嵌入式系統之客製化動態記憶體管理

蕭光哲
國立成功大學電機工程學系
hktsteve@rtpc06.ee.ncku.edu.tw

陳　敬
國立成功大學電機工程學系
jchen@rtpc06.ee.ncku.edu.tw

謝旻翰
國立成功大學電機工程學系
wasicc@rtpc06.ee.ncku.edu.tw

摘　　要

越來越多的應用軟體被移植至嵌入式系統上，如多媒體應用或是網路等相關應用程式。這些應用程式在執行期間由於有許多無法預期的資料產生，因此相當依賴動態記憶體的使用。嵌入式系統中因硬體資源相對缺乏，對於大量動態記憶體需求可能無法順利執行；又因一般類型之記憶體管理機制也不一定可以直接移植至嵌入式系統上，是故客製化動態記憶體管理機制有其必要性，若再搭配協助工具將可有系統化地處理客製化記憶體管理之選擇。本論文將客製化動態記憶體管理機制的實現分成兩部份：多種類型動態記憶體管理機制與協助選擇記憶體管理機制工具，並實作於 Zinix 微核心嵌入式系統環境，內容包含：函數庫中提供多種動態記憶體管理者，作業系統管理實體記憶體部份提供多種管理方式，與客製化工具等，以協助應用程式選取合適之記憶體管理者。本論文之實作成果使用 H.264 Decoder 與計算 3D 貼圖工具做為測試程式，於 TI 生產之 DaVinci 開發板上驗證。測試結果達到預期之效果，記憶體使用更有效率。

***關鍵詞*—動態記憶體管理、嵌入式系統、微核心**

一、簡　介

越來越多的應用軟體被移植至嵌入式系統行動平台上，例如多媒體應用或是網路服務等等相關應用。這些應用軟體在執行期間由於有許多無法預期的資料產生，因此相當依賴動態記憶體的使用。在如桌上型電腦之系統，許多一般類型之記憶體管理機制即可滿足上述所說明之應用；但對於嵌入式系統，由於硬體資源相對缺乏與有較多限制，對於大量動態記憶體需求之應用可能無法順利執行，且一般類型之記憶體管理機制也不一定適合直接移植至嵌入式系統上。因此客製化動態記憶體管理機制的提供成為一個更好的解決方案，可藉由不同的應用需求提供合適的動態記憶體管理機制使有限資源之嵌入式系統可被應用於不同情況，特別是針對有大量動態記憶體需求之應用。若客製化動態記憶體管理機制運作時，可配合以協助選擇之工具將可有助於系統化地處理客製化記憶體管理之實現，圖 1 所示即為上述概念。

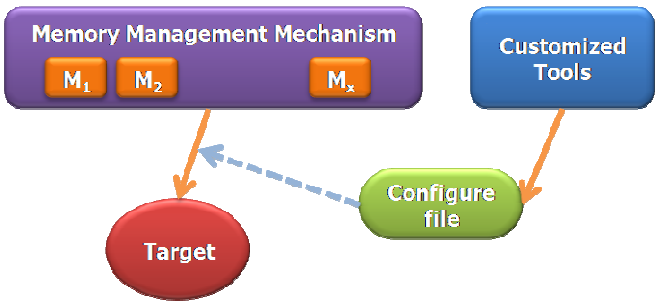

圖 1　選擇合適記憶體管理概念圖

本論文將客製化動態記憶體管理機制之實現分成兩部份：多樣化之動態記憶體管理機制與協助選擇記憶體管理機制之工具。藉由多種類型動態記憶體管理機制，應用軟體可依據其需求而選用較適合之動態記憶體管理機制；協助選擇記憶體管理機制之工具則依據應用程式對於動態記憶體需求情形，系統化地選擇合適之動態記憶體管理機制。系統發展人員使用協助選擇之工具取得應用之動態記憶體需求相關資訊與合適之機制，以建立具有客製化動態記憶體管理機制之應用軟體。

本論文說明對嵌入式系統之客製化動態記憶體管理之研究，內容規劃為第二節討論本論文

設計與實作所參考之主要相關文獻，一般常見之記憶體配置機制與分析方式皆在本節說明；第三節詳細說明本論文設計與實作之多種記憶體管理機制與實作環境；第四節詳細說明客製化記憶體管理輔助工具之設計與實作，以取得所需應用之記憶體使用情形並作為選擇合適記憶體管理方式之依據；第五節為實作之系統其相關效能測試比較；最後，第六節為本論文之結論。

二、相關研究

“Dynamic Storage Allocation, A Survey And Critical Review”[21] 將許多記憶體管理方式集合介紹，但在文獻提出時並沒有特別針對嵌入式系統的應用有任何考量。其針對記憶體管理機制主要三個部分：釋放之閒置區塊處理方式，閒置區塊何時分割，閒置區塊何時合併等，卻是很重要參考依據。而厚片配置[17] 主要是將常用之記憶體容量大小在釋放時先保留起來，而非與其他記憶體區塊合併。若記憶體區塊之容量大小可預先推算，則當有需求時可加快配置的時間。

文獻“Composing High-Performance Memory Allocators”[16] 中研究“Heap Layers”的基礎架構，提供有彈性及有效之記憶體管理方式，作者集合了 Lea Allocator[5] 與 Kingsley Allocator[21] 等比較有名的記憶體管理機制，透過 Heap Layers 覆蓋在原先的 malloc()之類的動態記憶體管理機制，再根據實際情形彈性選擇合適之記憶體管理機制。David Atienza 的研究論文[14][15]中也提供多種記憶體管理機制，且針對特定應用程式，如影像處理與 3D 計算相關方面，選擇較合適之記憶體管理機制，藉此達到較好之效果。

上述文獻中所提之特別記憶體管理機制，使得記憶體管理機制的選擇出現彈性，不過驗證的方式為使用 SPEC2000 驗證程式，而非實際應用程式，且在當時的研究尚未包含嵌入式系統，不過也引起注意。David Atienza 等人之相關研究，已針對嵌入式系統需求考量，而且使用實際應用程式，其研究中所選用的應用程式有其不同對記憶體需求之模式，但是實際驗證的方式為透過桌上型電腦模擬，且模擬的軟體亦為相關作者等人所自行研發之軟體套件，雖然呈現出不錯之效果，不過實際在嵌入式系統環境所測試的情況無法得知。另外在 David Atienza 等人之相關研究中，測試設計之記憶體管理次數為測試十次並取其平均值做為比較之參考依據，此為本論文測試實作結果時之參考。

分析應用程式與提供多記憶體管理討論到的是透過 C++這語言特性與 templates 結合物件導向之應用技術而產生的 Mixin[23]，對於系統的擴充有著許多彈性，但目前大部分作業系統在開發時，通常只使用 C 語言作為開發語言。因此若要利用此種方式實作記憶體管理，可能要先把系統部分更改，以及因為實作之平台為嵌入式系統，也必須要有合適之跨平台編譯工具，才能讓這些工作開始。相關研究[14] [15] 則提及了分析應用程式對記憶體需求的行為模式，並有較多著墨。從其中的實例可歸納出三種模式：

1.對動態記憶體須求為無法預測且類型繁多；
2.對動態記憶體的需求為固定範圍之類型；
3.存取動態記憶體的行為類似堆疊。

本論文實作參考文獻[14] [15] 之應用程式行為分析以選擇合適的記憶體管理機制。不過[14]與[15]之研究是透過模擬的方式分析出來，因此所得的數據並沒有考量實際 I/O 等情況，但針對記憶體管理機制的部份仍可以作為參考依據。

常見的作業系統中，Linux 採用 Binary Buddy 的方式管理頁面[19] ，其以滿足不同的記憶體頁面請求為目的，而採用會造成斷裂較高的頁面管理機制；Windows CE 則將所有頁面都先分割成同樣大小並放置於單一串列[12] ，但如果一次需要較大的頁面空間，則很有可能造成外部斷裂；而 µC/OS-II 採用多個分區記憶體管理[18] ，如果特定的頁面需求特別高時，不一定可以從其他分區中取得頁面。

由上述之作業系統可了解關於頁面管理的設計是越簡單越好，降低資料結構本身的複雜性以提高快速取的頁面的時間；另外因為這幾個作業系統都屬於要滿足大多數的應用，所以在頁面管理機制上有些取捨，但如果只是要滿足特定少數的應用，或許在頁面管理機制的設計上還可以有些調整，此為本論文相關實作的參考依據。

常見函數庫中，除了 uClibc[11] 提供不只一種記憶體管理機制可供選擇，Newlib[8] 與 diet libc[4] 皆只提供單一的記憶體管理機制可供選擇，而所提供的機制有另外針對記憶體容量需求情形而會有不同的處理方式。雖然 uClibc 提供了三種記憶體管理機制可在編譯前的選單設定，但若不深入研究所提供的記憶體管理機制，一般情況下很難知道較適合應用程式之設定。

上述三個函數庫所提供的記憶體管理機制，實際所管理的方式可分成三大類，其實就很像是 Lea Allocator[5] 所使用的方式。第一類為有幾個快取串列，每個串列皆是為特定大小的空間的倍數；第二類則是單一串列，根據需要而直接分割出所需記憶體容量，之後當有記憶體區塊被釋放時，再尋找合適的位置檢查是否可被合併並存放；第三類則是直接使用 Linux 所提供的 mmap()等相關函數。

總結以上各函數庫所提供的記憶體管理機制，會特別針對記憶體需求情形而有不一樣的處理方式，主要也是為了滿足不同的記憶體需求而規劃設計，但如果已經事先分析出記憶體需求情形為何，將可以降低設計的複雜性，且程式再執行時也可以省掉一些判定的時間已加快記憶體的配置，因此本論文之實作也將這些列入考量。

三、客製化記憶體管理之設計

(一) 記憶體管理架構

與記憶體管理有關的部份，如圖 2 所示可分成兩個層面來看：提供給應用程式使用之函數庫與作業系統之頁面管理。為了提供多個記憶體管理機制，首先透過架構圖了解與記憶體管理有關的部份。

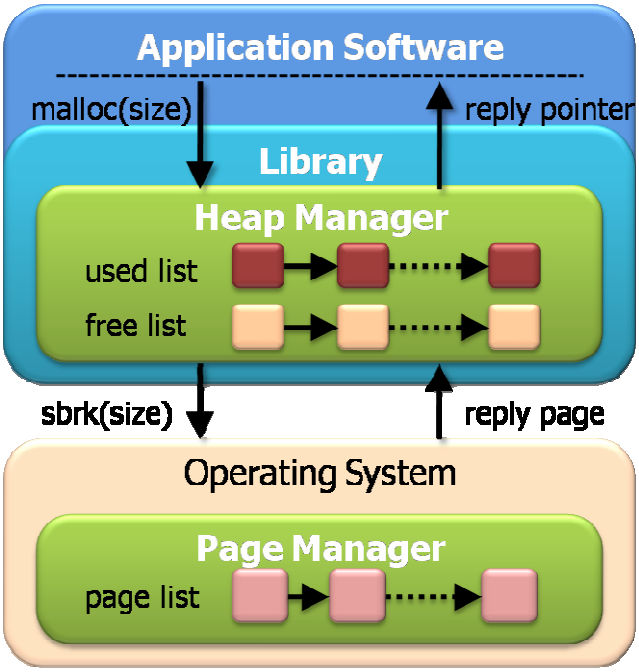

圖 2　動態記憶體管理概念圖

一般應用程式如果需要動態記憶體時，在 C 語言程式中會呼叫標準函數庫提供的函式 malloc()，而當不需使用這塊動態記憶體空間時，則可使用 free()函數釋放。應用程式透過函數庫向系統提出動態記憶體空間的需求，而系統所配置給應用程式的容量通常會大於實際需求，多餘的空間則透過函數庫協助應用程式管理。當應用程式再度有動態記憶體的需求時，則先從函數庫所管理的閒置區塊中尋找可滿足需求的空間；如果沒有滿足的空間，才向系統再次要求額外的記憶體空間。應用程式不再使用配置的動態記憶體空間時，透過 free()釋放這些空間。被釋放掉的閒置區塊，則交由函數庫協助管理，之後如果應用程式還需要動態記憶體空間時，可快速地從函數庫提供；因此這些被應用程式釋放的閒置區塊並非直接就釋放還給系統。

另外一部分與記憶體管理有關的是作業系統之頁面管理。作業系統為了管理實體記憶體空

間，並依據需求配置給需要的程式，如應用程式，驅動程式或是系統本身。因此作業系統會將實體記憶體空間規劃成許多區塊，每次配置以此區塊容量大小為單位配置給所需的程式，這些被配置的區塊，如果是應用程式，大部分都是等到程式執行結束才會歸還給系統，此為作業系統中頁面管理主要負責的部份。

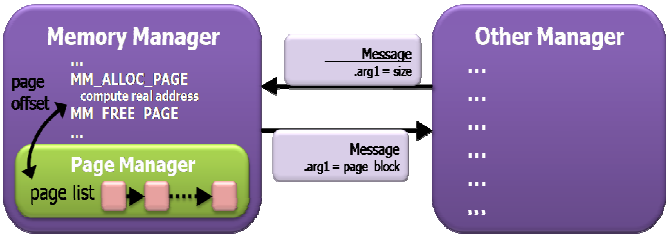

圖 3　作業系統之頁面管理

(二) 作業系統之頁面管理

一般桌上型電腦或工作站等作業系統中，因為在其上運行的程式種類很多且無法事先得知，為了滿足多樣化的需求以及管理上的問題，所以在頁面管理部份會採用可滿足大多數情況需求的管理方式；且為了管理方便因此必須多花一些空間成本，例如多使用一些記憶體空間以儲存額外在記憶體管理上會使用的資訊。但在嵌入式系統中，通常可預期會運行的應用程式為何，作業系統之服務功能為何，所以在頁面管理上如果只需要滿足這些需求，此時就可以採用針對應用所需之頁面管理方法作為頁面管理的機制。

本論文之設計於作業系統中如圖 4 所示提供三種頁面管理機制，以符合嵌入式系統屬於特別用途而產生的特性。所提供三種頁面管理機制分別為：Binary Buddy，Segregated Free List，與 Partition Allocator。處理頁面需求的流程如圖 3 所示，當系統接收到應用程式或是其他作業系統本身之服務管理者發出記憶體需求時，記憶體管理者會找尋合適的頁面大小並配置給應用程式或是其他服務管理者。因此有效尋找合適的頁面將影響應用程式的執行效率；如果是在微核心系統的架構中，提出需求必須透過訊息傳遞，因此這些額外的負擔也必須列入考量。

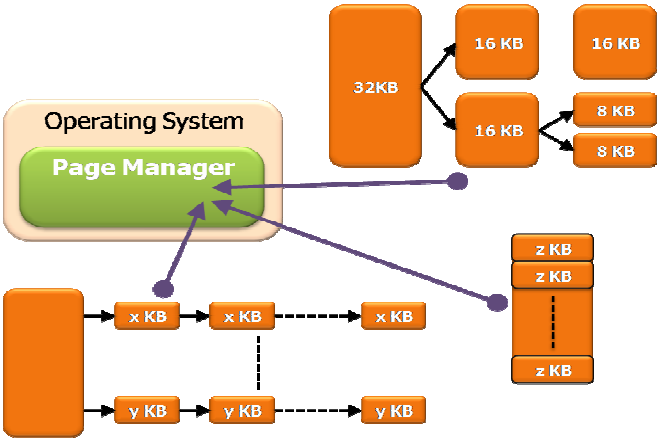

圖 4　作業系統內多種頁面管理機制

(三) 函數庫之閒置區塊管理

作業系統配置頁面給應用程式後，應用程式所需的容量大小通常不會超過作業系統所配置的，因此多餘的空間則交由函數庫中閒置區塊管理機制管理，如圖 5 所示。如果應用程式再次需要動態記憶體空間時，則函數庫之區塊管理機制會先找尋合適的空間，如果有足夠記憶體空間則配置給應用程式使用，否則會先向作業系統提出更多的記憶體需求；等待作業系統回應配置給一頁面後，閒置區塊管理機制會切割可滿足應用程式的需求大小配置給應用程式，多餘的空間將繼續由閒置區塊管理機制管理。

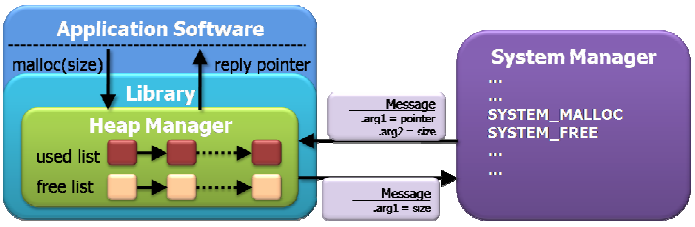

圖 5　函數庫之區塊管理

本論文在函數庫提供所提供的閒置區塊管理機制之設計共有三種，如圖 6 所示，分別為：Next-fit Address Order，Segregated Free List 與 Binary Buddy 等。在微核心系統的架構中，無論是向作業系統索取頁面空間或是釋放頁面都必須透過訊息傳遞索取；考量此因素所必須負擔額外的時間成本，所有已向系統索取的頁面統一在應用程式執行結束時再一併歸還給作業系統。

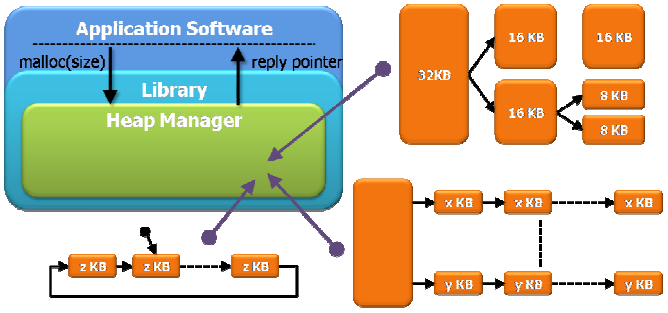

圖 6　函數庫內多種記憶體管理

(四) 記憶體管理選擇機制

在嵌入式系統上，會使用那些應用程式幾乎都已預先知道，甚至有些還可以最佳化。因此本論文決定選擇合適記憶體管理的時間點為編譯時就選定使用那種記憶體管理機制。

若要在編譯時決定使用之記憶體管理方式，根據過往經驗在實作上是透過函數的方式，當接收到對記憶體的需求時，一樣是透過原本的介面接收與回應，而不同的記憶體管理方式則是此函數中再呼叫合適的管理機制，使用此種設計方式好處為維護容易，且日後如果還希望加入更多的記憶體管理機制時，也可以快速而清楚地加入進系統。不過考量到實作在嵌入式系統上，如果每次為了記憶體的需求，還需要額外再呼叫函數，則因為在這之間有額外的負擔，如函數呼叫時，會把此時的狀態存進堆疊，然後才換新的函數執行，且執行過後還需要從堆疊取區原先函數的數值等，因此有些管理的成本必須列入考量。

(五) 實作環境

本論文實作所使用之硬體開發平台為德州儀器（Texas Instruments）所發展之 DaVinci[22] 參考設計開發板（Evaluation Board），如圖 7 所示，其處理器 TMS320DM6446[9] 為異質雙核心架構，由 ARM926EJ-S 與 TMS320C64x+所組成。DaVinci 實體記憶體最大支援至 256MB，本論文使用之開發板所提供之實體記憶體容量為 256MB。

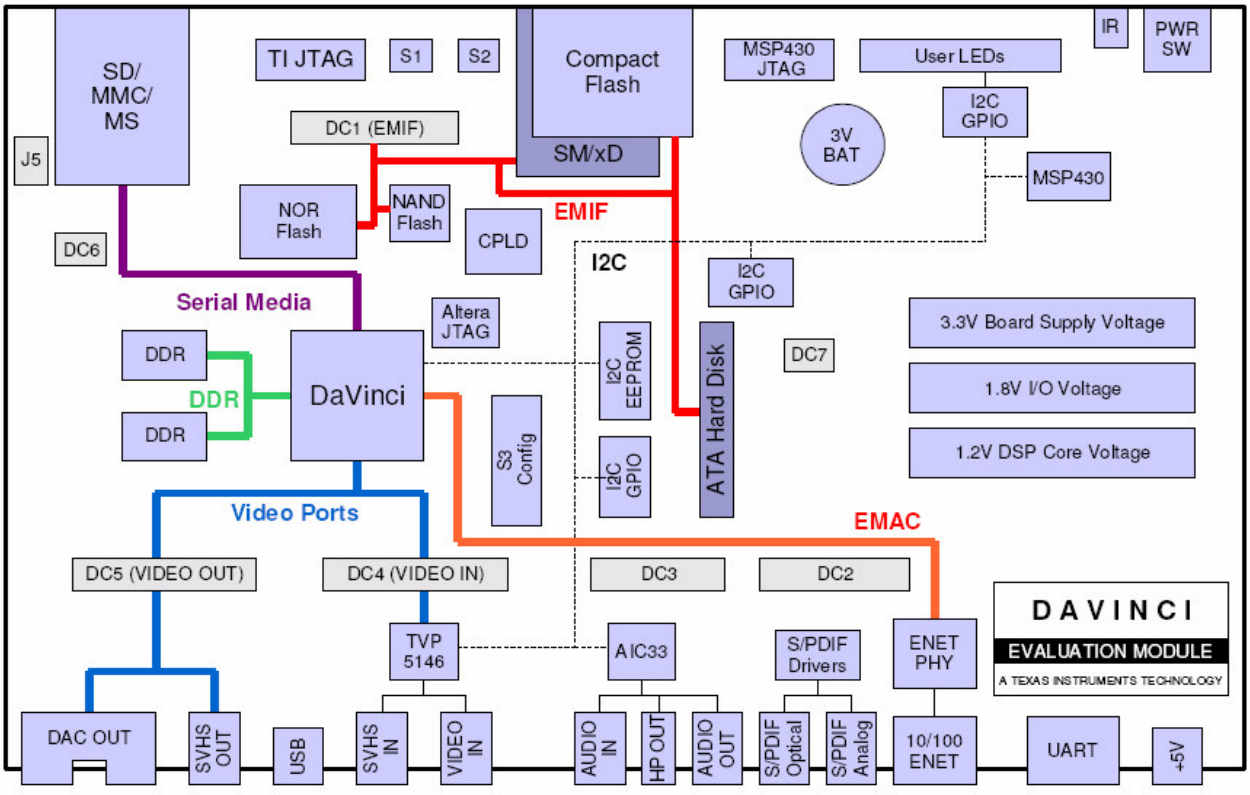

圖 7　TI DaVinci 開發板架構方塊圖

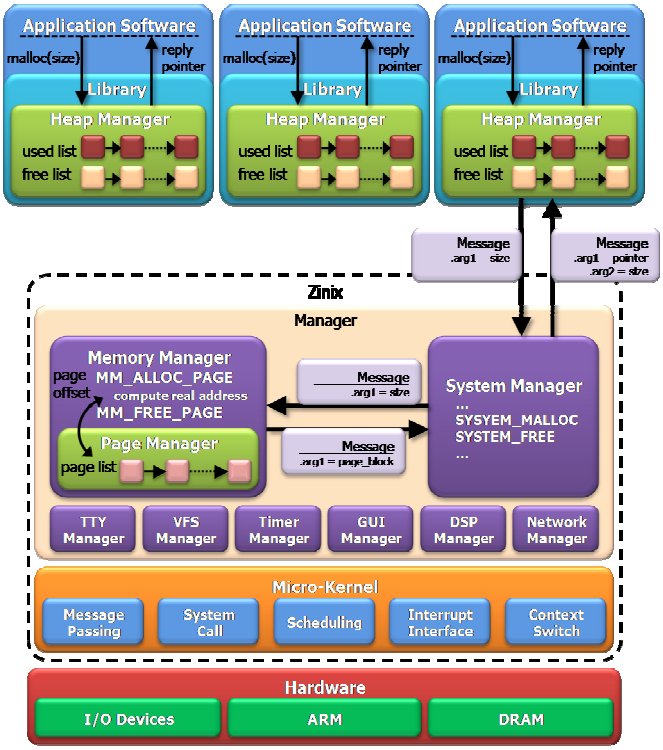

圖 8　Zinix 記憶體管理架構圖

本論文實作使用的作業系統為 ZINIX 微核心作業系統[1] 。ZINIX 微核心作業系統的架構分為四層，最底層為硬體層，往上分別為微核心層，管理者層與應用軟體層，如圖 8 所示，以下為各層之簡要說明：

1.硬體層：所有支援之硬體組成。

2.微核心層：負責之工作如下：

- 排程（Scheduling）
- 本文切換（Context Switch）
- 訊息傳遞（Message Passing）
- 啟動中斷處理函式（Interrupt Service-Routines）

3.管理者層：由伺服行程（Server Process）驅動程式行程（Driver Process）組成。按照微核心的定義，伺服行程與驅動程式行程是負責不同任務的使用者行程，在 Zinix 作業系統中並無明顯區別，稱為管理者(Manager)。

4.應用軟體層：應用程式可透過 API 呼叫底層管理者之各種功能完成工作。

Zinix 中之記憶體管理者（Memory Manager, MM）為伺服管理者之一，核心或伺服管理者有記憶體需求時都要對記憶體管理者發出 MM_ALLOC 的訊息傳遞，由記憶體管理者統一回應記憶體需求之請求。記憶體管理者之管理機制採用 Next-Fit Address Order 之設計。

提供給應用程式使用之函數庫方面，Zinix 支援動態記憶體需求與釋放之函數分別為 malloc() 與 free()，使用者程式可經由引用 include/stdlib.h 標頭檔使用此兩函數；每個使用者行程之 Heap 空間由各自獨立閒置區塊管理者（heap manager）管理，對於所取得之區塊管理機制為 Next-Fit Address Order。

四、客製化記憶體管理機制輔助工具

欲達到客製化記憶體管理，除了作業系統與函數庫提供多種機制可供選擇，更需要有其它方式協助選擇合適記憶體管理機制。若只有提供多種記憶體管理機制可供選擇而沒有其他訊息可供設定參考，必須完全依據使用者自己決定，只能稱為組態化（Configurable）記憶體管理。為了

達到客製化（Customized）記憶體管理，圖 9 所示為本論文提供客製化記憶體管理主要設計方式，分為兩部份：分析與選擇。在選擇合適記憶體管理前先分析應用程式動態記憶體需求情形，並以分析結果為選擇合適記憶體管理機制之依據。因此本論文設計與實作記憶體使用情形之分析程式，以分析應用程式在執行時針對動態記憶體需求情形，並產生相關設定檔以協助選擇合適記憶體管理機制。本節介紹分析程式所擷取的資料，分析的重點，與最後如何選擇合適的記憶體管理機制以達客製化；並說明分析與選擇操作流程與相關使用者介面。

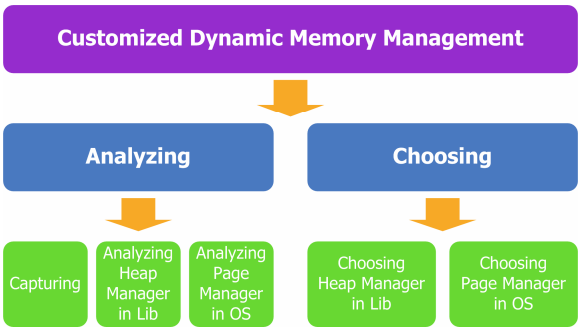

圖 9　客製化記憶體管理分析與選擇區塊圖

(一) 記憶體需求行為訊息之資料擷取

為了分析應用程式對記憶體需求的詳細情形，因此透過植入額外程式碼擷取應用程式對記憶體提出需求的時間點並記錄這些資料，如圖 10 所示。且本論文很重要的一點為儘量不修改應用程式本身的原始碼，因此植入的程式片段皆是在作業系統與函數庫之間，並使用條件編譯方式開啟或關閉其功能。因此選擇合適的植入點是首先考量的問題；因為植入之目的為擷取應用程式對記憶體需求的行為模式，所以植入點主要為跟記憶體管理有直接的關係的程式片段，並分為函數庫與作業系統兩部份。首先在作業系統中需要植入擷取應用程式對記憶體需求行為模式所考量之植入點，主要分成以下三部份：

1. 收到應用程式使用之函數提出記憶體需求訊息之時間點，表示作業系統行程管理者收到應用程式請求更多記憶體空間。
2. 行程管理者向記憶體管理者送出訊息表示需要實體記憶體頁面，而記憶體管理者收到訊息後會向其內部之頁面管理者索取記憶體頁面，頁面管理者找到合適的記憶體空間後，回傳給行程管理者。
3. 行程管理者收到從記憶體管理者回覆之訊息後，將配置的記憶體位址與空間大小回傳給應用程式之函數庫，以完成系統端的記憶體配置。

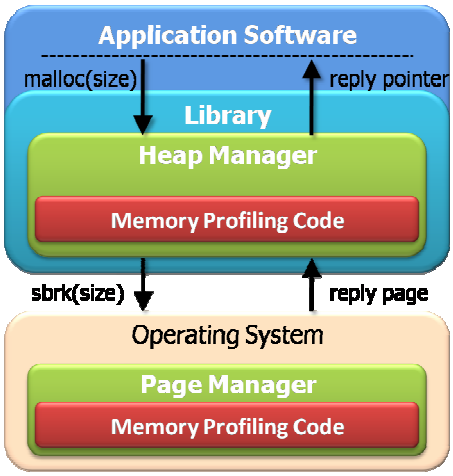

圖 10　加入擷取記憶體使用資訊概念圖

另外需要設立擷取應用程式行為的植入點為函數庫中有關動態記憶體的程式部份，稱為閒置區塊管理者，主要為 malloc()與 free()這兩個函數。當應用程式有動態記憶體的需求時，主要之步驟為：

1. 首先呼叫標準函數庫中 malloc()函數啟用閒置區塊管理者找尋可用記憶體區塊。
2. 若記憶體空間足夠則直接回覆應用程式。
3. 若沒有足夠的記憶體空間則透過訊息傳遞向作業系統索取頁面記憶體，之後分割頁面記憶體並回應給應用程式，其餘部分記憶體則由閒置區塊管理者管理之。

當應用程式不需要使用到先前得到的動態記憶體空間時，則是使用 free()函數釋放掉之前取得的記憶體空間。所以在標準函數庫中需要植入擷取應用程式行為的程式片段為此兩個函數。

為了以後分析記憶體使用情形所需要統計的資料，經過分析與研究參考文獻後，設計中所需分析之資料如下：

1. 應用程式的名稱及 PID；
2. 需求類型（配置記憶體或釋放記憶體）；
3. 需要配置的記憶體大小（只有在需要配置的情況）；
4. 應用程式提出需求的時間（開發板上之 tick 值）；
5. 函數庫實際配置的記憶體大小；
6. 應用程式目前要求的記憶體大小；
7. 目前記憶體管理配置給此應用程式的區塊大小；
8. 應用程式開始執行後累計提出要求的記憶體量大小。

(二) 分析擷取資料－函數庫之閒置區塊管理

圖 11 所示為分析應用程式之流程圖[3]。分析應用程式對記憶體需求之行為模式主要分為以下三個部份：

1. 讀入已從應用程式完整執行後，所擷取之資料並建立相關資料結構。
2. 分析執行過程之行為模式。
3. 根據分析結果選擇合適之記憶體管理方式。

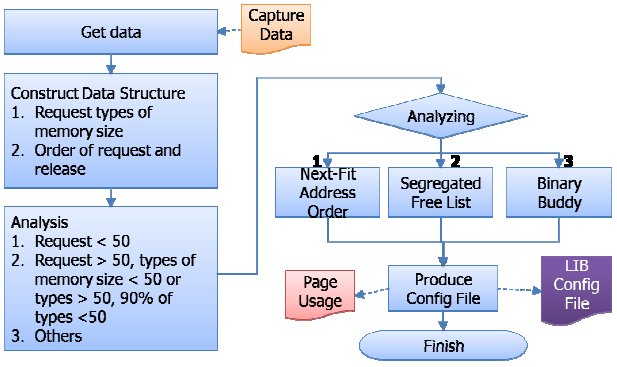

圖 11　函數庫閒置區塊使用情形分析之流程

分析程式除了會將結果另外存放於文字檔，還會產生供 Makefile 辨認之設定檔，作為編譯時連結合適的函數庫，如圖 12 所示[3]。另外產生頁面使用狀況的檔案供選擇作業系統所需的頁面管理程式之頁面分析程式使用。

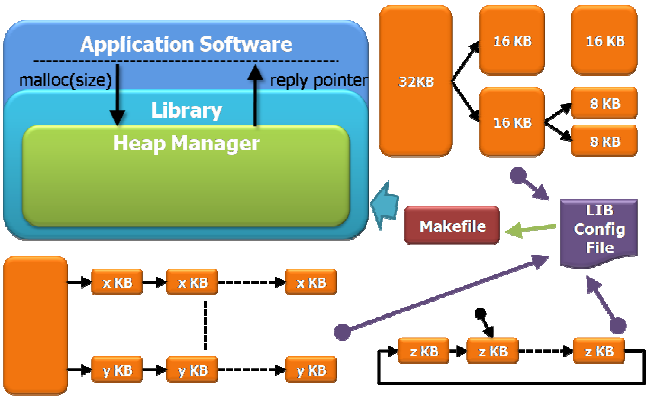

圖 12　函數庫透過編譯時決定記憶體管理機制

(三) 分析擷取資料－作業系統之頁面管理

當分析應用程式對記憶體需求的行為模式之分析程式完成後，另外將頁面使用情形整理成單一檔案，供爾後作業系統中頁面使用情形之分析程式使用，主要的分析流程如圖 13 所示[3]。首先讀入分析應用程式所輸出存放頁面使用相關資訊的檔案，整理所分析的應用程式所使用之頁面需求情形後，詢問使用者是否需要配置大頁面，如果沒有則設定 Segregated Free List，如果有則設定 Partition Allocator；如果頁面的需求類型種類超過八種以上，則設定 Binary Buddy。

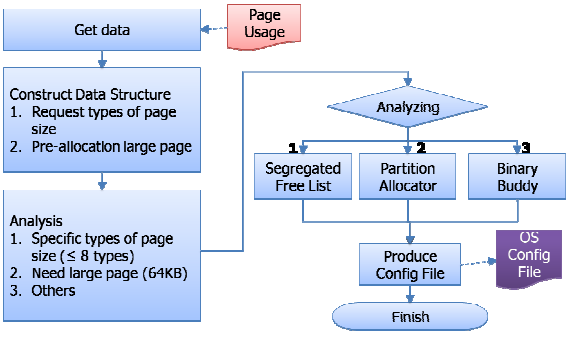

圖 13　作業系統頁面管理機制之分析流程

分析頁面程式最後會輸出 C 語言用的標頭檔與 Makefile 設定檔，C 語言用的標頭檔作為條

件編譯時使用，而 Makefile 設定檔作為要連結哪個頁面管理機制所編譯成的目的檔，如此完成所有分析的階段，如圖 14 所示。

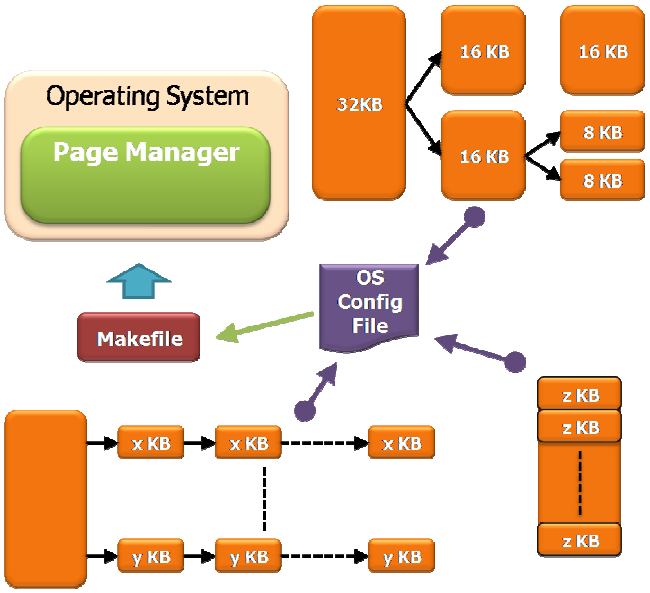

圖 14　作業系統透過編譯時決定頁面管理機制

(四) 客製化記憶體管理分析與選擇

操作流程圖 15 為客製化整體操作流程圖[3]，要選擇合適之記憶體管理方式主要分兩個部份：分析階段與選擇階段，分別是右半部份的上半區域與下半區域，左半部份主為分析前的擷取資料所需之操作流程。應用程式開發者需要瞭解的操作流程分為兩大步驟：

1. 分析應用程式對記憶體之需求：目的為收集應用程式對記憶體之行為模式，作為選擇合適記憶體管理依據。
2. 選擇合適記憶體管理：根據分析之結果，以選擇合適之記憶體管理方式達到更佳之記憶體管理。

五、效能分析

本節說明對上述實作之客製化動態記憶體管理機制之效能分析，包含：對應用程式使用動

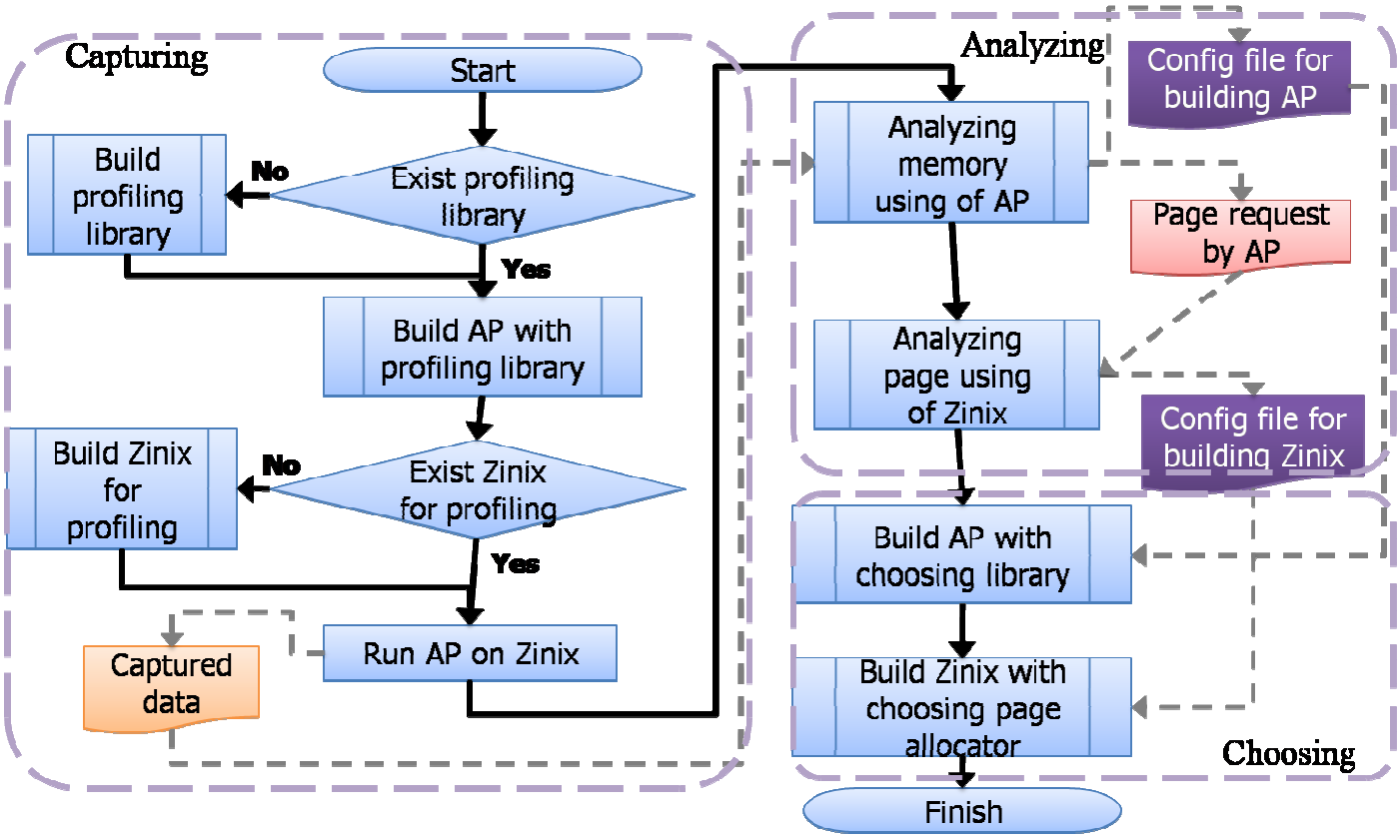

圖 15　客製化動態記憶體管理之整體操作流程

態記憶體行為模式分析程式之測試，應用程式使用此實作之效果等，目的在於測試並確認：

1. 函數庫之閒置區塊管理者可正常運作；
2. 作業系統之各頁面管理者可正常運作；
3. 記憶體使用情形之分析程式可分析應用程式記憶體需求狀況；
4. 作業系統頁面使用分析程式可有效分析頁面使用情況；
5. 應用程式根據記憶體使用情形之分析程式所選定之記憶體管理機制可讓應用程式執行時記憶體使用情形較預設情況佳。
6. 作業系統根據作業系統頁面使用之分析程式選定之頁面管理機制讓作業系統管理實體記憶體更有效率。

因此一過程中必須移植測試用之應用程式[3]，本論文使用為效能分析之測試程式為：計算 3D 貼圖程式與 H.264 Decoder。

(一) 測試程式一

Meteor 為一大量產生計算 3D 立體圖形中需使用幾個三角形貼成之工具與函數庫，主要性質為透過數學函數產生所需之立體圖形。

經過分析 Meteor.exe 所產生之輸出訊息後所得之動態記憶體使用情況，動態記憶體總共需求容量約為 16MB，不過系統實際配置之頁面記憶體卻有 26MB 左右，系統會額外多配置 60%左右的記憶體原因可能與 malloc 次數及特定記憶體容量大小有關。malloc 次數則高達 63 萬餘次，而且 57.7%的容量都集中在容量不大（以所介紹 Lea Allocator 之記憶體容量大小為區分）的記憶體需求[3]。函數庫之閒置區塊管理者為了管理每個區塊都需要額外的空間記錄相關資訊，這些資訊所佔用記憶體空間為 12 Bytes，且閒置區塊管理者是以此為單位配置記憶體，例如雖然測試程式所需之動態記體空間為 4 Bytes，不過在閒置區塊管理者配置後，需使用 24 Bytes 記憶體區塊配置，區塊之前面 12 Bytes 為記錄此區塊資訊之空間，後面的 12Bytes 真正只被測試程式使用的空間只有 4 Bytes，剩餘的 8 Bytes 則為內部斷裂。因此如果以 Meteor.exe 所需記憶體容量類型分析，其中 66%為只需 8Byte 的情況下，記憶體管理機制配置的區塊空間為 24 Bytes，在這之中的 50%為記憶體管理機制為了管理區塊而使用，且另外 16.7%為內部斷裂，因此這部份總共佔系統總配置的記憶體空間的 66%，從這邊可知為何系統會額外配置較多的記憶體空間給 Meteor.exe 在執行時期使用。

圖 16 為執行結果之比較，第 1 項與第 2 項之比較可明顯注意到執行時間大幅降低，效能增加了 60%，顯見在有大量動態記憶體之需求情況下，不同的閒置區塊管理者對於應用程式之執行有明顯之影響。第 2 項與第 3 項函數庫提供之閒置區塊管理者相同，但作業系統之頁面管理者不同，雖然只有提升 1%，原因在於向系統索取頁面空間時之過程需有兩次的訊息傳遞，即函數庫先送訊息至行程管理者，行程管理者再送訊息至記憶體管理者，中間損耗的時間不變，所以能提升的情況有限。若以第 1 項與第 3 項相比效能增加 61%，也驗證了本論文之設計針對動態記憶體方面有顯著改善，若之後處理器運算速度更快，記憶體管理相關部分之重要性將更加突顯。

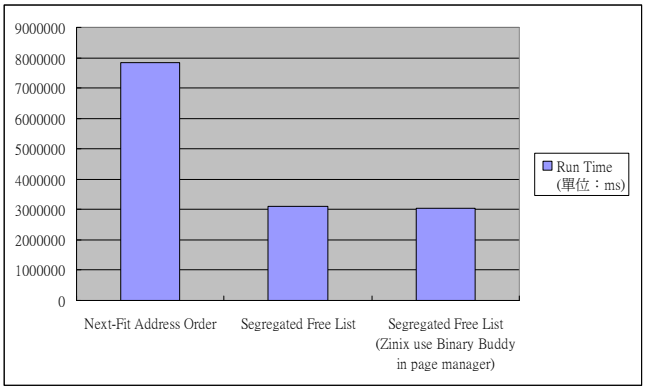

圖 16　Meteor 使用不同記憶體管理機制比較圖

(二) 測試程式二

分析 H.264 Decoder 動態記憶體需求情形之訊息檔案後，所得之動態記憶體需求之相關資訊如，可以看出總共提出需求與總共釋放容量很接近，另外 malloc 次數與 free 次數也很接近，表示 H.264 Decoder 在執行時期針對動態記憶體部分在使用過後有 99%記憶體使用量會再釋放出來，且在程式執行期間就會有釋放記憶體，因此雖然 H.264 Decoder 在執行時期總共提出需求量

接近 120MB，但實際上系統配置給 H.264 Decoder 用於動態記憶體部分不會超過 70MB；而動態記憶體使用的容量種類為 47 種，雖然每種類型記憶體容量之需求量不會遠大於其他記憶體容量類型，但是可明顯看出記憶體容量之類型則屬於比較多樣化且記憶體容量大小差異性很大，例如：8 Bytes 與 8MB 差異達一百萬倍[3]。

經過測試比較後，從圖 17 第 1 項與第 2 項可注意到只更換函數庫之閒置區塊管理者在效能方面雖然沒有很顯著的效果，即使第 3 項為更換作業系統記憶體管理者中頁面管理者後，大約只提升 3%之效能。其原因可能是測試的資料量約只有 100KB，導致要求記憶體的次數不多，加上在計算處理也用了不少的時間，因此可能造成效能沒有辦法有顯著的進展。而記憶體需求次數與資料量的大小成正比，故推測若資料量越大，則記憶體管理對於效能的影響越大。

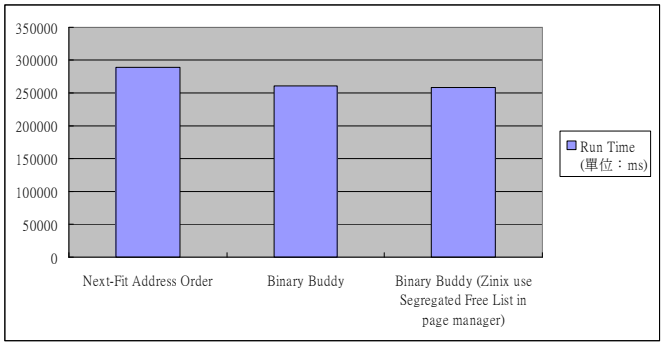

圖 17　H.264 使用不同記憶體管理機制比較圖

(三) 分析與討論

本論文使用測試程式為針對動態記憶體有大量需求之應用程式，並且先實際完整執行一次後取得動態記憶體的需求情形，主要分析的動態記憶體需求情形如下：

- 索取動態記憶體的次數
- 釋放動態記憶體的次數
- 提出動態記憶體要求時，記憶體需求的類型
- 針對提出需求時，實際配置記憶體容量大小

藉由索取或釋放動態記憶體的次數，瞭解測試對動態記憶體的依賴程度，次數越多代表在執行過程非常需要動態記憶體的使用。另外配置給測試程式的動態記憶體容量，通常會大於測試程式提出動態記憶體需求的容量大小，藉此知道真正使用記憶體的空間情形。

測試程式之一，Meteor，為計算 3D 立體圖形所需多少三角形構成之函數庫，其執行過程中對於動態記憶體的需求情形有以下特性[3]：

- 在執行過程中動態記憶體的需求次數非常大量，為 63 萬餘次。
- 動態記憶體需求容量的類型在本論文設定中屬於固定，本論文設定小於 50 種屬於固定需求類型，此測試程式只需 21 種類型。

分析工具根據以上所列特性自動選擇 Segregated Free List 為其函數庫中動態記憶體的管理機制，將常用的動態記憶體空間保留以加快下次的配置，故執行時間比原先的 Next-fit Address Order 縮短了 61%，可知在動態記憶體需求非常頻繁之情況下，管理機制對於執行時間有很大的影響。

測試程式之二，H.264 Decoder，其動態記憶體需求特徵如下[3]：

- 動態記憶體需求容量的類型屬於本論文設定之多種動態記憶體需求類型。
- 動態記憶體需求的容量大小差異頗大，最小為 8 Bytes，最大需求為 8MB 動態記憶體。

分析工具根據以上特性選擇函數庫中動態記憶體管理機制為 Binary Buddy，以盡可能保留較大的記憶體空間可供配置。由於 H.264 Decoder 在執行過程大部分時間屬於在處理畫面，及整個執行過程動態記憶體需求次數只有 227 次[3]，改變動態記憶體管理機制後此測試程式之整體執行時間只有縮短了 3%，可能原因為使用的測試資料量僅約為 100KB，若是資料量大則在處理過程中動態記憶體需求次數亦會跟著提升，故 Binary Buddy 取代原先使用的 Next-fit Address Order 確實有縮短時間的效果。

六、結　論

本論文設計與實作出透過分析應用程式對記憶體需求行為以提供合適之記憶體管理機制

達到客製化，因此系統更有彈性並可根據實際應用之動態記憶體使用情形，從多種記憶體管理機制中自動選取合適之記憶體管理者，藉著有效管理動態記憶體縮短實際應用之執行時間進而讓應用程式達到更好的執行效率。

最後透過 H.264 Decoder 與計算 3D 貼圖等測試程式實際驗證本論文實作之成果，從收集分析訊息以取得測試程式使用記憶體之情況為何，並根據測試程式特性選擇合適記憶體管理機制，藉此讓 H.264 Decoder 與計算 3D 貼圖等測試程式在動態記憶體的使用更有效率，達到在執行時間的縮短以加快取得執行結果，也證明本論文之設計與實作更貼近實際的應用。

誌　　謝

本論文之研究，承蒙行政院國科會補助部份研究經費，計畫編號：NSC 92-2220-E-006-002，NSC 98-2220-E-006-019，特此致謝。

七、參考文獻

[1] 洪文彬，“嵌入式微核心系統之設計與實作”，碩士論文，國立成功大學，2005。

[2] 戴顯權，陳瀅如，王春清，“多媒體通訊”，3rd edition，紳藍出版社，2006。

[3] 蕭光哲，“嵌入式系統之客製化動態記憶體管理”，碩士論文，國立成功大學，2009。

[4] “diet libc,” http://www.fefe.de/dietlibc/.

[5] “Dong Lea Memory Allocator,” http://gee.cs.oswego.edu/dl/html/malloc.html.

[6] “H.264/AVC Software Coordination,” http://iphome.hhi.de/suehring/tml/.

[7] “Meteor,” http://sourceforge.net/projects/meteor.

[8] “Newlib,” http://sourceware.org/newlib/.

[9] “TMS320DM644x Digital Media Processors,” http://focus.ti.com.cn/cn/lit/ml/sprt411b/sprt411b.pdf.

[10] “µC/OS-II,” http://www.micrium.com/products/rtos/kernel/rtos.html.

[11] “uCLibc,” http://www.uclibc.org/.

[12] “Windows CE,” http://www.microsoft.com/windowsembedded/zh-tw/products/default.mspx.

[13] D. Atienza, S. Mamagkakis, F. Catthoor, J. M. Mendias1, and D. Soudris, “Modular Construction and Power Modeling of Dynamic Memory Managers for Embedded Systems,” *Fourteenth International Workshop on Power and Timing Modeling, Optimization and Simulation*, pp. 510-520, 2004.

[14] D. Atienza, S. Mamagkakis, F. Catthoor, J. M. Mendias, and D. Soudris, “Reducing Memory Accesses with a System-Level Design Methodology in Customized Dynamic Memory Management,” *Workshop on Embedded Systems for Real-Time Multimedia*, pp. 93-98, 2004.

[15] D. Atienza, S. Mamagkakis, F. Catthoor, J. M. Mendias, and D. Soudris, “Dynamic Memory Management Design Methodology For Reduced Memory Footprint In Multimedia And Wireless Network Applications,” *Proceeding of Design, Automation and Test in Europe Conference and Exhibition*, pp. 532-537, 2004.

[16] E. D. Berger, B. G. Zorn, and K. S. McKinley, "Composing High-Performance Memory Allocators," *In Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation,* pp. 114-124, 2001.

[17] J. Bonwick, “The Slab Allocator: An Object-Caching Kernel Memory Allocator,” *In USENIX Summer,* pp. 87-98, 1994.

[18] J. J. Labrosse, ”MicroC/OS II: The Real Time Kernel,” CMP Books, 2002.

[19] M. Gorman, “Understanding the Linux Virtual Memory Manager,” Prentice Hall PTR, 2004.

[20] M. S. Johnstone, and P. R. Wilson, “The Memory Fragmentation Problem: Solved?” *International Symposium on Memory Management Proceedings of the First International Symposium on Memory Management,* pp. 26-36, 1998.

[21] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, “Dynamic Storage Allocation, A Survey and Critical Review,” *International Workshop on Memory Management*, pp. 1-116, 1995.

[22] Spectrum Digital Inc., “DaVinci EVM Technical Reference,” http://c6000.spectrumdigital.com/davincievm/revf/files/DaVinciEVM_TechRef.pdf, 2007.

[23] Y. Smaragdakis, and D. Batory, “Mixin-Based Programming in C++,” *Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering-Revised Papers*, pp. 163-177, 2000.