스트리밍 데이터의 선인출에 사용되는 참조예측표 교체 전략

임철후*, 김석일*, 전중남*
* 충북대학교 전자계산학과
e-mail: pedal7@hananet.net , {joongnam, ksi}@cbu.ac.kr

A Replacement Strategy for Reference Prediction Table Used in Prefetching Streaming Data

Chulhoo Lim*, Sukil Kim*, Joongnam Jeon*
*Dept of Computer Science, Chung-buk University

요 약

멀티미디어 응용프로그램은 처리데이터를 참조할 때 대부분 간격이 일정한 스트리밍 패턴으로 참조한다. 이 특성을 선인출 방법에 적용하여 멀티미디어 응용프로그램의 성능을 향상 시킬 수 있다. 이 논문에서는 하드웨어기반의 규칙 선인출 방법에서 참조예측표를 운영하는 방법을 제안한다. 크기가 제한되어 있는 참조예측표에 메모리 참조 명령어를 추가할 때 주소간격이 0인 행을 우선적으로 제거함으로써 비용절감의 효과를 가져 올 수 있다. 실험 결과 제안한 방법과 기존의 참조예측표를 FIFO 방식으로 운영하는 방법을 비교할 때 제안한 방법의 경우 참조예측표의 크기를 반으로 줄여도 거의 같은 효과를 볼 수 있었다.

1. 서론

CPU의 처리 속도는 지난 몇 년 동안 급속도로 발전해 왔으나 메모리 속도는 기술적인 약세로 인하여 CPU와 속도차이가 지난 수년간 지속적인 증가 추세를 나타내고 있다. 연구결과에 따르면 멀티미디어 응용프로그램의 전체 수행시간 중 25~50%가 메모리 참조에 소요되는 것으로 나타났다[1]. 멀티미디어 응용프로그램은 스트리밍 패턴으로 데이터를 참조하는 특성이 있다. 한편, 이러한 데이터의 자료 구조는 배열 형태가 일반적이며 배열에 대한 처리는 프로그램의 반복 구조로 표현되므로 스트리밍 데이터 참조는 규칙성이 존재한다[3,4]. 이 규칙성을 활용하여 캐시의 성능향상을 꾀하는 기법으로 선인출[2]에 관한 연구가 수행되었다.

선인출 방법에는 크게 하드웨어 기반의 선인출과 소프트웨어에 의한 선인출로 구분된다. 소프트웨어에 의한 선인출 방법은 정적 선인출이라고도 하며 프로그램이 컴파일되면서 컴파일러에 의해 규칙적인 데이터 엑세스 패턴을 감지하여 그 부분에 선인출코드를 삽입하여 실행코드를 만드는 방법이다. 하드웨어 기반의 선인출 방법은 프로그램의 수행 시 선인출 하드웨어에 의해 동적으로 선인출된다.

여러 하드웨어 선인출 방법 중 참조예측표(Reference Prediction Table)를 사용하는 규칙선인출의 경우 일정한 주소간격을 가진 메모리를 참조할 때 좋은 성능을 보인다[5]. 그러나 메모리참조명령어의 히스토리를 기록하여 선인출 명령을 발생시키는데 사용되는 참조예측표는 하드웨어로 구성되므로 비용과 직접적인 관계가 있다. 기존의 참조예측표의 운영은 FIFO로 운영하고 있다. 그러나 반복문 안에서 주소간격이 0인 메모리참조명령어는 스트리밍 데이터가 아니기 때문에 선인출하지 않으며, 따라서 참조예측표에 저장할 필요가 없다. 이것을 가능하게 하기 위해 참조예측표가 가득 찼을 때 새로운 메모리 참조 명령이 들어오면 우선적으로 주소간격이 0인 행을 제거함으로서 같은 캐시 미스 율을 유지하면서 참조예측표의 크기를 감소시킬 수 있을 것으로 기대된다.

2 규칙 참조 예측

규칙참조 예측은 메모리를 참조하는 명령어의 이전 참조 주소에 기반 하여 다음에 참조할 메모리 주소를 예측한다. 이러한 예측은 반복문에서 일어나는데 i번째 반복이 수행될 때 다음 반복, 즉, (i+1)번째 반복에서 사용될 메모리 주소를 예측하여 선인출 한다. 프로그램 카운터가 메모리참조명령어를 디코드 할 때, 참조예측표에 이 명령어와 일치하는 행이 있는지 검사하여 없으면 이 명령어를 참조예측표에 추가하고, 만약 일치하는 행이 있고 다음 반복에 사용될 것이 예측 가능하다면 선인출 명령을 발생시킨다.

본 연구는 한국과학재단 목적기초연구(R05-2002-000-01470-0) 지원으로 수행되었음.

2.1 참조예측표(Reference Prediction Table)

참조예측표는 메모리 참조명령어들에 대한 주소간격의 상호관계와 이전에 참조된 주소의 정보를 유지하여야 한다. 참조예측표의 행은 명령어주소에 의해 인덱스화 되며, 데이터를 액세스하는 유효주소와 주소간격뿐만 아니라 메모리참조명령어가 참조하는 주소의 규칙상태를 갖고 있어야 한다.

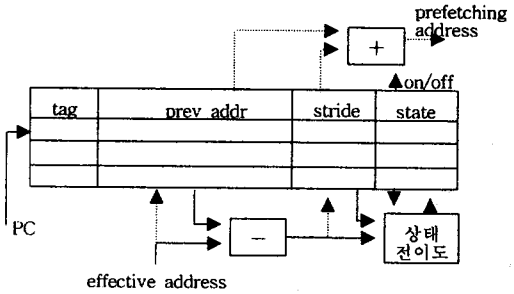

그림 1. 참조예측표의 구조

그림 1은 참조예측표 구조를 나타낸 것이다. 참조예측표 구조의 tag 필드는 메모리 참조 명령의 주소를 나타내고, prev_addr 필드는 tag를 갖는 메모리 참조 명령이 가장 마지막에 참조한 메모리 주소를 나타낸다. stride 필드는 마지막에 참조된 두 주소의 간격을 나타낸다. state 필드는 지난 메모리 참조 패턴에 따라 그림 2의 상태 전이도에서 보여지는 4개의 상태(2-bit)를 나타내며 이 상태에 의해 선인출의 유무를 결정한다.

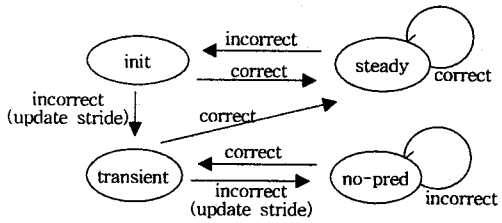

그림 2. 참조예측표의 상태 전이도

2.2 참조예측표의 운영과 문제점

기존의 참조예측표 방법은 새로운 메모리 참조가 발생하면 참조예측표 안에서 가장 오래된 행을 FIFO 방식으로 제거한다. 그림 3과 4를 이용하여 기존의 참조예측표 운영 방법을 제시하고자 한다.

그림 3은 주어진 배열의 모든 원소를 더하여 배열 B[1,2,3]에 넣는 간단한 프로그램이다. 이 프로그램에서 참조예측표에 들어올 수 있는 명령라인은 메모리참조명령어인 100, 104, 108, 112이다. 배열 A, B의 처음 주소는 각각 10000과 20000이고, 참조예측표는 크기가 3이고 초기에는 비어있으며 배열 A와 B의 어떤 원소도 캐시에 존재하지 않는다고 가정한다.

```
int A[100], B[10]
for i = 1 to 100
        B[1] += A[i];
        B[2] += A[i];
        B[3] += A[i];
end for
```

(a) 예제 코드

addr	instruction	comment	
100	lw r4, M[r2]	load A[i]	stride 4
104	lw r5, M[20000]	load B[1]	stride 0
108	lw r6, M[20004]	load B[2]	stride 0
112	lw r7, M[20008]	load B[3]	stride 0
116	addu r5, r5, r4	k += A[i]	
120	addu r6, r6, r4		
124	addu r7, r7, r4		
128	addu r2, r2, 1	increase i	
132	bne r2, r13, 100	; loop	

(b) 어셈블리 코드

그림 3. 예제 프로그램

tag	prev_addr	stride	state
100	10000	0	init

(a) 첫 번째 명령 실행 직후

tag	prev_addr	stride	state
100	10000	0	init
104	20000	0	init

(b) 두 번째 명령 실행 직후

tag	prev_addr	stride	state
100	10000	0	init
104	20000	0	init
108	20004	0	init

(c) 세 번째 명령 실행 직후

tag	prev_addr	stride	state
104	20000	0	init
108	20004	0	init
112	20008	0	init

(d) 네 번째 명령 실행 직후

그림 4. 크기가 3인 기존 방법의 참조예측표의 운영 예

전통적인 참조예측표의 운영방식에 따르면 프로그램 카운터에 의해 100, 104, 108번지의 메모리참조명령어가 수행될 때 참조예측표는 그림 4의 (a), (b), (c)와 같이 입력된다. 그러나 112번지의 명령어가 실행될 때 참조예측표가 가득 찼으므로 처음에 입력된 100번지 명령행이 참조예측표에서 빠져나가고 112번지 명령어가 들어오게 되며 참조예측표는 그림 4의 (d)와 같게 된다. 이 반복에서 모든 메모리 참조 명령이 참조할 메모리가 캐시에 없기 때문에 총 4개의 미스가 나게 된다. 두 번째 반복이 시작되면서 100번지 명령어가 참조예측표에 있는지 확인하고, 없으므로 가장 먼저 입력된 104번지 명령행이 참조예측표에서 빠져나가고 100번지 명령어가 들어오게 된다. 마찬가지로 104, 108, 112번지 명령어가 수행되면서 참조예측표에 없으므로 100번지 명령어와 같은 동작을 하게되고 매 반복이 끝날 때 참조예측표는 그림 4의 (d)와 같게 된다. 104, 108, 112가 참조하는 메모리는 이미 캐시에 있으므로 미스가 나지 않지만 100번지가 참조하는 메모리는 캐시에 없으므로 미스가 나게 된다. 이렇게 100번지는 주소간격이 일정함에도 불구하고 계속해서 참조예측표에 존재하지 않게 되므로 이 프로그램에서 100번의 반복이 끝날 때까지 100번의 미스를 일으키게 된다. 그러므로 이렇게 참조예측표 크기가 반복문 내에 있는 메모리참조명령어의 수보다 작을 때, 주소간격이 0인 매모리참조명령어로 인해 일정한 간격을 가진 스트리밍 데이터의 연속된 미스 발생 문제에 대한 새로운 참조예측표의 운영방식이 제안되어야 한다.

3. 제안하는 참조예측표 교체 알고리즘

참조예측표의 크기는 이상적으로 반복문 안에 포함되어 있는 메모리 참조 명령어의 수와 같거나 커야 한다. 그러나 이 수는 응용프로그램의 종류마다 다르다. 2.2절에서 설명한 참조예측표의 운영방식의 문제점은 주소간격이 0인 스칼라 데이터를 계속 유지하는 것과 참조예측표가 가득 찼을 때 새로운 메모리참조 명령어가 들어오면 가장

먼저 들어온 행과 교체하는 것이다.

이 문제점을 해결하기 위해 새로운 메모리 참조 명령이 들어오면 상태가 steady 이면서 주소간격이 0인 것을 가장 우선적으로 교체 대상으로 삼고 그 다음으로 가장 최근에 입력된 명령행을 교체 대상으로 삼는다.

그림 5의 알고리즘은 이러한 문제를 해결하는 교체정책으로서 IsExist() 함수는 참조예측표에 인수로 들어온 것이 존재하는지 검사하는 함수이고, Replace() 함수는 인수로 들어온 것과 입력된 메모리 참조명령과 교체하는 함수이다.

```
if (IsExist(입력된 메모리참조명령)) then
  참조예측표 갱신
else
  if (IsFull(참조예측표)) then
    if (IsExist((상태 = steady) & (주소간격 = 0))) then  ...(1)
      Replace(찾아진 메모리 참조 명령)
    else                                                 ...(2)
      Replace(가장 최근에 들어온 메모리 참조 명령)
  else
    참조예측표의 새로운 행을 할당받아 입력
```

그림 5. 제안하는 참조예측표의 운영 알고리즘

tag	prev_addr	stride	state
100	10000	0	init
104	20000	0	init
112	20008	0	init

(a) 첫 번째 반복 직후

tag	prev_addr	stride	state
100	10004	4	transient
108	20004	0	init
112	20008	0	steady

(b) 두 번째 반복 직후

tag	prev_addr	stride	state
100	10008	4	steady
112	20008	0	init
104	20000	0	init

(c) 세 번째 반복 직후

tag	prev_addr	stride	state
100	10008	4	steady
112	20008	0	steady
108	20004	0	init

(d) 네 번째 반복 직후

그림 6. 크기가 3인 제안된 참조예측표의 운영 예

그림 3의 프로그램을 크기가 3인 참조예측표의 제안된 운영방식에 따라 실행했을 때, 처음 반복에서 100, 104, 108번지의 메모리 참조 명령이 참조예측표에 입력되는 것은 기존의 방법과 같다. 그러나 112번지 메모리 참조 명령이 실행될 때 참조예측표가 가득 찼으므로 그림 5의 알고리즘에서 (2)에 의해 가장 최근에 입력된 명령행인 108번지 명령과 교체된다. 첫 번째 반복이 끝났을 때 참조예측표는 그림 6의 (a)와 같아지며 총 네 개의 미스가 일어난다. 두 번째 반복이 시작됐을 때 100, 104번지는 참조예측표에 있으므로 각 필드를 갱신한다. 108번지가 들어왔을 때 참조예측표가 가득 찼으므로 그림 5의 알고리즘에서 (1)에 의해 상태가 steady이고, 주소간격이 0인 104번지를 빼고 들어간다. 112번지는 이미 있으므로 자신의 각 필드를 갱신하므로 두 번째 반복이 끝났을 때 참조예측표는 그림 6의 (b)와 같게 되며 104, 108, 112번지는 이미 캐시에 있으므로 미스가 나지 않고, 100번지에서 한번의 미스만 일어난다. 세 번째 반복이 시작됐을 때 100번지가 이미 있으므로 각 필드를 갱신한다. 여기서 100번지의 상태가 steady가 되므로 선인출주소(prev_addr + stride)를 계산하여 선인출 명령을 발생한다. 104, 108, 112번지에 대해서는 앞의 반복과 동일한 방식으로 운영되어 세 번째 반복이 끝난 직후 참조예측표는 그림 6의 (c)와 같아지며 캐시 미스가 100번지에서 한번 일어난다. 그러나 다음 반복부터는 세 번째 반복에서 선인출된 참조주소가 캐시에 있으므로 미스가 나지 않으며 참조예측표의 100번지 상태가 steady이므로 계속해서 선인출 주소를 발생하게 되어 이 반복문이 끝날 때까지 전체 (반복문안의 메모리참조명령 개수(4) + (반복문내의 주소간격이 0이 아닌 메모리참조명령 개수(1) * 2))개의 캐시미스만이 일어난다.

기존의 참조예측표의 운영방식을 제안된 운영방식으로 바꿀 경우 2.2절에 보인 경우의 문제를 해결할 수 있으며 이에 따라 미스 수를 감소시킬 수 있고 참조예측표의 크기를 줄이더라도 더 좋거나 같은 효과를 볼 수 있다.

4. 실험 결과 및 분석

본 논문에서 제시하는 멀티미디어 응용프로그램에 대한 동적 미디어 데이터 검출에 따른 선인출 기법의 성능을 분석하기 위해, Digital Alpha DEC 시스템에서 트레이스 구동 시뮬레이션을 통하여 캐시의 수행을 모의 실험한 결과를 제시하고 이를 분석하였다. 그림 7은 이 논문에서 사용한 시뮬레이션 시스템의 구조이다. 응용프로그램의 명령어 트레이스를 생성하기 위하여Alpha CPU용 목적 코드를 이용하여 벤치마크 프로그램을 분석할 수 있도록 개발한 ATOM[6] 시뮬레이터를 사용하였다. 특히 이 실험에서는 데이터 적재/저장 명령어만을 트레이스하며 이 명령어들의 프로그램 카운터와 유효주소값을 기록하였다.

캐시의 수행과정을 모의하고 이에 관한 분석결과를 생성하기 위하여 트레이스 구동방식의 시뮬레이터인 dinero III[7]를 바탕으로 캐시 시뮬레이터를 구현하고 이를 CASIM이라 명명하였다. CASIM 시뮬레이터는 ATOM형 실행코드로부터 생성된 메모리 참조 명령어 트레이스를 입력으로 사용하여 적재/저장 명령어별 메모리 참조횟수와 캐시 미스 수 등의 결과를 산출하도록 개발한 시뮬레이터이다.

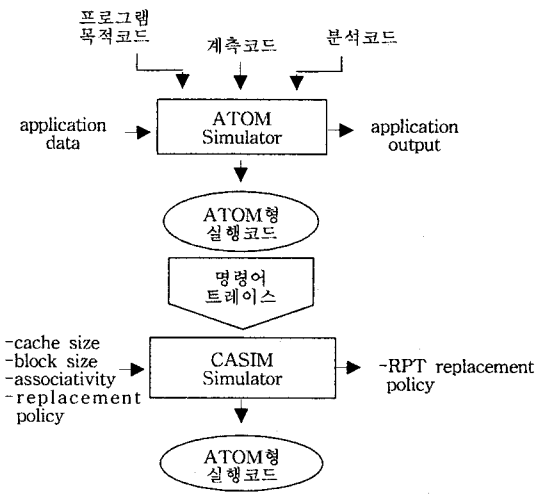

그림 7. 시뮬레이션 시스템

실험에서 사용된 벤치마크 프로그램은 멀티미디어용 벤치마크로 개발된 MediaBench[8] 벤치마크 중 jpeg 및 epic 프로그램을 선정하여 실험하였다. 표 1에 실험에 사용된 벤치마크 프로그램의 특징과 각 프로그램의 수행을 위하여 사용된 입력파일의 특징을 요약하였다.

	description	program module	data source	input size	image/frame size
JPEG	흑백 및 칼라 이미지를 위한 표준 압축 방식	cjpeg	image files of .ppm format	100K bytes	172×189
		djpeg	image file of .jpg format		
EPIC	실험용 이미지 압축 도구	epic	image files of .raw format	60K bytes	256×256
		unepic	image files of .E format		

표 1. 벤치마크 정보

그림 8에서는 본 논문에서 제시하는 참조예측표 운영방식을 비교 분석하기 위해 캐시의 세트 크기가 4이고 캐시의 크기가 8~256K까지의 구조에 기존의 방식을 이용하는 참조예측표 운영을 적용하여 캐시 미스 수를 조사해 보았다. 그림 8에서 epic은 캐시크기가 16K에서, cjpeg과 djpeg의 경우 캐시크기가 32K에서 캐시 미스 수가 크게 줄어들고 그 이상에서 차이가 거의 없음이 확인되었다. 이 실험을 통해 본 논문에서 제시하는 참조예측표 운영전략을 실험하는데 있어 캐시의 크기를 32Kbyte로 제한하였다.

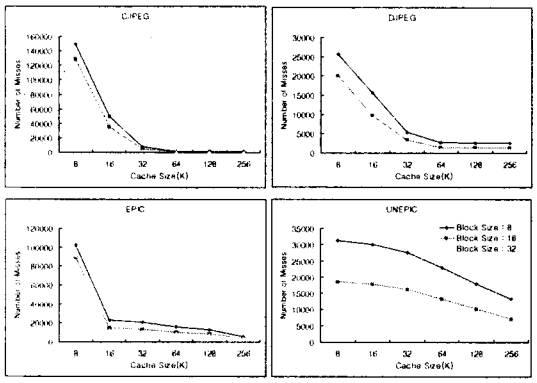

그림 8. 기존의 방식을 이용한 참조예측표 운영에서의 캐시 미스 수

그림 9는 캐시의 세트 크기가 4, 블록크기가 16byte, 캐시의 크기가 32Kbyte인 캐시구조에 기존 운영방식의 참조예측표를 적용한 캐시에서의 미스 수와 제안된 참조예측표를 적용한 캐시에서의 미스 수의 차이를 보여준다.

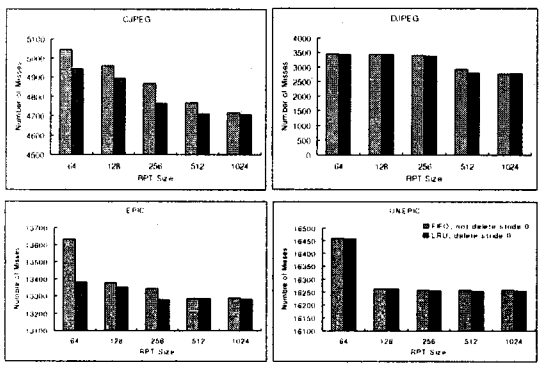

그림 9. 참조예측표 운영방법의 비교

그림 9에서 참조예측표의 크기가 64인 것과 128인 것에서, 제안된 운영방식을 사용하면 기존의 운영 방식을 사용한 참조예측표보다 그 크기를 절반으로 줄여도, unepic을 제외하고, 1~18개 이상의 미스 수의 감소를 보여준다.

이 실험의 결과 주소간격이 0인 것을 제거하지 않는 기존 운영방식의 128크기의 참조예측표를 사용하는 것과 주소간격이 0인 행을 제거하는 제안된 운영방식의 64크기의 참조예측표를 사용하는 구조의 성능이 같거나 미스 수에서 1~18개정도 이득을 볼 수 있다는 것을 알 수 있다.

5. 결론 및 향후 연구과제

본 논문에서는 선인출 방법에서 참조예측표의 제안된 운영방식을 이용하는 단일 캐시 구조에서의 효과를 측정하였다. 멀티미디어 응용프로그램을 처리할 때 참조예측표의 크기는 그림 9에서 보듯이 응용프로그램에서 가장 큰 반복문을 포괄할 수 있는 크기 이상의 참조예측표를 사용하는 것은 의미가 없다. 규칙 선인출은 하드웨어로 이루어진 동적 선인출 방법인 것을 고려해 볼 때 참조예측표의 크기를 절반으로 줄일 수 있다는 점에서 비용절감의 효과를 볼 수 있으며 향후 연구과제로 선인출에 따른 캐시 운영방법에 대한 연구가 필요하다.

참고문헌

[1] J. Fritts, "Multi-Level Memory Prefetching for Media and Streaming Processing," *Proceedings of International Congerence on Multimedia and Expo*, 2002

[2] S.P. VanderWiel, D. J. Lilja, "When Caches Aren't Enough: Data Prefetching Techniques," *IEEE Computer*, pp.23-20, July, 1997

[3] M.E. Wolf and M. S. Lan, "A Data Locality Optimizing Algorithm," *Proceedings of SIGPLAN'91 Conference on Programming Language Design and Implementation*, pp.30-44. June, 1991

[4] C. K. Luk, Optimizing the Cache Performance of Non_Numeric Applications, Ph. D. Thesis, *University of Toronto*, 2000.

[5] T.F. Chen and J. L. Baer, "Effective Hardware - Based Data Prefetching for High Performance Processors," *IEEE Transactions on Computers*, Vol. 44, No. 5, pp.609-623, May, 1995.

[6] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," *Proceedings of the ACM SIGPLAN* 94, pp.196-205, 1994

[7] M. D. Hill, Dinero III Cache Simulator, Technical Report, Computer Sciences Department, *University of Wisconsin, Madison*

[8] C. Lee, M. Potkonjak and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia Communications Systems," *Proceedings of the 30th Annual international Symposium on Micro architecture*, December, 1997.