個 section 標頭來描述這個 section 的相關性質。所有的 section 標頭都被集合在 section 標頭表中。而這個 section 標頭表的位置可以由 ELF 檔頭得知。

4.1.3 特殊的 section

在 ELF 格式檔案中有一些 section 所存的並不是程式的資料或程式碼，這些特殊的 section 內所存的資料可以幫助 linker 執行它的任務。而我們的系統主要會用到以下兩個特殊 section。

1. 字串表

當編譯器產生出目的檔時，會將所有的字串資料輸出至字串表中。字串表中的資料格式完全和 C 語言的字串相同。我們可以從 section 標頭的 sh_type 欄位來判斷此 section 是否爲字串表。一個字串表的 sh_type 欄位應該爲 SHT_STRTAB。

2. 符號表

符號表中包含映像檔中所有物件的資料。其中包含物件的名稱、位址、大小和相關的屬性。同樣的我們也可以從 section 標頭的 sh_type 欄位來得知這個 section 是否爲符號表。一個符號表的欄位爲 SHT_SYMTAB。

Figure 5: Object file format

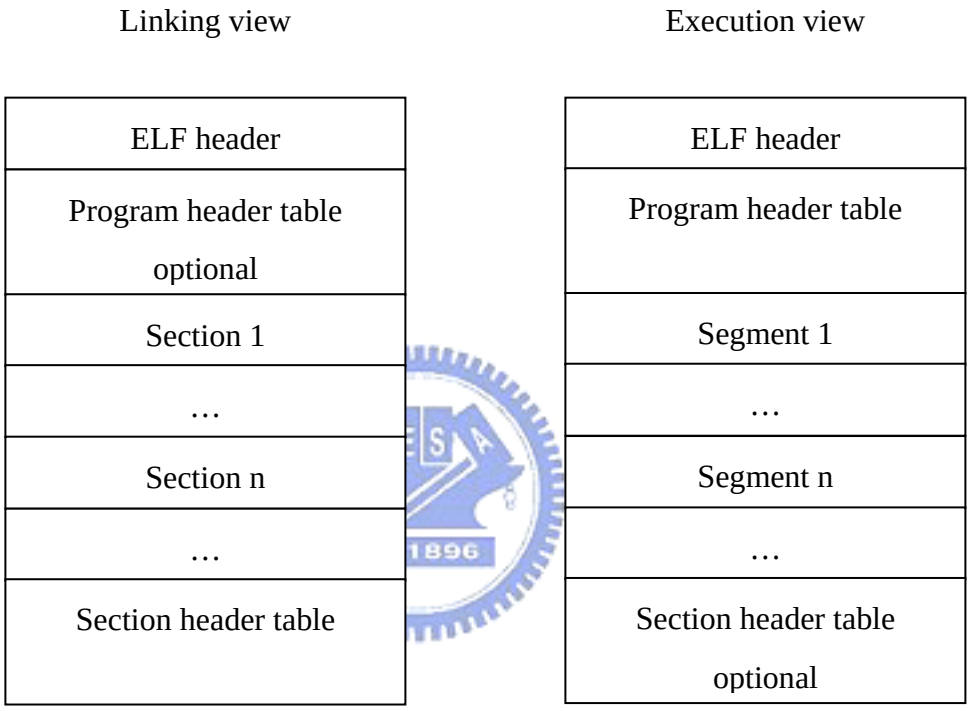
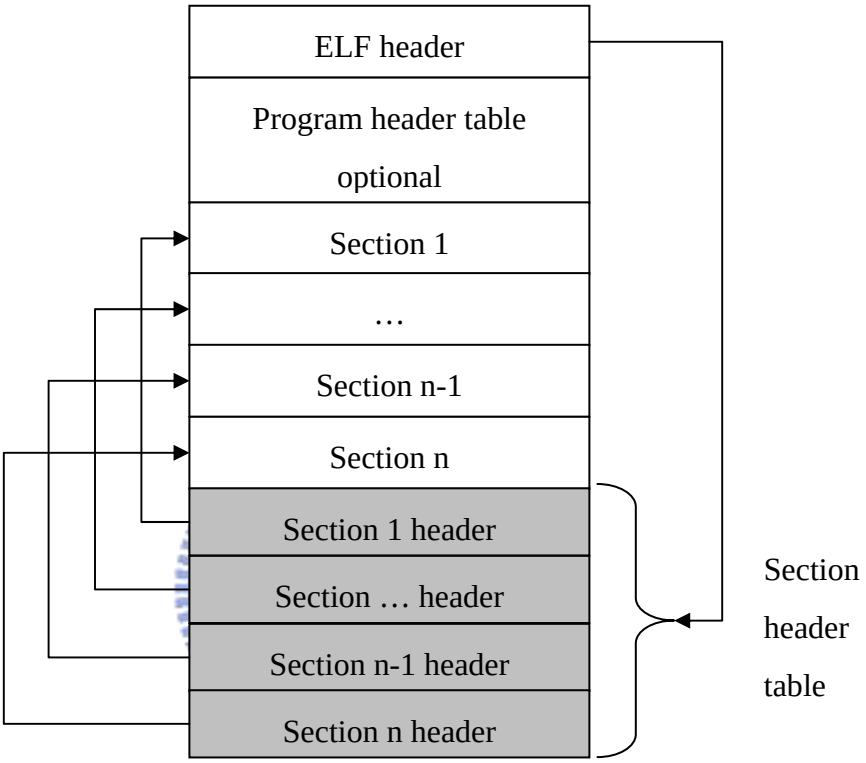


Figure 6: Relationship of section and section header



4.2 除錯資訊 [4]

這個標準被訂出來的目的是爲能符合 C、C++、FORTRAN77、Fortran90、Modula2 和 Pascal 的需求。目前 GCC 所產生出來的除錯資訊也屬於此標準。除錯資訊在 ELF 格式檔案中也有自己的 section。

4.2.1 除錯資訊結構

除錯資訊爲樹狀結構，每一個節點都有它代表的意義。每個節點都有屬性來描述這個節點的性質。爲了減少除錯資訊的大小，所有的除錯資訊都經過編碼。每一個編譯單元都有一棵除錯資訊樹來描述它。每一個節點內所描述的資

料需要透過 abbreviation table 中的指示才能解讀。而一個節點它可以描述一個變數或是函式，這必須將節點資料解碼後才能得知。GCC 所產生的除錯資訊會被放置在.debug_info 這個 section 中。

4.2.2 Abbreviation Table

除錯資訊樹中的每個節點都會對應到這個表中的一個元素，而多個節點可以對應到相同的元素。每個元素紀錄著它所代表的物件型態，如編譯單元、函式、變數等。接著是記錄這個節點有哪些屬性，每個屬性的型態。屬性的型態可能有字串、2 位元組的常數、4 位元的常數、位址、不定長度的整數等。GCC 所產生的 abbreviation table 會被放置在.debug_abbrev section 中。

4.2.3 行號矩陣

行號矩陣是連結程式原始碼和機器碼的主要橋樑。每個編譯單元都會對應到一個行號矩陣。矩陣中每一列都紀錄著行號所對應到的記憶體位址。在除錯器中設定中斷點時便是使用這項資訊來達成。但是要讀取這個矩陣的資料並不是那麼容易的。爲了要減少矩陣所佔用的空間，資訊經過特殊的壓縮。矩陣的資訊是由一個有限狀態機所產生出來的，而被儲存的資訊是這個有限狀態機的程式碼。透過執行這些程式碼，矩陣的資料就會依序被產生出來。

4.3 映像檔分析程式

爲了要移動物件，所有的非區域性指涉都必須被辨識，並且產生修正記錄，讓修正程式能夠正確搬移物件並修復這些非區域性指涉。

4.3.1 原始檔的前置處理

在 C 語言中有提供巨集的功能，爲了減低分析時的複雜度，必須消除巨集所造成的影響。因此在產生程式映像檔之前，必須利用 GCC 展開所有的巨集，並以展開後的編譯單元來進行編譯。對於 Linux 2.6 核心原始碼處理，我們可以修改 Makefile.build 這個檔案內的規則來達成。在編譯原始碼之前先命令編譯器展開巨集，消除巨集所造成的影響。

4.3.2 建立物件地圖

系統分析映像檔的第一步是建立物件地圖。物件地圖內記錄物件的位址、大小及屬性。分析程式會讀取映像檔的符號表，並且將所有全域物件加入到物件地圖中。對於區域物件爲了避免同名的衝突，系統必須將這些物件進行重新

命名的工作。這時分析程式會掃瞄除錯資訊，並且取得區域物件資訊，並將原始物件名稱加上編譯單元的名字，做爲物件的新名稱。

4.3.3 反組譯碼格式化

在進行映像檔分析之前，我們必須利用 objdump 這個程式對應像檔進行反組譯。然而反組譯後的輸出檔爲文字格式。爲了助往後的分析，系統會將此文字檔進行格式化，並且儲存於記憶體中。在反組譯碼檔案中，最開頭爲指令的位址，接下來的部分爲機器碼，最後爲可讀性佳的助憶碼。這些資料都會被讀取並記錄到記憶體中。

4.3.4 辨識可能字串物件

字串物件的資訊並不會在符號表或除錯資訊中出現，所以要定位出字串物件必須經由特殊的步驟。字串物件在映像檔中的格式與 C 語言的格式一樣，都是以 ASCII 碼 0 結尾。所以分析程式必須掃瞄所有的資料 section。一旦發現可能爲字串的物件，就將此位址記錄下來作爲後續分析使用。

4.3.5 尋找非區域性指涉

搜尋非區域性指涉的基本過程如下，首先決定目的物件，分析程式開始掃瞄編譯單元原始碼。當物件名稱被發現時，系統記錄發現物件行號。取得行號後利用除錯資訊的行號矩陣將行號轉換爲位址。接下來系統會分析相對應的反組譯碼，並辨識出非區域性指涉所在位址。之後分析程式會分析機器碼決定非區域性指涉如何被修正。

1. 一般全域物件產生的非區域性指涉

系統在進行掃瞄原始檔之前，會先建立一個全區域物件的串列。當掃瞄編譯單元時，會先判斷此 token 是否爲 identifier token。如果是此類的 token，且此 token 不是 C 的關鍵字，則分析程式會拿這個 token 去對全域物件串列進行比對。一旦發現此 token 爲全區域物件的名稱，系統會記錄這個物件的行號以利往後的分析。

2. 由字串物件所產生的非區域性指涉

當分析程式掃瞄編譯單元時，如果發現 token 爲字串時，則會比對此字串是否出現在字串串列中。如果比對符合，則字串物件會被加入到物件地圖中。然後系統會把字串物件當作目的物件，並找出它的非區域性指涉。

3. Static 物件所產生的非區域性指涉

如果一個 identifier token 帶有修飾字 static，則代表他是一個 static 物件。所以分析程式會爲它做更名的動作，更改後的名稱爲原始名稱加上編譯單元的名字，再利用更改後的名字進行比對的工作。

4. 初始化物件所產生的非區域性指涉

初始化物件包含用來初始化變數的資料，一般來說物件內的資料會在程式執行期間，被複製到變數中。但是並非所有的初始化物件都是如此，所以分析程式必須對每一種情況進行處理。

(1) 有實體的初始化物件

有實體的初始化物件可以在符號表中被發現，它們的名字爲 C.#.# ，井號代表一個數字。此數字由編譯器決定，所以分析程式無法直接由名稱來得知初始化物件的內容。因此當分析程式從編譯單元掃瞄到初始化物件後，必須對初始化物件的內容進行重建，並對所有在符號表中的初始化物件做比對，最後確定初始化物件的名稱。一旦比對完成，分析程式就可以知道初始化物件所在的位址，並用此位址和反組譯碼產生出修正記錄。

(2) 無實體的初始化物件

這種初始化物件會有下列的類型：被內嵌於指令的初始化物件、靜態區域變數的初始化物件、全域變數的初始化物件及靜態全域變數的初始化物件。

對於內嵌於指令的初始化物件，分析程式會對物件內的每一個參數進行分析，判斷它是否會產生非區域性指涉。如果參數爲字串或某物件的指標，那就代表分析程式必須掃瞄反組譯碼，並爲此參數產生修正記錄。

其餘的初始化物件都會被直接放置在變數中，所以分析程式並不需要掃瞄反組譯碼，只需要爲初始化物件中的每個非區域性指涉產生修正記錄。爲了產生修正記錄，分析程式必須知道變數的位址。全域變數的位址可以直接從物件地圖中取得。而靜態全域變數則可經過更名後，再從物件地圖得知。而靜態區域變數要經過特殊的處理才能得知它的位址。靜態物件在符號表中會被更名爲物件名稱加上一個數字，同樣的這個數字也是由編譯器決定。所以分析程式必須重建初始化物件的內容，用來尋找初始化

物件的位址。

5. 初始化物件中的非區域性指涉

初始化物件內也會包含非區域性的指涉，它們會是一個物件的指標，或者是一個字串。分析程式必須將初始化物件中的參數一個個取出，並利用物件地圖來產生修正記錄。對於字串物件，分析程式會比對候選字串串列，如果符合則將字串物件加入物件地圖中，並產生修正記錄。

4.3.6 修正記錄

修正記錄裡面記載如何修正非區域性指涉的資料。這些資訊包含非區域性指涉所在的位址、非區域性指涉所在物件、非區域性指涉參考的位址、非區域性指涉所參考的物件、修正欄位的大小及修正的方式。修正的方式有兩種。對於絕對位址定址只需要將新的物件位址填入修正欄位即可。而相對位址定址則要計算非區域性指涉所在位址和非區域性指涉參考位址相對的距離，之後在將新的值填入修正欄位中。

4.4 映像檔修正程式

映像檔修正程式可以利用物件分析程式所輸出的資料對映像檔進行重新排列的動作。所以它需要參考映像檔的物件地圖來作爲重新排列物件的依據。參考映像檔的物件地圖可以由 nm 這隻程式取得。

4.4.1 重建物件地圖

物件分析程式會將所有被辨識的物件列表，包括字串物件和初始化物件等。重建物件地圖同時也會將物件的內容讀入記憶體中，這是爲了方便接下來修正的動作。

4.4.2 重建存取地圖

由修正記錄，修正程式可以建立存取地圖。在修正記錄中有非區域性指涉所在物件及非區域性指涉所參考的物件，這可以讓程式推斷出存取關係。掃瞄完所有的修正紀錄後，系統就可以得到完整的存取地圖。

4.4.3 辨識可共享物件

修正程式會利用之前所提到的分類法爲物件分類。對於可以共享的物件，系統會給予它一個和參考物件相同的位址。而對於不能被共享的物件，修正程式會計算新的位址給這些物件。當所有物件都就定位後，修正程式便利用修正

記錄爲每個物件內的非區域性指涉做修正。在所有的物件都被修正後，修正程式必須掃瞄所有共享物件的內容。找出所有的隱藏污染源，並利用存取地圖找出所有和污染源有關的物件，並將給予這些物件新的位址。最後再對這些物件進行修正的工作。

4.4.4 輸出修正後映像檔

最後的步驟是將所有物件寫入到檔案中，並產生 ELF 格式所需要的資料。因爲某些物件已經有新的位址，所以在可被共享的 section 中會留下一些空洞。而這些空洞會造成這個 section 某些分頁不能被共享，所以修正程式會讀取參考映像檔的內容來填補這些空洞，讓整個 section 能夠被共享。

五、實驗

5.1 可共享物件評估

這個實驗的目的是爲了評估兩個映像檔之間可以共享物件的數量。不會被改寫的映像檔稱爲參考映像檔，而要被改寫的映像檔稱爲目的映像檔。分析程式會先掃瞄目的映像檔，並且爲目的映像檔建立存取地圖。接下來程式會分析參考映像檔，並利用參考映像檔的物件地圖來追蹤目的映像檔內的污染傳播。

追蹤的步驟如下，分析程式會先確定目的映像檔內的污染源。污染源有兩種，第一種是不存在於參考映像檔內的物件，另一個是物件在兩個映像檔內有不同的大小。這兩種物件是被修改過的，所以不能被共享。之後程式利用目的映像檔內的存取地圖來找出所有被污染源影響的物件，而剩餘的物件就是可共享的物件。

5.1.1 不同版本核心可共享物件數量評估

核心版本	2.6.11 – 2.6.17
編譯器版本	gcc-4.0.2

Table 1: 共享物件數量

Objects (shared)	11	12	13	14	15	16	17
11		2392	2203	1905	1205	1104	1047
12	2291		2347	2009	1191	1095	1040
13	2098	2338		2212	1264	1159	1098
14	1806	1999	2202		1394	1266	1187
15	1119	1181	1253	1389		2139	1935
16	1029	1087	1147	1259	2141		3050
17	968	1021	1071	1163	1912	3035	

objects (total)	
11	6109
12	5943
13	6063
14	6157
15	6269
16	6406
17	6554

Table 2: 共享物件大小總和 (bytes)

size (shared)	11	12	13	14	15	16	17
11		296375	270643	240255	151704	143622	138748
12	272432		302533	261436	143246	136147	131988
13	246194	302359		317405	180207	138885	133869
14	216399	261900	316632		209180	147744	140812
15	129503	143924	179863	208991		274309	253834
16	121701	135657	138203	147373	274018		398611
17	119232	131420	133063	137953	250281	395063	

size (total)	
11	1225089
12	1186068
13	1237299
14	1262038
15	1088606
16	1243640
17	1274471

Table 3: 共享物件數量 (百分比)

objects (%)	11	12	13	14	15	16	17
11		39.15	36.06	31.18	19.72	18.07	17.13
12	38.54		39.49	33.8	20.04	18.42	17.49
13	34.6	38.56		36.48	20.84	19.11	18.1
14	29.33	32.46	35.76		22.64	20.56	19.27
15	17.84	18.83	19.98	22.15		34.12	30.86
16	16.06	16.96	17.9	19.65	33.42		47.61
17	14.76	15.57	16.34	17.74	29.17	46.3	

Table 4: 共享物件大小總和 (百分比)

size (%)	11	12	13	14	15	16	17
11		24.19	22.09	19.61	12.38	11.72	11.32
12	22.96		25.5	22.04	12.07	11.47	11.12
13	19.89	24.43		25.65	14.56	11.22	10.81
14	17.14	20.75	25.08		16.57	11.7	11.15
15	11.89	13.22	16.52	19.19		25.19	23.31
16	9.78	10.9	11.11	11.85	22.03		32.05
17	9.35	10.31	10.44	10.82	19.63	30.99	

這個實驗是評估兩個映像檔之間有多少物件可以共享。而實驗的核心版本

從 2.6.11 到 2.6.17。表格的縱軸代表目的映像檔的版本，而橫軸代表參考映像檔的版本。由表格 1 中我們可以觀察到，兩個版本越相近的映像檔擁有越多可以共享的物件。表 2 則顯示出映像檔之間可供共享的物件大約是 100 Kbytes 到 400 Kbytes。從表 3 中觀察到 2.6.16 核心與 2.6.17 核心有高達 45%的物件可以做共享，而最低的部分也有 14.76%的物件可以做共享。表 4 可以看到共享的物件佔整個映像檔最高可達 30%，最少也有 10%。

核心版本	2.6.7 – 2.6.10
編譯器版本	gcc-3.3.2

Table 5: 共享物件數量

objects (shared)	7	8	9	10
7		2444	2117	1955
8	2452		2301	2123
9	2110	2281		2481
10	1908	2072	2442	

objects (total)	
7	5855
8	5934
9	6105
10	6152

Table 6: 共享物件大小總和 (bytes)

size (shared)	7	8	9	10
7		290084	224877	188821
8	291145		292813	252078
9	225179	291550		316801
10	185089	247864	313886	

size (total)	
7	1251931
8	1540872
9	1209351
10	1205304

Table 7: 共享物件數量 (百分比)

objects (%)	7	8	9	10
7		41.74	36.15	33.39
8	41.32		38.77	35.77
9	34.56	37.36		40.63
10	31.01	33.68	39.69	

Table 8: 共享物件大小總和 (百分比)

size (%)	7	8	9	10
7		23.17	17.96	15.08
8	18.89		19	16.35
9	18.61	24.1		26.19
10	15.35	20.56	26.04	

此實驗也是比對個映像檔之間的共享物件，因爲 2.6.7 到 2.6.10 的核心與 gcc-4.0.2 不相容，所以我們採用 gcc-3.3.2 來編譯。從上面的表格中，整體的趨勢和上一個實驗差不多。結論是版本越相近的核心可共享的物件越多，共享物件大小總和平均爲 200 Kbytes。

六、問題探討與總結

6.1 未來工作

目前的系統藉由掃瞄映像檔來建立物件地圖和存取地圖。爲了要得到更精確的結果，程式也會掃瞄原始碼。但是目前作法會完全忽略編譯器的資訊。對於同樣的原始碼，編譯器採用不同的最佳化策略也會產生不同的結果。沒有這部分的資訊，系統分析的結果也會不精確。

比較好的作法是利用編譯器來做物件移動，然而 gcc 並不能讓程式設計者指定物件的位址，這是 gcc 設計上的限制。改變 gcc 的設計是困難，折衷的辦法是分析映像檔後重新改寫原始碼，將原始碼重新排列並利用.org 這個 directive 來指定物件的位址。

6.2 結論

伺服器整合在未來是一種趨勢，利用虛擬機器的技術將硬體的資源做充分利用，節省硬體成本方便管理。在伺服器整合的環境中，類型相近的網路服務會被集中在一起，所以使用的作業系統和軟體也會十分相近。將這些軟體相同的記憶體分頁做共享，不僅可以減低記憶體的使用量，而且也可以降低 L2 快取失誤率，增加效能。但是由於程式原始碼和編譯器的影響，映像檔很難可以直接做共享。本論文利用物件重排的方法將映像檔改寫，增進映像檔的相似度讓共享可行。我們的實驗分析了 2.6.7 到 2.6.17 的 Linux 核心，結果顯示映像檔平均有 200 Kbytes 的物件可以共享，而 2.6.16 和 2.6.17 核心可共享部分更高達 389 Kbytes。這些物件平均佔整個映像檔 17%。

參考文獻

[1] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble, “Scale and Performance in the Denali Isolation Kernel”, 5th Symposium on Operating Systems Design and Implementation, 2002 Dec

[2] Carl A. Waldspurger, “Memory Resource Management in VMware ESX Server”, 5th Symposium on Operating Systems Design and Implementation, 2002 Dec

[3] Executable and Linkable Format. http://www.x86.org/intel.doc/tools.htmd

[4] DWARF Debugging Information Format. http://www.x86.org/intel.doc/tools.htm

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield, “Xen and the Art of Virtualization”, ACM Symposium on Operating Systems Principles, 2003

[6] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang,and Mendel Rosenblum. “Cellular Disco: Resource Management Using Virtual Clusters on Shared-Memory Multiprocessors”, ACM Transactions on Computer Systems, Vol. 18, No. 3, August 2000, Pages 229-262

[7] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. “Disco: Running Commodity Operating Systems on Scalable Multiprocessors”, ACM Transactions on Computer Systems, Vol. 15, No. 4, November 1997, Pages 412-447

[8] Samuel T. King, George W. Dunlap, and Peter M. Chen, “Operating System Support for Virtual Machines”, USENIX Annual Technical Conference, 2003 Jun

[9] Intel Vanderpool Technology for IA-32 Processors (VT-x) Preliminary Specification. http://www.intel.com/technology/computing/vptech/