如何最佳化多核心作業系統的擴充性與效能

在 SMP 平台上設計可擴充的即時應用程式

IntervalZero

摘要

當硬體平台升級到更新、更強、且擁有更多、更快核心的 CPU 時，我們會期望應用程式能跑得更快。理論上，更多的核心應該可以降低平均 CPU 負載，進而減少延遲。然而在許多情況下，應用程式並沒有跑得更快，CPU 負載還幾乎和舊的一樣。使用高階 CPU 的時候，甚至可能會出現一些破壞系統準確性的干擾。為什麼會這樣？又能怎麼改善呢？

答案很簡單：從一開始就為「可擴充性」而設計。除非應用程式在架構上被刻意設計成充分利用多核心環境，否則大多數 RTOS 應用程式在單核心與四核心 IPC 上的效能幾乎相同 — 這和大家預期的「RTOS 應用程式應該能線性擴充，在四核心 IPC 上的執行速度應該是單核心的四倍」剛好相反。如果系統不具備擴充性，四核心系統裡會有三個核心沒被用到。就算應用程式嘗試使用多核心，還是必須考慮記憶體存取、I/O、快取策略、資料同步等其他架構的最佳化，才能讓系統真正達到最佳擴充性。

儘管沒有任何系統能提供線性的擴充能力，我們仍然可以盡量讓每個應用程式逼近理論極限。這篇技術文章將說明幾項關鍵的架構策略，確保以 RTOS 為基礎的應用程式能獲得最佳擴充性。我們會探討 CPU 架構，解釋為什麼在使用更新或更強大的核心後，效能卻無法如預期提升，說明如何減輕干擾造成的影響，並提供硬體調整的建議以降低系統瓶頸。

介紹

本篇是為了 RTOS 使用者而撰寫的白皮書，探討即時與非即時應用程式同時在同一系統上執行的情境。為了維持即時應用程式的準確性，在理想情況下不應該和非即時應用程式共享任何硬體。但同時，如果雙方都能使用記憶體空間和同步事件，對整體系統還是會有所幫助。

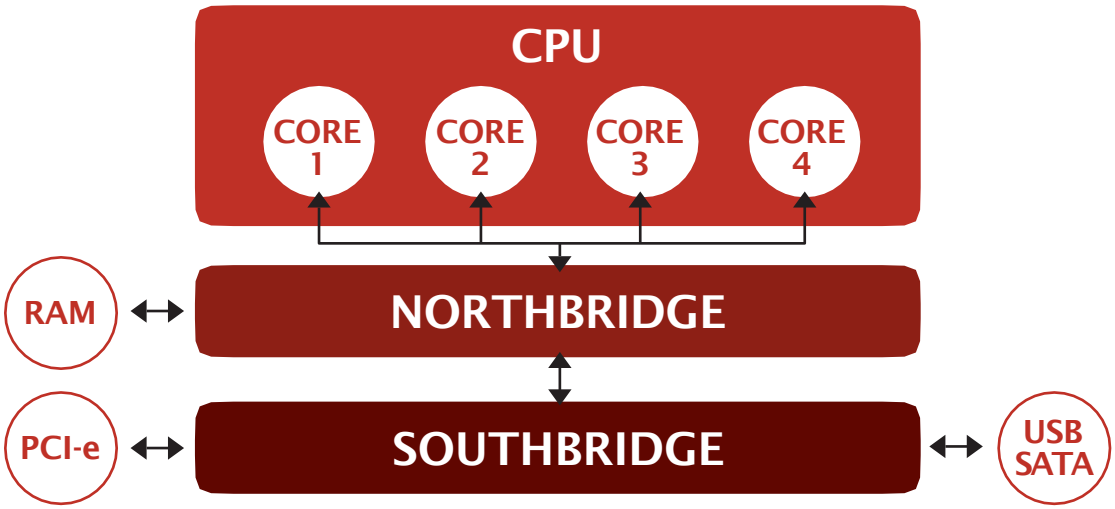
然而，這兩個目標是無法同時達成的。我們要不是使用一台專用的即時電腦，且必須透過匯流排協定來和非即時應用程式交換資料，不然就是讓兩種應用程式跑在同一台機器上，但勢必會共享 CPU 匯流排和快取。現今 CPU 核心的速度遠快於記憶體和 I/O 存取，因此干擾主要是來自於對這些資源存取時的競爭。

另外，在使用多核心時還有一個重要的考量點。應用程式中的不同執行緒通常會共用變數，因此必須同步這些變數的存取，才能確保數值的一致性。如果這件事在程式碼中沒有被處理，就會由 CPU 自動執行；但由於 CPU 並不了解整個程式的全貌，無法以最有效率的方式處理，因此會產生許多延遲。而這些延遲就是應用程式在兩個核心上不一定會比在單核心跑得快的原因。

這篇文章將會先探討與快取、記憶體以及 I/O 存取相關的 CPU 架構。然後再說明執行緒之間如何互動，和程式設計能如何幫助改善在多核心環境中的效能。最後，會提供實際問題的範例，並介紹可用來解決或將影響降到最低的方法。

本篇大多數的技術內容是根據 Red Hat 的 Ulrich Drepper 所撰寫的優秀文章《What Every Programmer Should Know About Memory》而來。如果時間允許，我們也建議閱讀該篇文章。

1. CPU 架構



1.1. 傳統架構: UMA (Uniform Memory Access，統一記憶體存取)

在這種架構中，所有核心都連接到同一條稱作前端匯流排（Front Side Bus，FSB）的匯流排上，這條匯流排再連到晶片組的北橋（Northbridge）。RAM 以及記憶體控制器也連接到同一個北橋。其他所有硬體則連到南橋（Southbridge），而南橋再透過北橋連接到 CPU。

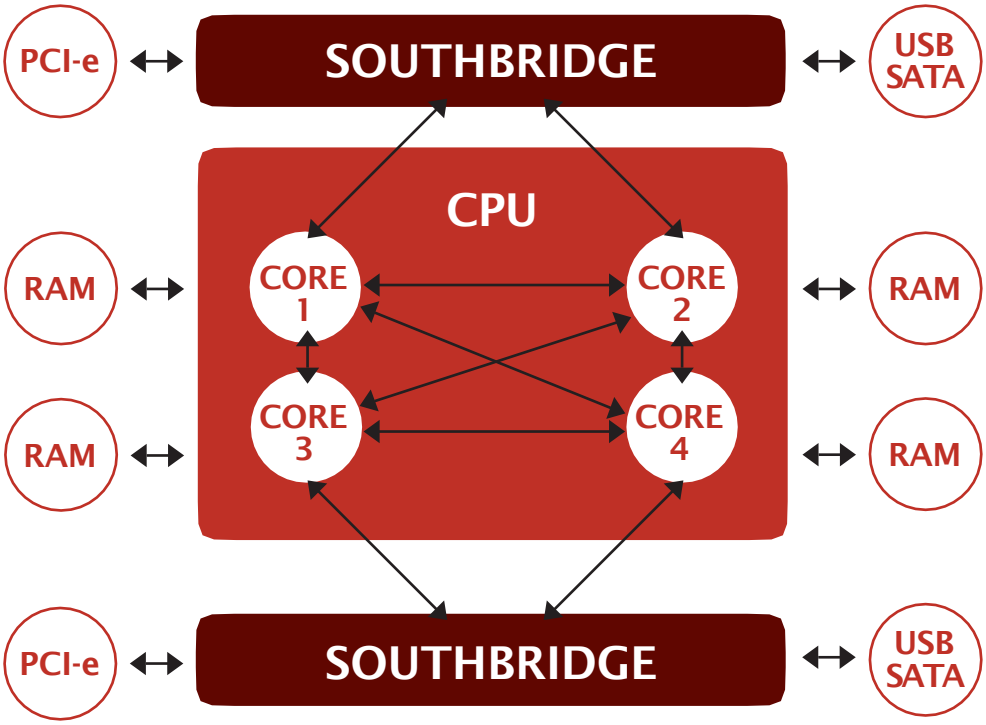
從這個設計我們就可以看出，北橋、南橋和 RAM 都是所有核心共享的資源，因此也被即時與非即時應用程式共用。除此之外，RAM 控制器通常只有一個埠，也就是說同一時間只有一個核心能存取 RAM。

CPU 的頻率多年來持續提升，而價格並沒有相對增加，但記憶體的情況就不一樣了。永續性記憶體（例如硬碟）的存取速度非常慢，因此引入了 RAM，讓 CPU 在執行程式碼與存取資料時，不必等待硬碟存取。

市面上有非常快速的靜態記憶體（Static RAM），但因為價格極高，所以在標準硬體中只能少量使用（大約幾 MB）。我們在電腦中一般所稱的 RAM 其實是動態記憶體（Dynamic RAM），雖然便宜很多，但也比靜態記憶體慢得多。存取動態記憶體需要耗費數百個 CPU 週期。在多個核心都存取這個動態記憶體的情況下，很容易看出 FSB 和 RAM 存取是傳統架構中最大的瓶頸。

1.2. NUMA (Non-Uniform Memory Access，非統一記憶體存取) 架構

為了消除 FSB 和 RAM 所造成的瓶頸，後來設計出一種新的架構，使用多個動態記憶體模組和多條用來存取這些模組的匯流排。每個核心甚至能擁有自己專屬的 RAM 模組。負責存取 I/O 的南橋也可以被重複配置，這樣不同的核心就能使用不同的匯流排來存取硬體。對於即時應用程式而言，這還有一個額外優點，就是不再需要與非即時應用程式共享這些資源。



NUMA 原本是為了多個處理器之間的互連而開發，但隨著現在處理器擁有越來越多的核心，這架構也被延伸應用在處理器內部。

NUMA 的設計也帶來了新的問題，因為變數只會儲存在單一 RAM 模組中，但可能有多個核心都需要存取這些變數。當要存取屬於其他核心的變數時，存取速度可能會慢的很多，因此應用程式應該特別針對這種架構來設計才能妥善利用。雖然我們會建議這種架構只用在專為 NUMA 設計的應用程式上，但當系統裡的核心數量超過四個時，UMA 架構中的 FSB 很容易就會過載，反而造成更多延遲。因此，NUMA 成為大型機器的主要架構。

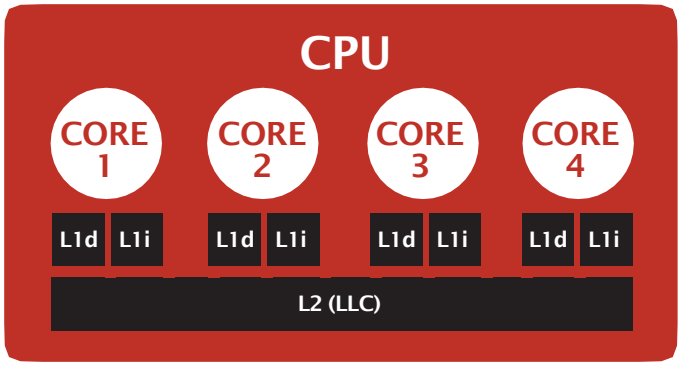
為了在享有 NUMA 優點的同時避免缺點，有些機器會設計成由共享一個 RAM 模組的處理器節點所組成。這樣一來，只使用單一節點內部核心的應用程式就不會受到 NUMA 影響，並且能正常運作。

由於 RTX64 目前不支援這種架構，我們將不再深入探討細節。

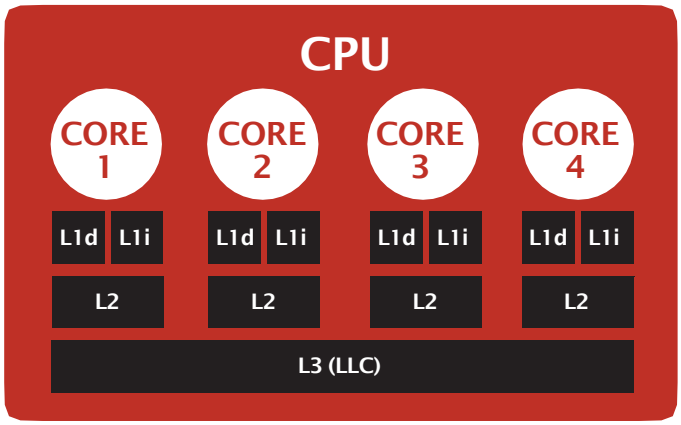
1.3. 記憶體與快取

如前所述，和存取靜態記憶體相比，CPU 存取動態記憶體的速度相當慢（平均要花上數百個週期才能存取一個字）。因此，現在的 CPU 都內建作為快取用的靜態記憶體，並組織成多個層級。當一個核心需要存取資料時，這些資料會先從主記憶體（動態記憶體）複製到最近的快取中，這樣核心就能夠以數倍快的速度存取這些資料。

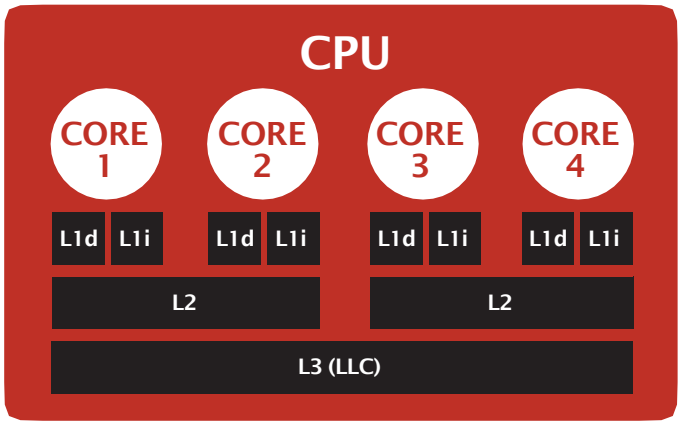
具有兩層快取的四核心 CPU



具有三層快取的四核心 CPU，第二層為獨佔式



具有三層快取的四核心 CPU，第二層由核心節點共享



快取層級 1（L1）會將指令（程式碼）與資料（變數）分開存放；其他層級則是整合式的。較高層級的快取比第一層級容量更大，速度也更慢，並且可能是單一核心專用或多個核心共用。容量最大的快取也稱為最後層快取（Last Level Cache，LLC），通常是由所有核心共同分享。以 CPU 週期為單位，各個快取層級的平均存取時間如下面的表格所示。

CACHE LEVEL	AVERAGE ACCESS TIME
LEVEL 1	~ 3
LEVEL 2	~ 15
LEVEL 3	~ 20
MAIN MEMORY	~ 300

如上表格，每當 CPU 因為資料不在快取中而必須等待存取主記憶體時，效能就會受到嚴重衝擊，這種情況稱為「快取遺漏」（cache miss）。如果主記憶體的資料是成批或依序存取，速度會快很多，但程式中的資料與指令存取很少是隨機的，所以 CPU 會嘗試預測接下來會用到哪些記憶體內容，並提前載入到快取中。這種技術稱為預取（prefetching），能大幅改善效能（可降低約 90% 的延遲）。

為了預測哪些資料應該放進快取，處理器會依據兩個原則：時態區域性（temporal locality）與空間區域性（spatial locality）。

- 時態區域性是指變數和指令通常會在短時間內連續被多次存取。這在處理迴圈和函式的區域變數時尤其明顯。
- 空間區域性是指位置相鄰且一起被定義的變數通常會一起被使用，而且下一行程式碼很有可能包含接下來要執行的指令。

時態區域性正是我們需要快取的原因。只有當資料會被多次存取，使用前先複製到本地緩衝區才有意義。為了善用空間區域性，以及 RAM 批次存取速度更快的特性，資料不是以位元組為單位來要求與傳輸，而是以通常長度為 64 位元組的快取行（cache line）為單位。此外，CPU 通常也會自動預取下一行。詳見第 2 節，程式設計師的工作就是讓資料與指令的存取順序盡可能的可預測，讓預取機制能有效運作。

1.4. 瓶頸

由於快取成本高，容量通常很小，所以並非與應用程式相關的所有資料和指令都能放在快取裡。此外，快取還是由所有連接到這個快取的核心上所執行的應用程式共享。這代表當一個應用程式或執行緒載入過多資料時，會把其他執行緒或應用程式仍想使用的舊資料驅逐且之後需要重新載入。核心上執行的程式碼愈多，在執行緒切換時指令被驅逐的機會就愈大。這種對快取的搶奪情況就稱為「記憶體爭用」（memory contention）。

當系統是單一插槽，或是在多插槽系統中的其中一個插槽同時包含即時與非即時核心時，最後一層快取（LLC）會被即時與非即時應用程式共享。因此，當非即時應用程式使用大量記憶體，例如播放高畫質影片的時候，就可能影響即時應用程式的效能。如果應用程式或執行緒所使用的資料量很小，就有機會選擇具有足夠快取的 CPU，讓整個資料保留在不會被其他應用程式影響的獨佔快取裡。

FSB 存取主記憶體的速度相對於 CPU 來說非常慢，因此光是單一核心進行大量資料載入，就可能占用所有的匯流排頻寬。這種情況稱為「匯流排爭用」（bus contention），而且當 CPU 具有四個或更多核心時，這種現象就會開始變得明顯。由於這個匯流排才是真正的瓶頸，通常把預算花在更快的 RAM 和晶片組匯流排上，會比花在更快的 CPU 上好；因為更快的 CPU 也只是等更久而已。

1.5. 資料同步

應用程式在兩個核心上執行的速度並不會比在單核心上更快的主要原因在於資料同步，同時程式碼序列化也會對執行延遲產生影響。如前所述，核心一向是透過最低層級的獨佔快取來存取資料，這表示如果兩個核心要存取一個變數，這變數就必須同時存在於兩個核心的快取中；而當這個變數的值被修改時，兩個核心裡的快取都必須被更新。CPU 必須確保整個系統的資料一致性，這可能就會造成巨大的延遲，通常是因為一個核心必須去窺探另一個核心的快取資料，以確保資料的完整性。

為了維持這種一致性，CPU 會使用定義快取行狀態的 MESI 協定。

- MODIFIED：數值已被這個核心修改過，因此這是系統中唯一有效的副本。
- EXCLUSIVE：只有這個核心在使用這個變數。不需要發送變更信號。
- SHARED：這個變數存在於多個快取中。如果數值發生變更，必須通知其他核心。
- INVALID：尚未有變數被載入，或數值已被其他核心變更。

每個快取行的狀態由各個核心各自維護。為了做到這一點，核心必須監看所有對主記憶體的資料請求，並通知其他核心自己已經有正被讀取或已修改的變數。每當一個核心想要存取在另一個核心快取中被修改的變數時，新數值就必須同時送到主記憶體以及正在讀取的核心。存取這個值的速度會變得和沒有快取一樣慢。如果有一個變數被一個核心頻繁寫入，同時又被另一個核心讀取，就會發生以下情況：

ACTION	CORE 1 STATUS	CORE 2 STATUS
Core 1 reads the value	EXCLUSIVE	INVALID
Core 2 reads the value	SHARED	SHARED
Core 1 modifies the value	MODIFIED	INVALID
Core 2 reads the value	SHARED	SHARED
Core 1 modifies the value	MODIFIED	INVALID
Core 2 reads the value	SHARED	SHARED

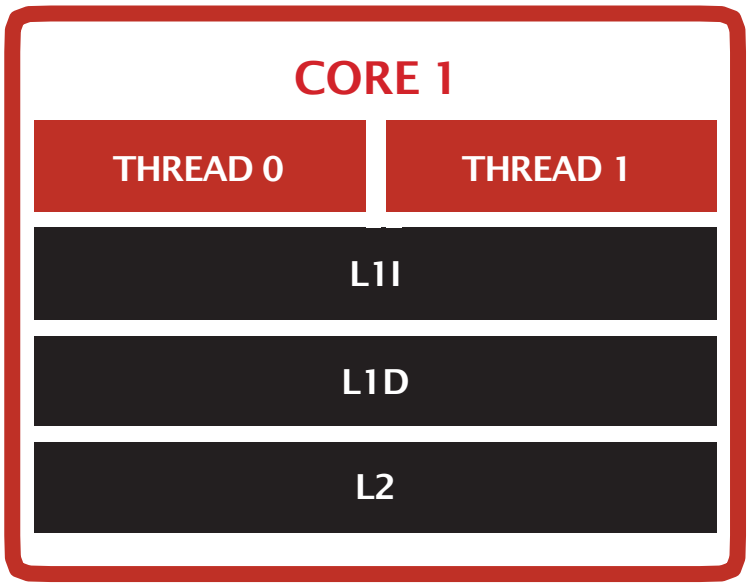
在這種情況下，核心 2 每次都必須從主記憶體讀取數值，等於失去了快取的優勢。而核心 1 則必須在每次修改值的時候，都要在 FSB 上發送「所有權請求」（Request For Ownership，RFO），並且在每次核心 2 要求讀取這個值時更新主記憶體。

當兩個執行緒位於不同核心時，存取這個值的速度會比在單一核心上慢非常多，並且還會增加 FSB 上的流量負擔。

指令在這方面的問題要小得多，因為指令通常是唯讀的。在這種情況下，不需要知道有多少核心正在使用這些指令。自我修改程式碼雖然存在，但風險很高而且極少被使用，所以在這裡就不多做說明。

1.6. 多核心與超執行緒

多個核心和單一核心上的多個執行緒看起來用途相似，但共享資源的方式卻截然不同。因此，實際使用的方式幾乎是相反的。



在多核心架構下，每個核心都有自己的層級 1 快取並且是成倍配置的。這代表系統上可用的快取變多了，但這些快取之間必須進行同步。

在超執行緒架構下，兩個執行緒共享同一個層級 1 快取，因此在最壞的情況下，可用的快取容量可能只有一半。

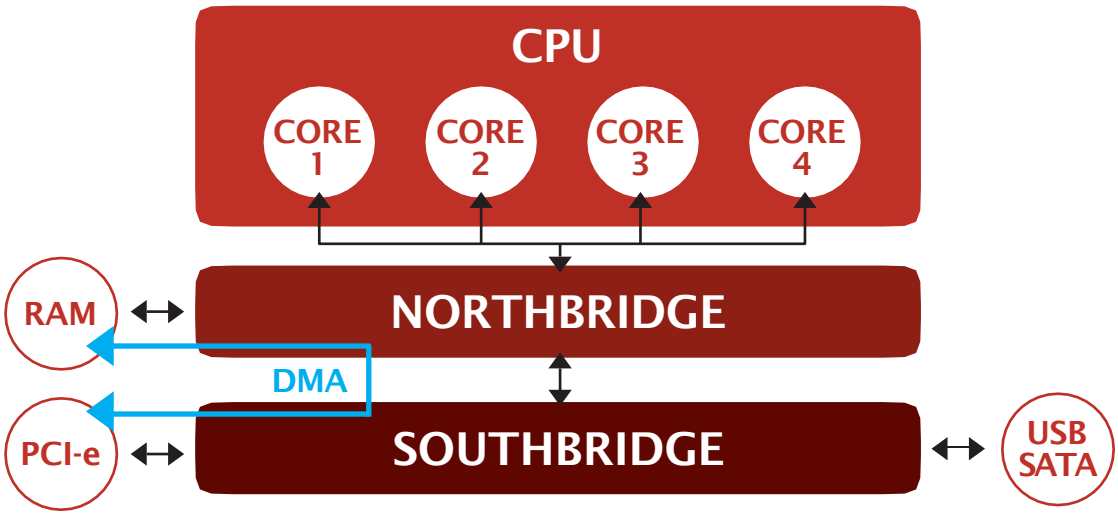
因此，在多核心架構下，程式設計師必須限制各核心執行緒之間的共享資料量以避免同步延遲。在超執行緒架構下，層級 1 快取是在超執行緒之間共享的，這代表如果每個執行緒使用的資料不同，就會產生快取爭用，使得系統必須更頻繁地從主記憶體載入資料。

在這種情況下，只有在同一組資料集上執行獨立運算時，效能才會提升，但這通常屬於特殊情況。所以在多數情況下，超執行緒並不會改善效能，因此我們建議停用。另外，由於核心與層級 1 快取都是由兩個超執行緒共享，因此這兩個執行緒應該全數用於即時應用程式，或全數用於非即時應用程式。

1.7. DMA (Direct Memory Access，直接記憶體存取)

I/O（無論是 PCI-e、USB 或其他任何類型）的存取是由 CPU 指令控制的。這表示當像 NIC 這類裝置以中斷訊號通知有資料更新時，CPU 必須查詢資料並傳送到主記憶體，而這會在 FSB 上增加大量不必要的負載。

為了因應高速匯流排，便開發出直接記憶體存取（DMA）功能。使用 DMA 時，裝置會通知 CPU 資料已更新，並直接把資料傳送到主記憶體而不需要 CPU 採取任何動作。



如果 CPU 打算立刻使用這筆資料，可使用一個稱為直接快取存取（DCA）的新功能，將資料同時複製到主記憶體與 CPU 快取中。

2. 記憶體與多核心程式設計

2.1. 快取機制

快取是完全由處理器控制，且無法由程式設計師修改，但被執行的方式可能會對程式效能造成巨大的影響。不同快取中的資料可能有所不同，或是低層級快取中的資料可能會在高層級快取中被複製。資料重複保存會讓高層級快取看起來可用空間變小，但如果兩個核心要存取由高層級快取所共享的同一個變數，就能利用快取來同步數值，而不必一路存取到主記憶體。

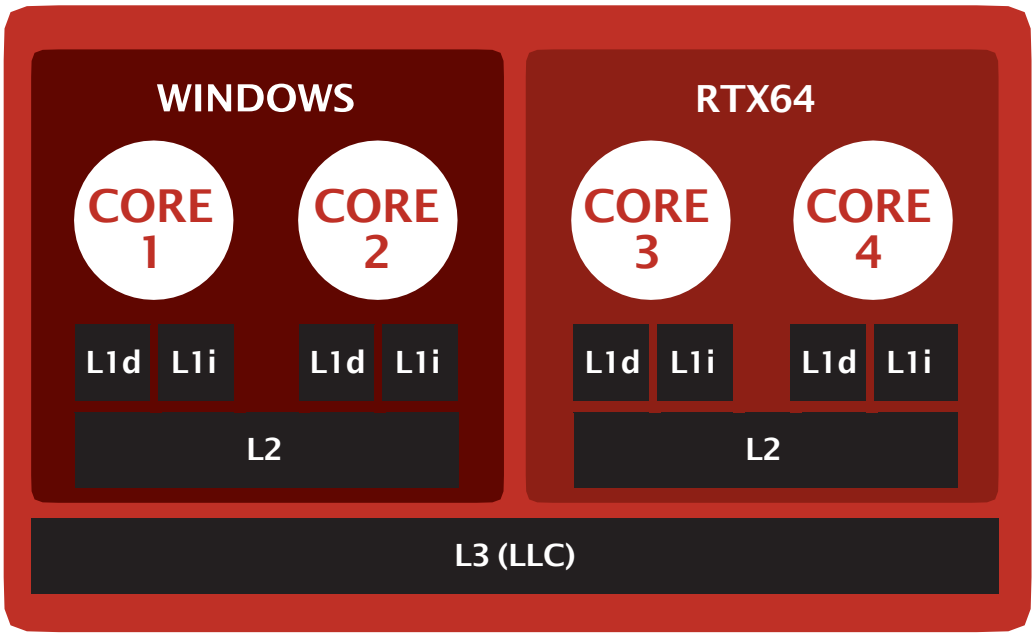
我們可以假設快取永遠是滿的，畢竟在電腦運作幾分鐘後這也是事實。因此，任何新加入快取的資料都會導致另一個快取行被「驅逐」，也就是被複製回下一層快取；而下一層快取如果沒有包含被驅逐的值，接著就會再把這個值寫回主記憶體。預設情況下，快取中最久未使用的值會被驅逐，不過有些新的處理器會用更複雜的演算法來決定要驅逐哪個值。處理器也具備用來強制啟用或繞過這些快取機制的記憶體管理指令，但使用這些功能非常依賴特定的硬體和應用程式，且需要投入額外的開發時間。本文將不詳述這些功能，但在前言中提到的 Ulrich Drepper 論文中有更完整的說明。

2.2. 獨佔快取與共享快取

如 1.3 節所述，現在有多種會影響效能的快取架構。每個執行緒要分配到哪個核心上執行的選擇，應該取決於執行緒之間的互動方式以及可用的快取資源。

理想的情況下，共享許多變數的執行緒應該在同一個核心上執行，而在不同核心上執行的執行緒則不該共享變數。但如果真能做到這點，那麼只有小型應用程式會在單一核心上執行。規模較大的應用程式則需要使用多個核心來執行不同的模組，同時仍必須共享資料和同步模組。

因此，程式設計師需要辨識哪些變數是共享的、哪些不是，並盡可能讓更多區域變數保留在獨佔快取中，同時利用共享快取來存放共享變數。以 RTX64 這種兩套作業系統且各自擁有專屬核心的特定情況來說，最理想的配置是具備三層快取：層級 1 快取由每個核心獨佔；層級 2 快取區分為 RTX64 核心快取與 Windows 核心快取；最後一層快取則由所有核心共享。這樣 Windows 應用程式就無法污染 RTX64 的層級 2 快取，而不同 RTX64 核心上的執行緒也能共享變數，且不必依賴與非即時環境共用的硬體資源。不過，這種理想情況只有在 RTX64 應用程式中最常用變數不超過層級 2 快取容量時才會成立。



2.3. 變數宣告最佳化

為了善用多核心與快取的最佳化，必須辨識出不同類型的變數並以不同的方式處理。變數類型包括：

- 由單一核心使用的變數。這些變數應該只出現在一個層級 1 的快取中。雖然可能會被驅逐到 L2/LLC，但不會同時出現在多個 L1 快取。
- 唯讀變數（在開始初始化後就不再更動）。這類變數可以由多個核心共享且不會有效能問題。
- 大多是唯讀的變數。
- 經常修改的變數。若變數經常被多個核心修改，或是在共享狀態下被某個核心修改，存取速度就會變慢。因此，這類變數應該集中組合以避免干擾其他變數。

雖然有編譯器定義可確保對齊並明確標示變數類型，但其實透過正確宣告變數即可達到效能上的改善。

- 變數是以 64 位元組的快取行為單位被存取的，因此最好確認一個快取行中只有一種類型的變數。為了達到這點，可將變數分組放入長度為 128 位元組倍數的結構中。
- 結構的總大小應該盡可能精簡，而且變數在結構中會被對齊（在 64 位元系統上可假設以 64 位元對齊），因此較小的變數類型應該集中組合在一起。例如，將兩個 int 或四個 short 分在同一組。
- 預取會載入下一個快取行，因此第一個被使用的變數應該放在結構的開頭，且變數應依照被使用的順序進行宣告。
- 應該把會一起被消耗的「大多是唯讀」和「經常修改」的變數分組在一起，這樣就能在一次操作中全部被更新。

如果不遵守這些規則，可能會出現一種稱為「偽共享」（false sharing）的非預期狀況。這會發生在一個應該是唯讀、或只由單一核心使用的變數，卻因為和另一個被不同核心修改的變數位於同一個快取行上，而被標記為無效。在這種情況下，即使原本不該如此，存取第一個變數的速度會變得緩慢。所以將不同類型的變數放置在不同的快取行中，可以確保這種情況不會發生。

存取共享且經常修改的變數可能會變得緩慢，甚至在 FSB 過載時受到干擾。這些變數不該被那些需要高度準確性、且必須維持極小抖動（jitter）的執行緒所存取。為了達成這點，可能有用的做法是建立每個核心專屬的中繼變數，讓高度準確性的執行緒存取這些變數，再由較低優先權的執行緒負責把值同步給共享變數。

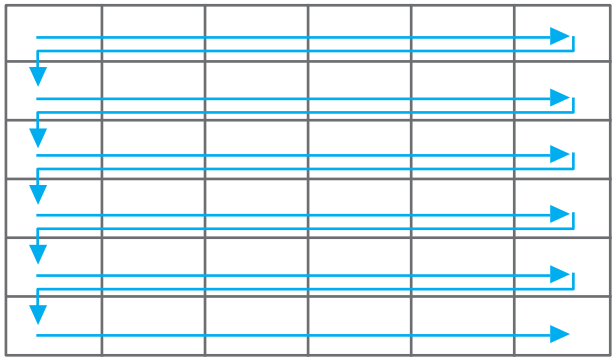
2.4. 變數存取最佳化

當需要處理超出快取容量的大型資料集時，可能有必要以最佳化快取使用的方式來編寫程式碼。這類操作可能是影像分析或其他大型矩陣運算，也只有當矩陣規模大到無法放入快取時，才需要考慮本節內容。由於資料必須從主記憶體載入，因此應以能善用快取與 RAM 技術的方式來取用資料：

- 如果可行，應將資料集拆分為能容納於快取的更小集合，並在移至下一個集合之前，完成對該單一集合的所有操作。
- 資料應依照在記憶體中定義的順序來存取，這樣預取就能降低載入時間。

我們將以兩個各有 2000 行列的正方形矩陣 A 和 B 的乘法作為範例，說明程式碼可能的修改方式。

矩陣在記憶體中的定義方式為陣列的陣列。因此，變數的組織方式如下：

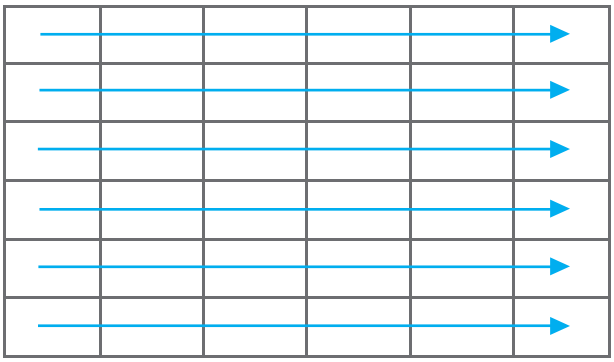


完成矩陣乘法的標準程式碼非常簡單，會使用三個 for 迴圈：

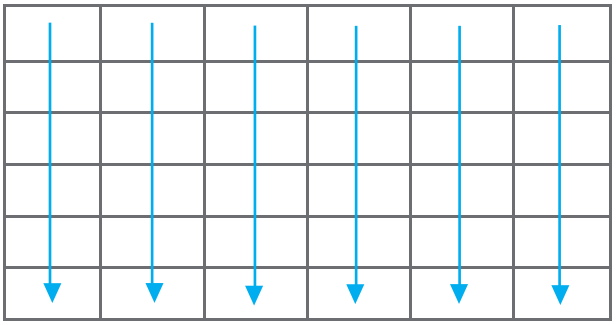
```
for (int i = 0; i < 2000; i++) {
        for (int j = 0; j < 2000; j++) {
                for (int k = 0; k < 2000; k++)
                        Result[i][j] += A[i][k] * B[k][j];
        }
}
```

在這段程式碼中，矩陣會以不同的方式被消耗：

A:



B:



在這種簡單的邏輯下，第一個矩陣的資料只會被處理一次，而且是按照在記憶體中的排列順序來存取。然而，第二個矩陣的資料會被載入許多次，且存取順序對處理器來說是隨機的。

在這種情況下，程式設計師應該嘗試把運算拆分成能放進 L1d 快取的較小資料集，並在移至下一個資料集之前，盡可能先完成對當前資料集的使用。

```
int SetSize = DataSetSize / CellDataSize; // 64 / 8
int Iteration = 2000 / SetSize;
for (int i = 0; i < Iteration; i+= SetSize) {
        for (int j = 0; j < Iteration; j+= SetSize) {
                for (int k = 0; k < Iteration; k+= SetSize)
                        { for (int i2 = 0; i2 < SetSize; i2++) {
                                for (int j2 = 0; j2 < SetSize; j2++) {
                                        for (int k2 = 0; k2 < SetSize; k2++)
                                                Result[i][j+SetSize*i2+j2] +=
                                                        A[i][k+SetSize*i2+k2] * B[k][j+SetSize*k2+j2];
                                }
                        }
                }
        }
}
```

為了簡化運算式，可使用指標（pointers）來表示矩陣：

```
int SetSize = DataSetSize / CellDataSize; // 64 / 8
int Iteration = 2000 / SetSize;

for (int i = 0; i < Iteration; i+= SetSize) {
        for (int j = 0; j < Iteration; j+= SetSize) {
                for (int k = 0; k < Iteration; k+= SetSize)
                        { for (int i2 = 0; i2 < SetSize; i2++) {
                                R2 = &Result[i][j + SetSize * i2];
                                A2 = &A[i][k + SetSize * i2];
                                for (int j2 = 0; j2 < SetSize; j2++) {
                                        for (int k2 = 0; k2 < SetSize; k2++)
                                                { B2 = &B[k][j + SetSize *
                                                k2]; R2[j2] += A2[k2] +
                                                B2[j2];
                                        }
                                }
                        }
                }
        }
}
```

為了簡化運算式，可使用指標（pointers）來表示矩陣：

```
for (int i = 0; i < Iteration; i+= SetSize) {
        for (int j = 0; j < Iteration; j+= SetSize) {
                for (int k = 0; k < Iteration; k+= SetSize)
                        { for (int i2 = 0; i2 < SetSize; i2++) {
                                R2 = &Result[i][j + SetSize * i2];
                                A2 = &A[i][k + SetSize * i2];
                                for (int k2 = 0; k2 < SetSize; k2++)
                                        { B2 = &B[k][j + SetSize * k2];
                                        for (int j2 = 0; j2 < SetSize; j2++)
                                                R2[j2] += A2[k2] + B2[j2];
                                }
                        }
                }
        }
}
```

這種程式碼修改可把運算時間減少 75%，差異非常可觀。不過，由於這會讓程式碼更複雜並引入新的變數，因此只有在處理的資料量大到足以讓效能的提升抵過額外開發時間時，才值得採用。

2.5. 最佳化程式碼的可預測性

在原始碼層面需要手動處理的工作相對少很多。編譯器會知道正確的排序與最佳化規則，並且自動套用。但指令的預測只要出錯，造成的延遲會比資料預測錯誤更嚴重，因為指令在被 CPU 使用前必須先解碼。因此，若可行，程式碼應盡可能減少預測錯誤。

在預設或正常的執行路徑中，應盡量避免分支程式碼。當程式碼符合預期條件時，不應引發分支或跳躍。若某個條件具有最可能會發生的值，最可能的案例程式碼就應該接著執行，因為這路徑會被預先載入。這種情況常發生在檢查錯誤或不正確參數的時候。我們可以預期事情在大多數情況下正常運作，且提供的資料是有效的。此時，正常操作應該接在條件判斷之後，而錯誤處理程式碼則可放置在較遠的位置。

CPU 還有另一種最佳化機制，稱為「亂序執行」（Out Of Order execution，OOO）。也就是當 CPU 偵測到兩個指令彼此不相關時，會先開始執行第二個指令以節省處理時間。這通常不會影響效能，除非第二個指令會使用第一個指令也需要的資源（例如從主記憶體讀取資料），進而顯著延遲第一個指令的執行。CPU 的記憶體管理指令可用來防止 OOO 發生，但這同樣需要對硬體以及記憶體管理指令的運作方式有精準的理解。

2.6. 序列化程式碼

一個應用程式的許多組成部分，包括所使用的函式庫，會被運行在不同核心上的多個執行緒所呼叫。除非複製大部分的應用程式和作業系統程式碼，否則無法完全避免序列化程式碼；但那樣反而會對效能造成更嚴重的損害。

為了避免並行存取程式碼中序列化的部分，作業系統會使用一種稱為自旋鎖（spinlock）的內部互斥鎖。自旋鎖讓執行緒在等待所需資源時，不會把核心釋放給另一個執行緒。自旋鎖通常只用於非常短的等待時間，遠短於排程器的定時器週期。

近年的高階處理器新增了一項功能，用來降低序列化程式碼的影響：交易式暫存器（transactional registers）。在交易式暫存器中執行的操作不會立刻套用到一般暫存器上。這表示，第二個執行緒不需要使用自旋鎖來等待資源，而是直接在交易式暫存器中進行運算，等資源被釋放後，且在此期間讀取的暫存器未被修改時才會套用結果。在超過 90% 的情況下，讀取的暫存器不會被修改，因此第二個執行緒可以在不用等待的情況下完成執行。

3. 執行上的實務問題

3.1. 多核心環境下的效能下降

問題

即時應用程式的速度太慢，或造成 CPU 負載接近 100%，因此新增了一個 RTX64 核心以提升效能。然而，加入額外核心後效能不但沒有改善，甚至反而下降。

原因

這很可能是因為資料同步延遲所造成。如果應用程式不是針對多核心環境而開發，加上有大量變數被不同核心上的執行緒所共享，就會導致快取失去作用，並且由於需要不斷存取主記憶體或觸發快取一致性機制，效能很可能會因此下降。

可能的解決方法

最佳的解決方法很可能是依照本文第 2 節的指引來修改應用程式。

若希望在不修改軟體的情況下提升效能，增加核心數的方法針對這個特定應用程式是行不通的。相較之下，提高所使用的單核心頻率，以及可能的話提升 RAM 的頻率會更有效。

3.2. 分離執行緒變數無法提升效能

問題

即時應用程式已修改為可在多核心上執行。每個核心上的執行緒只共享少量變數，但這些修改似乎並沒有明顯的提升效能。

原因

如果分離不同核心使用的變數仍無法提升效能，那很可能是發生了偽共享。如第 2.3 節所述，我們需要確保非共享變數不會和共享變數位於同一條快取行上。變數並非由快取個別載入，而是以 128 位元組的快取行為單位進行載入。

可能的解決方法

如第 2.3 節的說明重新檢視變數宣告。將不同類型的變數分組到不同的結構中，並讓這些結構佔用完整的快取行，以確保在記憶體中是相互隔離且不會產生干擾。

3.3. 不同 RTX64 核心上的行程會互相干擾

問題

兩個獨立的應用程式已設定在兩個不同的 RTX64 核心上執行，但當其中一個進行大量運算時，會導致另一個應用程式出現延遲。

原因

即使已竭盡所能將兩個核心分離，它們仍然會共享某些資源。就算使用的 I/O 在不同的 PCI-e 線路上且沒有共享記憶體，CPU 的最後一層快取（LLC）仍會被兩個應用程式共同使用，而用來存取 I/O 與 RAM 的前端匯流排（FSB）也是如此。因此，在 LLC 或 FSB 或兩者中都可能發生爭用。

當一個應用程式進行大量運算時，通常也會使用大量的資料。這些資料會汙染 LLC，迫使另一個應用程式再次從 RAM 請求資料。載入這些資料也會在 FSB 上產生大量流量，進而可能導致壅塞。如果另一個應用程式也需要存取 RAM 資料，這些存取時間就會變得更長。

可能的解決方法

第一個解決方案是限制第一個應用程式的運算速度，以避免汙染快取和過載 FSB。另一個方法則是選用快取容量更大的 CPU 和頻率更高的 RAM 模組，以降低大量運算帶來的影響。

Intel 在少數非常高階的處理器上開發了一項稱為「快取配置技術」（Cache Allocation Technology，CAT）的技術，可為特定處理器保留快取空間。Intel 還宣布了另一項技術稱為「記憶體匯流排配置」（Memory Bus Allocation，MBA），可為特定核心保留 FSB 頻寬。這些技術目前只有在最新一代的高階處理器系列上提供，並從 RTX64 3.4 版本開始支援。

3.4. 更多 Windows 核心造成更高的延遲

問題

某個 RTX64 應用程式被部署到一個更快且擁有更多核心的新 CPU 上。這些新增的核心已指派給 Windows。雖然系統的 Windows 或 RTX64 端都沒有任何變更，但 RTX64 應用程式卻出現了延遲。

原因

這是由匯流排爭用所造成的。連接 CPU 核心與 RAM 的前端匯流排（FSB）是一種速度遠低於 CPU 核心的有限資源。FSB 是由所有核心共享，而且任何單一核心都有可能完全佔用。系統中很可能有某個 Windows 應用程式在大量消耗資料，這個應用程式會使用指派給 Windows 的所有核心，透過 FSB 以最快速度存取資料。隨著 Windows 核心數量增加，分配給 RTX64 核心的 FSB 頻寬比例就減少了，因而增加存取主記憶體的延遲。

可能的解決方法

理想的解決方法是改用 NUMA 平台或為 RTX64 保留頻寬。但若將平台改為 NUMA，會需要對應用程式進行重大修改，而且 RTX64 目前尚未支援 NUMA。Intel 提供一項用於配置頻寬的技術，稱為「記憶體頻寬配置」（Memory Bandwidth Allocation，MBA），此技術從 RTX64 3.4 版本開始支援。

目前可行的方法是限制系統核心數量以避免爭用（當核心數超過四個時，爭用問題會變得非常顯著），或是提高 FSB 頻寬。如果情況允許，我們建議購買速度更快的 RAM，並確認晶片組支援這種更高的頻率。

RTX64 支援 CAT 與 MBA（Intel RDT）

Intel 在高階處理器中提供了例如快取分配技術（CAT）、記憶體匯流排配置（MBA）以及交易式暫存器（transactional registers），以減輕效能影響並保護關鍵執行緒免於干擾。這些功能屬於 Intel 資源控管技術（Resource Director Technology，RDT）的一部分。RTX64 透過將這些功能連結即時執行緒的優先權與親和性設定，讓開發者可以在不需要修改程式的情況下使用較新的處理器。此外，更新一代的 Intel 處理器還支援 Intel 時間協調運算（Time Coordinated Computing，TCC），為即時運算帶來更高層次的最佳化。

結論

即時作業系統能讓開發者以撰寫 Windows 應用程式的方式來撰寫即時應用程式，並負責處理和 Windows 之間的排程與資源隔離。不過，仍有部分資源例如處理器快取和前端匯流排頻寬是共享的，而在近期的處理器中變成了系統的瓶頸。這引發了一些常讓人覺得難以解釋或違反直覺的效能問題，並進一步影響整體系統的擴充性。

在本文中，我們探討了造成這些效能問題的原因、如何透過硬體選擇來提升效能，以及有助於解決問題的替代作法。我們也介紹了 Intel 能進一步改善效能的新技術 CAT 與 MBA，且已由 RTX64 支援。藉由這些資訊與技術，我們可以針對多核心系統最佳化應用程式並提升擴充性，進而為整個組織帶來更好的成果。

如需進一步了解如何解決效能挑戰，或 RTX64 支援的最新功能，請聯絡您的英特蒙窗口，或直接與我們聯絡。

選擇硬體平台

請依循下方準則以判斷哪些元件對系統的影響最大。

a. 處理器

即時系統最需要的是穩定性，而不是瞬間效能爆發或省電。Atom 處理器能提供良好的效能，相比之下行動處理器的穩定性永遠不盡理想。若系統支援，應該停用像超執行緒（Hyper-threading）和睡眠狀態這樣的功能。

獨立的 RTX64 應用程式應使用專屬的核心，並且與 Windows 使用的核心分開。但為了避免爭用問題，我們建議將核心數量限制在四個以內，或確保 Windows 應用程式不會出現大量資料消耗的突發狀況。

b. 晶片組與 RAM

對於負載很重的應用程式而言，RAM 存取會成為效能瓶頸。問題不在於 RAM 的容量大小；我們只需要確保應用程式使用的所有資料都能放得下即可，多出來的空間不會帶來任何改變。相反地，RAM 與晶片組匯流排的頻率非常關鍵，並會決定 CPU 存取所有使用資料的速度。因此，我們應該在預算允許的範圍內，盡可能選擇頻率最高的硬體規格。

快取的容量大小與配置方式，往往比 CPU 頻率對效能的影響更大。一般來說，快取越大，程式執行的速度就越快；但隨著快取容量增加，存取延遲通常也會隨之上升。若快取容量仍小於資料集的大小且需要頻繁存取 RAM，加大快取就能降低整體延遲；但如果快取容量已經大於資料集大小，再增加快取反而會適得其反。

如果一個 RTX64 應用程式會使用多核心，那麼只讓 RTX64 核心共享層級 2 快取可能會有幫助（請參考第 1.3 節）。

c. I/O 裝置

任何可能的瓶頸和轉換延遲都應該避免，因為光是 RAM 存取端就已經帶來足夠多的限制。

任何連接到 RTX64 的裝置都應該有自己的 PCI-e 通道以避免延遲。新一代的處理器只支援 PCI-e，因此傳統 PCI 裝置其實是由硬體晶片分組再連結到 PCI-e。如果可能，應盡量避免這種情況，因為這類額外的晶片會導致無法控制的延遲。只有直接連接到晶片組的裝置，才能提供良好的效能。

所有 PCI-e 相關的睡眠和省電選項都應該停用。這些選項位於 BIOS 中且通常無法由使用者修改，因此應該要求電腦供應商提供正確配置的 BIOS。