补充材料

冯睿骐

January 2023

1 Belady 方法

下面对 Belady 方法的最优性给出一个证明[1]。

为了简便，我们考虑 fully-associative 的情况，这可以容易地推广到 set-associative 情况。假设某一时刻，一个采用 Belady 策略的缓存块内的内存内容为 $X = \{x_1, x_2, x_3, ..., x_n\}$，而采用其他策略的缓存块为 $Y = \{y_1, y_2, y_3, ..., y_n\}$ [2]

考虑这样一个函数 $f : \{x\} \to \mathbb{R}$，将一个内存块映射到它未来第一次被访问的时间。在初始时刻 t_0，$X = Y$。第 i 次 cache miss 时，假设新的内存块是 z。那么 Belady 算法将会将 $X \bigcup \{z\}$ 中的元素

$$x_{mi} = \underset{\xi \in X \bigcup \{z\}}{\operatorname{argmax}} f(\xi) \tag{1}$$

替换为 z。将像第一次替换后这样，满足

$$Y \setminus X = \{\underset{\xi \in X \bigcup \{z\}}{\operatorname{argmax}} f(\xi)\} \tag{2}$$

且

$$X \setminus Y = \{\eta\}, \eta \in X \bigcup \{z\} \tag{3}$$

的状态记作 *Init*。

若 $\forall t \in [t_0, f(x_{m1}))$，Belady 方法的缓存没有发生 cache miss (将这个条件记作 *Simp*). 那么由于 $\forall \eta \in X, f(\eta) \in [t_0, f(x_{m1})]$，其他策略对任意元素 η 的替换都会使得任意访问次数后的用时 t_y 不低于 Belady 方法的用时 t_x，如示意图1所示。

[1] 找了一段时间也没有找到，所以就自己瞎写了个，也不知道对不对

[2] 假设缓存已经填满了，所以两个集合的大小相同。

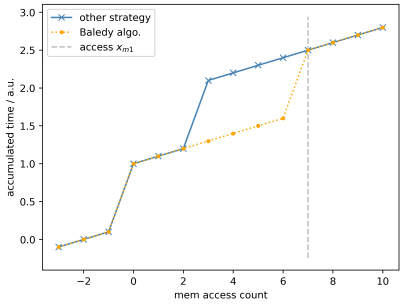

图 1: 灰色虚线表示 $t = f(x_{m1})$ 时刻。第一次 cache miss 发生在 0 的位置。假设 x_{m1} 在第 6 次被访问，而其他策略替换的 η 在第 2 次被访问。可以发现 Belady 方法的耗时总是不大于其他策略的耗时。当 Belady 方法没有 $[t_0, f(x_{m1}))$ 内的 cache miss 时，其他策略至多有一个 cache miss，因为替换后 $X \setminus Y = \{\eta\}$。

在条件 *Simp* 下，开始时由于 $X \setminus Y = \{\eta\}$，所以 $\eta \notin Y$ 且 $\eta \in X$，其他策略会有一次 cache miss。这次 miss 后会将3中的 η 加入 Y，因此这之后 $Y \setminus X$ 不变但

$$X \setminus Y = x_n \tag{4}$$

其中 x_n 是这次其他策略替换掉的内存内容[3]。很类似地，当访问 x_{m1} 时 Belady 方法发生 cache miss 后，$X \setminus Y$ 不变但

$$Y \setminus X = x_l \tag{5}$$

其中 x_l 是这次 Belady 策略替换掉的内容。而且由于

$$x_l = \underset{\xi \in X_{\text{before } x_{m1}} \bigcup \{x_{m1}\}}{\operatorname{argmax}} f(\xi) \tag{6}$$

所以在访问 x_{m1+1} 时，X 和 Y 满足 *Init* 状态。

下面我们证明任何情况都能被分解成 *Simp* 条件：

若 $t < f(x_{m1})$ 时，Belady 方法在 x_{in} 处遇到一次额外的 cache miss，那么记 Belady 方法将 x_{in} 替换为 x_{m2}。假设替换前 $X \setminus Y = S^0_{XY}$, $Y \setminus X = S^0_{YX}$, 又因为替换后 $X \setminus Y$ 中新增的元素 s_{XY} 源于它从 Y 中被删除，所以 $s_{XY} \in Y$。因此 $s_{XY} \in Y \not\subset X \setminus Y$，$s_{XY} \notin S^0_{XY}$；同理 $s_{YX} \notin S^0_{YX}$。又因为 $S^0_{YX} = \{x_{m1}\}$，所以如果从 Y 中删去 x_{m1}，那么等价于形成了新的 *Init*

[3]如果这里根本不加入 η 好像就有点问题了，Belady 方法为了保留最近的下一次访问在缓存中，应该要允许不替换新的内存内容。

条件；如果从 X 中删去 x_n，情况相同；如果从 Y, X 中分别删去 x_{m1}, x_n，那么此时 $X = Y$，下一次 cache miss 时出现 *Init* 情况；如果从 X, Y 中分别删去其他元素，那么一定在 S_{XY}, S_{YX} 中各自增添了一个元素，分别为 η' 与 x_{m2}。在最后一种情况中，虽然相当于图2中所示的两个过程在时间上叠加，但是关于 *Init* 状态在 *Simp* 条件下保持不变的证明不受到影响。

此外，虽然图1和2中的例子，Belady 算法的总命中率并不更低，但对于一系列有限长的指令，x_{mi} 可能出现在无限远，也就是替换了指令的剩余部分不再访问的内容，这就能让 Belady 算法的总命中率低于其他策略。

综上所述，执行相同访问时，总有 $t_x \leq t_y$。

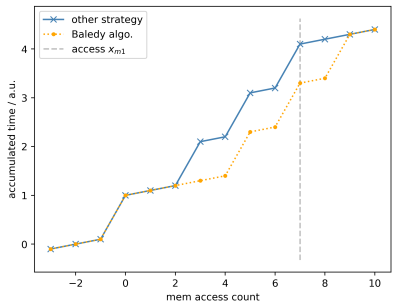

图 2: 灰色虚线表示 $t = f(x_{m1})$ 时刻。假设 x_{m1} 在第 6 次被访问，而其他策略替换的 η 在第 2 次被访问。假设额外的 cache miss 发生在第 4 次访问，它对应的 Belady 方法和其他策略的新的 cache miss(x_{m2} 和 η') 分别发生在第 8 和 6 次访问。两个 $t_x \leq t_y$ 的函数叠加的结果总有 $t_x^{new} \leq t_y^{new}$

2 Block Multiplication

2.1 Proof

Consider this multiplication:

$$C = AB \tag{7}$$

and thus the element in C can be written as

$$c_{ij} = a_{ik} b_{kj} \tag{8}$$

If A and B can be divided into block matrices

$$A = \begin{pmatrix} A_{00} & \cdots & A_{0m} \\ \vdots & & \vdots \\ A_{n0} & \cdots & A_{nm} \end{pmatrix} \tag{9}$$

$$B = \begin{pmatrix} B_{00} & \cdots & B_{0l} \\ \vdots & & \vdots \\ B_{m0} & \cdots & B_{ml} \end{pmatrix} \tag{10}$$

The block multiplication is

$$C_{ij} = A_{ik} B_{kj} \tag{11}$$

where

$$A_{ik} B_{kj} = \{ a^{ik}_{i'k'} b^{kj}_{k'j'} \} \tag{12}$$

so Eq.11 can be written

$$c^{ij}_{i'j'} = a^{ik}_{i'k'} b^{kj}_{k'j'} \tag{13}$$

where Einstein Summation Convention is adopted. To prove Eq.13, we need to insert the definition of block matrices, which is

$$x^{ij}_{i'j'} = x_{i'+r(i),j'+c(j)} \tag{14}$$

where $r(i)$ is the row of the upper side of the block (i, j) and $c(j)$ is the column of the left side of the block. In Eq.13 the summation k in turn transforms into the summation over $r_B(k)$ and $c_A(k)$:

$$c^{ij}_{i'j'} = a_{i'+r_A(i),k'+c_A(k)} b_{k'+r_B(k),j'+c_B(j)} \tag{15}$$

Noting

$$r_C(i) = r_A(i) \tag{16}$$

$$c_C(j) = c_B(j) \tag{17}$$

and we are done.

2.2 Algorithmic time complexity

Suppose a $n \times n$ square matrix is divided into $m \times m$ blocks. For each block the time $T_i = (\frac{n}{m})^3 \times m^3 = T_0$ where T_0 is the time complexity of regular matrix multiplication. The total time would be $m^2 T_0$. Therefore we cannot shrink the size of our matrix without cost. A trade-off must be carefully dealt with in cache-friendly programming.