미디어 데이터의 빠른 참조를 위한 캐시 운영 전략

문 현 주† · 김 석 일††

요 약

스트리밍 형태로 처리되는 멀티미디어 응용 데이터는 공간적 지역성은 큰 대신 시간적 지역성이 낮은 특징이 있다. 이 논문에서는 멀티미디어 응용 데이터에 내재된 메모리 참조의 규칙성을 활용하는 동적 선인출 기법을 제안하였다. 제안된 기법은 배열을 작은 블록으로 나누어 블록별로 계산을 수행하는 응용 프로그램의 경우에 기존의 방법과 비교하여 선인출 에러를 크게 줄일 수 있다. 여러 가지 미디어 벤치마크에 대한 실험 결과, 제안된 기법이 기존의 선형 선인출 기법에 비하여 예측의 정확도가 높고 성능이 우수함을 확인하였다.

A Cache Managing Strategy for Fast Media Data Access

Hyun Ju Moon† · Sukil Kim††

ABSTRAT

Multimedia data processing in streaming pattern contains high spatial locality and low temporal locality. This paper has proposed a dynamic data prefetching scheme that fully exploits the regularity between memory addresses referred consecutively. Compared to the existing data prefetching scheme, the proposed scheme can reduce data prefetching error when an application divides an array into smaller blocks and processes them block by block. Experimental results on various media benchmark programs show the proposed scheme predicts memory addresses more accurately and results in better performance than existing prefetching schemes.

키워드 : 동적 데이터 선인출(Dynamic Data Prefetching), 규칙적인 메모리 참조(Regulated Memory Reference), 스트리밍 패턴(Streaming Pattern), 캐시 시스템(Cache System)

1. 서 론

Moore의 법칙에 따르면 프로세서의 속도는 매 18개월 마다 2배가 되나 메모리의 속도는 매년 10% 정도의 비율로 향상되어 왔다. 이러한 발전 속도의 차이로 인해 메모리 참조 시간을 줄이는 것이 응용 프로그램의 빠른 처리에 관건이 되고 있다[1]. 메모리 참조 시간을 줄이기 위한 전통적인 방법의 하나가 주 메모리와 프로세서 사이에 캐시를 두어 주 메모리에 대한 접근 시간의 평균치를 줄이는 것이다. 이와 관련하여 캐시 블록의 대체 전략(replacement strategy), 적은 비용으로 보다 우수한 효과를 얻기 위한 계층 캐시 구조[2-4], 한번에 여러 라인의 블록을 주 메모리로부터 읽어오도록 하는 기법[5, 6] 등이 연구되었다.

최근에는 프로세서 또는 연산처리기가 필요로 하는 데이터나 명령어를 필요한 시점보다 훨씬 이전에 주 메모리로부터 캐시로 읽어오도록 하여 데이터나 명령어의 적재 시간을 줄이는 방법이 제안되었다. 명령어 재배치는 데이터를 주 메모리로부터 레지스터로 적재하는 시간이 긴 점을 보완하기 위하여 접근할 메모리 주소가 알려지는 시점에서 명령어를 실행하도록 목적코드를 만드는 방법이다. 이 방법은 데이터가 필요한 시점 이전에 데이터를 메모리 장치에 요구하는 기법으로 컴파일 시에 명령어 재배치가 이루어지므로 일종의 정적 데이터 선인출 기법이라고 볼 수 있다[7-12].

이와는 반대로 동적인 선인출 기법은 주 메모리를 참조할 때에 지난 일정 기간 동안에 이루어졌던 주 메모리의 참조 기록을 참고하여 앞으로 필요할지도 모를 데이터의 주소를 먼저 계산하여 이 데이터를 함께 자동적으로 주 메모리로부터 캐시로 적재하는 기법이다[5, 6, 13-16]. 이 기법은 데이터의 인출 주소의 변화량이 일정한 경우에 효과가 매우 높으나 인출 주소가 불규칙적으로 변화하는 경우에는 오히려 캐시의 오염을 증가시켜 성능을 저하시키는 원인이 되기도 한다. 따라서 동적 선인출 기법은 일정한 패턴으로 메모리

※ 이 논문은 한국과학재단 목적기초연구(R05-2002 000-01470-0) 지원으로 수행되었음. 이 논문은 2003학년도 나사렛대학교 학술 연구비 지원에 의해 연구되었음.
† 정 회 원 : 나사렛대학교 정보과학부 교수
†† 종신회원 : 충북대학교 전기전자컴퓨터공학부 교수
논문접수 : 2003년 11월 12일, 심사완료 : 2004년 1월 26일

를 참조하는 멀티미디어 응용 프로그램의 경우에 적합하다.

멀티미디어 응용 프로그램에서는 데이터를 대단위 배열의 형태로 저장하고 배열을 여러 개의 서브 블록으로 나누어 이들 서브 블록들을 연속적으로 읽어 들이면서 계산에 사용하는 스트리밍 패턴이 종종 사용된다[17, 18]. 또한, 스트리밍 패턴은 공간 지역성(spacial locality)이 높은 대신 시간 지역성(time locality)이 낮은 특징이 있다[18-20]. 즉, 스트리밍 데이터는 한번 사용된 후에는 다시 사용되지 않는 경향이 있으나 이웃한 데이터가 연속적으로 사용되는 특징이 있다. 따라서 스트리밍 데이터 구조는 공간 지역성이 높으므로 캐시 성능을 높이는데 동적 선인출 기법이 매우 유효한 수단이다. 이 논문에서도 스트리밍 데이터에서 캐시 효과를 높일 수 있는 동적 선인출 기법을 제안한다.

이 논문의 구성은 다음과 같다. 제 2장에서는 이 논문에서 제안하는 동적 선인출 기법의 토대가 되는 one block prefetching[21], reference prediction prefetching[22], streaming buffer[23] 기법을 소개하고, 이들 기법이 가지고 있는 제한점을 밝혔다. 제 3장에서는 하나의 미디어 데이터를 여러 개의 서브 블록으로 나누고 각각의 서브 블록에 대한 계산을 수행하는 응용 프로그램에 적합한 새로운 동적 선인출 기법을 제안하였다. 또한, 제안한 기법을 구현하고 이에 적합한 메모리 제어기를 설계하였다. 제 4장에서는 여러 가지 멀티미디어 벤치마크에 대하여 기존의 선인출 기법과 제안한 선인출 기법을 적용하였을 경우에 발생되는 캐시 미스 횟수의 변화를 측정하였다. 그 결과, 이 논문에서 제안한 선인출 기법의 성능이 기존의 기법에 비하여 크게 향상되었음을 확인하였다. 마지막으로 제 5장에서는 논문의 결론을 도출하였다.

2. 관련 연구

스트리밍 데이터를 구성하는 자료 구조는 배열 형태가 일반적이다. 또한, 이들 배열의 처리는 프로그램의 반복 구조로 표현되므로 스트리밍 데이터 참조 주소는 일정한 규칙성이 존재한다[17, 18]. 따라서 이러한 스트리밍 데이터 참조 주소의 규칙성을 활용하면 프로그램을 실행하면서 필요한 스트리밍 데이터의 주소를 예측하여 선인출 할 수 있다.

가장 간단한 선인출 기법은 어떤 메모리 블록을 참조할 때에 이 블록과 이웃한 몇 개의 블록을 선인출하는 방법이다[21]. 이 방법은 선인출할 블록들의 주소가 실제로 참조하려는 블록의 주소와 이웃하고 있으므로 선인출 주소를 결정하는 방법이 매우 간단한 장점이 있으나 응용 프로그램이 메모리 블록들을 순서대로 참조하는 경우에만 효과적이다. 예를 들어 몇 개의 블록을 건너뛰는 패턴을 보이는 응용 프로그램의 경우에는 선인출 하여 캐시에 저장된 메모리 블록들을 대부분 사용하지 않는 문제, 즉 캐시 오염(cache pollution) 문제가 발생한다. 이러한 현상은 동시에 몇 개의 메모리 블록을 선인출하는 경우에 보다 심각하므로 한 개의 메모리 블록을 선인출하는 것이 일반적이다.

이웃한 메모리 블록을 선인출할 경우에 발생하는 캐시 오염 문제를 해소하기 위한 방법으로 캐시와 메모리 사이에 버퍼를 두고 선인출한 메모리 블록을 임시로 버퍼에 저장하고 실제로 이 블록이 참조될 때에만 캐시로 옮기는 기법이 제안되었다[23]. 이 방법은 캐시 오염을 일으킬 수 있는 선인출 블록이 더 이상 캐시에 적재되지 않으므로 버퍼가 충분히 큰 경우에는 여러 개의 메모리 블록을 선인출하는 경우에도 성능 저하나 캐시 오염이 증가되지 않는 장점이 있다. 그러나 선인출하는 메모리 블록의 크기가 버퍼의 크기를 넘는 경우에 이를 처리하는 하드웨어가 매우 복잡해지는 단점이 있다. 또한, 이 버퍼를 채용하는 것은 버퍼의 운용 방식이 캐시와 동일하므로 결국 버퍼를 사용하는 것은 캐시의 크기를 늘리는 것과 동일하므로 비용의 증가가 수반되는 문제점이 있다.

Chen이 제안한 선인출 기법은 프로그램 내의 특정한 주소의 명령어가 반복해서 참조하는 메모리 주소를 기억하였다가 이 주소가 일정한 차이를 두고 변화하는 경우에 이를 예측하여 메모리 블록을 캐시로 적재하는 기법이다[14]. 일반적으로 미디어 데이터는 루프에 의하여 반복적으로 참조되는 프로그램 특징을 지니고 있으므로 미디어 데이터를 참조하는 명령어 마다 명령어의 적재 주소를 기억해두고 이 명령어가 데이터를 참조할 때마다 먼저 번에 명령어가 실행될 때에 참조하였던 데이터 주소와 비교하여 두 데이터 주소의 차이, 즉 거리(stride)를 계산하고, 이 차이를 현재 참조하려는 데이터의 주소에 더하여 앞으로 참조할 것으로 예상되는 데이터의 선인출 주소를 계산하는 것이다. 따라서 이 기법은 일정한 패턴으로 메모리를 참조하는 스트리밍 데이터의 처리시에 매우 유용하다.

예를 들어 $A_{n\times n}$ 배열의 원소 a_{ij}가 행 우선(row major) 방식으로 메모리에 저장되어 있다고 가정하고, 어떤 응용 프로그램이 배열 A를 $a_{11}, a_{12}, \cdots, a_{21}, a_{22}, \cdots, a_{nn-1}, a_{nn}$의 순서로 참조한다고 가정하면 a_{11}와 a_{12}의 참조가 완료되어야만 a_{13}을 선인출하는데 필요한 거리를 다음과 같이 계산할 수 있다. 즉,

$$d = \alpha(a_{12}) - \alpha(a_{11}).$$

여기서 α는 주어진 원소의 메모리 주소값이다.

따라서 a_{13}의 선인출 예측 주소

$$\hat{\alpha}(a_{13}) = \alpha(a_{12}) + d.$$

여기서 메모리 주소 $\hat{\alpha}(a_{13})$의 원소를 선인출하였고 $\alpha(a_{13}) = \hat{\alpha}(a_{13})$라면, a_{13}이 필요한 시점에서는 이미 a_{13}이 선인출

되어 캐시에 적재되어 있으므로 캐시 미스가 발생하지 않는다. 마찬가지로 a_{14}부터 a_{nn}에 이르는 원소를 순서대로 참조하는 과정에서 캐시 미스가 발생하지 않는다.

만일 응용 프로그램이 배열 A를 여러 개의 작은 블록으로 나누고 각각의 서브 블록을 처리하는 이중 구조의 루프로 구성된 프로그램 구조의 경우에는 다른 결과를 초래한다. 예를 들어 배열 $A_{n\times n}$를 $m\times m$개의 서브 배열로 나누어 처리한다고 가정하자. 즉,

$$A_{n\times n}=\begin{Bmatrix} B_{11} & B_{12} & \cdots & B_{1m} \\ B_{21} & B_{22} & \cdots & B_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ B_{m1} & B_{m2} & \cdots & B_{mm} \end{Bmatrix}.$$

여기서 B_{ij}는 크기가 $\frac{n}{m}\times\frac{n}{m}$인 서브 배열이다. 즉,

$$B_{p\times p}=\begin{Bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{p1} & b_{p2} & \cdots & b_{pp} \end{Bmatrix}.$$

여기서 $p=n/m$.

만일 응용 프로그램이 서브 배열 $B_{11}, B_{12}, \cdots, B_{1m}, B_{21}, \cdots, B_{2m}\cdots, B_{m1}, B_{m2}, \cdots, B_{mm}$의 순서로 처리한다고 가정하고, 모든 서브 배열 B_{ij}은 행 우선순위에 따라

$$b_{11}, b_{12}, \cdots, b_{1p-1}, b_{1p}, b_{21}, b_{22}, \cdots, b_{2p-1}, b_{p2}, \cdots, b_{pp-1}, b_{pp},$$

의 순서로 계산이 이루어진다고 가정하자.

이 경우에 Chen의 방법을 적용하면, b_{11}, b_{12},는 선인출 예측이 불가능하나 그 이후로 b_{1p-1}, b_{1p}를 참조하는 경우에는 이미 선인출에 의하여 캐시에 적재된 원소를 참조할 수 있다. 그러나 b_{1p}를 참조할 때에 선인출 하는 메모리 주소 $\hat{a}(b_{21})$는 다음과 같이 계산된다. 즉,

$$\hat{a}(b_{21})=a(b_{1p})+d\ ,\ \ d=a(b_{1p})-a(b_{1p-1}).$$

그런데 b_{21}은 b_{1p}와 연속한 주소에 저장되어 있지 않으므로 b_{21}의 참조시

$$a(b_{21})\neq\hat{a}(b_{21}).$$

즉, b_{21}의 선인출 예측이 정확하지 않다. 따라서 b_{22}의 선인출도 정확하지 않다. 마찬가지로 b_{i1}, b_{i2}, $i=1,2,\cdots,n/m$에서 선인출 예측은 정확하지 못하다. 이상으로부터 배열 A를 서브 블록으로 나누고, 이 블록을 임의의 순서로 처리하는 응용 프로그램의 경우에 나타나는 최대 선인출 예측 에러의 수는 $2(n/m)^2$이다. 따라서 미디어 데이터를 서브 블록으로 나누어 처리하는 응용 프로그램에 Chen이 제안한 기법을 적용하는 경우에는 선인출 에러가 매우 많이 일어남을 알 수 있다.

뿐만 아니라 Chen의 방법은 스트리밍 데이터를 처리하는 루프 내의 모든 명령어에 대한 적재 주소와 메모리 참조 주소를 저장해야 하므로 복잡한 자료구조가 필요하며, 이를 이용하여 선인출 주소를 결정하는 복잡한 논리회로가 필요한 단점이 있다. 이 논문에서는 미디어 데이터를 서브 블록으로 나누어 처리하더라도 선인출 에러의 발생빈도를 크게 줄일 수 있는 선인출 기법을 제안한다.

3. 2차원 참조 예측 기법

3.1 메모리 참조 규칙성

제 2장에서와 같이 미디어 데이터를 서브 블록으로 나누어 처리하는 경우에 Chen의 기법에서는 선인출 예측 에러가 많이 발생하는 것은 각 서브 블록을 구성하고 있는 서브 배열이 연속된 메모리내에 저장되어 있지 않기 때문이다. 따라서 서브 블록에 대한 선인출 예측 에러를 줄이기 위해서는 서브 블록을 연속된 메모리내에 재배치하거나 서브 블록을 구성하고 있는 원소를 참조할 때에 열과 행이 바뀔 때에 참조한 원소간의 거리(stride)가 급격히 달라지더라도 이를 예측할 수 있는 방법이 필요하다.

참조할 원소간의 거리가 달라지는 경우는 비단 서브 블록을 처리하는 경우에만 발생하는 것이 아니다. 예를 들어 (그림 1)과 같이 $n\times n$ 배열에서 숫자로 표시된 순서로 배열 원소를 참조하는 경우에도 거리의 불규칙한 변화가 나타난다. (그림 1)(a)는 대각선 방향으로 행마다 2개의 연속한 원소를 참조하는 경우로 참조할 원소의 거리는

$$1,\ n,\ 1,\ n,\ \cdots,\ 1,\ n,$$

와 같이 변화한다. 즉, 행이 바뀔 때마다 참조할 원소의 메모리 내 주소의 변화가 급격하다.

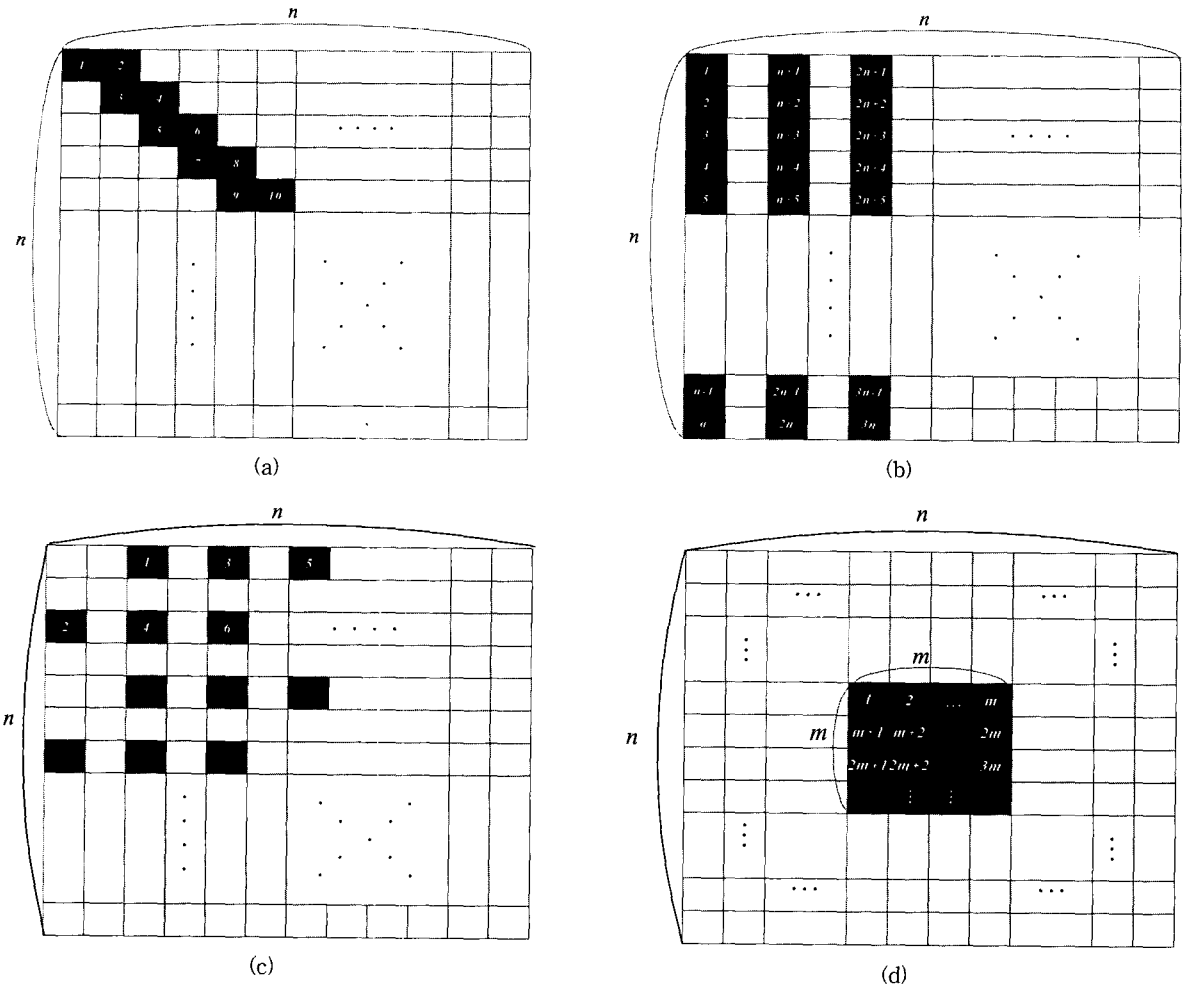
(그림 1)(b)는 행 우선 방식으로 저장된 배열을 열 우선 방식으로 참조하는 예이다. 이 경우에는 원소간 거리의 변화가

$$n,\ n,\ \cdots,\ n,\ -n(n-1)+2,\ n,\ n,\ \cdots,\ n,\\ -n(n-1)+2,\ n\cdots$$

와 같다. 즉, 열이 바뀔 때마다 원소간 거리가 급격하게 변화하므로 Chen의 기법을 적용할 때에는 열이 바뀔 때마다 선인출 예측 에러가 발생한다.

(그림 1)(c)는 연속하지 않은 원소를 참조하는 예로 원소간 거리의 변화가 (그림 1)(a)의 경우와 같이

$$2n-1,\ -2n+3,\ 2n-1,\ -2n+3,\ \cdots,\ 2n-1,\ -2n+3,\ \cdots$$



(그림 1) 여러 가지 메모리 참조 패턴

으로 불규칙적이다.

(그림 1)(d)는 배열의 일부분($m \times m$ 서브 배열)만을 참조하는 경우이다. 이 때 원소간 거리의 변화를 살펴보면,

$$1, 1, \cdots, 1, n-m+1, 1, 1, \cdots, 1, n-m+1, 1, 1, \cdots$$

과 같다. 따라서 원소간 거리의 변화는 (그림 1)(b)와 유사함을 알 수 있다. 이 경우에 전체 원소의 참조 횟수 중에서 원소간 거리가 불규칙하게 변화하는 횟수의 비율은 $1/m$이다. 즉, m과 n이 클수록 선인출 예측에러의 발생 가능성이 적어짐을 알 수 있다. 그러나 n이 작은 경우에는 선인출 예측에러의 발생빈도가 높아 원소간 거리의 급격한 변화를 예측하는 방법이 필요하다.

이상의 4가지 예에서 (그림 1)(a)와 (그림 1)(c)는 메모리를 참조할 때마다 원소간 거리의 급격한 변화가 반복되므로 Chen의 기법으로는 메모리 선인출을 위한 주소의 예측이 불가능하다[14]. 즉, Chen의 기법은 (그림 1)(b) 또는 (그림 1)(d)와 같이 원소간 거리가 일정한 경우에 효과적이다. 그러나 이 경우에도 열이 바뀌는 경우에는 선인출 예측에러가 발생한다.

3.2 2차원 거리에 의한 선인출 예측

Chen의 기법을 적용하는 경우에 (그림 1)(b)과 같은 단순한 메모리 참조패턴의 경우에 열이 바뀔 때마다 선인출 예측 에러가 발생하는 것은 Chen의 기법이 1차원 방향의 거리를 기억하기 때문이다. 즉, 메모리에 배열 원소가 연속적으로 저장되어 있을 때에만 첫 부분을 참조하는 경우를 제외하고는 나머지 원소의 참조시 선인출 에러가 발생하지 않는다. 그러나 (그림 1)(d)와 같이 배열을 서브 블록으로 나누어 계산하는 경우에 서브 블록의 원소들은 연속된 메모리 주소에 저장되어 있지 않다. 따라서 Chen의 기법을 적용하는 경우에는 매우 많은 선인출 예측 에러가 발생하게 된다.

만일 두 개의 거리를 활용한다면 (그림 1)(d)에서 서브 블록 원소에 대한 선인출 예측 에러를 줄이는 것이 가능하다. 즉, 메모리 참조에 따른 거리의 변화가

$$1, 1, \cdots, 1, n-m+1, 1, 1, \cdots, 1, n-m+1, 1, 1, \cdots$$

과 같다고 가정하면 m개의 연속된 배열 원소를 참조한 후 다음 열로 바꾸고 첫 번째 행부터 배열원소를 참조하는 과정을 반복하게 된다. 따라서 거리가 일정한 경우에는 Chen의 방법에서와 같이 선인출 주소

$$\hat{a}(a_{i+1}) = a(a_i) + S_1.$$

여기서 S_1은 행 방향으로의 거리이며, $S_1=1$. 또한, a_{i+1}은 a_i의 주소에 의하여 선인출할 원소를 의미한다.

원소간 거리의 급속한 변화(여기서는 $n-m+1$)가 발생하는 경우에는 이 거리를 S_2에 저장해 놓고 행과 열이 동시에 바뀌기 직전에 선인출 주소를 S_2를 이용하여 예측할 수 있다. 즉,

$$\hat{a}(a_{i+1}) = a(a_i) + S_2.$$

여기서 S_2는 열 방향으로의 거리이며, (그림 1)(d)에서 $S_2 = n-m+1$.

따라서

$$\hat{a}(a_{i+1}) = \begin{cases} a(a_i)+S_1 & \text{if stride is } \mid \text{arly changed} \\ a(a_i)+S_2 & \text{otherwise} \end{cases}$$

즉, 행과 열이 바뀌기 직전의 원소를 알 수 있다면 거리의 급격한 변화에도 불구하고 정확한 선인출 주소를 예측할 수 있다. 행과 열이 동시에 바뀌는 지점을 찾는 과정은 다음절에서 설명할 것이다.

(그림 1)(b)에서

$$n, n, \cdots, n, -n(n-1), +2, n, n, \cdots, n, -n(n-1), +2, n \cdots$$

와 같이 원소간 거리가 변화하는 경우에도 S_1과 S_2를 이용하여 선인출 주소를 정확히 예측할 수 있다. 즉,

$$S_1 = n, S_2 = -n(n-1)+2.$$

여기서 (그림 1)(d)의 경우와 마찬가지로 행과 열이 바뀌기 직전의 원소를 알 수 있다면 거리의 급격한 변화에도 불구하고 정확한 선인출 주소를 예측할 수 있다.

(그림 1)(a)에 보인 메모리 참조패턴의 경우에

$$1, n, 1, n, \cdots, 1, n,$$

와 같이 거리가 변화하는 경우에도 S_1과 S_2를 이용하여 선인출 주소를 정확히 예측할 수 있다. 즉,

$$S_1 = 1, S_2 = n.$$

따라서 이 경우도 (그림 1)(d)의 경우와 마찬가지로 행과 열이 바뀌기 직전의 원소를 알 수 있다면 거리의 급격한 변화에도 불구하고 정확한 선인출 주소를 예측할 수 있다.

3.3 행과 열의 변경 지점 예측

2차원 배열의 원소를 참조하는 과정에서 행과 열이 동시에 바뀌는 지점을 예측하는 절차는 2차원 배열의 선인출 주소 예측에 필수적이다. 이 논문에서는 거리가 일정할 경우에는 이 값을 1차원 거리로 간주하고, 거리가 변경될 때에는 이를 2차원 거리로 간주하여 이를 해결하였다. 예를 들어,

$$b, b, b, b, a, b, b, b, b, a, b, b, \cdots$$

와 같이 거리가 변화할 경우에 $S_1 = b, S_2 = a$로 설정하며, 이 때 S_1이 b로 설정된 이후에 값이 b에서 a로 변화할 때까지의 참조한 원소의 수(여기서는 $L=4$)를 저장해 놓는다면, S_2가 a로 설정된 이후부터 S_1이 b인 동안에 원소를 L개만큼 참조한 후에는 열과 행이 동시에 변경되는 것을 예측할 수 있다. 즉, 위에 보인 예에서 두 번째로 거리가 a로 변경되는 지점을 정확히 판단할 수 있다. 만일 거리 S_1이 L개의 원소를 참조하기 전에 변화되거나 L개가 계속된 이후에 선인출한 주소를 잘못 예측하거나 그 때의 거리가 a가 아닌 경우에는 S_1과 S_2를 초기화하여 새로운 배열의 참조 패턴을 찾도록 한다.

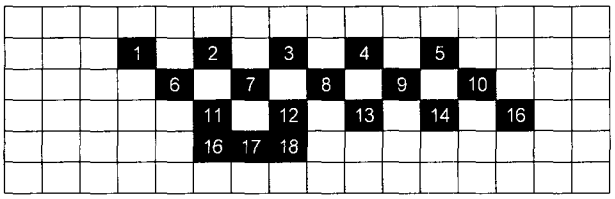
(그림 2)는 2차원 거리에 의하여 선인출 주소를 예측하는 알고리즘이다. (그림 2)에서 en_2D는 S_2를 설정하기 위하여 거리가 변화하는 첫 번째 지점을 찾기 위한 변수이며, en_2D가 세트 될 때까지 C_R을 증가시킨다. 여기서 C_R는 거리가 변화하지 않는 동안 참조하는 원소의 수를 저장하는 변수이다. C_v는 거리 값이 일정한 동안 참조하는 원소의 개수를 세어서 C_R과 비교하기 위한 변수이다. 즉, 거리가 일정하며, C_v 값이 C_R보다 작을 때까지는 같은 열의 원소를 참조하는 경우이며, C_v이 C_R와 같은 지점이 열과 행이 바뀌는 지점이다.

```
while I_pc(A_addr) is begin issued do
  if I_pc do Not exist in 2DRPT then
    // Let 2DRPT be a 2D Reference Prediction Table.
      Put I_pc to 2DRPT ; //
    create entry for I_pc in 2DRPT.
    2DRPT(I_pc, P_addr) = A_addr ;
    2DRPT(I_pc, S_1) = 0 ;
    2DRPT(I_pc, S_2) = 0 ;
    2DRPT(I_pc, C_R) = 1 ;
    2DRPT(I_pc, C_v) = 0 ;
    do nothing ; //No prefetch
  else// entry for I_pc exists in 2DRPT.
  C_v++ ; //increase C_v.
  end if

  if ( C_v == C_R && en_2D==true) then// prefetch by S_2
      prefetch A_addr + S_2
```

```
    C_v = 0 ;
  else // prefetch by S_1
    if S_1 == 0 then // do nothing
    else
      prefetch A_addr + S_1 ;
      if A_addr == P_addr + S_1 then //prefetch expectation is correct
        if en_2D == false then //no stride change in 1-Dim
          C_R++ ;              //continue counting no. of elements
        end if                 //otherwise C_R is fixed.
      else//prefetch expectation is fault
        if en_2D == false then //new stride has met here
          en_2D = true ;
        end if// set new S_2
        C_v = 0 ;
        S_2 = A_addr - P_addr ;
        end if
      end if
    end if
  end while
```

(그림 2) 2차원 참조 예측 알고리즘



(a) 참조패턴

Sequence	C_v	C_R	en_2D	P_{addr}	S_1	S_2	prefetch prediction error
1	0	0	0	537050384	0	0	○
2	1	1	0	537050386	2	0	○
3	2	2	0	537050388	2	0	×
4	3	3	0	537050390	2	0	×
5	4	4	0	537050392	2	0	×
6	4	0	1	537050401	2	9	○
7	4	1	1	537050403	2	9	×
8	4	2	1	537050405	2	9	×
9	4	3	1	537050407	2	9	×
10	4	4	1	537050409	2	9	×
11	4	0	1	537050418	2	9	×
12	4	1	1	537050420	2	9	×
13	4	2	1	537050422	2	9	×
14	4	3	1	537050424	2	9	×
15	4	4	1	537050426	2	9	×
16	0	0	0	537050384	0	0	○
17	1	1	0	537050385	1	0	○
18	2	2	0	537050386	1	0	×

(b) 예측표의 변화

(그림 3) 배열 원소의 2차원 참조 예측

(그림 3)(a)는 2차원 배열을 구성하는 미디어 데이터의 일부 원소를 참조하는 것을 보인 예이다. (그림 3)(b)는 (그림 3)(a)의 데이터 참조에 대하여 이 논문에서 2차원 참조 예측 기법을 적용하였을 경우에 선인출 테이블의 변화를 보인 것이다. (그림 3)(b)에서 C_v와 C_R 값이 모두 0인 부분과 *en_2D*이 *true*이며 C_v=0인 경우가 선인출 예측 에러가 발생하는 위치이다. 즉, 처음 원소를 인출하는 경우와 S_2가 0에서 9로 바뀌는 시점에서 선인출 예측 에러가 발생한다.

4. 실험 및 분석

이 논문에서 제안한 2차원 참조 예측 기법의 성능을 분석하기 위하여 여러 가지 벤치마크에 대한 명령어 트레이스를 생성하고 명령어 트레이스 중에서 *load* 명령어가 참조하는 메모리 주소를 토대로 선인출 주소를 예측하였다. 또한 예측된 선인출 주소의 데이터도 캐시로 적재하도록 하였다. 이 과정에서 실제로 벤치마크 프로그램 실행 중에 선인출 데이터가 실제로 *load* 해야 하는 데이터인가를 확인하여 선인출 예측의 정확성을 측정하였다.

명령어 트레이스는 Alpha CPU용 목적 코드를 분석할 수 있도록 개발한 ATOM 시뮬레이터를 사용하였다[24]. 선인출 예측의 정확성을 측정하는 과정은 위스콘신 대학에서 개발한 트레이스 구동 방식의 시뮬레이터인 Dinero III를 바탕으로 여러 가지 선인출 기법별로 캐시 시뮬레이터를 구현하였다[25]. 즉, 캐시 시뮬레이터는 ATOM 시뮬레이터가 생성한 명령어 트레이스를 입력받아 *load/store* 명령어의 실행 갯수(메모리 참조 횟수)와 캐시 미스 횟수 등을 보고하도록 하였다.

〈표 1〉 벤치마크 프로그램

program module	description	data source	input size	image/ frame size
mpeg2enc	transforming tool for digital video stream (compression)	image frames of Y.U.V format	4 frames	256×242
mpeg2dec	transforming tool for digital video stream (decompression)	mpeg video streams	4Mbytes	256×242
cjpeg	compression standard for image	image files of .ppm format	100Kbytes	172×189
djpeg	decompression standard for image	image files of .jpg format	50Kbytes	172×189
epic	image compressing tool	image files of .raw format	60Kbytes	256×256
unepic	image decompression tool	image files of .E format	4Kbytes	256×256

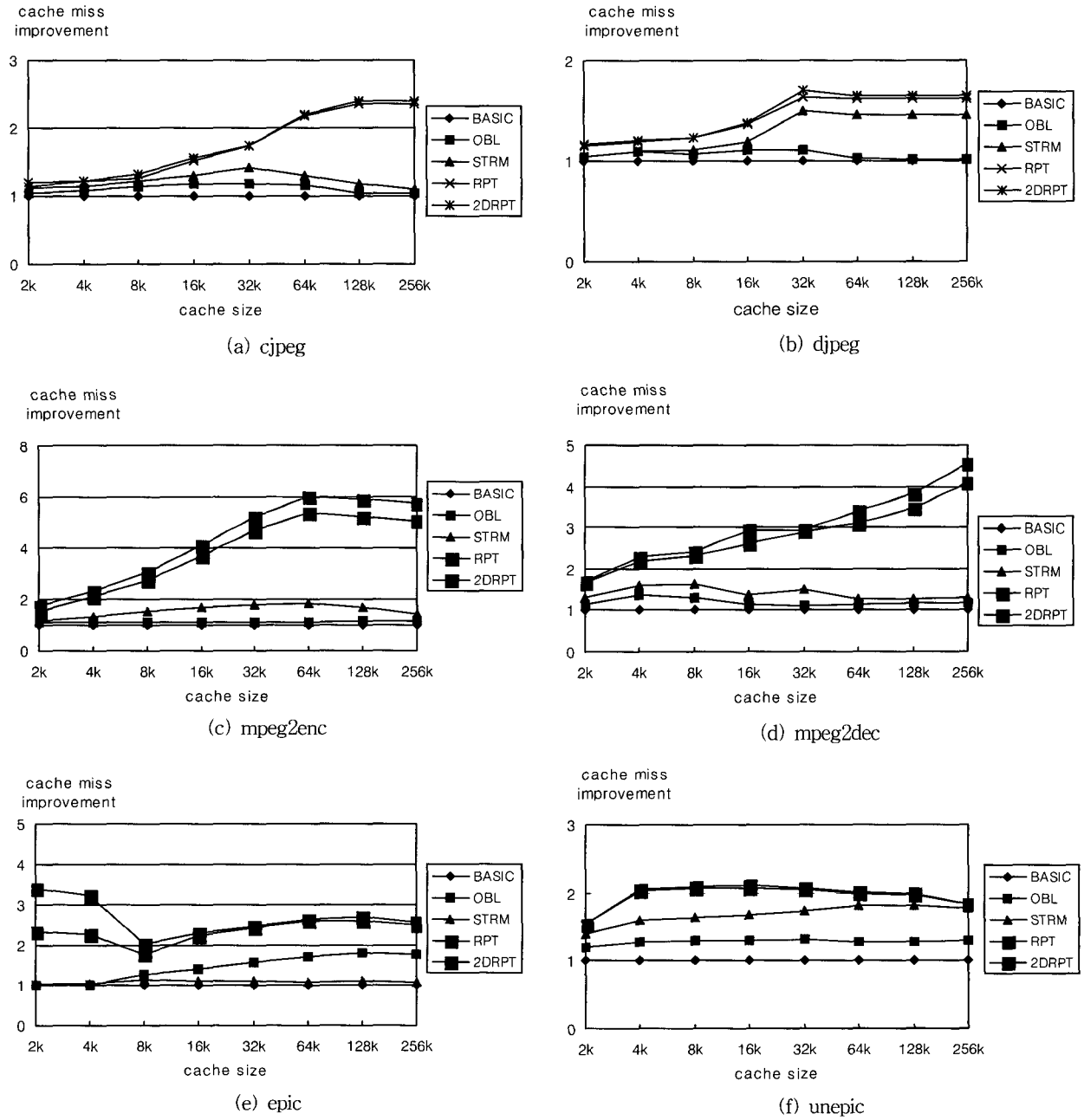
실험을 위한 응용 프로그램으로는 MediaBench 벤치마크를 구성하는 MPEG, JPEG 및 EPIC 프로그램을 선정하였

다[26]. 각 프로그램은 <표 1>과 같이 인코더와 디코더로 구성되어 있다.

(그림 4)는 선인출 기법을 전혀 사용하지 않은 경우(BASIC)를 기준으로 어떤 메모리 블록을 참조할 때에 이 블록과 이웃한 몇 개의 블록을 선인출 하는 경우(OBL), 캐시와 메모리 사이에 버퍼를 두고 선인출한 메모리 블록을 임시로 버퍼에 저장하는 경우(STRM), 하나의 거리 값만을 이용하여 선인출 주소를 예측하는 Chen의 기법(RPT), 2차원 참조 예측(2DRPT) 기법을 사용한 경우의 캐시 미스 감소율을 측정한 결과이다.

(그림 4)에서 알 수 있듯이 2DRPT 기법이 기존의 선인출 기법들에 비하여 캐시 미스의 감소율이 가장 큰 것을 알 수 있다. 즉, 2DRPT 기법이 기존의 기법에 비하여 선인출 예측이 보다 정확함을 보여준다. 특히 Chen의 RPT 기법에 비하여 성능이 우수함을 알 수 있다. MPEG 모듈의 경우, 2DRPT 기법에 의한 성능 향상이 뚜렷하게 나타났으며 이는 MPEG 모듈이 2차원 배열을 처리하는 과정에서 불규칙한 데이터 참조가 부분적으로 많이 있음을 보여주는 것이다.

(그림 5)는 (그림 4)에서 유사한 성능을 보이는 Chen의

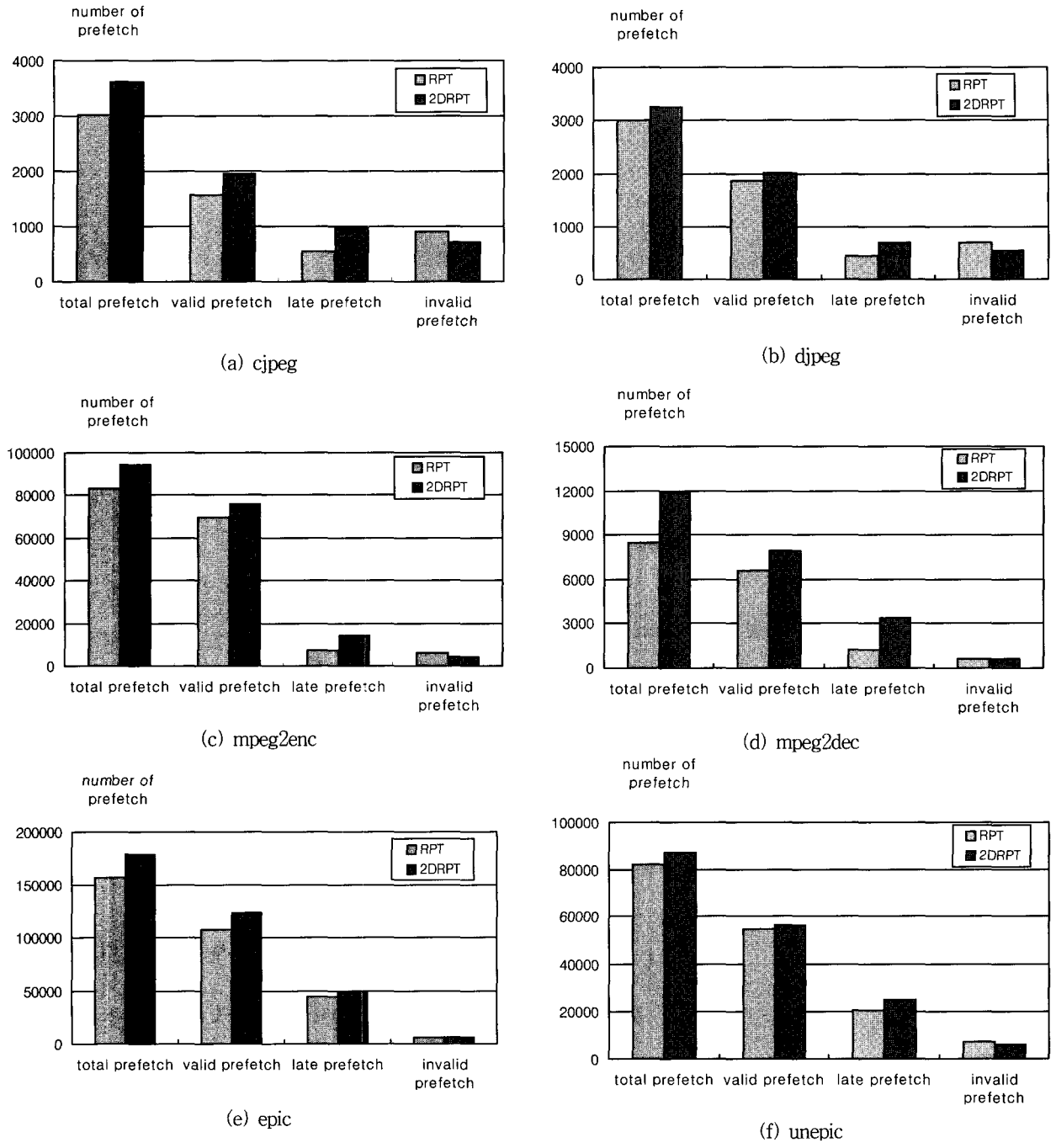


(a) cjpeg (b) djpeg

(c) mpeg2enc (d) mpeg2dec

(e) epic (f) unepic

(그림 4) 선인출 기법에 따른 캐시 미스 횟수의 감소율



(그림 5) 선인출 데이터의 유효성 비교

RPT 기법과 이 논문에서 제안한 2DRPT 기법을 적용한 경우에 선인출 데이터의 실제 참조 여부를 비교한 것이다. 이를 위하여 선인출 된 총 데이터 블록의 수(total prefetch)와 유효 선인출 블록(valid prefetch) 즉, 선인출 된 후에 실제로 참조된 메모리 블록의 수, 그리고 지각 선인출 블록(late prefetch)의 수 및 무효 선인출 블록(invalid prefetch) 즉, 선인출 되었으나 참조되지 않은 메모리 블록의 수를 비교하였다.

(그림 5)에서 알 수 있듯이 2DRPT 기법을 사용한 경우 선인출 되는 데이터의 수와 유효 선인출(valid prefetch) 및 지각 선인출(late prefetch) 데이터의 수가 Chen의 RPT기법을 적용하였을 때에 비하여 증가한 것을 알 수 있다. 그러나 무효 선인출(invalid prefetch) 데이터의 수는 현저히 감소한 것을 알 수 있다. 이는 2DRPT 기법이 RPT 기법에 비하여 선인출 예측의 정확도가 높은 것을 보여주는 것이다.

5. 결 론

이 논문에서는 멀티미디어 응용 프로그램의 데이터 참조 규칙성을 활용하여 프로그램의 메모리 참조 속도를 향상시키는 2차원 참조 예측(2DRPT) 기법을 제안하였다. 선인출을 위한 기존의 참조 예측 기법이 일정하게 유지되는 원소간 거리를 이용하는 반면, 제안된 2차원 참조 예측 기법에서는 일정하게 유지되는 거리 뿐 만 아니라 반복적으로 급격히 변화하는 거리도 선인출 할 메모리 주소 예측에 활용한다. 즉, 배열 원소가 참조되는 순서에 따라 일정하게 유지되는 원소간의 거리(S_1)와 급격히 변화하는 원소간의 거리(S_2)를 모두 저장하고 참조되는 원소의 위치에 따라 S_1 또는 S_2를 이용하여 선인출 할 주소를 계산한다.

블록 알고리즘의 유용성을 조사하기 위해 멀티미디어 처리용 벤치마크인 MediaBench 중 MPEG, JPEG, EPIC, 프로그램에 대한 캐시 미스율을 측정하였다. 실험 결과, 2차원 참조 예측(2DRPT) 기법이 기존 선인출 기법들에 비하여 우수한 성능을 나타냈으며 특히 기존의 참조 예측(RPT) 기법에 비하여 최대 45%의 성능 향상을 나타냈다. 2DRPT 기법은 블록 패턴을 더 많이 갖는 응용에서 성능이 더욱 우수할 것으로 판단된다. 또한 2DRPT 기법에 의하여 선인출 된 데이터 중 실제 참조되는 데이터의 비율이 RPT 기법에 비하여 높은 것으로 나타나 2DRPT 기법이 정확도 측면에서 우수한 것을 입증하였다.

향후에는 2DRPT 기법에 의해 정확하게 예측된 데이터 중 실제 참조되는 시점 이후에 캐시로 적재되는 지각 선인출의 영향을 줄임으로써 선인출의 효과를 극대화 하는 연구를 수행하고자 한다.

참 고 문 헌

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

[2] J. L. Baer and W. H. Wang, "Architectural Choices for Multi-level Cache Hierachies, *Proceedings of 16th International Conference on Parallel Processing*ierarchies," pp.258-261, 1987.

[3] K. Gharachorloo, A. Gupta and J. Hennessy, "Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors," P*roceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.245-257, 1991.

[4] J. R. Goodman, *Cache Consistency and Sequential Consistency*, Technical Report TR-1006, University of Wisconsin-Madison, February, 1991.

[5] F. Dahlgren, M. Dubois and P. Stenstrom, "Fixed and Adaptive Sequential Prefetching in Shared-memory Multiprocessors," *Proceedings of 1993 International Conference on Parallel Processing*, pp.I56-63, August, 1993.

[6] N. P. Jouppi, "Improving Direct-mapped Cache Performance by the Addition of a Small Fully associative Cache and Prefetch Buffers," *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 364-373, May, 1990.

[7] C. H. Chi and K. K. Fang, "Compiler Driven Data Cache Prefetching for High Performance Computers," *Proceedings of Regional 10th Annual International Conference*, Vol.2, No.9, pp.274-278. August, 1994.

[8] M. H. Lipasti et al., "SPAID : Software Prefetching in Pointer and Call-Intensive Environments," *Proceedings of 28th International Symposium on Microarchitecture*, pp. 231-236, 1995.

[9] C. K. Luk and T. C. Mowry, "Compiler-Based Prefetching for Recursive Data Structures," *Proceedings of 9th International Conference on Architectural Support for Programming Languages and Operating Systems, ACM Press*, pp.222-233, 1996.

[10] T. Mowry and A. Gupta, "Tolerating Latency through Software-controlled Prefetching in Shared-memory Multiprocessors," *Journal of Parallel and Distributed Computing*, Vol.12, No.2, pp.87-106, June, 1991.

[11] T. C. Mowry, S. Lam and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," *Proceeding of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM Press, pp.62-73, 1992.

[12] P. Ranganathan, V. S. Pai , H. Abdel-Shafi, S. V. Adve, "The Interaction of Software Prefetching with ILP Processors in Shared-Memory Systems," *ACM SIGARCH Computer Architecture News*, Vol.25 No.2, pp.144-156, May, 1997.

[13] D. F. Zucker, M. J. Flynn and R. B. Lee, "A Comparison of Hardware Prefetching Techniques for Multimedia Benchmarks," *Proceedings of International Conference on Multimedia Computing and Systems*, pp.236-244, June, 1996.

[14] 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.

[15] J. Fritts, W. Wolf and B. Liu, "Understanding Multimedia Application Characteristics for Designing Programmable Media Processors," *Proceedings of SPIE*, Vol.3655, pp.2-13, January, 1991.

[16] S. P. VanderWiel and D. J. Lilja, "A Compiler-Assisted

Data Prefetch Controller," *Proceedings of International Conference on Computer Design*, pp.372-377, October, 1999.

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

[18] M. E. Wolf and M. S. Lam, "A Data Locality Opitmizing Algorithm," *Proceedins of SIGPLAN'91 Conference on Programming Language Design and Implementation*, pp. 30-44, June, 1991.

[19] S. Carr, K. S. McKinley and C. W. Tseng, "Compiler Optimization for Improving Data Locality," *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.252-262, October, 1994.

[20] S. Sohoni, Z. Xu, R. Min and Y. Hu, "A Study of Memory System Performance of Multimedia Applications," A*CM SIGMETRICS*, June, 2001.

[21] A. Smith, Sequential Program Prefetching in Memory Hierachies, *IEEE Computer*, 11(2), pp.7-21, 1997.

[22] T. F. Chen and J. L. Baer, "Effective Hardware-Based Data Prefetching for High Performance Processors," *IEEE Transactions on Computers*, 44(5), pp.609-623, May, 1995.

[23] N. P. Jouppi, Improving Direct-mapped Cache Performance by the Addition of a Small Fully associative Cache and Prefetch Buffers, *Proceedings of the 17th Annual International Symposium on Computer Atchitecture*, 364-373, May, 1990.

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

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

[26] 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 Microarchitecture*, December, 1997.

문 현 주

e-mail : hjmoon@kornu.ac.kr

1995년 충북대학교 컴퓨터과학과(학사)

1997년 충북대학교 대학원 전자계산학과 (이학석사)

2003년 충북대학교 대학원 전자계산학과 (이학박사)

2003년~현재 나사렛대학교 정보과학부 전임강사

관심분야 : 데이터 선인출, 분산 병렬 처리, 고성능 컴퓨팅, 그리드 컴퓨팅 등

김 석 일

e-mail : ksi@cbucc.chungbuk.ac.kr

1975년 서울대학교 전기공학과(공학사)

1975년~1990년 국방과학연구소 선임 연구원

1984년 연세대학교 대학원 전자계산학과 (공학석사)

1989년 North Carolina State University(공학박사)

1990년~현재 충북대학교 전기전자컴퓨터공학부 교수

2002년~현재 충북대학교 전기전자컴퓨터공학부 학부장

관심분야 : 병렬 컴퓨터 구조, 슈퍼컴퓨팅, 병렬처리 언어, 이기종 분산처리, 시각장애인 사용자 인터페이스