Słowniki w modelu *cache-oblivious*

Tomasz Dudziak

na podstawie wystąpienia z 10 Stycznia 2011

1 Wprowadzenie

1.1 Hierarchia pamięci

Podstawowym aspektem każdego algorytmu jest jakiś wewnętrzny stan, który w rzeczywistym programie komputerowym reprezentowany jest przez rejestry procesora, pamięć RAM i, w niektórych przypadkach, jakąś formę pamięci zewnętrznej. Koszt dostępu do każdego z tych rodzajów pamięci składa się z dwóch komponentów: opóźnienia (*latency*, *round-trip time*) i przepustowości (*bandwidth*).

$$\text{koszt dostępu do pamięci} = \text{opóźnienie} + \frac{\text{ilość danych}}{\text{przepustowość}} \tag{1}$$

Przepustowość można zwiększać stosując odpowiednią szerokość magistrali ale opóźnienie jest twardo związane z użytą technologią i z dołu ograniczone przez prawa fizyki – im większa pamięć tym jest fizycznie większa a informacja nie może podróżować szybciej niż światło.

Dlatego jedynym sposobem na dalsze zmniejszanie średniego kosztu dostępu do pamięci jest pozyskiwanie danych w większych porcjach (czynnik "ilość danych"we wzorze powyżej) oraz przechowywanie często wymaganych danych w pamięci o mniejszym opóźnieniu. We współczesnych komputerach istnieje cała hierarchia wielu (np. 5,7) pamięci podręcznych, w których szybsza ale mniejsza pamięć przechowuje często pozyskiwane bloki z większej, wolniejszej.

Patrząc na program bardzo trudno zgadnąć w jaki sposób będzie się zachowywał na takiej architekturze i co będzie w pamięci podręcznej na poszczególnych etapach wykonania programu. Chcielibyśmy mieć narzędzie w postaci teoretycznego modelu, który umożliwiałby analizowanie wydajności algorytmów i w miarę wiernie brał pod uwagę wielopoziomowe pamięci podręczne ale jednocześnie nie byłby tak skomplikowany jak ich rzeczywista, sprzętowa implementacja.

1.2 Model cache-oblivious

Maszyna w modelu cache-oblivious interpretuje takie same programy jak maszyna RAM ale posiada pamięć dwóch rodzajów: szybką pamięć podręczną rozmiaru M i dowolnie dużą pamięć główną (podobnie jak w modelu RAM). Obie pamięci podzielone są na bloki rozmiaru B.

W momencie gdy program żąda dostępu do pewnej komórki pamięci następuje jedna z trzech akcji:

1. Blok zawierający żądaną komórkę znajduje się w pamięci podręcznej – zwracana jest przechowywana wartość, nie następuje dostęp do pamięci głównej.

2. Blok zawierający żądaną komórkę nie znajduje się w pamięci podręcznej; w pamięci podręcznej jest wolne miejsce na przynajmniej jeden blok – cały blok sprowadzany jest do cache'a.

3. Blok zawierający żądana komórkę nie znajduje się w pamięci podręcznej; w pamięci podręcznej nie ma żadnego wolnego miejsca – sprowadzany jest cały blok i umieszczany w miejsce bloku, który zostanie użyty najpóźniej w przyszłości.

Zauważmy, że taka maszyna nie jest możliwa do zaimplementowania ze względu na to, że algorytm wyboru bloków do usunięcia z cache'a jest *offline*. Istnieją stosunkowo proste strategie *online* (np. LRU, FIFO), które przy rozsądnych założeniach są gorsze od optymalnej jedynie o niewielki stały współczynnik [6]. Stosowanie rzeczywistych strategii w modelu bardzo skomplikowałoby analizę i uczyniłoby ją zależną od implementacji.

Podstawową miarą złożoności algorytmów będzie dla nas teraz ilość transferów z pamięci głównej do podręcznej: $MT(N) = MT_{B,M}(N)$. Nasz model będzie brał pod uwagę zysk związany z korzystaniem z niedawno odwiedzanych komórek pamięci (*temporal locality*) i umieszczaniem wspólnie wykorzystywanych danych obok siebie w pamięci (*spatial locality*).

Wymagamy by algorytm nie wykorzystywał parametrów B i M. W ten sposób klasa programów dla maszyny RAM i cache-oblivious jest dokładnie taka sama i nasz model ma znaczenie tylko przy analizie. Istnieje wynik teoretyczny mówiący, że wydajny algorytm w cache-oblivious będzie wydajny dla szerokiej klasy modeli z dowolną liczbą poziomów pamięci podręcznej [6].

1.3 Słowniki w modelu cache-oblivious

Opisane tutaj struktury znane są jako *cache-oblivious B-tree* i zostały zaczerpnięte z [3]. Zarówno statyczny jak i dynamiczny słownik umożliwia wyszukiwanie w $O(\log_B N)$ w pesymistycznym przypadku a operacje modyfikujące dynamiczny słownik kosztują $O(\log_B N + \lg^2 N/B)$ w sensie zamortyzowanym.

Warto zauważyć, że dolne ograniczenie na sortowanie przy użyciu porównań w modelu pamięci zewnętrznej to $\Theta(\frac{N}{B}\frac{\log_B N}{\log_B M})$[1] więc zastosowanie serii wstawień do opisanego słownika nie prowadzi do optymalnego algorytmu sortowania.

Znane są różne alternatywy dla opisanych struktur, m.in.:

- COLA (Cache-Oblivious Lookahead Array [4]) umożliwia wyszukiwanie w $O(\lg N)$ ale optymalne wstawianie w $O((\lg N)/B)$ w senise zamortyzowanym.
- Drzewa wykładnicze [2] są koncepcyjnie prostsze i umożliwiają wykonywanie wszystkich podstawowych operacji w $O(\log_B N)$ w najgorszym wypadku (nie-zamortyzowane).
- Shuttle Tree [4] pozwala na optymalne wyszukiwanie w $O(\log_B N)$ i wstawianie w zastraszającym:

  $$O\left(\frac{\log_B N}{B^{\Theta(1/(\log\log B)^2)}} + \frac{\log^2 N}{B}\right)$$

  To jest asymptotycznie szybciej od przedstawionej tutaj struktury o ile jest taka stała c, że:

  $$B \leq (\log N)^{1+c/\log\log\log^2 N}$$

2 Statyczny słownik

2.1 Złożoność przeszukiwania binarnego

Najprostszą strukturą realizującą statyczny słownik jest posortowana tablica, w której wyszukujemy elementy binarnie. Okazuje się, że w modelu cache-oblivious taka struktura nie jest asymptotycznie optymalna.

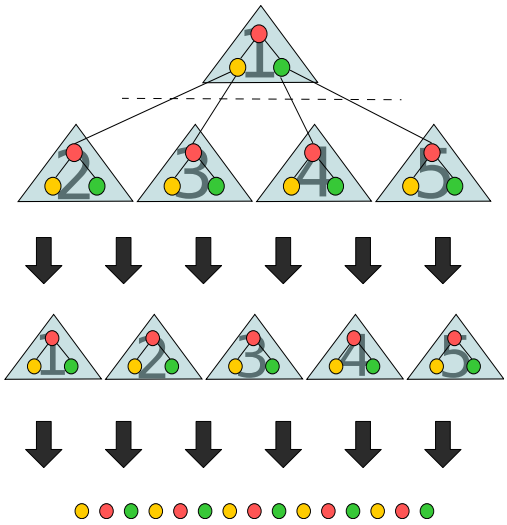
Wyszukując binarnie w tablicy rozmiaru N wykonamy $\Theta(\lg N)$ zejść rekurencyjnych, z których pierwszych $\Theta(\lg N - \lg B)$ będzie wymagało sprowadzenia całego świeżego bloku z pamięci a ostatnie $O(\lg B)$ będzie operować tylko i wyłącznie na jednym bloku. Zatem liczba dostępów do pamięci jest rzędu $\Theta(\lg N - \lg B) = \Theta(\lg \frac{N}{B})$. To o wiele gorzej od dolnego ograniczenia na pesymistyczny czas pojedynczego wyszukiwania, które wynosi $O(\log_B N)$.

Zauważmy, że gdy w trakcie wyszukiwania binarnego sprowadzamy do pamięci podręcznej cały blok z powodu pewnego elementu, którego klucz chcemy sprawdzić, dostajemy za darmo sporo sąsiednich elementów. Jednak ze względu na naturę algorytmu te elementy w większości wypadków nie będą nigdy potrzebne, a nawet jeśli będą to w dość odległej przyszłości. Potrzebny jest alternatywny sposób rozmieszczania kluczy w tablicy tak, by lokalność była bardziej wykorzystywana.

2.2 Reprezentacja van Emde Boasa

Jednym ze sposobów reprezentacji drzewa binarnego w postaci tablicy jest wypisanie jego elementów w kolejności *inorder*. W przypadku pełnych drzew binarnych mając indeks pewnego węzła i i jego wysokość w drzewie h (gdzie liście są na wysokości 1 a wartość h jest największa dla korzenia) możemy wyliczyć indeksy jego synów za pomocą wzoru $i \pm 2^{h-2}$ [5].

Na posortowaną tablicę 2^{h-1} elementów można spojrzeć jak na reprezentację *inorder* pewnego drzewa BST. Wtedy przeszukiwanie binarne w tablicy dokładnie odpowiada wyszukiwaniu w drzewie. Statyczny słownik skonstruujemy stosując alternatywną metodę rozkładania węzłów drzewa binarnego w tablicy zwaną *reprezentacją van Emde Boasa*.



Rysunek 1: Sposób konstrukcji reprezentacji van Emde Boasa.

Reprezentacja vEB jest określona rekurencyjnie: Drzewo będące pojedynczym liściem reprezentujemy jako tablicę długości 1 zawierającą jego klucz. Drzewo T mające N wierzchołków rozcinamy horyzontalnie w połowie otrzymując T_0 – drzewo złożone z wierzchołków T o głębokości nie większej niż $\lceil h(T)/2 \rceil$ – oraz drzewa $T_1, \ldots, T_k$ powstałe z wierzchołków o większej głębokości. Reprezentacja drzewa T jest konkatenacją reprezentacji drzew $T_0, T_1, \ldots, T_k$.

Mając dane binarne drzewo wyszukiwań (nawet implicite w reprezentacji inorder) można łatwo skonstruować reprezentację vEB z definicji. Jeśli razem z kluczem przechowywać będziemy indeksy synów (i ewentualnie ojca o ile zachodzi taka potrzeba) to przechodzenie po tak reprezentowanym drzewie jest bardzo proste. W odróżnieniu od reprezentacji inorder, indeksy synów nie wyrażają się łatwym wzorem ale [5] pokazuje jak pozbyć się jawnie trzymanych indeksów stosując pomocniczą tablicę rozmiaru $O(\lg N)$.

2.3 Poziomy szczegółowości

Załóżmy, że mamy tablicę, która reprezentuje pewne drzewo binarne w reprezentacji vEB. Przypuśćmy chwilowo, że jest to drzewo o wysokości 4, jak na rysunku 1. Możemy wyobrazić sobie jedno zejście rekurencyjne procedury konstruującej drzewo i pomyśleć o drzewie o wysokości 2, w którym korzeniem jest trójkąt 1 z rysunku, a jego synami trójkąty 2, 3, 4 i 5. Jest to drzewo 4-arne, w którym każdy wierzchołek ma w środku pełne drzewo binarne wysokości 2. Zauważmy, że drzewo przechowywane w każdym wierzchołku jest reprezentowane w postaci vEB w spójnym fragmencie tablicy.

I tak na każdą tablicę reprezentującą drzewo wysokości h za pomocą vEB możemy patrzeć na różnych poziomach szczegółowości:

Poziom 0: Drzewo złożone z jednego wierzchołka zawierającego drzewo binarne wysokości h.

Poziom 1: Drzewo wysokości 2 — korzeń i $2^{h/2}$ synów — w którym każdy wierzchołek przechowuje drzewo binarne wysokości $h/2$.

Poziom 2: Drzewo wysokości 4, w którym każdy wierzchołek wewnętrzny ma $2^{h/4}$ synów a w wierzchołkach przechowywane są drzewa binarne wysokości $h/4$.

...

Poziom i: Drzewo wysokości 2^i, w którym każdy wierzchołek wewnętrzny ma $2^{h/2^i}$ synów a w wierzchołkach przechowywane są drzewa binarne wysokości $h/2^i$.

...

Poziom $\lg h$: Drzewo wysokości $2^{\lg h} = h$, w którym każdy wierzchołek wewnętrzny ma $2^{h/2^{\lg h}} = 2^{h/h} = 2$ synów a w wierzchołkach przechowywane są drzewa binarne wysokości $h/h = 1$. To odpowiada oryginalnemu drzewu binarnemu.

Ogólniej, drzewo na poziomie $i+1$ konstruujemy z drzewa na poziomie i wyobrażając sobie każde przechowywane wewnątrz wierzchołka drzewo binarne na poziomie 1 i łącząc tak otrzymane drzewa zgodnie z krawędziami zewnętrznego drzewa.

Zauważmy, że na każdym poziomie prawdziwe jest, że drzewa przechowywane w węzłach drzewa-drzew reprezentowane są przez spójny fragment tablicy. Można myśleć o tym tak, że każda tablica odpowiedniego rozmiaru reprezentuje nam nie jedno, ale całą rodzinę drzew o różnej arności. Jeśli reprezentowane drzewo binarne było drzewem BST to drzewo na poziomie i przypomina B-drzewo o arności $2^{h/2^i}$. Stąd ta struktura bywa czasem nazywana *cache-oblivious B-tree*.

2.4 Złożoność wyszukiwania

Wyszukiwanie przeprowadzamy jak w normalnym drzewie BST, przechodząc ścieżką od korzenia do odpowiedniego węzła. Wyobraźmy sobie dane drzewo o wysokości h na poziomie szczegółowości d:

$$d = \left\lceil \lg\left(\frac{h}{\lg(B+1)}\right)\right\rceil \tag{2}$$

Na tym poziomie każdy wierzchołek przechowuje drzewo o $2^{h/2^d} - 1$ wierzchołkach. Możemy obliczyć, że:

$$2^{h/2^d} - 1 \leq 2^{h/(h/\lg(B+1))} - 1 = 2^{\lg(B+1)} - 1 = B \tag{3}$$

Więc każde takie drzewo mieści się w jednym bloku pamięci. Nadrzędne drzewo na poziomie d ma wysokość:

$$2^d = 2 \cdot 2^{d-1} \leq 2 \cdot 2^{\lg(h/\lg(B+1))} = 2h/\lg(B+1) \in O(h/\lg B) = O(\lg_B N) \tag{4}$$

Przejście ścieżką korzeń-liść w drzewie BST na najwyższym poziomie szczegółowości odpowiada przejściu ścieżką w drzewie na poziomie d gdzie w każdym odwiedzanym wierzchołku przechodzimy ścieżką korzeń-liść przechowywanego tam drzewa binarnego. Ponieważ drzewa przechowywane w wierzchołkach mają rozmiar mniejszy niż B, przejście po takim pojedynczym drzewie wymaga maksymalnie dwóch dostępów do pamięci (dwóch a nie jednego bo tablica reprezentująca drzewo może leżeć na połączeniu pomiędzy blokami). Zatem, ze wzoru 4, cała wyszukiwanie wymaga nie więcej niż $O(\lg_B N)$ dostępów do pamięci.

3 Dynamiczny słownik

3.1 Packed-memory structure

Packed-memory structure (znane też jako problem Ordered File Maintenance) jest prostą strukturą danych przechowującą ciąg elementów w tablicy. Każda komórka tablicy albo zawiera jakiś element albo jest pusta. Struktura gwarantuje, że każdy spójny podciąg k przechowywanych elementów mieści się w spójnym fragmencie tablicy rozmiaru $O(k)$. Wstawianie i usuwanie elementów ze struktury może wymagać prostego przeorganizowania pewnego fragmentu tablicy rozmiaru $O(\lg^2 N)$ w sensie zamortyzowanym.

Oznacza to, że w modelu cache-oblivious operacje `Insert` i `Delete` kosztują $O(\frac{\lg^2 N}{B} + 1)$ a operacja `Next` zwracająca indeks następnej niepustej komórki kosztuje $O(\frac{1}{B})$ (wszystkie koszty zamortyzowane), co jest atrakcyjną alternatywą dla klasycznej listy z dowiązaniami. Dodatkową zaletą bywa możliwość prostego sprawdzania w czasie stałym czy pewien element występuje przed innym w przechowywanym ciągu (w liście z dowiązaniami może to wymagać przejrzenia całej listy).

Szczegóły implementacji i analizę tej struktury można znaleźć np. w [3].

3.2 Konstrukcja dynamicznego słownika

Struktura będzie przechowywać elementy jako posortowany ciąg w packed-memory structure a w celu przyśpieszonego wyszukiwania będzie utrzymywać specjalne statyczne drzewo wyszukiwań jako indeks do PMS-a. Drzewo wyszukiwań będzie przechowywane w reprezentacji van Emde Boasa.

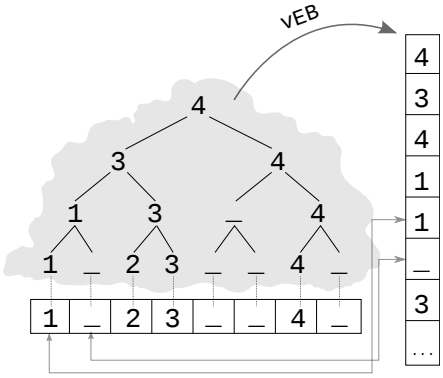
Drzewo wyszukiwań będzie pełnym drzewem binarnym, którego liście odpowiadają komórkom w packed-memory structure (niezależnie czy te komórki przechowują jakiś element czy są puste). Węzły wewnętrzne drzewa przechowują maksimum z kluczy swoich synów ("pusty klucz"jest mniejszy od wszystkiego, tj. liczony jak $-\infty$).

Drzewo jest statyczne w sensie struktury ale wstawianie i usuwanie elementów może wymagać uaktualniania kluczy w węzłach wewnętrznych tak, by niezmiennik pozostał spełniony.

3.2.1 Wyszukiwanie

Wyszukujemy schodząc rekurencyjnie w dół drzewa. Jeśli węzeł ma klucz pusty to całe poddrzewo nie zawiera elementów i nie warto schodzić niżej. Przypuśćmy, że pewien węzeł wewnętrzny ma klucz β i posiada dwóch synów o niepustych kluczach α i β. Z niezmiennika drzewa wiemy, że wszystkie klucze w lewym poddrzewie są nie większe niż α (a przynajmniej jeden z nich jest równy α). Ponieważ niepuste liście w drzewie są posortowane rosnąco wiemy, że wszystkie elementy w prawym poddrzewie są nie mniejsze niż α. Zatem szukany element x będzie znajdował się w lewym poddrzewie jeśli $x \leq \alpha$ a w prawym poddrzewie w przeciwnym wypadku.

Wiemy, że rozmiar packed-memory structure jest $O(N)$ więc statyczne drzewo-indeks ma $O(N)$ węzłów. Jak ustaliliśmy w rozdziale 2.4 przejście ścieżką od korzenia do liścia w drzewie binarnym reprezentowanym w vEB kosztuje $O(\log_B N)$ dostępów do pamięci.



Rysunek 2: Przykładowa dynamiczna struktura reprezentująca zbiór kluczy $\{1, 2, 3, 4\}$.

3.3 Wstawianie i usuwanie

Operacje modyfikujące drzewo rozpoczynają się od znalezienia elementu do usunięcia bądź miejsca do wstawienia nowego elementy za pomocą operacji opisanej powyżej. Znając indeks możemy dokonać modyfikacji w PMS co zaowocuje zmianą pewnego spójnego fragmentu tablicy rozmiaru k (gdzie k będzie amortyzować się do $O(\lg^2 N)$). Następnie przechodzimy postorder całe poddrzewo złożone z przodków liści odpowiadających zmodyfikowanym komórkom PMS-a, uaktualniając klucze tak by przywrócić odpowiedni niezmiennik.

Spróbujemy oszacować ilość transferów z pamięci potrzebną na wykonanie tego przejścia postorder przez $O(\log_B N + \frac{k}{B})$. Wyobraźmy sobie drzewo na poziomie szczegółowości d takim jak zdefiniowano równością 2 w rozdziale 2.4. Podzielimy odwiedzaną część drzewa na trzy strefy:

Strefa A: Dwa najniższe poziomy drzewa (liście i ojcowie liści).

Strefa B: Najniższy wspólny przodek liści odpowiadających zmodyfikowanym elementom PMS-a wraz z wszystkimi jego potomkami, którzy nie są w strefie A.

Strefa C: Ścieżka od korzenia do najniższego wspólnego przodka rozpoczynającego strefę B.

Podliczymy dostępy do pamięci wykonane w trakcie odwiedzania węzłów z każdej strefy osobno.

Węzły najniższego poziomu będą odwiedzane tylko raz. Węzły należące do przedostatniego poziomu będą odwiedzane przed i po każdym odwiedzeniu węzła najniższego poziomu. O ile $M \geq 4$ to po powrocie z węzła najniższego poziomu jego ojciec będzie wciąż w cache'u. Zatem w strefie A wykonamy $O(\frac{k}{B})$ dostępów do pamięci.

Analizując strefę B wyobraźmy sobie na chwilę prawdziwe, binarne drzewo na najwyższym poziomie szczegółowości odpowiadające tej strefie. Ma tyle liści ile jest odwiedzanych węzłów na poziomie szczegółowości d na górze strefy A, tj. $O(\frac{k}{B})$. Drzewo binarne o $O(\frac{k}{B})$ liściach ma nie więcej niż $O(\frac{k}{B})$ wierzchołków więc nawet zakładając pesymistycznie, że zmarnujemy dostęp do pamięci na każdy z nich nie przekraczamy $O(\frac{k}{B})$.

Ścieżka korzeń-LCA jest na pewno krótsza niż $O(\log_B N)$ bo taka jest wysokość całego drzewa na tym poziomie szczegółowości więc w strefie C płacimy $O(\log_B N)$.

Zatem w sumie zapłacimy $O(\log_B N + \frac{k}{B})$. Z własności PMS-a wiemy, że składnik $O(\frac{k}{B})$ amortyzuje się do $O(\frac{\lg^2 N}{B})$ więc ostatecznie procedury modyfikujące drzewo kosztują $O(\log_B N + \frac{\lg^2 N}{B})$ w sensie zamortyzowanym.

Literatura

[1] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31:1116–1127, September 1988.

[2] Michael A. Bender, Richard Cole, and Rajeev Raman. Exponential structures for efficient cache-oblivious algorithms. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, ICALP '02, pages 195–207, London, UK, UK, 2002. Springer-Verlag.

[3] Michael A. Bender, Ziyang Duan, John Iacono, and Jing Wu. A locality-preserving cache-oblivious dynamic dictionary. *J. Algorithms*, 53:115–136, November 2004.

[4] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-oblivious streaming b-trees. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '07, pages 81–92, New York, NY, USA, 2007. ACM.

[5] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. Cache oblivious search trees via binary trees of small height. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '02, pages 39–48, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[6] Eric Demaine. 6.851: Advanced data structures; lecture 19. http://courses.csail.mit.edu/6.851/spring07/scribe/lec19.pdf, 2007.