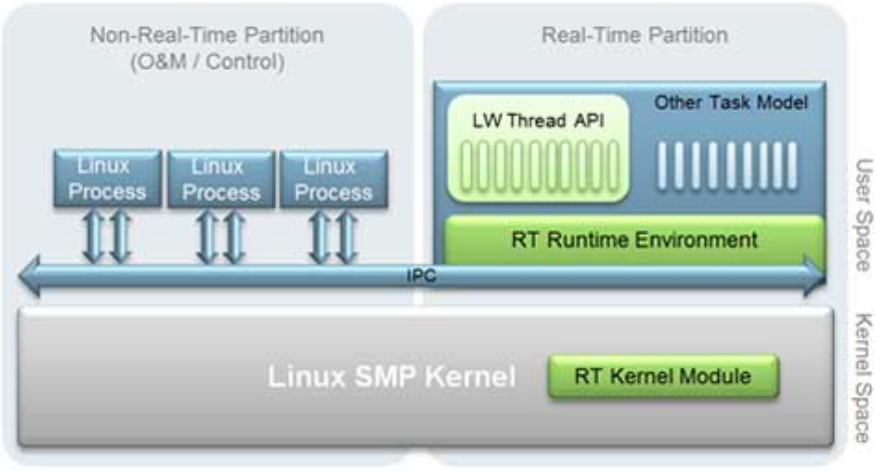
SMP Linux 即時效能分析和改進

簡介

本報告整理在 i.MX6Q Sabresd 硬體搭配 Linux PREEMPT_RT 種種即時效能分析和改進策略，我們希望除了掌握目前的研究進度外，也做為未來建構以 Linux 為基礎的即時系統的評估依據。

NXP 公司出品 (2015 年 NXP 併購 Freescale 後，繼承後者原本的 i.MX 產品線) 的 i.MX6Q Sabresd (以下簡稱 "i.MX6Q") 提供 4 核心 ARM Cortex-A9，我們將這樣的 SMP 架構依據需求，切割 SMP 運算資源為**即時**以及**非即時**兩個執行環境，讓即時任務在即時環境裡享有獨占的資源而不被非即時環境裡的事件所干擾，進而提昇即時任務的效能。整個系統架構圖如下：



- Real Time Partitioning

在 SMP Linux 環境裡，系統資源主要由以下兩類組成：

- CPU
- Memory & Coherence unit

而系統的事件主要有

- Interrupt
- system call

對於系統事件，我們主要探討這些事件發生時，對於即時效能的影響程度。

為了評估和量化整個系統的即時效能，我們導入一套 motion control 測試程式 (以下簡稱 "mctest")，作為量化效能的依據，從而控制變因去設計實驗，探討針對資源的區隔和上述系統事件帶來的衝擊。

mctest 效能評估

本程式取自真實環境中，機器人控制程式的演算法實做，得以在 Linux 作為獨立即時應用程式的形式運作，並用來系統的即時效能。在 motion 的運算過程中，我們希望能夠減少被打斷 (preempt) 的機率，我們進一步改寫主體程式碼，使其得以在 Linux 核心模式 (kernel space) 運作，作為一個 LKM (loadable Linux kernel module)。

以下是預期的執行時間：

- 單次 motion 運算若沒受到其他因素的影響，執行時間應**小於 10 us**
  - 通常為 6 us，已開啟 gcc 的 ARM NEON 最佳化

- 單次執行時間**大於 10 us** 者，我們定義為**超時**，代表此次 motion 被外部因素拖慢
- 首度執行 mctest 之際，我們會測量到遠大於後續的執行時間，前者往往大於 30 us。推測應是剛執行 motion 程式時，伴隨著大量的 cache miss 所致，我們稱為 "cold start"。在報告中，我們略去所有 cold start 的數據。

系統資源

CPU資源的區隔

資源區隔的作用是為了降低即時任務的外部干擾，並試圖縮減即時環境內的 context switch 發生次數，為此，我們可透過 SMP affinity，將 CPU 資源區分。舉例來說，透過 kernel command line 可設定 `isolcpus`，指定從核心排程中希望排除的 CPU 清單，所以，`isolcpus=3` 即代表將 CPU3 從核心排程裡移除，一旦該 CPU被從移除後，除非特別指定 (例如透過 taskset)，否則不會被排到任何 task。

為了行文的便利，我們將這樣因為即時處理需求而隔離的 CPU 為 "isolated cpu"，而且為了區分 kernel 和 processor core，中文的「核心」特指作業系統核心，而後者則簡稱為 "core"。

Memory & Coherence Unit

在 SMP 環境下，我們無法對記憶體資源做實體區隔。因此，我們要探討存取記憶體資源時會造成即時效能的影響。在典型 ARM 架構中，I/O是採用 memory mapping 的方式，因此，I/O 的存取也可視作記憶體存取。在下圖可見，我們使用的 CortexA9 MP-core 之間共享 L2 cache，並受到 SCU 的監督以及管理。

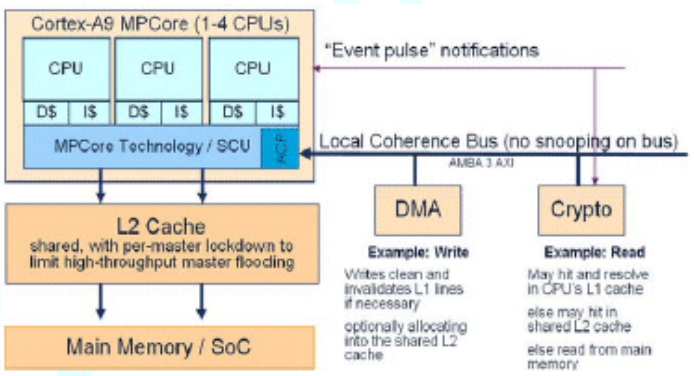


Fig. 3. Accelerator Coherence Port

- Memory & SCU in CortexA9

實驗目標: 探討前述 motion 的執行過程中，lock 及週邊 I/O 的存取對效能的影響

- 說明: 原本的 mctest 實做中，進入和離開 motion 演算法的時間點都追加了 GPIO 操作，用以觀察 memory coherence 的影響，但我們發現 GPIO 的操作會影響效能。
- 實驗數據

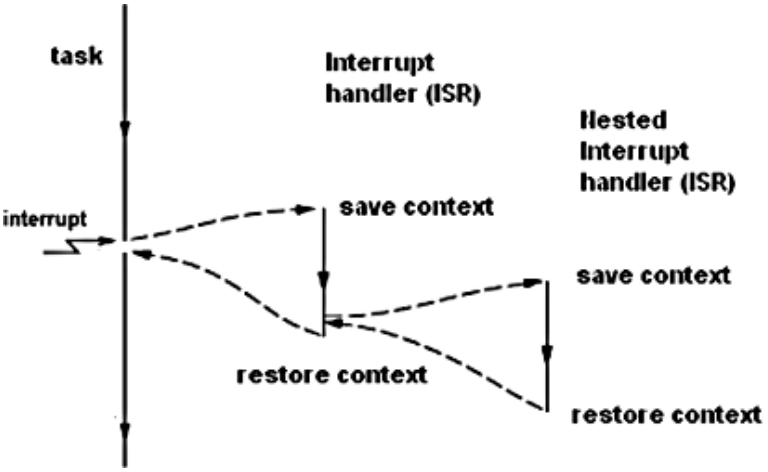
  M1: 原本的 mctest 程式
  M2: 保留 GPIO 存取的函式，但去掉lock/unlock
  M3: 保留 GPIO 存取的函式，但使用 raw_spin_lock_irqsave/raw_spin_unlock_irqrestore 取代原本的 spin_lock_irqsave() / spin_unlock_irqrestore()
  M4: 保留 GPIO 存取的函式，改用 loca_irq_save/local_irq_restrore
  M5: 移除所有 GPIO 存取

	超時次數(>10us)	最大超時
M1	41	12.67us
M2	7	11us
M3	19	11.67us
M4	6	12.67us
M5	1	10.33us

- 實驗結論: 由上可見，lock 及週邊 I/O 存取都會拖慢 motion。Lock 的存取涉及到其他核心之上程式競爭關係，而週邊 I/O 存取則會致使 memory-mapped 區域的存取，進而增加在 AXI Bus 上與其他核心競爭的機率。

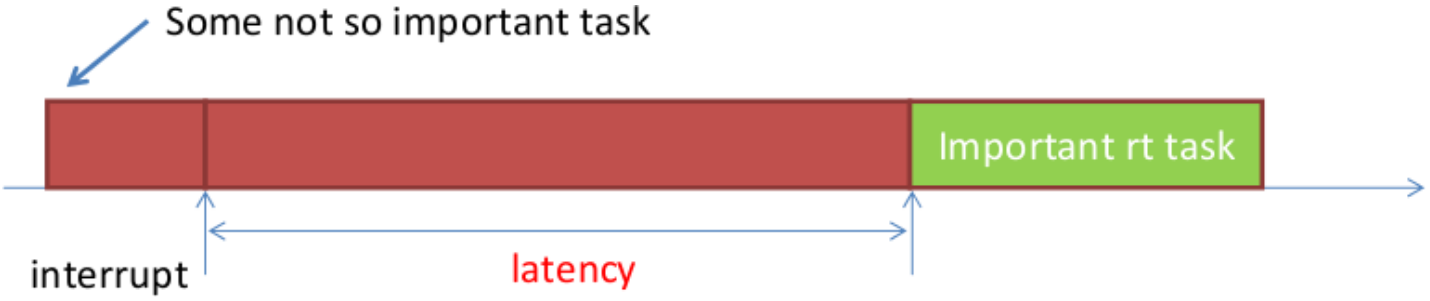
- **未來研究方向**:
  - 降低實體記憶體存取次數。對於共用資源，譬如 lock，我們可以考慮 LWRT 裡提到的方向，使用 shared memory 來增加共用資源的 locality
  - 借鏡 kdump 保留記憶體的方式，實現記憶體隔離

系統事件



Interrupt 事件

在事件驅動的設計中，interrupt 的觸發往往會影響整體即時處理的效能，為此，我們必須確認 interrupt latency 和 duration 的時間開銷是可預測的 (deterministic)。對於 non-critical Interrupt (不是做為即時任務的觸發事件)，我們應當降低這些 interrupts 帶來的衝擊。



round-robin 一類的排程演算法仰賴系統提供穩定且微小的時間間隔 (time-slice 或 quantum)，也就是週期性的 tick interrupt，由過往的經驗指出，量化 tick interrupt latency 有助於找出即時處理的效能瓶頸。我們設計實驗去計算 tick_program_event 設定的時間和 High Resolution Timer (HRT，以下均簡稱為 HRT) 實際發生的時間之間的差值，測量結果指出，在 i.MX6Q 硬體上 tick interrupt latency 平均為 2.336 us，但最差的情況可達 13.67 us。

IRQ Affinity

我們可透過 Linux 核心提供的 IRQ Affinity 設定，將 non-critical interrupt 設定給非即時環境的 CPU 處理。例如:

- **echo 2 > /proc/irq/90/smp_affinity**

就是將 IRQ 90 設定給 CPU1 處理 (bit mask: 0b0010)

Tickless kernel

自 Linux 核心 3.10 開始，引入新的 full tickless operation 機制，降低週期性觸發的 Tick 對系統的影響。原本週期性 timer interrupt 對即時系統的衝擊主要是以下兩項：

- tick / timer interrupt 在 SMP 環境下，每顆 core 都必須處理和執行 ISR
- 打斷高優先權的即時任務的執行，因而降低即時效能

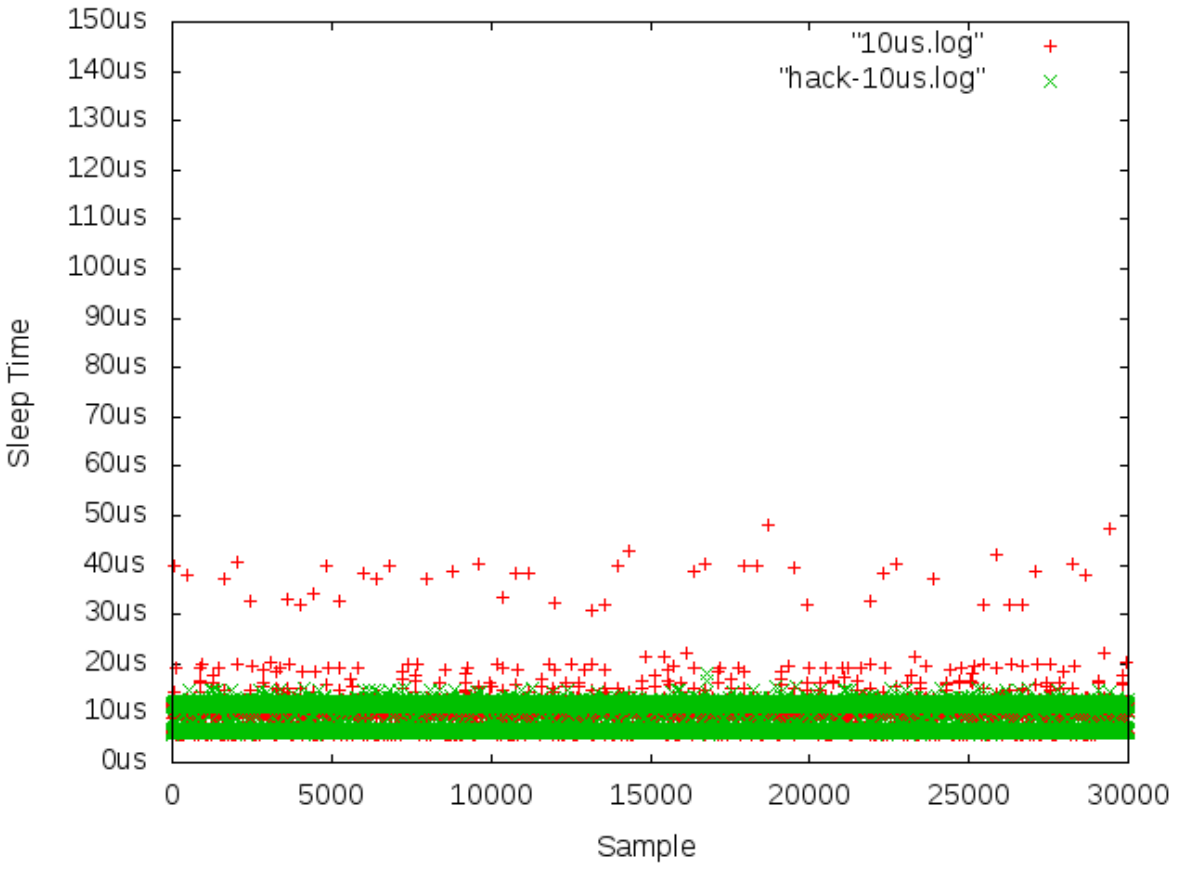
一旦 full tickless 核心組態開啟後，可在 isolated cpu 中大幅降低 tick / timer interrupt 發生的次數。對應的 Linux 核心設定如下：

- Kernel Configurations:
  - CONFIG_NO_HZ_FULL
  - CONFIG_RCU_NOCB_CPU
  - CONFIG_RCU_NOCB_CPU_ALL
- Kernel Command Line: `nohz_full=3 isolcpus=3 nohz=on`

  由於 CPU3 被標注為需要隔離，CPU3 在一般狀況下將不會被 Linux 排程器排入任務，當然也就可以充分運作 tickless。

開啟 tickless 組態後，要留意其行為與 PREEMPT_RT 會有衝突，詳情可參考郵件論壇上的討論。目前的暫時解決方案是，將 timer lock 從 spinlock (在 PREEMPT_RT 中，spinlock 本質上就是rtmutex) 變更為 raw_spinlock，這作法的後遺症是會帶來 timer latency 的增加。

綜合前述，我們設計透過 tickless 核心組態來避免Tick Interrupt 影響的實驗。結果如下圖，HRT 週期設定為 10 us。圖片中紅色標記沒開啟 tickless (預設組態) 時，最大的 latency 達到 50 us，而綠色標記則是開啟 tickless 核心組態，latency 收斂於 20 us 之內。



System Call 事件

當我們使用 lmbench 一類 Linux 常見效能測量 (benchmarking) 的工具，會發現 i.MX6Q 環境中出現頻繁的延遲，即使已引入 isolated cpu 來隔離系統資源。深入觀察後，發現在非 isolated cpu 環境執行特定 system call (系統呼叫，以下簡稱 syscall) 之際，isolated cpu 上高優先權即時任務會因此被拖慢回應時間，為此，我們設計實驗，在其中 3 core 中執行參考 lmbench 內含 syscall 的大量案例 (instance)，並指定在最後一個

core 裡頭執行 mctest，倘若我們統計 mctest 超時次數，即可推測受到 syscall 的影響也愈大。

為了排除 memory coherence 的影響，我們對照測試中的 GPIO 操作的處理和消除狀況，進行 10,000 次 mctest 的執行後，可得到以下數據：

- Motion w/o GPIO Access

	超時次數(>10us)	最大超時(us)
fstat	0	*
getppid	0	*
gettimeofday	0	*
open	22	11.33
read	1	10.00
stat	0	*
write	1	*
fifo	0	*
pipe	0	*
sem	1	10.00
signal handler install	0	*
signal handler catch	0	*
fork	8450	30.67
exec	4530	26.33

* 代表未超時，不列出

- Motion with GPIO Access

	超時次數(>10us)	最大超時(us)
fstat	64	12.66
getppid	103	13.67
gettimeofday	87	12.67
open	164	14.67
read	91	13.33
stat	0	*
write	76	14.00
fifo	103	13.33
pipe	109	13.33
sem	101	13.00
signal handler install	76	12.30
signal handler catch	71	14.30
fork	7651	42.33
exec	6552	36.00

* 代表未超時，不列出

由上述實驗結果可進一步得知：

- 進行 GPIO 操作，連帶要考慮 memory coherence 時，會受到非 isolated cpu 的 syscall 所影響，尤其以 `open` 造成的影響最大
- 一旦消除 GPIO 操作後，memory coherence 因而減少，此時僅有少數 syscall 會造成效能影響，如 `open`

結論以及未來研究方向

本文探討 Linux PREEMPT_RT 核心組態在 SMP 環境下的即時效能影響因素。首先我們藉由 SMP affinity，在特定的 CPU 上隔離即時任務，以降低非即時任務帶來的衝擊，過程中我們發現由於記憶體由若干 CPU 所共享，Linux 並沒有進行實體區隔，於是就伴隨著 memory coherence 的時間成本，進而導致最差情況下延遲時間 (worse case execution time) 的增加。另外，我們也發現頻繁的系統呼叫實際上也會影響到即時效能，即便系統呼叫不發生於隔離的即時環境。為此，我們提出兩項效能改善機制：

- 透過啟用 tickless 核心組態，降低即時任務所在 CPU 受到 timer 中斷的影響
- 分析個別系統呼叫對即時效能的影響，並予以分類，之後可透過特定的程式開發模型來降低效能衝擊

為了進一步提昇即時效能，我們必須更細緻地切割 SMP 環境中所有系統資源，於是針對前述 memory coherence 的議題，有以下兩項解決方案：

- 在 shared memory 的基礎上，對於資料的存取方式做 locality 的最佳化，以減少需要進行 memory coherence 的機會
- 借鏡 kdump 保留記憶體的方式，保留不會被虛擬記憶體對應到的實體空間，特別保留給即時任務使用