PR-Tree: 메인 메모리에서 선반입을 적용한 확장된 R-tree 색인 기법†

강홍구°, 김동오, 홍동숙, 한기준
건국대학교 컴퓨터공학과
{hkkang°, dokim, dshong, kjhan}@db.konkuk.ac.kr

PR-Tree: An Extended R-Tree Indexing Method using Prefetching in Main Memory

Hong-Koo Kang°, Dong-O Kim, Dong-Sook Hong, Ki-Joon Han
Dept. of Computer Engineering, Konkuk University

요 약

최근 프로세서와 메인 메모리간의 속도 차이가 커지면서 캐시 실패가 메인 메모리에서 동작하는 R-Tree의 성능 저하에 미치는 영향이 커짐에 따라 캐시 실패를 줄여 캐시 성능을 개선하려는 연구가 많이 진행되고 있다. 일반적인 캐시 성능 개선 방법은 엔트리 정보를 줄여 노드에 더 많은 엔트리를 저장함으로써 펜-아웃(fanout)을 증가시키고 캐시 실패를 최소화한다. 그러나 이러한 방법은 엔트리 정보를 줄이는 추가 연산으로 인해 갱신 성능이 떨어지고, 노드간 이동시 발생하는 캐시 실패는 여전히 해결하지 못하고 있다.

본 논문은 이를 해결하기 위해 선반입(prefetching)을 적용한 확장된 R-Tree인 PR-tree(Prefetching R-Tree)를 제안하고 평가하였다. PR-Tree는 펜-아웃을 증가시키고 트리의 높이를 낮추기 위해 실제 캐시 라인의 정수 배인 노드를 생성하고, 선반입을 적용하여 노드 캐시로 인한 메모리 지연을 최소화하였다. 또한 접근할 노드를 선반입하여 노드간 이동시 발생하는 캐시 실패도 최소화하였다. PR-Tree는 실험에서 R-Tree보다 검색 연산에서 최대 38%의 성능 향상을 보였으며, 갱신 연산에서도 최대 30%의 성능 향상을 보였다.

1. 서 론

최근 프로세서 속도와 메인 메모리 속도의 차이가 커짐에 따라 캐시 메모리를 얼마나 효과적으로 사용하는가 하는 문제가 메인 메모리 인덱스의 성능에 결정적인 영향을 미치게 되었다[6]. 이에 캐시 성능을 개선한 인덱스 구조와 알고리즘에 관한 연구가 다양하게 진행되어 왔다[1,4,6,7,8].

Rao와 Ross는 메인 메모리 인덱스의 설계에 있어서 캐시 성능의 중요성을 제기하고 read-only OLAP 환경에서 이진 탐색 트리나 T-Tree보다 빠른 검색 성능을 보이는 CSS-Tree(Cache-Sensitive Search Tree)[6]와 B+-Tree에서 캐시 성능을 향상시키는 CSB+-Tree를 제안하였다[7]. Inga와 Peter는 노드 크기를 캐시 라인 크기에 맞추고, MBR의 불필요한 정보를 제거하여 캐시 라인에 보다 많은 정보를 저장하기 위한 pR-Tree(partial R-Tree)를 제안하였다[8]. 그리고, Kim과 Cha는 엔트리의 MBR을 압축하여 캐시에 보다 많은 엔트리를 포함하기 위한 CR-Tree를 제안하였다[4]. 이러한 방법들은 여러 측면에서 인덱스 구조에서 발생할 수 있는 캐시 실패를 줄였지만, 여전히 캐시 라인 크기보다 큰 노드를 캐시에 적재할 때나 노드간 이동시 캐시 실패가 발생하는 문제가 있다. 또한, 갱신시 엔트리 정보를 재구성하거나 압축하는 추가 비용으로 인해 성능이 떨어지는 단점이 있다.

본 논문은 이러한 문제점을 개선하기 위해 선반입 기법을 R-Tree에 적용하여 캐시 실패를 줄이고 갱신 연산에서 추가 비용이 발생하지 않는 PR-Tree (Prefetching R-Tree)를 제안하고 평가하였다. PR-Tree는 펜-아웃을 증가시키고 트리의 높이를 낮추기 위해 크기가 실제 캐시 라인의 정수 배인 노드를 사용한다. 그리고 여러 캐시 라인을 포함하는 노드를 캐시에 적재할 때 발생하는 메모리 지연을 최소화하기 위해 캐시 라인 단위로 캐시에 선반입한다. 또한 노드간 이동시에 발생하는 캐시 실패를 줄이기 위해 접근이 필요한 자식 노드를 미리 캐시에 적재한다.

PR-Tree는 선반입을 이용하여 노드 크기와 접근하는 노드 개수에 비례하여 성능이 향상된다. 따라서 노드 크기가 크거나, 접근하는 노드 개수가 많은 질의에 보다 효율적이다. 실험에서 PR-Tree는 R-Tree보다 검색 연산에서 최대 35%의 성능 향상을 보였으며, 갱신 연산에서는 최대 30%의 성능 향상을 보였다.

본 논문의 구성은 다음과 같다. 1장의 서론에 이어서, 2장의 관련 연구에서는 R-Tree, 기존의 캐시 성능 개선 인덱스, 선반입 기법을 설명한다. 3장에서는 본 논문에서 제시하는 PR-Tree를 기술하고, 4장에서는 PR-Tree의 실험 결과를 설명한다. 마지막으로, 5장에서 결론을 언급한다.

2. 관련 연구

이 장에서는 R-Tree를 설명하고 캐시 성능의 문제

† 본 연구는 한국과학재단 목적기초연구(과제번호: R01-2001-000-00540-0)지원으로 수행되었음.

점을 살펴본다. 그리고, 기존의 캐시 성능을 개선한 인덱스들을 분석한다. 마지막으로, 캐시 성능을 향상시키기 위한 선반입 기법에 대해 설명한다.

2.1 R-Tree

R-Tree는 B-Tree를 공간 인덱스에 맞게 변형한 것으로서, B-Tree와 마찬가지로 높이 균형 트리이고 객체에 대한 참조는 리프 노드에만 존재한다. 공간 객체를 표현하는데 최소 사각형(MBR: Minimum Bonding Rectangle)을 사용하는 R-Tree는 공간 객체를 찾기 위하여 적은 수의 노드만 방문하면 되도록 설계되었으며, 트리 구조의 동적인 생성을 지원하기 때문에 갱신을 검색과 혼합하여 진행할 수 있다[2].

R-Tree의 중간 노드 형태는 (*p, RECT*)이며, 이때 *p*는 자식 노드에 대한 포인터이고, *RECT*는 자식 노드에 있는 모든 최소 사각형을 둘러싸는 사각형이다. R-Tree의 리프 노드 형태는 (*oid, RECT*)이고 이 때, *oid*는 실제 공간 객체를 의미하며, *RECT*는 공간 객체에 대한 최소 사각형이다.

노드의 최대 엔트리 수가 M이고 최소 엔트리 수를 m(m<M/2)이라 할 때 R-Tree는 다음 특성을 가진다.

(a) 루트 노드(root node)는 리프 노드가 아니라면 적어도 2개의 자식을 가진다.

(b) 모든 중간 노드는 루트 노드가 아니라면 적어도 m에서 M개사이의 자식을 가진다.

(c) 모든 리프 노드는 루트 노드가 아니라면 적어도 m에서 M개사이의 자식을 가진다.

(d) 모든 리프 노드는 같은 높이에 있다.

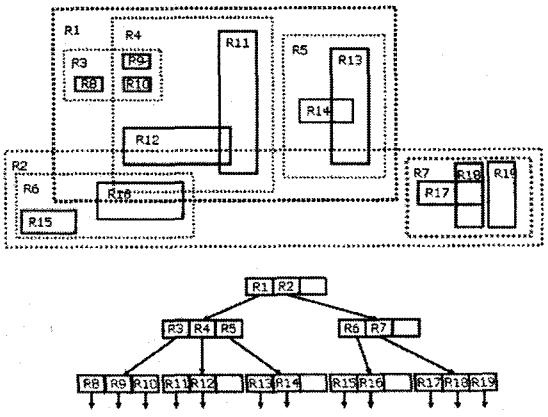

그림 1. R-Tree

그림 1은 R-Tree의 구조를 보여주고 있다. 그림 1에서 실선으로 표현된 사각형은 리프 노드의 엔트리에 의해 참조하는 실제 공간 객체의 MBR이고, 점선으로 표현된 사각형은 R-Tree 인덱스 노드의 MBR이다.

R-Tree는 디스크 기반 인덱스로 디스크 I/O를 효과적으로 줄이기 위해 설계되어서 블록이 작은 캐시 메모리에는 적합하지 못하다. 그림 2는 R-Tree에서 캐시 실패 비용을 나타낸다. 그림 2에서 보듯이 전체 수행 시간에서 캐시 실패로 인한 지연 시간이 매우 높다. 특히, 메인 메모리에 상주하는 R-Tree의 경우 디스크 I/O가 없기 때문에 캐시 실패를 효율적으로 줄이는 연구가 필요하다.

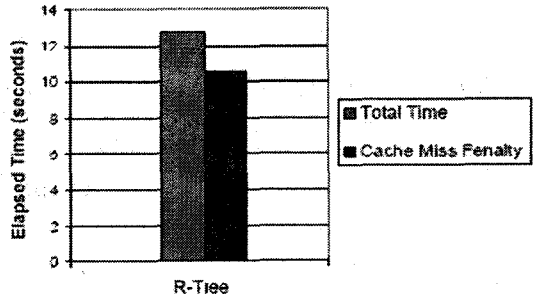

그림 2. R-Tree에서 캐시 실패 비용

2.2 기존의 캐시 성능 향상 인덱스

인덱스에서 캐시 성능 향상에 관한 연구는 다양한 방법으로 진행되어 왔다. 대표적인 인덱스에는 CSB+-Tree, pR-Tree, CR-Tree가 있다.

CSB+-Tree는 첫번째 자식 노드의 포인터를 제외하고 나머지 자식 노드의 포인터를 제거하여 자식 노드를 메모리에 연속적으로 저장하는 B+-Tree의 변형이다[7]. 자식 노드의 위치는 첫번째 자식 노드로부터 메모리 offset을 계산하여 알 수 있다. 연속적으로 저장된 노드의 크기는 캐시 라인 크기로 맞추고, 펜-아웃을 두 배로 늘려 트리의 높이를 낮추었다. 따라서 B+-Tree보다 트리 탐색에서 캐시 실패가 줄어들게 된다. 그러나 갱신시 연속적인 저장 구조를 유지하기 위한 추가 연산이 필요하다.

pR-Tree는 부모 MBR의 좌표값과 겹침이 발생하는 자식 MBR의 좌표값을 제거하여 캐시 라인에 보다 많은 정보를 저장하기 위한 R-Tree의 변형이다[8]. 캐시 성능을 위해 CSB+-Tree와 같이 엔트리에서 포인터를 제거하고, 노드 크기를 캐시 라인과 같게 하였다. 그러나 엔트리의 수가 적을 때는 좋은 성능을 보이지만 엔트리 수가 증가함에 따라 부모 MBR의 좌표값과 겹치는 좌표의 비율이 크게 줄어들어 성능이 나빠지고, 갱신시 연속적인 저장 구조를 유지하기 위한 추가 연산이 필요하다.

CR-Tree(Cache-conscious R-Tree)는 압축한 MBR을 키로 사용하는 R-Tree이다[4]. 자식 노드의 MBR을 부모 노드 MBR에 대한 상대적 좌표로 표현하고(RMBR), 상대 좌표가 일정한 비트로 표현될 수 있도록 양자화한다(QRMBR). 그러나 MBR이 QRMBR이 되면 원래의 MBR과 약간의 오차가 발생할 수 있고, 이러한 오차에 의해서 추가적인 결과(false hit)가 발생할 수 있다. 또한, 갱신시 MBR의 압축과 복원에 대한 추가 연산이 필요하다.

2.3 선반입 기법

캐시 실패로 인한 메모리 지연은 데이타를 캐시에

선반입함으로써 줄일 수 있다. 본 논문에서 이용한 선반입 기법은 프로세서가 자체적으로 제공하는 하드웨어적 방법과 프로그램 소스에 선반입 명령어를 삽입하는 소프트웨어적 방법을 모두 사용하였다.

하드웨어적 방법은 프로세서가 자동으로 데이타를 선반입하는 것으로, 선반입할 페이지를 예측하기 위해 지역성(locality)과 순차성(sequencing)과 같은 데이타 특성을 사용한다. 지역성은 크게 공간적 지역성과 시간적 지역성으로 나눌 수 있는데, 공간적 지역성이란 최근에 참조된 캐시 블록을 기준으로 인접된 페이지들이 가까운 미래에도 참조될 확률이 높다는 것을 나타내는 특성이고, 시간적 지역성은 최근에 참조된 캐시 블록의 페이지는 가까운 미래에 참조될 확률이 높다는 것을 나타내는 특성이다. 순차성은 공간적 지역성과 관련이 있는 것으로, 참조된 페이지의 바로 다음 주소에 위치한 페이지가 참조될 확률이 높음을 나타내는 특성이다[9].

소프트웨어적 방법은 프로그램 소스에 선반입 명령을 추가하는 것으로, 이를 위해 현대의 마이크로 프로세서는 선반입을 수행하기 위해 몇 가지 fetch 명령을 지원한다. Fetch 명령의 수행은 간단히 프로세서 레지스터에 명령을 올리는 것과 같이 간단하고, 속도도 매우 빠르다. Fetch 명령은 non-blocking 메모리 연산이기 때문에 캐시에서 동작하는 다른 메모리 연산들에 우선하여 수행된다. 그리고, 일반적으로 선반입은 수행 도중 예외 상황을 발생시키지 않기 때문에 프로그램의 동작에 불필요한 영향을 주지 않는다. 그러므로, 소프트웨어적 방법에서 가장 중요한 문제는 선반입 스케줄링(prefetch scheduling)이다. 즉, 프로그램 코드 사이에 fetch 명령을 적당한 위치에 배치하는 것이다.

3. PR-Tree

본 논문에서 제안하는 PR-Tree는 노드 크기를 캐시 라인 크기의 정수 배로 맞추고, 노드가 가지는 캐시 라인의 실패로 인한 메모리 지연 시간을 줄이기 위해 노드를 캐시 라인 단위로 선반입한다. 이러한 방법을 통해 노드가 캐시에 적재될 때 발생하는 캐시 라인 실패를 줄일 수 있다. 그리고, 노드 크기를 크게 함에 따라 트리의 높이가 낮아지고 캐시 실패가 줄어들어 메모리 지연 시간을 줄일 수 있다. 이처럼 PR-Tree는 선반입 기법을 이용해 검색, 삽입, 삭제 연산에서 발생하는 캐시 실패를 줄여 노드 접근 시간을 줄이는 공간 인덱스 구조이다.

3.1 노드 구조

PR-Tree의 노드는 R-Tree의 노드 구조와 비슷하지만 캐시 라인 크기의 정수 배에 맞도록 엔트리의 개수를 제한하는 것이 R-Tree와 다르다. 이렇게 함으로써 노드를 캐시 라인 단위로 나누어 선반입할 때 공간 효율성을 얻을 수 있다. 예를 들어, 캐시 라인 크기가 32byte일 경우에는 노드의 크기는 64+160n 바이트 (n>0)에 맞추어 64, 224, 384, 544 … 64+160n 바이트가 된다. 반면에 캐시 라인 크기가 64바이트인 경우는 노드 크기가 64+320n 바이트 (n>0)에 맞추어 64, 384, 704 … 64+160n 바이트가 된다. 그림 3은 32바이트 캐시 라인 2개에 맞춰진 PR-Tree 노드를 보여준다. 여기서 노드가 가지는 엔트리는 3개가 된다.

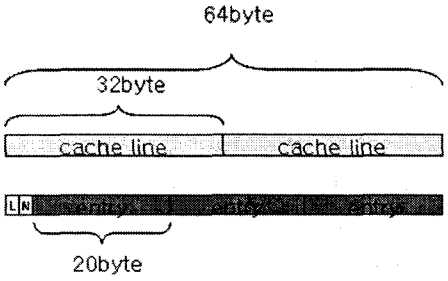

그림 3. PR-Tree의 노드 구조

3.2 알고리즘

이 절에서는 PR-Tree의 검색, 삽입, 삭제 알고리즘을 설명한다.

3.2.1 검색 알고리즘

PR-Tree의 검색 연산은 루트 노드에서 시작하여 질의에 포함되는 하위 레벨의 노드로 이동하는 과정을 반복하여 리프 노드의 객체를 찾아내는 동작을 말한다. PR-Tree의 검색 알고리즘에서는 두 가지 선반입 알고리즘을 적용한다.

첫번째 알고리즘(그림 4)은 노드 크기가 작은 경우로 현재 참조되는 노드의 모든 자식 노드를 선반입한다. 자식 노드를 모두 선반입하면 겹치지 않는 자식 노드들도 선반입되지만 노드가 작기 때문에 선반입 연산과 질의 영역과 겹치는 자식 노드의 포인터를 얻는 연산의 병렬 구간이 짧다. 그러므로 일반 R-Tree 연산과 비교하여 큰 성능 향상을 얻을 수 없다. 따라서, 노드 크기가 작은 경우에는 전체 자식 노드를 선반입하는 것이 메모리 지연을 보다 효율적으로 줄일 수 있다.

```
begin
  Results = ∅
if (R is not a leaf node) then // Search internal node
  prefetch (all entry's child);
  if (entry's child pointer is not NULL) then
   for each entry (p, RECT) of R do
     if (RECT overlaps W) then
           CHILD = node pointed to by p ;
           SEARCH(CHILD, W∩RECT);
     end if
   end for
  end if
else // Search leaf node
  if(entry's child pointer is not NULL) then
   for each entry (p, RECT) of R do
    if RECT overlaps W then
     OBJECT = spatial object pointed to by p
     Results += OBJECT
   end if
  end for
 end if
return Results
end
```

그림 4. 검색 알고리즘 1

두번째 알고리즘(그림 5)은 노드 크기가 큰 경우로 현재 참조되는 노드의 연산으로 자식 노드의 포인터를 얻었을 경우에 해당 자식 노드를 선반입하는 방법이다. 노드의 크기가 큰 경우에는 선반입 연산과 질의 영역과 겹치는 자식 노드의 포인터를 얻는 연산의 병렬 구간이 길다. 따라서 노드 크기가 큰 경우에는 필요한 자식 노드만 선반입하는 것이 메모리 지연을 보다 효율적으로 줄일 수 있다.

```
begin
  Results = Ø
if (R is not a leaf node) then // Search internal node
  if (entry's child pointer is not NULL) then
      prefetch (entry's child);
    for each entry (p, RECT) of R do
      if (RECT overlaps W) then
            CHILD = node pointed to by p
            SEARCH (CHILD, W∩RECT)
      end if
    end for
 end if
else // Search leaf node
 if (entry's child pointer is not NULL) then
   for each entry (p, RECT) of R do
     if (RECT overlaps W) then
        OBJECT = spatial object pointed to by p
        Results += OBJECT
     end if
   end for
end if
return Results
end
```

그림 5. 검색 알고리즘 2

3.2.2 삽입 알고리즘

PR-Tree에 공간 데이타를 삽입하는 과정의 처음은 검색 과정과 마찬가지로 트리의 루트 노드부터 시작하여 각 노드의 엔트리에 포함된 모든 하위 노드 MBR 정보를 계산하여 객체를 삽입할 때 크기 확장이 최소로 되는지를 판단한다. 확장이 최소가 되는 하위 노드가 발견되면 그 하위 노드를 루트로 하는 서브 트리를 계속해서 찾아 내려가는데 이때 선반입 명령을 적용한다. 그리하여 리프 노드에 도달했을 때에 객체 삽입이 이루어진다. PR-Tree의 삽입 알고리즘은 그림 6과 같다.

```
begin
 if (R is not a leaf node) then // Search internal node
   for each entry (p, RECT) of R do
     prefetch (entry's child );
     if (RECT overlaps IR) then
       CHILD = node pointed to by p
       INSERT (CHILD, IR)
     end if
   end for
 else // Insert into leaf node
   ADD (IR, R) // Insert IR into R
   if (R overflow) then
     SPLIT (R)
   end if
 end if
end
```

그림 6. 삽입 알고리즘

3.2.3 삭제 알고리즘

PR-Tree는 R-Tree와 마찬가지로 공간 데이타를 삭제하기 위하여 객체에 대한 참조가 저장되어 있는 리프 노드를 모두 찾아낸 후 그 리프 노드들의 엔트리에서 객체에 대한 참조를 삭제한다. PR-Tree은 객체와 교차가 발생하는 모든 리프 노드에 객체에 대한 참조를 유지하기 때문에 공간 데이타 삭제 시 여러 개의 리프 노드들에서 삭제 작업이 필요하게 된다. PR-Tree의 삭제 알고리즘에서도 검색 알고리즘과 같이 노드 크기에 따라 두 가지 선반입 알고리즘을 적용한다. 그림 7은 현재 노드의 연산 전에 전체 자식 노드를 선반입하는 알고리즘이고, 그림 8은 현재 노드의 연산으로 얻은 자식 노드를 선반입하는 알고리즘을 나타낸다.

```
begin
 if (R is not a leaf node) then // Search internal node
  prefetch (all entry's child );
  for each entry (p, RECT) of R do
    if (RECT overlaps IR) then
      CHILD = node pointed to by p
      DEL (CHILD, IR)
    end if
  end for
else // Remove from leafe node
  DEL (IR, R)
end if
end
```

그림 7. 삭제 알고리즘 1

```
begin
 if (R is not a leaf node) then // Search internal node
  for each entry (p, RECT) of R do
    prefetch (entry's child );
    if (RECT overlaps IR) then
      CHILD = node pointed to by p
      DEL (CHILD, IR)
    end if
  end for
else // Remove from leafe node
  DEL (IR, R)
end if
end
```

그림 8. 삭제 알고리즘 2

3.3 노드 크기 설정

PR-Tree의 캐시 성능을 최적화하기 위해 노드 크기가 캐시 라인에 적합해야 한다. 이런 노드 크기에 영향을 미치는 요소는 두 가지가 있다. 첫째는 노드를 캐시에 적재할 때에 캐시 라인 실패 시간(T_1)이고, 둘째는 파이프라인된 캐시 라인 실패 시간(T_{next})이다. 캐시 라인 크기의 정수배가 되는 노드는 처음 캐시 라인을 적재할 때 캐시 라인 실패가 발생하나 다음 캐시 라인부터는 파이프라인에 적재되는 시간 지연만 발생한다. 따라서 검색에서 걸리는 전체 메모리 지연시간은 접근하는 노드 개수와 노드당 접근 시간의 곱이 된다.

검색에서 접근하는 노드의 개수는 다음과 같다. 질의 사각형에 객체가 포함되는 확률은 전체 데이타 영역에서 질의 사각형이 차지하는 비율이다[3]. 예를 들어, 노드 N과 (Wx, Wy)의 크기를 가진 사각형 W을 가지고 있다고 가정하면, W의 오른쪽 상단 모서리로 점 질의한 확률로서 N.mbb가 W를 교차하는 확률을 평가할 수 있다. 실제로 N.mbb와 W의 교차는 그림 9처럼 두 가지 경우가 발생할 수 있다.

그림 9의 왼쪽은 질의 사각형(W1)의 오른쪽 상단 모서리가 N.mbb에 포함되는 것을 나타내고, 오른쪽은 질의 사각형(W2)의 오른쪽 상단 모서리가 N.mbb의 오른쪽으로 Wx 만큼 왼쪽으로 Wy 만큼 팽창된 부분에 포함되는 것을 보여준다.

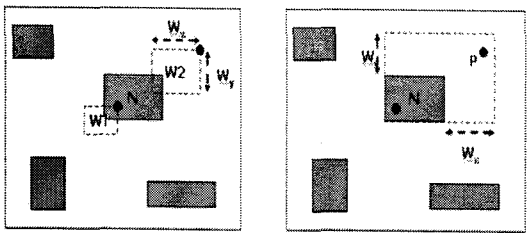

그림 9. 질의 사각형의 구성

만약 오른쪽 상단 모서리가 사각형 W=(N.mbb.xmin + Wx, N.mbb.ymin + Wy) 안에 들어가고 W가 N.mbb를 교차한다면, 공간 객체들이 균일하게 검색 공간에 위치하고 있기 때문에 비용은 W에 관한 점 질의처럼 평가될 수 있다. 정리하면 범위 질의 동안 탐색되는 노드의 수 WQ(w)는 아래 식 1과 같이 계산될 수 있다.

$$WQ(w) = \sum_{i=1}^{n} (Si,x + Wx) \times (Si,y + Wy) \quad \cdots \text{(식 1)}$$

식 1은 R-Tree, R*-Tree의 특정 타입 그리고 정적, 동적 구성모드에 관계없이 R-Tree의 특성에 의존한다. 이때 전체 트리에서 검색으로 인한 평균 메모리 지연 시간(T_{total_miss})은 아래 식 2와 같다.

$$\begin{aligned} T_{total-miss} &= WQ(w) \times (T_1 + (d-1) \times T_{next}) \\ &= T_{next} \times WQ(w)(\frac{T_1}{T_{next}} + (d-1)) \end{aligned} \quad \cdots \text{(식 2)}$$

T_1과 T_{next}는 하드웨어의 성능에 따르므로 전체 메모리 지연 시간은 WQ(w)와 d에 의해 결정된다. 노드당 줄이는 메모리 지연시간은 T1이고 트리 전체로는 WQ(w)*T1이 된다. 따라서 노드 크기는 d, WQ(w)의 조합에 대해 캐시 실패 비용을 계산하여 최적의 크기를 선택한다.

4. 실험 결과

본 논문에서는 LINUX 환경 하에서 C 언어로 R-Tree, PR-Tree를 각각 구현하였다. 실험에 사용된 컴퓨터 시스템은 Intel Pentium3 1GHz이고 L1, L2 캐시의 라인 크기는 32바이트이며, 메인 메모리의 용량은 1GB이다.

검색 성능 실험에 사용될 공간 객체는 0에서 1사이의 float형으로 전체 영역에 균등하게 분포하도록 10,000개의 객체를 임의로 생성하고, 크기는 한 변의 평균을 0.0001로 하였다.

검색에 사용할 질의 영역은 전체 데이타 영역의 70%이상을 포함하는 것으로 하였고, 임의로 10,000번의 범위 질의를 수행하여 R-Tree와 성능을 비교하였다. 실험 결과는 그림 10과 같이 대체로 노드 크기가 커질수록 검색 성능이 좋아지고, 선반입을 이용한 성능 향상이 일정한 것을 볼 수 있다. 이는 노드 접근에서 발생하는 메모리 지연을 줄였기 때문이다. PR-Tree의 검색 속도는 그림에서 보듯이 최대 35%가 향상되었다.

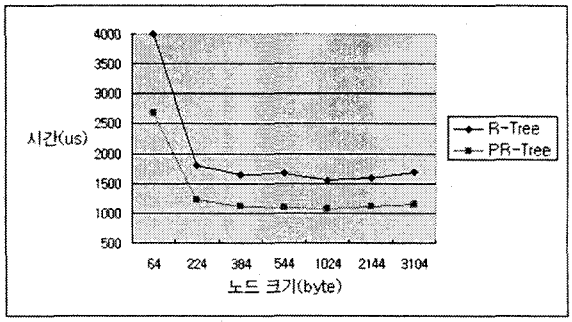

그림 10. 검색 연산시 성능 비교

R-Tree는 노드간 겹침을 허용하기 때문에 객체가 불규칙적으로 분포할 경우 다중 검색 경로가 발생하여 검색 성능이 떨어진다. 그러나 PR-Tree는 다중 검색 경로가 발생할 경우 빈번한 노드간 이동에서 발생하는 메모리 지연을 줄이므로 성능을 보다 많이 향상시킬 수 있다.

공간 객체의 불균형 분포에 대한 검색 연산 실험에서 사용될 공간 객체는 0에서 1사이의 float형으로 전체 영역에 불균등하게 분포하도록 10,000개의 객체를 임의로 생성하고, 객체의 크기는 한 변의 평균을 0.0001로 하였다. 검색에 사용할 질의 영역은 전체 데이타 영역의 30%를 포함하는 것으로 하였고, 임의로 10,000번의 검색 질의를 수행하여 R-Tree와 성능을 비교하였다.

실험 결과 그림 11과 같이 노드 크기가 클수록 성능이 향상되는 것을 볼 수 있고, R-Tree보다 최대 38%의 성능 향상을 보였다.

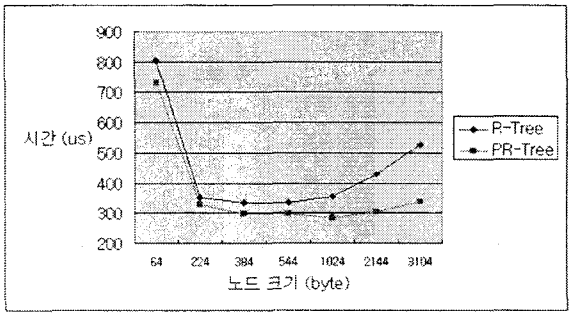

그림 11. 불균등 분포에서 검색 연산시 성능 비교

삽입 연산의 실험 설정은 우선 균등 분포된 공간 객체 10,000개를 갖는 트리를 생성하고, 삽입할 객체의 크기는 한 변의 평균을 0.001로 하였다. 삽입 객체를 임의로 10,000번 삽입하여 그 평균을 구하여 R-Tree와 성능을 비교하였다.

결과는 그림 12과 같이 대체적으로 노드 크기가 클수록 성능이 나빠지고, 선반입을 이용한 성능 향상은 다소 좋아지는 것을 볼 수 있다. 이는 접근하는 노드의 수가 트리 높이와 같기 때문이다. PR-Tree의 삽입 속도는 그림에서 보듯이 최대 11%가 향상되었다.

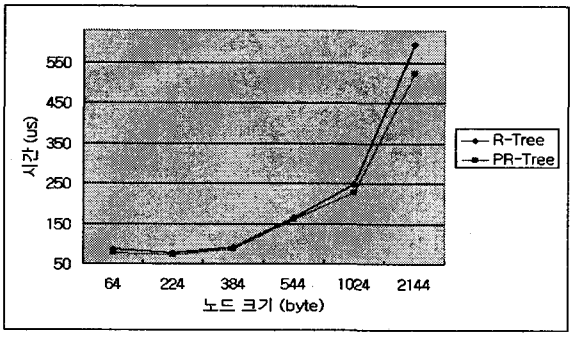

그림 12. 삽입 연산시 성능 비교

삭제 성능의 실험 설정은 우선 균등 분포된 공간 객체 10,000개를 갖는 생성하고, 삭제할 객체는 한변의 길이가 평균 0.001이 되도록 하였다. 삭제 객체를 임의로 10,000번 삭제하여 그 평균을 구하여 R-Tree와 성능을 비교하였다.

결과는 그림 13과 같이 대체적으로 노드 크기가 클수록 성능이 좋아지고, 선반입을 이용한 성능 향상이 일정한 것을 볼 수 있다. 이는 노드 접근에서 발생하는 메모리 지연을 줄였기 때문이다. PR-Tree의 삭제 속도는 그림에서 보듯이 최대 30%가 향상되었다.

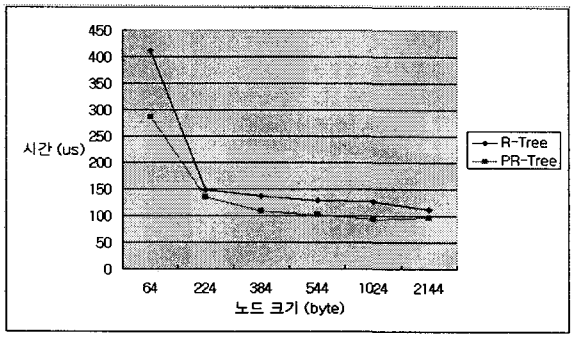

그림 13. 삭제 연산시 성능 비교

5. 결론

프로세서 속도와 메인 메모리 속도의 차이가 커짐으로써 메모리에 상주하는 인덱스에서 캐시 실패는 성능에 상당한 영향을 미친다. 이를 해결하기 위해 많은 연구가 있었으나, 기존에 제안된 캐시 성능 개선 인덱스는 노드간 이동시 여전히 캐시 실패가 발생하고 노드 구조의 재구성이나 엔트리의 압축, 포인터 제거로 인한 추가 연산으로 갱신 성능이 나빠지는 단점이 있다. 본 연구에서는 이를 해결하기 위해 선반입 기법을 이용하여 노드간 이동시 발생하는 캐시 실패를 줄이고 트리의 펜-아웃을 늘려 트리의 높이를 낮춤으로써 R-Tree의 검색 및 갱신 성능을 향상시키는 PR-Tree를 제안하였다.

PR-Tree는 실험을 통해 R-Tree보다 검색 연산에서 최대 38%의 성능 향상을 보였고, 갱신 연산에서 최고 30%의 성능 향상을 보였다. 또한, 데이타 객체의 불균형 분포로 인한 밀집 지역에 대한 범위 질의에서도 좋은 성능을 보였다. 향후 과제로는 실제 메인 메모리에서 동작하는 응용 시스템에 적용해보고 시스템의 특성에 따른 선반입 스케줄링 개선이 있다.

참고 문헌

[1] Chen, S., Gibbons, P., Mowry, T., and Valentin, G., "Fractal Prefetching B+-Trees : Optimizing Both Cache and Disk Performances," Proceedings of ACM SIGMOD Conference, 2002, pp.157-168.

[2] Guttman, A., "R-Trees: a Dynamic Index Structure for Spatial Searching," Proceedings of ACM SIGMOD Conference, 1984, pp.47-54.

[3] Kamel, I., and Faloutsos, C., "On Packing R-Trees," Proceedings of ACM CIKM Conference, 1993, pp.490-499.

[4] Kim, K., Cha, S., and Kwon, K., "Optimizing Multidimensional Index Tree for Main Memory Access," Proceedings of ACM SIGMOD Conference, 2001, pp.139-150.

[5] Mowry, T., Lan, S., and Gupta, A., "Design and Evaluation of a Compiler Algorithm for Prefetching," Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems, 1992, pp.62-73.

[6] Rao, J., and Ross, K. A., "Cache Conscious Indexing for Decision-Support in Main Memory," Proceedings of VLDB Conference, 1999, pp.78-89.

[7] Rao, J., and Ross, K. A., "Making B+-Trees Cache Conscious in Main Memory," Proceedings of ACM SIGMOD Conference, 2000, pp.475-486.

[8] Sitzmann, P., and Stuckey, "Compacting Discriminator Information for Spatial Trees," Proceedings of Australasian Database Conference, 2002, pp.167-176.

[9] VanderWiel, S., and Lilja, D., "Data Prefetch Mechanisms," ACM Computing Surveys, Vol. 32, 2000, pp.174-199.