Ghent University

Algoritmen en Datastructuren 3

Project Btree

Jan-Pieter Baert

Academiejaar 2019-2020

Inhoudsopgave

1 Implementatie

1.1 Algemene structuur

In het bestand *utils/globs.h* definiëren we eerst het aantal sleutels van de btree, dit moet een even getal zijn en noemen we '*NUMBER_OF_BTREE_KEYS*', voor de leesbaarheid van het verslag noemen we dit hierna *NB*.

De algemene structuur van de implementatie is een btree waarin we de ouder, het aantal elementen, zijn kinderen en zijn elementen bijhouden. De kinderen van een btree zijn een lijst van btrees (of een *NULL*-pointer bij bladeren), de lengte van deze lijst is '*NB*+1' De elementen van een btree zijn een lijst van btreeElement's, de lengte van deze lijst is *NB*.

1.2 Keuzes

1.2.1 Grafstenen

In dit project hebben we gekozen voor grafstenen, deze grafstenen worden aangeduid door de waarde van een element gelijk te stellen aan een **NULL**-pointer. We hebben gekozen voor grafstenen omdat we in de cursus van AD3 (oefening 19, p 32) zien dat het verschil in diepte vrij klein blijft. Dit zelfs voor een zeer laag percentage sleutels die geen grafsteen zijn.

1.2.2 Structuur

We hebben gekozen om de elementen rechtstreeks in de btree te steken en niet pointers van de elementen. Initiëel kozen we ervoor om de pointers zelf in de btree op te slaan, maar we merkten echter dat het opslaan van de elementen zorgde voor een sterke versnelling van het programma. De reden dat dit sneller is, is omdat alle elementen, van één laag in de btree, in één cache blok geladen kunnen worden, dit zorgt voor betere lees- en schrijfsnelheden.
Om de cache optimaal te gebruiken, maken we gebruik van de cachegrootte (die aan de compiler als *CACHESIZE* wordt meegeven) om NB te berekenen, zodanig dat deze nog net volledig in één cacheblok past.

2 Caching

2.1 Dynamisch tov. statische btrees

Om statische btrees te simuleren is er een optie bij het genereren van testen om eerst al de toevoegoperaties te doen en pas daarna de andere operaties (waarbij de btree dus niet meer moet veranderen). Om dynamische btrees te simuleren gebruiken we input met een variatie aan verschillende operaties, waar de boom dus gedurende het volledige proces kan veranderen door toevoegoperaties.
Om onze btrees te testen gebruiken we de volgende configuratie:
$CACHESIZE = 8192$, $CACHEASSOC = 8$, $CACHELINESIZE = 64\text{B}$

grootte	100000	200000	300000	400000	500000
statisch	17.0%	18.3%	19.0%	18.6%	18.3%
dynamisch	15.1%	16.7%	17.3%	17.8%	18.1%

Tabel 1: Het verschil in cache miss % in statische tov. dynamische btrees

In tabel 1 zien we dat er weinig verschil op te merken is tussen het statisch en dynamisch zijn van een boom. Een dynamische input zorgt wel voor een gemiddeld kleinere boom bij de instructies, dit kan het kleine verschil tussen statisch en dynamisch verklaren.

2.2 Invloed van cachegrootte

Om de invloed van de cachegrootte te bekijken, zullen we zowel voor een dynamische als voor een statische input (voor beiden 100000 testoperaties) kijken naar het percentage cache-missers.

cachegrootte (in bytes)	512	1024	2048	4096	8192
statisch	14.9%(660M)	9.3%(236M)	8.7%(129M)	11.8%(938M)	16.9%(641M)
dynamisch	14.2%(316M)	8.9%(115M)	8.6%(67M)	10.9%(515M)	15.1%(368M)

Tabel 2: Het verschil in cache-missers percentage (en totaal aantal refs) tussen verschillende cachegroottes

In tabel 2 zien we dat 2048 bytes de beste configuratie blijkt te zijn om het aantal cache misses zo klein mogelijk te houden. We merken ook dat het totaal aantal refs ook het minst is bij 2048 bytes en dat het veel groter is bij 4096 bytes.