제18회 한국정보처리학회 추계학술발표대회 논문집 제9권 제2호 (2002.11)

lpCSB+ - 트리: 레벨 프리페칭 기법을 이용하는 향상된 주기억장치 상주형 색인구조

홍 현 택 피 준 일 송 석 일 유 재 수
충북대학교 정보통신공학과 및 컴퓨터정보통신연구소
{hongry, pji, prince}@netdb.chungbuk.ac.kr, yjs@cbucc.chungbuk.ac.kr

lpCSB+ - tree : An Enhanced Main Memory Index Structure Employing Level Prefetching Technique

Hyun Taek Hong, Jun Il Pee, Seok Il Song, Jae Soo Yoo
Dept. of Computer and Communication Engineering, Chungbuk National University

요 약

주기억장치 상주형 색인구조에서는 2차 캐쉬 실패가 성능에 매우 큰 영향을 미친다. 기존에 제안된 주기억장치 상주형 색인구조들은 2차 캐쉬 실패를 고려하긴 했지만 여전히 트리의 각 레벨을 접근할 때는 2차 캐쉬 실패가 발생한다. 본 논문에서는 이러한 문제점을 인식하고 트리 순회 시 각 레벨을 방문할 때도 캐쉬 실패가 발생하지 않는 주기억장치 색인구조를 제안한다. 제안하는 색인구조는 다음 레벨에서 방문할 가능성이 있는 노드들을 프리페칭하여 다음 레벨을 방문할 때도 캐쉬 실패가 발생하지 않도록 한다. 또한, 기본적인 구조는 노드그룹 개념을 이용하여 노드의 팬-아웃을 증가시키는 CSB+-트리에 기반하지만 CSB+-트리의 단점인 분할 비용의 증가문제를 해결하기 위한 방법을 제안한다. 시뮬레이션을 통해 기존의 색인구조와 비교하여 제안하는 색인구조의 우수성을 보인다.

1. 서 론

현대 컴퓨터 구조에서는 중앙처리장치(CPU)의 속도와 주기억장치의 접근속도의 차이가 점점 커지면서 이들 간에 병목현상이 발생한다. 최근의 중앙처리장치는 이 병목현상을 해소하기 위해 캐쉬기법을 도입하고 있으며 일반적으로 1차 및 2차 캐쉬를 가지는 계층적 기억장치 구조를 가지고 있다.

최근 주기억장치의 가격이 하락하면서 주기억장치 데이터베이스 관리 시스템(Main Memory Database Management System:MMDBMS)이 널리 쓰이고 있는데 디스크 I/O가 발생하지 않는 MMDBMS에서는 주기억장치와 중앙처리장치 간의 속도 차이가 전체 성능에 미치는 영향이 크다. 이에 관련한 기존연구의 보고에 의하면 특히 2차 데이터 캐쉬 실패가 성능저하에 가장 큰 영향을 미친다[1].

특히, 2차 데이터 캐쉬 실패는 순차적 데이터접근을 하지 않는 색인구조와 같은 데이터 구조에 더 큰 영향을 미친다. 최근 이를 극복하기 위한 연구가 활발히 진행중이며 그 결과 CSS-트리[2], CSB+-트리[3], pkB-트리[4], CR-트리[5], pB+-트리[6]와 같은 캐쉬실패를 고려하는 색인구조들이 제안되었다. 이 색인구조들은 여러 측면에서 색인구조에서 발생할 수 있는 캐쉬 실패를 줄였지만 여전히 트리의 각 레벨을 접근할 때는 캐쉬 실패가 발생하고 있으며, 이는 성능에 큰 영향을 미치고 있다.

본 논문에서는 이러한 문제점에 착안하여 각 레벨에서도 캐쉬 접근 실패가 발생하지 않는 색인구조를 제안한다. 제안하는 색인구조인 lpCSB(level prefetching CSB)+-트리는 프리페치 기법을 사용하여 현재 접근하는 노드의 자식과 손자 노드들을 캐쉬에 미리 올려놓음으로써 트리의 다음 레벨을 접근할 때 캐쉬 실패가 발생하지 않도록 한다.

본 논문의 구성은 다음과 같다. 2장에서 관련연구에 대하여 논의한 뒤, 3장에서 제안하는 lpCSB+-트리의 구조와 삽입, 탐색 방법을 서술하고 탐색 성능을 분석한다. 4장에서는 시뮬레이션을 통하여 제안하는 기법의 우수성을 증명하고 5장에서 결론을 맺는다.

2. 관련연구

CSS(Cache Sensitive Search)-트리는 OLAP 환경을 위해 고려된 색인구조로서 키들을 물리적으로 연속된 공간에 저장하여 포인터를 제거하였다. 한 노드의 자식노드 위치는 k-array 기법을 사용하여 계산된다. 이로인해 한 노드가 포함할 수 있는 엔트리의 개수가 증가하였고, 보다 효과적으로 캐쉬 실패 횟수를 줄일 수 있었다. 하지만 변경이 발생할 경우 전체 트리를 재구성해야하는 단점이 있다.

CSB+(Cache Sensitive B+)-트리는 CSS-트리의 기법을 OLTP 환경을 위해 B+-트리에 적용한 형태로서 변경에도 효과적으로 대처할 수 있다. 한 노드의 자식노드들을 물리적으로 연속된 공간인 노드 그룹에 저장하고, 부모 노드에는 자식 노드들을 갖고있는 노드 그룹을 향한 포인터 한 개만 저장함으로서 부모노드의 포인터를 줄이는 기법을 사용하였다.

CSS-트리와 CSB+-트리가 포인터를 줄이는 기법을 사용한 반면 pkB-트리와 CR-트리는 키의 크기를 줄이는 기법을 사용하였다. 부분 키(partial key) 기법을 사용한 pkB-트리는 전체 키 중에서 고정된 일부분만을 키로 이용하여 캐쉬 실패 횟수와 키 비교 비용을 줄인다. 이 기법은 전체 키를 사용하지 않으므로, 가변길이의 키나 크기가 큰 키의 경우에도 효과적으로 색인을 구성할 수 있다는 장점이 있다.

CR(Cache-conscious R)-트리는 다차원 색인구조인 R-트

※ 본 연구는 2001년도 한국학술진흥재단(KRF - 2001 - 041 - E00233)의 지원으로 수행되었음

리의 키를 양자화(Quantization)를 통하여 압축하는 기법을 사용하였다. MBR(Minimum Bounding Rectangle) 영역을 일정한 개수로 분할한 뒤 부모의 왼쪽 하단 좌표를 기준으로 값을 표시함으로써 자식 노드들의 영역을 나타내는 값을 효과적으로 줄일 수 있다. 하지만 이 경우 원래의 MBR보다 크게 영역이 설정될 수 있다는 단점이 있다.

프리페치 기법을 B+-트리에 도입한 pB+(prefetching B+)-트리는 기존의 방법에서 노드의 크기를 캐쉬라인의 크기로 제한했던 것과 달리 노드 크기를 캐쉬라인의 정수 배로 늘렸다. 또한 범위 검색을 위하여 Jump-pointer 기법을 제시하였다. 프리페치 기법은 접근하고자하는 기억장치의 데이터를 미리 캐쉬로 가져오는 기법으로서 CPU의 연산과는 독립적으로 동작한다. 노드가 여러 개의 캐쉬라인으로 구성되어있을 경우 노드의 첫 번째 캐쉬라인을 접근하는 동안 나머지 캐쉬라인을 프리페칭한다면, 노드가 여러 개의 캐쉬 라인으로 구성되어있다 하더라도 노드를 접근하는 동안 캐쉬 실패는 발생하지 않게 된다. 하지만, 이 방법에서도 여전히 트리의 높이에 비례하여 캐쉬 실패가 발생하게 된다.

3. IpCSB+-트리

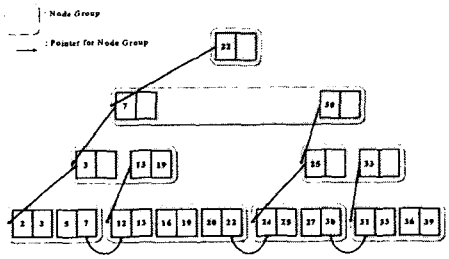
3.1 개요

위에서 언급된 색인 구조들의 공통적인 문제점은 각 레벨에서 캐쉬 실패가 발생한다는 것이다. [6]에서는 이 문제를 인식하고 자식 노드나 손자 노드들을 병렬로 프리페치하는 등의 해결책과 그에 대한 문제점도 지적하고 있다. 지적된 문제점으로 가장 어려운 문제는 접근하게 될 자손 노드들을 미리 알 수 없다는 점이다. 자손 노드들을 프리페치 하기 위해서는 그 노드들의 주소를 알아야하고, 또한 자식 노드들을 프리페치 한다고 하더라도 팬-아웃(fan-out)이 크기 때문에 프리페치 할 양이 너무 많게 된다.

이러한 문제점에 착안하여 본 논문에서는 현재 접근중인 노드의 손자 노드들을 프리페치하는 접근 방법을 사용하며, 기본 구조는 CSB+-트리를 사용한다.

CSB+-트리의 문제점으로 지적되는 것은 색인에 사용되는 키들을 정수로 한정했고 형제 노드들을 모두 연속된 공간에 저장하여 변경을 수행할 때 어렵다는 것이었다. 본 논문에서는 변경에 대한 비용을 줄이고 더불어 프리페치 기법을 적용하기 위해서 CSB+-트리의 구조를 수정했다. 또한, CSB+-트리의 변경 비용을 줄이기 위한 다른 방법으로 새로운 삽입 알고리즘을 제시하였다.

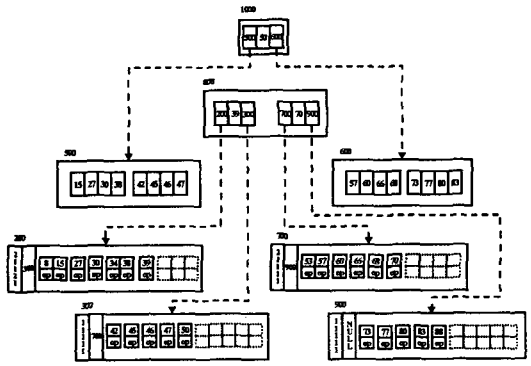
제안하는 색인 구조에 대한 설명을 돕기 위해서 먼저 CSB+-트리의 구조에 대해서 좀 더 자세히 기술해 본다.



(그림 1) CSB+-트리의 구조

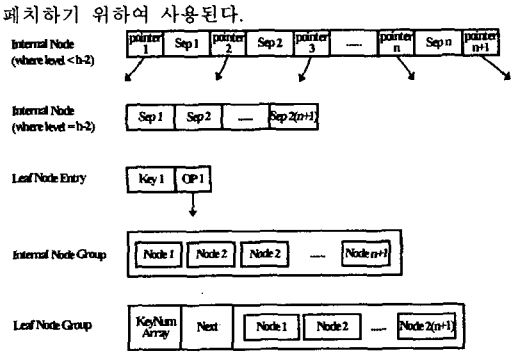
그림 1은 CSB+-트리의 전체적인 구조를 보여준다. CSB+-트리는 균형 다원 탐색 트리이다. d차의 CSB+-트리에 있는 각 노드는 $m(d \le m \le 2d)$개의 키들을 포함한다. CSB+-트리는 한 노드의 자식 노드들을 물리적으로 연속된 공간인 노드 그룹에 저장하고, 임의의 자식 노드의 위치는 계산에 의하여 얻게된다. CSB+-트리의 노드는 자식 노드 그룹의 첫 번째 노드 포인터만을 저장하여 공간 활용도를 높인다. 따라서, CSB+-트리는 B+-트리에 비해서 한 노드 당 저장할 수 있는 키의 개수가 더 많아지게 된다. 예를 들어서 노드 크기가 64바이트이고 키와 포인터의 크기가 모두 4바이트라면 B+-트리는 노드 당 7개의 키를 포함할 수 있는 반면에 CSB+-트리는 14개의 키를 포함할 수 있다. 제안하는 색인구조는 CSB+-트리의 위와 같은 속성들을 상속받는다.

3.2 구조



(그림 2) IpCSB+-트리의 구조

그림 2는 제안하는 색인 구조의 한 구성 예를 보여준다. 비 단말 노드의 자식 노드들은 물리적으로 연속적인 공간에 저장된다. h를 색인 트리의 높이라고 하고 루트노드의 레벨을 0으로 할 때, 제안하는 색인 구조에서 레벨이 h-2 보다 적은 비 단말 노드들은 프리페치하게 될 손자노드들의 노드 그룹에 대한 포인터들을 가지고 있다. 이 포인터는 다음 레벨에서 접근할 노드가 결정이 되면, 그 노드의 자식노드들을 프리페치하기 위하여 사용된다.



(그림 3) 노드와 노드 그룹의 구조

그림 3에서는 제안하는 색인 구조의 노드와 노드 그룹 구조를 보여주고 있다. 레벨이 h-2보다 작은 비 단말 노드는 손자 노드 그룹들에 대한 포인터와 자식노드들에 대한 구분자로 구성되어 있고, 레벨이 h-2에 해당하는 비 단말 노드는 손자 노드들이 존재하지 않으므로 포인터를 가질 필요가 없으며 단지 구분자로만 구성되어 있다. 따라서, 전체 트리에 존재하는 포인터의 수는 CSB+-트리와 동일하다. 단말 노드의 경우는 노드의 크기에 제한이 없고 각 엔트리는 키값과 객체에 대한 포인터로 구성이 된다. 노드의 크기에 제한을 두지

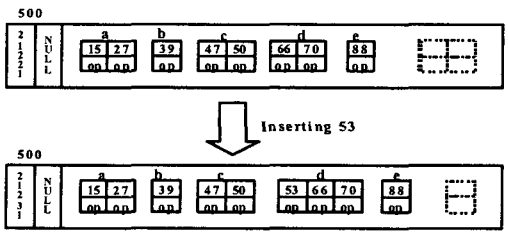
않은 대신에 단말 노드 그룹에서는 각 노드들이 몇 개의 엔트리로 구성되어 있는 지에 관한 정보를 유지하고 있어야한다. 이 값들은 노드 그룹의 맨 앞에 배열의 형태로 갖고 있게 되며, 범위 탐색을 위한 다음 노드 그룹을 위한 포인터는 단말 노드 그룹에 한 개만 존재하면 된다. 이런 정보를 위한 추가적인 공간에 대한 부담은 하나의 노드 그룹에 포함되는 노드들의 개수가 작아서 4~8비트면 충분하므로 무시할 수 있다.

3.3 삽입

제안하는 색인구조의 삽입알고리즘은 두 단계로 수행된다. 첫 번째 단계는 새로운 엔트리를 삽입할 단말노드를 찾는 단계로서 맨 처음 루트 노드를 접근한다. 하지만 루트노드를 접근하기 전에 루트노드의 자식노드들을 병렬로 프리페치한다. 루트노드와 자식 노드그룹에 대한 포인터는 색인 설명자에 미리 기록이 되어 있다고 가정한다. 루트노드에서 삽입자가 다음 방문할 자식노드를 결정하면 그 자식노드의 자식들을 병렬로 프리페치한다. 즉, 두 레벨 다음에 접근하게 될 가능성이 있는 노드들을 캐쉬에 올려놓는 것이다. 이때 루트노드의 손자노드들의 노드그룹에 대한 포인터는 루트노드에 저장된다. 다음에 삽입자는 결정된 자식노드를 방문한다. 삽입자는 이러한 과정을 다음 레벨에서도 반복해서 새롭게 삽입할 엔트리를 위한 단말 노드를 찾는다.

두 번째 단계에서 삽입자는 새로운 엔트리를 단말노드에 삽입한다. 이때 넘침이 발생하면 분할을 수행하여 넘침을 처리한다. 단말 노드에서의 분할 알고리즘은 CSB+-트리의 변경 비용을 줄이는 형태로 진행된다. 제안하는 색인 구조의 단말 노드들은 그 크기가 고정되어 있지 않다. 단말 노드 그룹의 크기는 최대만큼 이미 확보되어 있으며 그 공간을 활용하여 단말 노드의 크기를 증가시키는 형태로 삽입이 진행된다. 즉, 분할은 노드 그룹이 꽉 찼을때만 발생하게 된다.

그림 4에서와 같이 한 단말 노드 그룹이 메모리 주소 500번지에 저장되어있고, 단말 노드의 개수는 5개인 경우를 가정하자. 만약 단말 노드의 엔트리 개수를 2개로 제한한다면 새로운 엔트리 53은 d에 삽입되면서 넘침을 유발할 것이고, 그로 인해 노드 d는 분할되며 노드 그룹의 노드의 개수가 최대 5이므로 다시 노드 그룹이 분할될 것이다. 그러나 본 논문에서 제안하는 분할알고리즘은 그림 4에서처럼 노드 d를 분할하지 않는다. 대신에 노드 그룹의 여유공간으로부터 새로 삽입되는 엔트리를 위한 공간을 할당해서 사용한다. 즉, 노드 d의 크기만 커지게 되는 것이고, 노드 그룹은 분할되지 않는다. 제한하는 색인 구조에서의 분할은 노드 그룹의 예약된 공간이 모두 사용되었을 때만 수행된다. 비 단말 노드의 분할은 CSB+-트리와 유사하다.



(그림 4) 단말 노드 그룹에 대한 삽입 방법

3.4 탐색

탐색 알고리즘은 삽입의 첫번째 연산과 유사하다. 루트 노드를 접근하기 이전에 색인 설명자에 기술된 포인터를 참조하여 루트 노드와 자식 노드들을 프리페치한다. 그리고 다음 순회할 자식 노드 A가 결정되면 루트노드에 포함된 포인터를 참조하여 A의 자식 노드들을 포함하고 있는 노드 그룹을 프리페치하고, A노드에 대한 탐색을 수행한다. 이러한 과정을 단말 노드에까지 진행한다. 이렇게 되면 탐색 수행중에 각 레벨에서 발생하는 L2 캐쉬 실패 횟수를 줄일 수 있다.

3.5 탐색 성능 분석

제안하는 색인 구조의 탐색성능과 pCSB+(prefetch CSB+)-트리의 탐색 성능을 캐쉬 실패 회수 관점에서 수학적으로 분석한다. 높이가 h인 색인 트리의 탐색 연산의 총 수행시간(T_{total})를 다음의 수식 1로 표현할 수 있다.

$$T_{total} = T_1 + \sum_{l=1}^{h-1} \left\{ \alpha T_1 + wT_p \left(m - \alpha - pn \right) \right\} \quad \text{(수식 1)}$$

$$\alpha = \begin{cases} pn = 0, & 1 \\ pn > 0, & 0 \end{cases}$$

$$T_1 = T_c + \left\{ T_p \left(w - 1 \right) \right\}$$

T_1은 두개 이상의 캐쉬 라인으로 구성된 하나의 노드를 적재하는데 걸리는 평균 지연시간이다. 이 시간은 첫 번째 캐쉬라인을 적재하는데 걸리는 캐쉬 실패 지연 시간(T_C)과 하나의 노드를 구성하는 나머지 캐쉬라인을 프리페치하는데 걸리는 지연시간으로 구성된다. T_P는 하나의 캐쉬라인을 프리페치하는데 걸리는 시간이며, pn은 하나의 노드그룹을 구성하는 노드 중 이미 프리페치된 노드들의 개수이고, w는 하나의 노드 그룹을 구성하는 캐쉬 라인의 수이다. pn이 0보다 크면 최소한 노드그룹의 첫 번째 노드는 이미 프리페치되어 있다는 것이다. 그렇지 않다면, 노드그룹의 첫 번째 노드를 로드하는데 완전 캐쉬 실패 지연 시간만큼 지연되게 된다. tn은 노드 그룹에서 실제로 접근할 자식노드의 위치이다. tn이 pn과 같거나 pn보다 작다면 캐쉬 실패로 인한 지연은 발생하지 않으며 프리페치로 인한 지연도 발생하지 않는다. 만일 tn이 pn보다 크고 pn이 0보다 크다면 프리페치로 인한 지연이 발생한다. 이처럼 pn은 색인 트리의 성능에 심각한 영향을 끼친다. 만일 pn이 0보다 크다는 것을 확신할 수 있다면 매우 큰 성능향상을 기대할 수 있다.

pCSB+-트리의 T_{total}은 각 레벨에서 캐쉬실패를 경험해야 하므로 수식 2와 같다.

$$T_{total} = \sum_{l=0}^{h-1} T_1 \quad \text{(수식 2)}$$

제안하는 색인구조의 탐색성능은 전적으로 pn와 h에 의존적이다. 만일 pn이 0이면 각 레벨을 접근할 때마다 T_1 지연이 발생하는데, 이러한 pn은 노드에서의 처리시간에 의존적이다. 일반적으로 lpCSB+-트리의 h는 pCSB+-트리보다 높은데, 제안하는 방법에서는 손자노드들을 프리페치해야 하므로 트리의 노드 크기를 pCSB+-트리만큼 확장할 수 없기 때문이다.

단순히 제안하는 색인 트리의 탐색 성능을 수식을 통해서 pCSB+-트리와 비교할 수는 없다. 왜냐하면 탐색성능에 큰 영향을 미치는 h와 pn이 성능에 큰 영향을 미치는 요인이기 때문이다.

4. 시뮬레이션

시뮬레이션에 사용된 시스템은 펜티엄-IV 1.7GHz 프로세

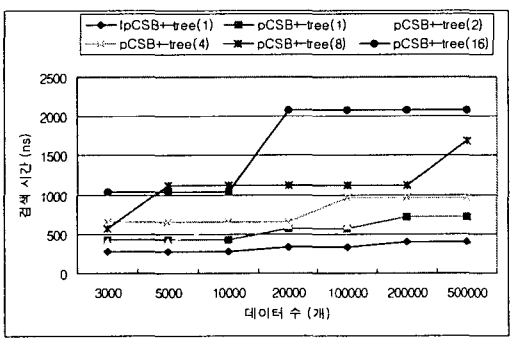
서에 256Mbyte의 메모리를 가지며, 운영체제는 RedHat 리눅스 7.1을 사용하였고, 컴파일러는 gcc 2.96이었다. 실험에 사용된 파라미터들이 표1에 나타나있다.

시뮬레이션에 사용된 데이터 집합은 균등 분포의 정수 500,000개로서 데이터 개수를 변경시키며 시뮬레이션을 수행하였고, 또한 색인 구성 시 한 노드의 크기는 1~16 캐쉬 라인수로 변경시키며 수행하였다. 시뮬레이션에 사용된 펜티엄-IV CPU의 총 L2 캐쉬의 크기는 256KBytes이고, 하나의 L2 캐쉬 라인의 크기는 64Bytes였다. 캐쉬 검색 시간과 관련하여 만약 검색하고자 하는 영역이 캐쉬에 없을 경우 발생하는 캐쉬 실패 지연시간은 88.2 ns였으며 하나의 캐쉬 라인을 프리페치 하는데 드는 지연시간은 5.5 ns이었다. 추가적으로 검색을 수행할 영역이 캐쉬 라인 상에 있을 때, 캐쉬 라인 상에서 원하는 키 값을 찾는데 소요되는 시간은 54.2 ns였다.

<표 1> 시뮬레이션 파라미터

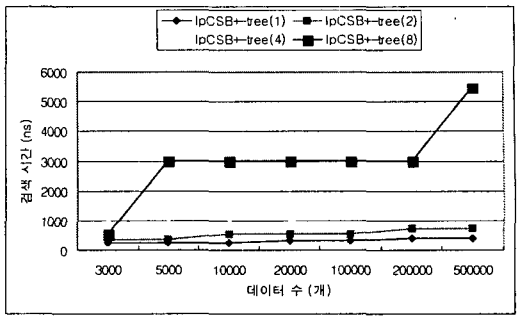
항목		값
시스템 파라미터	전체 L2 캐쉬 크기	256 KByte
	L2 캐쉬 라인 크기	64 Byte
	캐쉬 실패 비용	88.2 ns
	캐쉬 프리페치 비용	5.5 ns
	캐쉬라인 탐색 비용	54.2 ns
성능 파라미터	데이터 집합	균등분포 500,000 개
	노드 크기	1 ~ 16 캐쉬라인

시뮬레이션을 통해 제안하는 색인 기법을 pCSB+-트리와 검색 시간 관점에서 비교를 하였다. 먼저 그림 5는 데이터의 개수를 3000~500,000개까지 변경시키면서 lpCSB+-트리와 pCSB+-트리를 실험한 결과이다. 성능 평가시 lpCSB+-트리의 경우 노드의 크기를 하나의 캐쉬라인 크기로 하였으며, pCSB+-트리의 경우에는 노드의 크기를 1~16개의 캐쉬라인으로 변경하면서 시뮬레이션을 수행하였다. 시뮬레이션 결과 lpCSB+-트리가 pCSB+-트리보다 모든 경우에서 우수한 성능을 나타내었다. pCSB+-트리의 경우 한 노드의 크기를 하나의 캐쉬 라인으로 하였을 경우 가장 우수한 성능을 나타내었으며, 제안하는 lpCSB+-트리는 이보다 검색 시간측면에서 약 20% 향상된 결과를 나타내었다. 이는 상위노드에서 자식노드를 탐색하는 과정에서 원하는 자식노드를 프리페치하였기 때문이다.



(그림 5) 데이터수와 노드 크기 변화에 따른 pCSB+-트리와 lpCSB+-트리의 검색시간

그림 6은 lpCSB+-트리의 노드 크기를 여러 개의 캐쉬 라인으로 하였을 경우에 대한 성능 평가이다. 성능평가 결과 lpCSB+-트리의 경우에는 하나의 캐쉬 라인 크기로 하였을 경우 가장 좋은 성능을 보이고 있다. 그 이유는 노드의 크기가 커질수록 프리페치해야하는 손자 노드그룹내의 노드의 개수가 지수적으로 증가하여 노드그룹 적재시간이 급격히 증가하기 때문이다.



(그림 6) 데이터수와 노드크기의 변화에 따른 lpCSB+-트리의 검색시간

5. 결론

본 논문에서 제안한 lpCSB+-트리는 기존에 제안된 캐쉬를 고려한 주기억장치 상주형 색인구조들이 공통적으로 트리의 각 레벨을 접근할때 캐쉬 실패가 발생하는 문제를 해결하였다. 제안하는 색인구조는 CSB+-트리의 구조에 기반하며 프리페칭 기법을 사용하여 현재 접근 중인 노드의 자식과 손자 노드들을 미리 캐쉬에 올려놓는 방법을 이용하였다. 그리고, CSB+-트리의 문제점인 분할 비용을 줄이기 위해 새로운 분할 알고리즘을 제안하였다. 시뮬레이션 결과 제안하는 방법이 기존의 방법에 비해 검색측면에서 20% 정도 성능이 향상됨을 볼 수 있었다. 향후연구에서는 실제 환경에서 다양한 실험을 통해 제안하는 색인구조의 성능을 분석한다.

6. 참고문헌

[1] Anastassia Ailamaki, David J. DeWitt, Mark D Hill and David A. Wood, "DBMSs on a Modern Processor: Where Does Time Go?", In Proceedings of VLDB Conference, pp. 266-277, 1999.

[2] Jun Rao and Kenneth A. Ross, "Cache Conscious Indexing for Decision-Support in Main Memory", In proceedings of VLDB conference, pp. 78-79, 1999.

[3] Jun Rao and Kenneth A. Ross, "Making B+-Trees Cache Conscious in Main Memory", In proceedings of ACM SIGMOD Conference, pp. 475-486, 2000.

[4] Philip Bohannon, Peter McIlroy and Rajeev Rastogi, "Main-Memory Index Structures with Fixed-Size Partial Keys", In proceedings of ACM SIGMOD Conference, pp. 163-174, 2001.

[5] Kihong Kim, Sang K. Cha and Keunjoo Kwon, "Optimizing Multidimensional Index Trees for Main Memory Access", Proceedings of ACM SIGMOD Conference, pp. 139-150, 2001.

[6] Shimin Chen, Phillip B. Gibbons and Todd C. Mowry, "Improving Index Performance through Prefetching", In proceedings of ACM SIGMOD Conference, 2001.]