設計與實作一個適用於無線感測網路下雙節點系統之檢查點與重啟點機制

李育翰[1]
[1]國立中興大學資訊科學與工程學系
s9656007@cs.nchu.edu.tw

張軒彬[2]
[2]國立中興大學資訊科學與工程學系暨資訊網路與多媒體研究所
hpchang@cs.nchu.edu.tw

摘要—感測網路常肩負著長週期工作的使命，同時許多感測節點部署的地區可能面臨著嚴酷的環境考驗，容易導致節點在長時間的工作週期中產生錯誤而喪失功能。因此，本論文以雙節點架構為基礎，為感測網路實作了一個檢查點和重啟點的機制。除此之外，基於成本和電源的考量，感測節點上的微控制器通常不具備記憶體管理單元。因此，我們也為此平台設計了三種增量式檢查點的機制。在每一檢查點執行的週期裡，我們善用了 SANDBOXING 的概念來追蹤記憶體變更的區域。此三類增量式檢查點主要不同在於產生的檢查點檔案大小與系統執行時的額外負擔；產生較小的檢查點檔案隱含著有較大的執行負擔。

我們以實作本系統於 mica2 mote 感測節點上。從實驗結果，我們發現僅需不到 6.5 毫秒的時間就可完成所需的操作。因此，藉由本論文提出的機制，不僅能提升感測網路的可靠度，並且其執行時的負擔是極輕微的。

***關鍵字*: 檢查點/重啟點、容錯、感測網路**

***Abstract*—Wireless sensor networks are expected to work for a long period of time. Furthermore, lots of wireless sensor network applications are deployed in the rigorous environments. As a result, sensor nodes are usually prone to errors and would miss their function during their lifetime. In this paper, on the basis of dual-mote architecture, we thus implement a checkpoint/restart scheme for wireless sensor nodes.**

In addition, due to the considerations of cost and energy consumption, the microcontroller in each wireless sensor node usually does not have the memory management unit. Consequently, we also design three incremental checkpoint schemes in the MMU-Less sensor nodes. We leverage the idea of sandboxing to track the memory regions changed during each checkpoint period. The three incremental checkpoint mechanisms differ in the checkpoint data size and system execution overhead. The smaller the checkpoint data size is, the larger the execution overhead.

We have implemented our system in the Mica2 mote. For the experimental result, the operations take less than 6.5ms on the sensor node platform. Consequently, our proposed scheme can effectively increase the reliability of the sensor networks with neglected overhead.

Keywords: checkpoint/restart, fault tolerance, sensor networks

1. 簡介

現今許多無線感測網路[6]佈署的區域可能會面臨著嚴酷的自然環境考驗，例如：無人的荒野森林，氣候凜冽的炙熱地區…等等，進而導致感測系統硬體裝置的錯誤或毀損。對於某些重要的應用而言，如果感測結點無法順利運作，有可能會導致無法彌補的傷害或損失。

然而感測節點的微控制器上大都不具有記憶體管理元件，因此無法有效的在硬體的協助下對系統做保護的動作，無論是軟體邏輯上的錯誤或是硬體瞬間的訊號錯誤，皆有可能讓系統陷入無可預期的行為，輕則記憶體內的資料毀損，重則可能導致作業系統全面性的當機(kernel panic)，此時只能仰賴 watch dog timer 來重新啟動系統，但此時系統執行狀態已經流失。

有鑑於此，本論文目標為替感測節點設計專屬的系統狀態保存與回復的機制，系統狀態得以在任何時間點上復原，此外，我們也引入 dual mote 的概念，以雙節點系統的方式再度提升感測點工作的可靠度。最後，我們也提出在 MMU-Less 嵌入式環境平台上的增量式檢查點(incremental checkpoint)機制，並根據應用程式模組行為的不同，提出數種不同特性的增量式檢查點方法，希望能以減少電源消耗為依歸達

成長效執行的理想。我們已將上述機制實作在 mica2 motes 上。就我們所知，本系統是首先在無線感測平台上實現全系統狀態備份與重啟的設計。

在接下來的章節中，第二節會介紹傳統電腦在容錯方面相關研究。第三節的系統設計與實作中會詳細介紹本系統的架構及實作方法。第四節實驗結果會介紹實驗平台及設計以及相關實驗數據。最後則是結論及未來工作。

2. 背景

我們的研究是基於 SOS 作業系統，因此，在 2.1 小節我們會簡介 SOS 作業系統。而在 2.2 小節中，我們會介紹有關於檢查點與重啟點機制的相關研究。

2.1. SOS 作業系統

SOS[3]由一個共同的核心與多個動態載入模組構成所需的軟體運算環境。核心的部分實作了非搶奪式的優先權排班器、訊息傳遞機制、動態記憶體管理、動態應用程式模組管理等機制。

在排班器方面，SOS 為 event-driven 的排班方式且採取三層設計的優先權佇列。每層佇列內部以 FIFO 的方式來管理，CPU 在處理目前訊息所指派的工作時，將不被其他訊息所中斷，因此可能發生系統中許多高優先權的訊息在等某低優先權訊息完成其工作的情況。

核心和應用程式模組之間以 jump table 做為溝通的橋梁，當模組需要對核心的函式進行呼叫時，應用程式模組必須透過核心內的 jump table 來完成，jump table 記錄了核心內所提供的系統服務的函式位置。藉由 jump table，當系統核心升級時，只要 jump table 的結構沒有改變，應用程式也不需要重新編譯(re-compiled)。

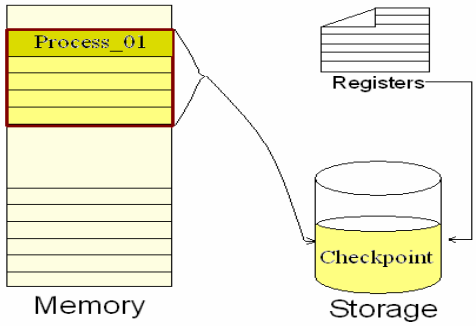
SOS 提供兩種應用程式模組間的溝通方式：非同步和同步。非同步的溝通方式是透過訊息傳遞，所有訊息都必須經由排班佇列，讓排班器(scheduler)將訊息從來源模組送往目的模組，以這種方式傳遞資料具有較高的延遲性。同步呼叫方式則是模組間彼此直接呼叫其所提供函式的機制。模組可透過 function registration，在核心管理的 function control block(FCB)結構的協助下，將自身的函式提供給別的模組呼叫，當模組想使用別的模組函式前，則需向核心做 function subscription，取得對方函式的進入點，以便以後執行時可直接呼叫函式，實現低延遲性的溝通的方法。

2.2. Checkpoint 相關研究

Checkpoint 機制[1-2, 5, 8]為增加系統容錯能力的多種方法之一，其主要的特點為回復正常執行環境於發生錯誤的時間點之前。因此只要硬體平台是無損的前提下，不管是任何無法想像或預期的系統錯誤類型，checkpoint 機制都能成功的修復我們的系統。一個設計良好的 checkpoint/restart 機制能夠支援我們所保存的系統資料，不單是在原本的主機平台上重新啟動，更可讓另一台主機也依需求藉由接收而來的資訊，更新成我們所需要的系統狀態。

Libckpt[7]為一著名適用於單核心處理器的 checkpoint 工具，它在 UNIX 環境下以不修改應用軟體的方式達到程式執行狀態的備份功能。當系統運行出錯時，可在核心的協助之下讓程式重新回復到備份時間點時的正常狀態。

在傳統的設計中，系統管理者會設定一個合適於工作平台的 threshold 時間，當時間遞減歸零時系統便會執行 checkpoint 動作，，如圖一所示，系統會將被 checkpoint 的程式其記憶體內容與暫存器的值等資訊備份到外部儲存體上，然後再重設時間計時器數值，等待下一週期的 checkpoint。



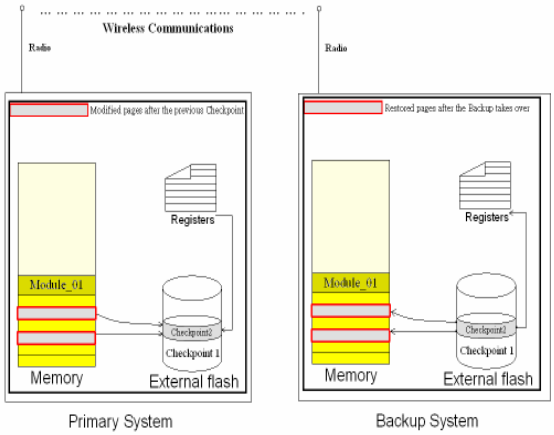
圖一： Traditional checkpointing in Libckpt

在 Libckpt 中，也提出以下幾種針對 checkpoint overhead 的優化方法：

1. **Incremental Checkpointing**: 每一次執行 checkpoint 時，跟上一次的 checkpoint 相比，兩次 checkpoint 間，可能只有一小部分的記憶體內容有經過變更。若我們每次都備份全部的記憶體內容，無形中便浪費了外部儲存體的空間，也間接影響備份動作的效率。因此在高階的硬體平台上，提出了使用分頁保護硬體裝置(page protection bits)，現今多內建於處理器的記憶體管理單元(Memory Management Unit)中，以一個分頁(4K)為單位，用來辨識兩次 checkpoint 動作間記憶體更動的部分，然後只儲存這些變更的部分到外部儲存體。
2. **Forked Checkpointing**:當我們替應用程式產生 checkpoint 備份檔案時，在傳統的架構下應用程式的執行將會被迫轉成閒置的狀態，等待 checkpoint 動作完成時才再回復執行，所以，當被備份程式的資料區間很龐大的時候，checkpoint 行為很容易造成系統整體執行上的效能不佳。因此如果平台上的實體記憶體足夠多，我們可先將程式的資料區間複製到記憶體的另一個區塊，接著 checkpoint 此一區塊的記憶體內容到外部儲存體，同一時間，應用程式也可以使用自己原本的空間。兩者交替著使用各自所屬的記憶體內容，減少了原本應用程式無法回應操作的延遲時間。
3. **User-Directed Checkpointing**: 為了能更改善 checkpoint 行為的效能，避免無謂的紀錄和寫入時間，第三種最佳化的想法為由應用程式設計師自行在程式碼中埋入我們所提供的函式呼叫，此函式呼叫能幫助系統在 checkpoint 行為時準確的抓取有意義的資料變數位址在予以備份，或是透過提供的函式自行啟動觸發 checkpoint 動作，如此一來可以讓 checkpoint 動作時間點更有彈性也更具實質意義。

3. 系統架構

在我們的設計中，嘗試將兩顆 mote 以無線傳輸方式連接組合成一組平台，如圖二所示。Primary system 週期性的將系統狀態送到 backup system，當 primary system 上的硬體毀損時，backup system 會接手 primary 系統上原本的工作，以增強系統運作時的可靠度。本論文對系統增強容錯能力的方法為適時的為系統建立檢查點和重啟點(checkpoint/restart)，並試著用類似 sandbox[4, 9]的技巧對記憶體存取進行追蹤，以達成 incremental checkpoint 的目標。而為了能更精確的量化需要保存的記憶體內容，我們提出了多個 heuristic incremental checkpoint 的方法，針對不同工作類型的應用程式模組，藉此相互評估最合適的 checkpoint 方法以達成低耗電的目標。



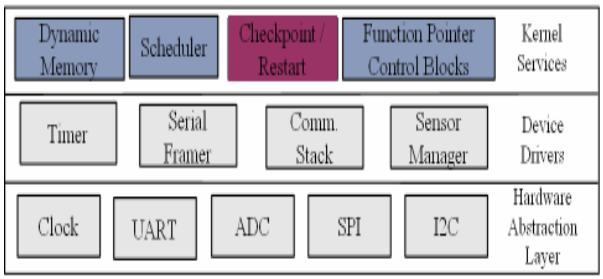
圖二：雙節點架構並以無線網路連接

3.1. 基本型檢查/回復實作架構

為方便我們所保留下來的系統狀態能便捷在兩個系統間相互的傳遞和重啟，在我們基本型 checkpoint/restart 的設計裡：checkpoint 和 restart 動作皆由同一個系統模組來負責，彼此共享同一連續區段的記憶體空間，如此一來可確保有裝載此模組的兩個節點系統彼此 memory Layout 是相同的，也同時有著 checkpoint 和 restart 的能力。使用相同的 memory Layout 不僅可以簡

化我們系統實作時的困難度，在記憶體的使用上我們也可以讓工作時的 checkpoint 和 restart states 使用重複的記憶體區段，以減少記憶體的花費。

如圖三所示，我們的 checkpoint/restart 機制在 SOS 作業系統中，採取以靜態模組的型式擴充 SOS 核心的部分，使其成為系統裡服務的一個元件，可週期性的對整個系統狀態執行 checkpoint 動作，然後儲存到位於 mica2 板上的外部 flash 裝置與以保留。因為我們是針對整個系統的狀態做備份，因此必須讓我們的 checkpoint/restart 模組在進入工作狀態之後，是處於不受作業系統和應用程式模組影響的情況，也就是說必須能獨立完成工作不需要作業系統的協助才行，以避免 checkpoint 的動作與作業系統彼此汙染相互的狀態，其達成的方法，下一小節會指出。

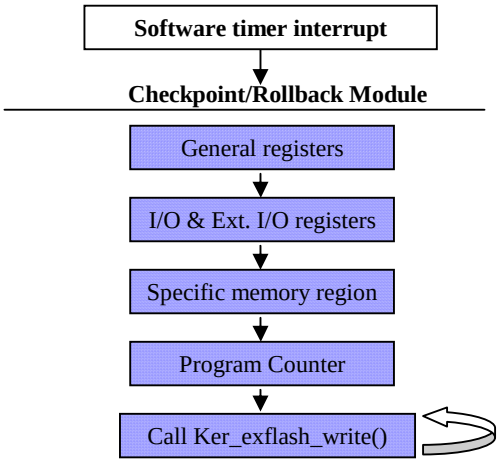


圖三：SOS 系統元件圖

在我們的機制裡，checkpoint 觸發時的邏輯與步驟如下：

1. 當我們所設定的週期時間到期時，SOS 的軟體計時器會產生 timeout 訊息到排班佇列中，等待 checkpoint/restart 模組的接收。
2. 再來即要確定系統內所有與外部 I/O 相關的模組服務是否都已經結束，才不會造成日後 restart 時系統資料不一致的情形。很幸運的，由於 SOS 採用 non-preemptive FIFO 的排班設計，所以當 checkpoint/restart 模組收到訊息開始執行時，則表示其餘的工作皆已告一段落。因此，checkpoint 運作的過程中，其餘模組沒有機會取得 CPU，也不會有機會再改變執行狀態。唯一可能造成系統狀態生改變的因素為外部中斷事件，所以要避免不必要的外部中斷發生，例如：外部的無線傳輸訊號…等等。做法上為在 checkpoint 動作的期間關掉所有的外部中斷只開啟 External Flash Interrupt，即可避免無謂的干擾影響我們 checkpoint 執行前原始的系統狀態。
3. 當備份完成後，重設相關的系統參數，離開 checkpoint/restart 模組返回排班器回復正常執行。

在 restart 的步驟裡，則為 checkpoint 的反向操作，但改採訊息傳遞的方式觸發，一旦 checkpoint/restart 模組接收到 rollback 的訊息，即立刻在當前的作業系統執行重啟的動作。



圖四：Checkpoint 的過程圖

3.1.1 Checkpoint 的實作步驟

在實作的部分，我們分成 Checkpoint 和 Restart 兩大部分來做要點的描述。在 Checkpoint 的實作方面，如圖四所示，我們分為四個步驟一一簡介。

一. 暫存器的部分

Checkpoint 一開始執行時，首先是暫存器的內容保存，在 AVR 的架構下暫存器分成三大類：

(a) General registers：32 個
(b) I/O registers：63 個
(c) Extended I/O registers：160 個

由於 (a)類暫存器為當前 MCU 運算最常使用的一群，在 avr 指令集中多數的指令操作都要仰賴此類暫存器的輔助，為避免接下來的運算改變其中的內容，所以在 checkpoint 一開始的階段就使用 PUSH 指令（因為 PUSH/POP 操作時不需要(a)類暫存器的協助即可把指定的暫存器內容存到記憶體中），將其保存到 stack 記憶體區中，接著連帶也將為(b)類暫存器上的狀態暫存器(SREG)保存進去，以確保系統狀態的完整。

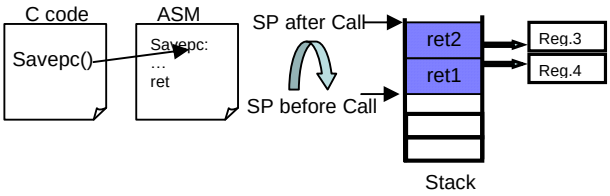
此時屬於(b)類的 stack pointer 也已更新完畢，我們將 stack pointer 保存在 checkpoint/restart 模組所屬的狀態記憶體區域(state memory area)裡。緊接著我們藉助 Avr-libc 函式庫提供的 memcpy()函式保存剩餘的暫存器(包含 I/O 暫存器、延伸 I/O 暫存器)資訊到 checkpoint/restart 模組的狀態記憶體區裡。注意，我們不將剩餘的暫存器放到 stack 中是為了不使 stack 區域過度的龐大，所以僅讓以 push 指令保存的 general registers 儲存到 stack 中，其餘暫存器(如：I/O Registers 和 Extended I/O Registers)則透過 memcpy()放到 checkpoint/restart 模組的狀態記憶體區域裡，以防止 stack 高度超過 256Bytes 的情況(Avr-libc 中 stack 預設的最大值)。

二. 特定的記憶體區塊備份

由於本設計的主要目標之一為整個系統狀態的完整保留，因此當 checkpoint 執行時，某些特定的記憶體區塊需另外備份，包含：

1. SOS 所提供的外部 Flash 讀寫模組(Exflash)的 .data section (35 bytes)
2. 系統 Stack area 當下的內容 – 取 256 bytes

我們也將上述兩個區域的記憶體內容暫存在 checkpoint/restart 模組的狀態記憶體區塊內。先對這些區域做內容複製是為了避免接下來的 checkpoint 動作會污染其原本的內容。這是因為 checkpoint 期間會委託 Exflash 模組來控制外部 Flash 的讀寫動作，因此，Exflash 模組的狀態會一直隨著時間改變，為避免備份到錯誤的 Exflash 模組狀態，故在 Exflash 模組動作之前先對其做備份。同樣的，在 SOS 的系統架構中，執行期間任何模組的函示呼叫都是使用同一塊 stack 區域，所以我們在呼叫 Exflash 模組提供的函數時，其執行期間所產生的區域變數也會置於此一 stack 中；再者系統的 Flash 中斷服務常式(Interrupt Service Routine)也會使用此一 stack 區域來完成中斷發生時所需的服務。這是因為我們外部 Flash 裝製讀寫動作的完成與否，需要仰賴中斷機制來告知作業系統。基於上述原因，所以我們也事先的為 stack 區域做備份。



圖五：擷取 Program Counter

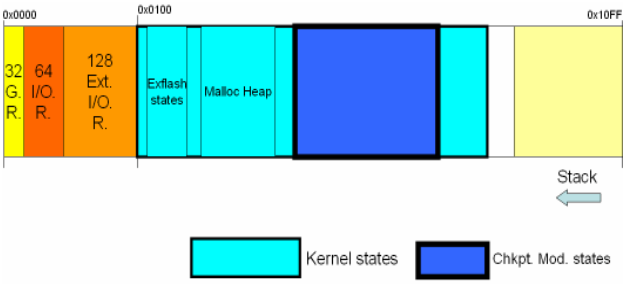
三. 保存 Program Counter 的內容

最後，我們保存 *program counter* 的內容，在 AVR 架構下有一點令人困擾的設計，就是我們無法從任何暫存器或特定的記憶體位址來獲得目前的 PC(Program Counter)值。為了達成擷取當前 PC 值，我們使用了一個小技巧，藉助了 Call 指令的特性，如圖五所示。我們實作一個由 AVR 組語所寫成的副程式，並且不傳入任何的參數值。當進入副程式後馬上將堆疊最上層的 2bytes 複製一份到暫存器中，此即為我們所需要的 PC 值。

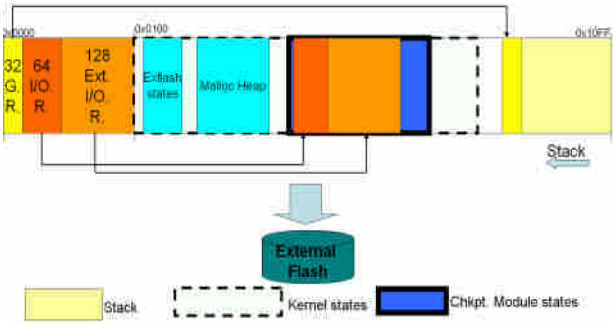
四. 備份記憶體內容到外部 Flash 記憶體

皆下來，呼叫 SOS 的 external flash 寫入模組(exflash)。對 SOS 本身而言，模組每服務一個訊息所花費的時間需要越短越好，若是停滯的太久會顯的 soft real-time 效率也極度不彰，所以其驅動程式模組的設

計上，也會盡可能一有等待 I/O 硬體狀態變化的空檔，便將 MCU 控制權交還給核心的情形，同樣的，exflash 模組的運作也是如此，每一次寫完一個 block 之後，exflash 模組會送出一個 message，將 MCU 控制權交還給核心。但是對我們接下來的寫入動作而言，這是不樂見的情況。在我們的 checkpoint/restart 模組將資料備份到外部 flash 的期間，我們希望整個系統狀態有所變化的地方越少越好，才越能正確的讓 checkpoint 模組備份整個作業系統的狀態。



圖六：The ordinary memory image



圖七：Memory image during checkpointing

因此，我們在 exflash 模組的內部新添加了一個變數 chkpt_switch，用以控制 exflash 模組在 checkpoint 發生後 execution flow 的範圍。由於外部 flash 裝置反應的速度遠小於我們 MCU 的執行，所以在此期間我們讓 MCU busy waiting，來等待 flash 裝置每一步的讀寫動作完成。我們就一遍又一遍的呼叫 ker_exflash_write()函式，來傳入全部的記憶體位址達成備份的目標，如此一來在寫入過程中核心記憶體狀態會改變的範圍就會被固定在 exflash 模組的區域和 stack area 這一區塊。

圖六為系統平時的記憶體影像，經由以上的步驟後，系統的記憶體影像會成如圖七分布。

3.1.2 Restart 的實作步驟

在 restart 整個系統狀態的實作中，有以下幾點要特別的注意：

1. 在 rollback 的過程中，當我們在做記憶體資料的回填時(將資料從外部 flash 複製到記憶體)要避開正在運作中模組的記憶體變數的區域，包含：external flash 和 heckpoint/restart 模組，才不會破壞我們 rollback 動作的執行。這是因為 rollback 的過程中需要 exflash 和 checkpoint/restart 模組的執行，如果將之前備份的資料蓋過這兩個模組的記憶體空間，會破壞這兩個模組在 restart 過程中的執行狀態。換言之，這兩個模組的狀態回復必須是最後的步驟。
2. 在填回 stack 記憶體區域的過程中，在程式的撰寫上要完全的避免使用區域變數，改採用全域變數的方式來運作，如此才能確保 stack 內容的無汙染與程式碼功能上的正確性，rollback 完成的當下各層 frame 資訊才得以維持，各種突發的作業系統呼叫也才能維持無誤的執行狀態，維持 checkpoint 模組執行前的架構。
3. 由於 AVR 架構沒有 PC 暫存器，故無法直接的將 checkpoint 時的 PC 值寫回，所以我們需要取巧其 ijmp 指令，我們將之前 checkpoint 保留下的 PC 值填入 Z 暫存器，再呼叫 ijmp，如此一來便有如重設 program counter 的效果。

下圖八為我們執行 rollback 時實作的流程圖。重啟系統的流程相似於 checkpoint 步驟的反方向進行，首先一開始在收到 restart 訊息後，我們調整 chkpt_switch 變數使系統進入 checkpoint/restart 模組壟斷執行的情況。一次次的呼叫 ker_exflash_read()函式把先前保存記憶體影像從外部 Flash 儲存體中

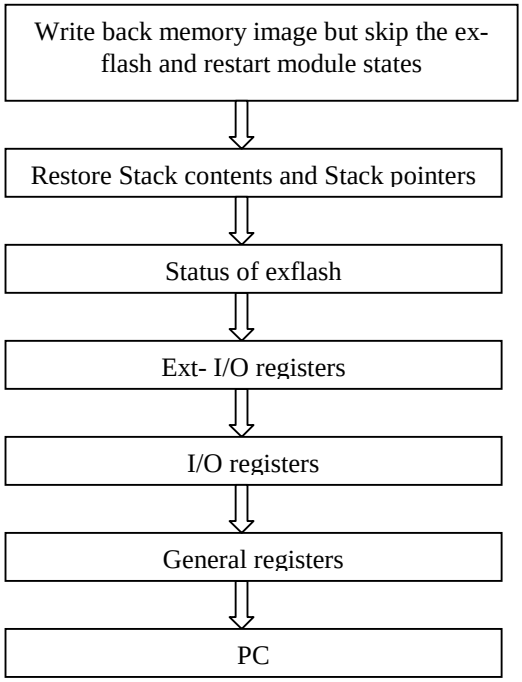
讀出，然後先寫入到 checkpoint/restart 模組的暫存緩衝區裡，開始判斷此次寫回記憶體的資料區段是否會影響 checkpoint/restart 模組和 exflash 模組的執行，略過會受影響的記憶體區間，其餘則直接寫入各自的記憶體區段。

此時記憶體影像全從外部 flash 讀出寫回主記憶體區後，下一步即開始著手先前特定記憶體區塊（即 exflash 模組）資料的回存。我們先回復 stack area 的內容，重設 stack pointer 的位置讓接下來如果發生函式呼叫的情況時 stack 能從正確的高度開始增長，所以接下來的記憶體操作皆可用 memcpy() 函式來進行。其次，我們從 checkpoint/restart 模組的 states 記憶體區裡將資料讀出，回復因為 rollback 過程導致變化的 exFlash 模組的狀態，並重新設定 chkpt_switch 變數。之後，我們回復置於 stack 中的 general registers 和其餘存在 checkpoint/restart module state 裡的 extended I/O registers，以重整目前的硬體平台狀態；然後，再從 stack 區裡 POP 出一般常用的 32 個工作暫存器將其回存還原，最後再更新狀態暫存器的內容，完成系統狀態的回存。

最後，我們要重設 Z 暫存器，透過 ijmp 跳回前次 checkpoint 函式結束前的程式碼附近，使的函式在結束前能自然地回收正確數量的區域變數，維護 function epilogue 執行時的完整，回收正確的 frame 大小釋放記憶體空間以確保 stack 結構的正確，再將 CPU 執行權交給排班器。

3.1.3 改進

基本型 checkpoint 在我們的設計實作當中，還可以對其執行效能做一個簡單的優化，在 SOS 作業系統的架構設計裡，有 2048 bytes 左右的 malloc_heap 空間，由核心模組控制其使用的情形，可讓系統中的其他模組執行動態分配記憶體，或是當模組動態插入系統的時候來存放模組的狀態，其大部分屬空白的記憶體內容；所以我們只要讀取其中的管理資料結構 BlockHeaderType 搭配 malloc_heap 的資訊便可以在基本型 checkpoint 執行時省去一部分無謂的備份動作。



圖八：Rollback Procedure

3.2. 增量式檢查點實作方法

為了在先天缺乏硬體 MMU 的情況達成 incremental checkpoint 的目標，以節省我們為數不多的 flash 空間，並使系統能在較省電的方式下執行狀態備份。在本篇論文中，我們也提出了三種不同的增量式檢查點實作方法。

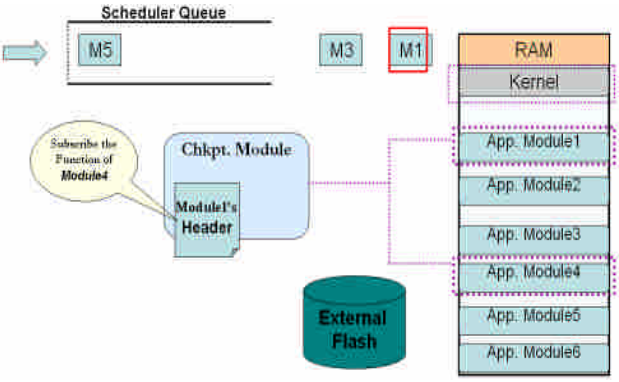
3.2.1 方法一：藉由檔頭資訊

首先我們分析欲執行模組間彼此的 function dependencies，具體做法為，每一個合法的應用程式模組皆有專屬的模組檔頭，以利動態連結時 Linker 取得所需的資訊。相似的，我們也可從中獲得此應用程式模組與其它模組的關聯，並將其紀錄追蹤。如圖九所示，在第一個藍色方框指出了所有欲訂閱位於其他模組裡的函式數目，第二藍色方框即為此模組欲使用之另一模組的函式編號和模組編號。當 Checkpoint 執行時，如圖十所示，我們便觀

察系統執行過(從排班佇列離開)的模組有哪些，再配合檔頭資訊找出其餘可能的相關聯模組記憶體區域，最後將其複製後寫入外部 flash 裝置裡。

```
Static mod_header_t mod_header SOS_MODULE_HEADER = {
    .mod_id    = MOD_FN_C_PID,
    .state_size = sizeof(fn_state_t),
    .num_timers = 1,
    .num_sub_func = 2,
    .num_prov_func = 0,
    .platform_type = HW_TYPE
    .processor_type = MCU_TYPE
    .code_id = ethons(MOD_FN_C_PID),
    .module_handler = module,
    .funct = {
        [0] = USE_FUNC_get_node_id(MOD_FN_S_PID, MOD_GET_NODE_ID_FID),
        [1] = USE_FUNC_set_led(MOD_FN_S_PID, MOD_SET_LED_FID),
    },
}
```

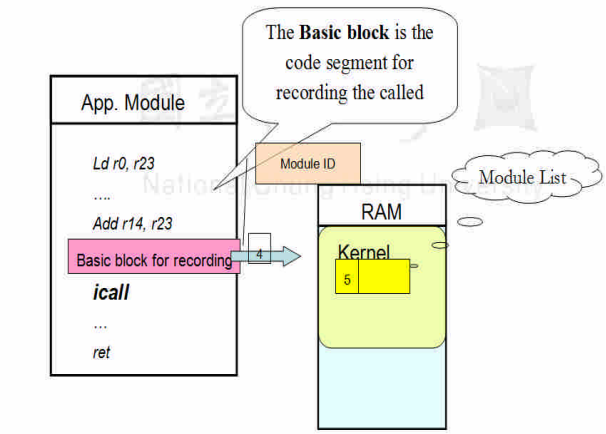
圖九：模組檔頭



圖十：方法一示意圖

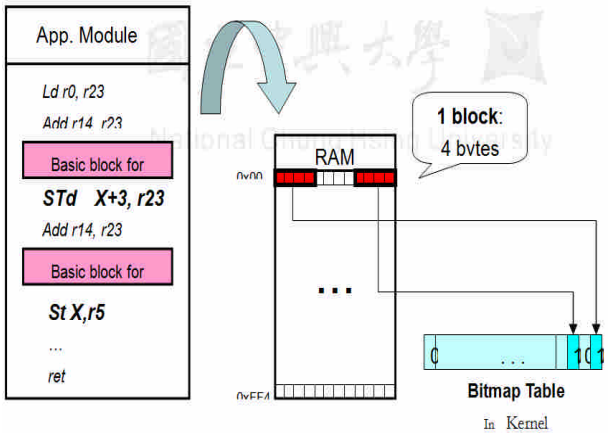
3.2.2 方法二：紀錄模組呼叫的指令

如 2.1 章節所提及模組和模組之間，可藉由同步呼叫(synchronous function call)的方式直接讓目的模組的函式開始服務，再者動態的同步呼叫也未必每次在模組執行時皆會發生。所以方法二追蹤動態同步呼叫的確切情形，如果在執行期間函式呼叫真的成立，我們便在 checkpoint 動作時，除了備份離開排班佇列(透過 scheduler 執行過)的模組的記憶體範圍，再藉由參考置於 kernel 中的 module_list 結構來備份經由同步呼叫執行過的模組記憶體範圍。



圖十一：方法二示意圖

如圖十一所示，在實作上我們可以捕捉和 icall 指令相關的變數內容，例如系統裡的 pid_sp 變數，以追蹤模組呼叫的狀況。再藉由參考置於 kernel 中的 module list()結構來備份經由同步呼叫執行過的模組記憶體範圍。



圖十二：方法三示意圖

3.2.3 方法三：紀錄記憶體寫入的指令

攔截欲執行模組程式碼中所有的記憶體更動指令，如: ST、STD、STS，以最精確的方式記錄 run-time 時會變更的記憶體位址。此策略最能確實的節省備份的資料量達到較佳的 incremental checkpoint 成果，但卻對模組的執行造成最高的負擔，也可能間接造成多餘的電力消耗。

在應用程式模組執行時，我們在核心處用如圖十二所示，一個 bitmap table 來紀錄記憶體修改的位址，我們攔截每一條寫入的指令，將其在 table 中所對應的 bit 處設

為真；為了減少 bitmap table 對記憶體造成的額外負擔，我們也可彈性的取 4、8…bytes 為單位形成一個 block 來做為每一個 bit 的對應單位。當 checkpoint 發生時我們再根據 table 的資料，對為真(true)的位址做記憶體內容的備份。

4. 實驗結果

我們實驗使用硬體平台為 Mica2[11]系列的 MPR400，其上載有 Atmega128L 微控制器(8Mhz)、無線通訊裝置及外部 Logger Flash(Atmel At45DB031B)。該 Atmega128L 控制器有 128Kbytes 的內部可執行 Flash 記憶體，4096bytes 的 Static RAM，還有具備 4Kbytes 的 EEPROM、眾多的外部 I/O 匯流排…等。本論文使用 programming board 為 Mib510。我們將 PC 端編譯好的系統核心和模組影像經由序列埠送到 Mib510 透過其上的 ISP(In-System Processor Atmega16L)協助燒錄到相接的 Mica2 內部 flash 記憶體。

實驗所使用的 SOS 版本為最終版的 2.0.1，SOS 系統本身採標準 C 的語法，搭配 Avr-libc 函式庫[10]的規範所建構而成。

表一： Code section size for blank SOS Kernel and our implementation (SOS_Chkpt)

Code(Bytes)	Raw		Optimization	
SOS	84038	-	44170	-
SOS_Chkpt	94232	+12.13%	50270	+13.81%

4.1 程式碼大小

在我們基本型檢查點的實作方面，在與原本 SOS 相比較，我們列出 text 區段的大小，結果如表一所示，我們分別列出在編譯時未採最佳化編譯策略(RAW 欄位)與開啟使用最佳化(Optimization)時程式大小上的增幅。

如表二所示，我們在 SOS 加入了基本型備份以及回復機制後，記憶體用量增加的部分主要為儲存某些會在 checkpoint 過程中改變的系統狀態，使用量上約增加了 15%。

表二： Memory overhead for blank SOS kernel and our implementation (SOS_Chkpt)

Memory	Bytes	
SOS	2950	-
SOS_Chkpt	3482	+15.2%

4.2 執行時間

此部份列出當我們基本型的 checkpoint/restart 模組執行備份和回復 4096 bytes 記憶體內容時系統的延遲時間，並分別以 Mica2 上所測得的 CPU clock cycle 和毫秒為單位來表示，我們取 16 次執行後所得的平均值，結果如表三、表四所示。

表三：執行檢查點時的系統延遲

latency	clock cycle	ms
checkpoint operation	41445	5.6213

表四：執行重啟點時的系統延遲

latency	clock cycle	ms
rollback operation	46492	6.3059

表五：備份檔傳輸時間

latency	clock cycle	ms
state transmission	626209	84.935

最後，我們也測量了主要節點和備份節點間，在傳輸並保存備份資料時系統需要付出的時間負擔。這當中過程的行為模式為：

1. 主要節點從外部 flash 將資料讀出
2. 透過無線通訊將系統狀態送至備份節點
3. 備份節點將資料寫入外部 flash 保存

同樣的，我們取 16 次執行後的平均值，結果如表五所示。

5. 結論

在本篇論文中，我們提出了一個適用於感測節點的 checkpoint/restart 機制，並在 SOS 系統上實現全系統狀態保留與回復。在雙節點的架構下改善了系統的容錯能力，增加無線感測節點的可靠度。我們也

提出了三種減少 checkpoint 資料量的策略，以減少系統需額外付出的運作成本。

在未來的工作裡，我們可以藉由 checkpoint/restart 狀態保存與復原的能力對系統做多次的狀態備份，再規劃一套判定運作中系統功能性正常與否的機制，使我們能提早發現不正常狀態的時間點；避免單一感測節點出錯後還繼續運作，送出不正確的資料給其它合作的節點，對感測網路整體的工作運行產生更嚴重的危害。

參考文獻

[1] E. N. Elnozahy, David B. Johnson, and W. Zwaenepoel, “The Performance of Consistent Checkpointing”, Proceedings of the 11th Symposium on Reliable Distributed Systems, pp. 39-47, 1992.

[2] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, and Fabrizio Petrini, “Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers”, Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, p. 9, 2005.

[3] Chih-Chieh Han, Ram Kumar Rengaswamy, Roy Shea, Eddie Kohler and Mani Srivastava, “A Dynamic Operating System for Sensor Nodes,” In Proceedings of the 3rd ACM MobiSys, pp. 163 – 176, 2005.

[4] R. Kumar, E. Kohler, and M. Srivastava, ”Harbor: Software based memory protection for sensor nodes”, Proceeding of the 6th International Symposium on Information Processing in Sensor Networks, pp. 340-349, 2007.

[5] Kai Li, Jeffrey F. Naughton, and James S.Plank, “Low-Latency, Concurrent Checkpointing for Parallel Programs”, IEEE Trans. on Parallel and Distributed Systems, Vol.5, No.8, pp. 874-879, Aug. 1994.

[6] K. Lorincz, D. Malan, T. Fulford-Jones, A. Nawoj, A. Clavel, V. Shnayder, M. Welsh and S. Moulton, “Sensor Networks for Emergency Response: Challenges and Opportunities”, IEEE Pervasive Computing, Vol.3 No.4, pp.16-23, October 2004.

[7] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li, “Libckpt: Transparent Checkpointing under Unix”, Proceedings of the Usenix Winter 1995 Technical Conference, New Orleans, LA, January, 1995.

[8] Golden G. Richard III and Mukesh Singhal, “Using Logging and Asynchronous Checkpointing to Implement Recoverable Distributed Shared Memory”, Proceedings of the 12th Symposium on Reliable Distributed Systems, pp. 58-67, 1993.

[9] Robert Wahbe, Steven Lucco, Thomas E. Anderson, Susan L. Graham, “Efficient software-based fault isolation”, Proceedings of the fourteenth ACM symposium on Operating Systems Principles, pp.203-216, December, 1993.

[10] The AVR-LIBC document: http://www.nongnu.org/avr-libc/user-manual/

[11] Berkeley mica motes, http://www.xbow.com/