計算機組織與 RISC-V CPU 設計 · 第 4 章

快取記憶體

記憶體階層 / 區域性原理 / 映射方式 / 替換與寫入策略 / 多層快取

摘要

記憶體系統的設計目標是「以接近最便宜技術的單位成本, 提供接近最昂貴技術的存取速度」, 而達成此目標的關鍵就是記憶體階層與快取 **(cache)**。本章先建立記憶體系統的特徵框架 (容量、存取時間、傳輸率等), 說明階層為何有效——參考區域性原理; 再深入快取設計的所有核心要素: 容量、映射函數 (直接、全關聯、集合關聯)、替換演算法、寫入策略、行大小與多層快取。本章對應 Stallings 第 4 章。

目錄

1 記憶體系統概觀

1.1 記憶體系統的特徵

特徵	說明
位置	處理器內 (暫存器)、內部 (主記憶體、快取)、外部 (磁碟、磁帶)
容量	以位元組或字為單位
傳輸單位	字 (word)、區塊 (block)
存取方法	循序、直接、隨機存取、關聯 (associative)
效能	存取時間 T_A、記憶體週期時間、傳輸率
實體型態	半導體、磁性、光學
實體特性	揮發性/非揮發性、可抹除性

1.2 記憶體階層

設計上存在三難: 容量大、速度快、成本低無法同時滿足。存在以下權衡: 存取時間愈快, 每位元成本愈高; 容量愈大, 每位元成本愈低、但存取愈慢。出路是不依賴單一記憶體技術, 而是建構階層:

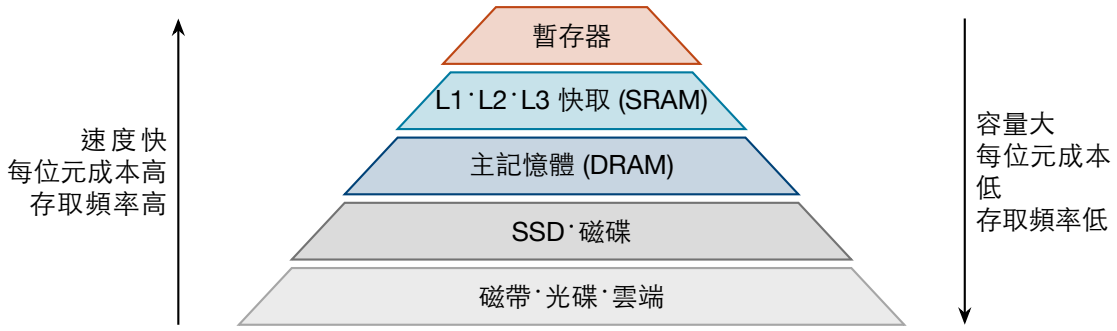


圖 1 記憶體階層: 向下容量遞增、成本遞減、速度遞減

階層之所以有效, 關鍵是存取頻率也向下遞減——這由下述原理保證。

定義 **1.1** (參考區域性 Locality of Reference)**.** 程式執行期間, 處理器對記憶體的參考 (指令與資料) 有群聚傾向:

- 時間區域性: 最近被參考的位置, 近期內很可能再次被參考 (迴圈、熱點變數);
- 空間區域性: 被參考位置的鄰近位置, 很可能即將被參考 (循序執行的指令、依序走訪的陣列)。

兩層記憶體的平均存取時間

設第 1 層存取時間 $T_1 = 0.01\,\mu s$、第 2 層 $T_2 = 0.1\,\mu s$, 命中率 H 為在第 1 層找到資料的比例。平均存取時間:

$$T_{avg} = H \times T_1 + (1 - H)(T_1 + T_2)$$

當 $H = 0.95$:$T_{avg} = 0.95(0.01) + 0.05(0.11) = 0.015\,\mu s$——接近快速層的速度, 卻擁有大容量層的成本結構。區域性讓高命中率成為常態。

2 快取記憶體原理

快取是介於處理器與主記憶體之間、容量小而速度快的記憶體。運作原則:

1. 處理器要讀取某字時, 先檢查快取;
2. 命中 **(hit)**: 直接從快取取得, 速度極快;
3. 未命中 **(miss)**: 把包含該字的整個區塊 (數十位元組) 從主記憶體讀入快取, 再把該字交給處理器——因為空間區域性, 鄰近資料很可能馬上被用到。

主記憶體被視為 $M = 2^n/K$ 個區塊 (位址 n 位元、區塊 K 字); 快取有 m 條行 **(line)**, 每行含一個區塊、一個標籤 **(tag)** 與控制位元 (有效位元、髒位元)。$m \ll M$, 因此需要映射函數決定區塊放哪一行, 以及替換演算法決定踢掉誰。

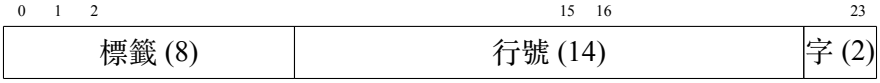
3 快取設計要素

3.1 映射函數

設快取 64 KB、行大小 (區塊)4 B(即 16K = 2^{14} 行), 主記憶體 16 MB(24 位元位址)。

3.1.1 直接映射 (Direct Mapping)

每個區塊只能放進唯一一行:$i = j \bmod m$(j 為區塊號、m 為行數)。位址被切成三段:



- 優點: 簡單、便宜——查一行、比一個標籤即可。
- 缺點: 每個區塊位置固定。若程式恰好反覆使用兩個映射到同一行的區塊, 將持續互踢 (顛簸 **thrashing**), 命中率驟降。

3.1.2 全關聯映射 (Fully Associative)

區塊可放入任意一行; 位址只分「標籤 + 字」。查找時須同時比對所有行的標籤 (關聯記憶體)。彈性最大、無顛簸問題, 但比較電路複雜且昂貴, 只用於小容量結構 (如 TLB)。

3.1.3 集合關聯映射 (Set Associative)

折衷方案: 快取分為 v 個集合, 每集合 k 行 (k-way); 區塊 j 固定屬於集合 $i = j \bmod v$, 但可放入該集合的任一行。查找時只需並行比較 k 個標籤。$k = 2 \sim 16$ 即可逼近全關聯的命中率, 是現代快取的主流 (如 8-way L1、16-way L2)。

	直接映射	**k** 路集合關聯	全關聯
區塊可放位置	1 行	集合內 k 行	任意行
標籤比較次數	1	k(並行)	全部行 (並行)
硬體成本	低	中	高
顛簸風險	高	低	無

3.2 替換演算法

直接映射無選擇餘地; 關聯映射需要在集合 (或全快取) 中挑一行犧牲。皆以硬體實現:

- **LRU**(最近最少使用): 效果最佳、最常用;2-way 時只需每行 1 個 USE 位元。
- **FIFO**: 先進先出, 環形緩衝即可實現。
- **LFU**(最不常使用): 需計數器。
- 隨機: 實測僅略遜於使用式演算法, 硬體最簡。

3.3 寫入策略

- 寫穿 **(write-through)**: 每次寫入同時更新快取與主記憶體。記憶體永遠一致 (對多處理器、DMA 友善), 但產生大量記憶體流量。可搭配寫入緩衝緩解。
- 寫回 **(write-back)**: 只寫快取並設定髒位元 **(dirty bit)**; 該行被替換時才寫回主記憶體。流量最小, 但主記憶體可能暫時過期,I/O 必須經過快取或配合一致性協定 (第 16 章 MESI)。
- 寫入未命中時:**write-allocate**(先把區塊載入再寫, 通常配 write-back) 或 **no-write-allocate**(直接寫主記憶體, 通常配 write-through)。

3.4 行大小

行加大, 初期因空間區域性而命中率上升; 過大則 (1) 可放的區塊數變少、(2) 行內遠端資料被用到的機率降低, 命中率反而下降。8–64 B 是常見甜蜜點 (現代 CPU 多為 64 B)。

3.5 快取數目與階層

- 多層快取:L1(晶片內, 最小最快)+ L2(較大)+ L3(共享)。L1 未命中時到 L2, 再到 L3、主記憶體。
- 分離式 **vs.** 統一式:L1 幾乎都採指令/資料分離 (哈佛式), 消除提取單元與執行單元對快取的競爭——這對管線化設計至關重要 (第 12 章; 我們的 Verilog CPU 也將採分離的指令/資料記憶體介面)。L2/L3 則多為統一式。

重點觀念

快取對程式設計師透明, 但對效能絕不透明: 同一演算法, 快取友善 (依序走訪、資料重用) 與快取不友善 (大步幅、隨機存取) 的寫法可差達一個數量級。理解本章, 就理解了「為什麼程式跑不快」最常見的答案。

4 實例: 典型三層快取階層

	L1(每核)	**L2**(每核)	**L3**(共享)
容量	32–64 KB ×2(I/D 分離)	256 KB–2 MB	8–96 MB
關聯度	8-way	8–16-way	16-way
延遲 (週期)	3–5	12–20	30–60
行大小	64 B	64 B	64 B
寫入策略	write-back	write-back	write-back

RISC-V 高效能核心 (如 SiFive P 系列) 同樣採用此種階層; 而微控制器級的 RV32 核心 (如本教材將實作者) 常直接以 SRAM 作為緊耦合記憶體, 不設快取——階層設計永遠服務於應用需求與成本。

5 本章重點回顧

本章要點

- 記憶體階層以「小而快」搭配「大而慢」, 靠區域性原理達成高命中率。
- $T_{avg} = HT_1 + (1-H)(T_1 + T_2)$: 命中率主宰平均存取時間。
- 映射: 直接 (1 處)、集合關聯 (k 處)、全關聯 (任意); 現代主流為 4–16 路集合關聯。
- 替換:LRU 最常用; 寫入:write-through(一致但流量大)vs. write-back(高效但需一致性機制)。
- L1 指令/資料分離支援管線; 多層快取逐級擴大容量、增加延遲。

6 複習問題

1. 時間區域性與空間區域性的差異? 各舉一個程式範例。
2. 快取 128 KB、行大小 64 B、4 路集合關聯、位址 32 位元。求: 行數、集合數, 以及位址中標籤/集合索引/區塊內偏移的位元數。
3. 直接映射為何可能顛簸? 集合關聯如何緩解?
4. 比較 write-through 與 write-back 的優缺點; 何時必須關心快取與主記憶體的一致性?
5. 行大小加大為何先提升後降低命中率?
6. L1 為何採指令/資料分離? 這與管線有何關係?
7. 命中率 97%、L1 存取 1 ns、主記憶體 60 ns, 求平均存取時間; 若命中率降至 90% 呢?

參考資料

[1] W. Stallings, *Computer Organization and Architecture*, 6th ed., Chapter 4.

[2] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed., Appendix B.

[3] U. Drepper, “What Every Programmer Should Know About Memory,” 2007.