# 칩의 크기가 제한된 단일칩 프로세서를 위한 레벨 1 캐시구조

주 영 관[†] · 김 석 일[††]


## 요 약

이 논문에서는 단일 칩 프로세서에서 제한된 공간의 레벨 1 캐시를 구성하고 있는 선인출 캐시 $L_P$와 요구인출 캐시 $L_1$의 합이 일정할 때, $L_1$와 $L_P$의 크기의 적정한 비율을 실험을 통하여 분석하였다. 실험 결과, $L_1$와 $L_P$의 합이 16KB일 경우에는 $L_1$을 12KB, $L_P$를 4KB로 구성하고 $L_P$의 선인출 기법과 캐시교체정책은 각각 OBL과 FIFO을 적용시키는 레벨 1 캐시 구조가 가장 성능이 우수함을 보였다. 또한 이 분석은 $L_1$와 $L_P$의 합이 32KB 이상인 경우에는 $L_P$의 선인출 기법으로는 동적필터 기법을 사용하는 것이 유리함을 보였고 32KB의 공간이 가용한 경우에는 $L_1$을 28KB, $L_P$를 4KB로, 64KB가 가용한 경우에는 $L_1$을 48KB, $L_P$를 16KB로 레벨 1 캐시를 분할하는 것이 가장 좋은 성능을 발휘함을 보였다.


# A Level One Cache Organization for Chip-Size Limited Single Processor

YoungKwan Ju[†] · Sukil Kim[††]


## ABSTRACT

This paper measured a proper ratio of the size of demand fetch cache $L_1$ to that of prefetch cache $L_P$ by imulation when the size of $L_1$ and $L_P$ are constant which organize space-limited level 1 cache of a single microprocessor chip. The analysis of our experiment showed that in the condition of the sum of the size of $L_1$ and $L_P$ are 16 KB, the level 1 cache organization by constituting $L_P$ with 4 KB and employing OBL and FIFO as a prefetch technique and a cache replacement policy respectively resulted in the best performance. Also, this analysis showed that in the condition of the sum of the size of $L_1$ and $L_P$ are over 32 KB, employing dynamic filtering as prefetch technique of $L_P$ are more advantageous and splitting level 1 cache by constituting $L_1$ with 28 KB and $L_P$ with 4 KB in the case of 32 KB of space are available, by constituting $L_1$ with 48 KB and $L_P$ with 16 KB in the case of 64 KB elicited the best performance.


**키워드 : 캐시구조(cache architecture), 선인출기법(prefetch technique), 온칩 캐시(on-chip cache), 단일칩 프로세서(one chip Processor)**

## 1. 서 론

컴퓨터가 데이터베이스, 멀티미디어 처리, 범용 등 사회의 다양한 분야에 활용되면서 보다 빠른 계산력을 필요로 하게 되었다. 이에 따라 빠른 연산이 가능한 프로세서 구조에 대한 연구가 활발히 진행되어 오고 있다. 빠른 연산을 가능하도록 하는 전통적인 방법이 프로세서의 클록 속도를 높이는 방법이다. 그러나 이 방법에 의한 속도 향상은 이미 물리적인 한계에 봉착하여 더 이상 클록속도를 높이는 것이 불가능한 상태가 되었다. 이에 따라 동시에 여러 개의 명령어를 실행하도록 하는 명령어 파이프라인 구조[14], 여러 개의 연산처리기를 동시에 활용하는 슈퍼스칼라 프로세서 구조[15], 서브워드 병렬성을 활용하는 구조[16] 등에 관한 연구가 제안된 바 있다.

또한, 목적코드에서 명령어의 빠른 수행을 저해하는 원인을 제거하는 기법에 대한 연구도 활발히 진행되었다. 예를 들어, 분기예측 기법은 분기조건을 추측하고 추측한 결과에 따라 다음에 수행될 명령어를 선택하여 실행하도록 하는 기법이다[17]. 이 기법은 분기조건의 추측이 올바른 경우에는 분기조건의 실행이 완료되기 이전에 실행한 명령어 스트림의 결과를 그대로 사용할 수 있으므로 더 많은 명령어의 실행이 가능하다. 물론 분기조건의 실행 결과가 틀리면 이에 따른 페널티가 발생하는 단점이 있기는 하지만 단위시간당 실행되는 명령어의 수를 늘리는 방법의 하나이다.

† 준 회 원 : 충북대학교 전자계산학과 박사과정
†† 종신회원 : 충북대학교 전기전자컴퓨터공학부 교수, 유비쿼터스바이오정보기술연구센터 연구소장
논문접수 : 2005년 2월 19일, 심사완료 : 2005년 3월 23일

메모리 참조 지연시간을 줄이기 위한 방법으로는 캐시와 메모리를 계층적으로 구성한 구조[1, 2, 6], 여러 개의 메모리 모듈을 동시에 참조하여 메모리 대역폭(memory bandwidth)을 증가키는 메모리 인터리빙 구조[18] 등이 있다.

근래에는 보다 적극적으로 메모리 참조 지연시간을 줄이는 기법이 연구되고 있다. 명령어 재배치(instruction reorder)[28]는 메모리를 참조하는 명령어가 메모리 지연시간을 필요로 하므로 메모리 지연시간에 의하여 명령어의 파이프라인 사이클이 진행되지 못하는 문제를 해소하기 위하여 참조할 메모리 주소가 확인되는 시점에서 메모리 참조 명령어를 실행하도록 메모리 참조 명령어를 재배치하여 메모리 지연시간을 줄이는 기법이다. 이 방법은 데이터를 필요로 하는 시점 이전에 필요한 데이터를 메모리 장치에 요구하는 기법으로 컴파일 시에 명령어 재배치가 이루어지므로 일종의 정적 데이터 선인출 기법이라고 볼 수 있다[3].

최근에는 캐시 제어기가 어떤 명령어의 실행 시에 필요한 피연산자나 데이터를 사전에 캐시에 적재하도록 함으로써 명령어의 실행 시에 피연산자나 데이터를 참조함에 따른 지연을 최소화 하는 동적 선인출 기법이 활발히 연구되고 있다[1, 6, 12, 13]. 즉, 메모리를 참조할 때에 지난 기간 동안에 참조되었던 메모리의 참조 주소를 토대로 앞으로 필요할지도 모를 데이터의 주소를 예측하여 이를 메모리로부터 캐시로 적재하도록 한다. 따라서 이 기법은 응용 프로그램이 실행되는 과정에서 참조할 데이터의 주소를 예측해야 하므로 과거의 참조주소로부터 얼마나 정확한 예측을 할 수 있는가에 의하여 선인출 기법의 효과가 결정되는 특징이 있다.

범용 프로그램은 일반적으로 한번 참조한 데이터를 반복해서 사용하는 재사용성이 매우 높다. 그러나 멀티미디어 응용 프로그램의 경우에는 이웃한 데이터를 연속적으로 참조하는 지역성이 좋은 대신 반복해서 사용하는 재사용성이 낮은 특징이 있다. 즉, 멀티미디어 응용 프로그램을 주로 수행하는 미디어 프로세서는 지역성이 좋으나 재사용성이 낮은 멀티미디어 응용 프로그램에 적합한 구조를 지니고 있다[19, 20]. 반면에 범용 프로그램을 주로 처리하는 프로세서의 경우에는 재사용성을 효과적으로 활용하는 구조로 구성되어 있다[21, 22]. 따라서 미디어 프로세서에서 범용 프로그램을 실행하는 경우나 범용 프로세서에서 멀티미디어 응용 프로그램을 실행한다면 그 효과가 떨어질 수밖에 없다. 그러나 개인용 컴퓨터와 같은 대부분의 범용 컴퓨터가 범용과 멀티미디어 응용 프로그램 등의 다양한 응용 프로그램을 수행하고 있다. 따라서 범용 컴퓨터용 선인출 기법과 캐시 구조는 데이터의 지역성과 재사용성간의 타협에 의하여 효과적인 구성이 가능하다. 이 논문에서는 범용 프로세서에 적용될 수 있는 선인출 캐시구조를 제안하였다.

## 2. 관련연구 및 연구동기

### 2.1 선인출 알고리즘

프로세서의 캐시에 적용되는 선인출 기법은 매우 광범위하게 연구되었다. OBL(one block lookahead)기법은 어떤 메모리 블록을 참조할 때에 이웃한 블록을 선인출하도록 하는 방법이다[11]. 이 기법은 선인출 주소가 참조하는 데이터의 메모리 주소에 의하여 결정되므로 캐시 제어기의 구성이 매우 간단한 장점이 있다. 즉, 미디어 응용 프로그램과 같이 메모리 블록을 차례대로 참조하는 경우에 매우 효과적이다. 그러나 범용 프로그램과 같이 몇 개의 블록을 건너뛰면서 메모리를 참조하는 경우에는 선인출의 효과가 거의 없으며, 캐시 오염(cache pollution)을 유발한다[26, 27].

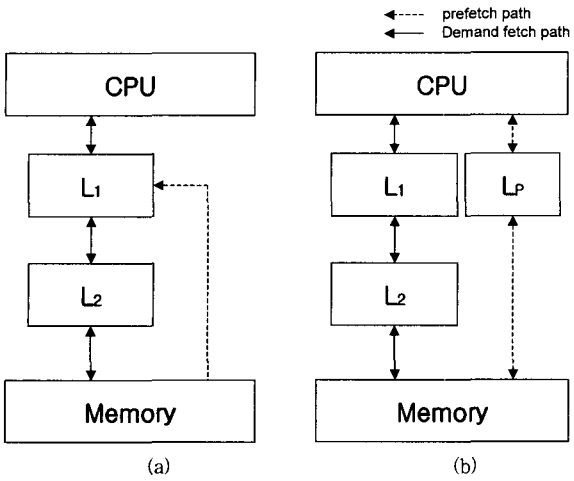
참조예측표(Reference Prediction Table) 기법[4]은 루프와 같이 특정한 주소의 메모리 참조 명령어가 반복해서 수행될 때에 참조하는 메모리 주소가 일정한 간격으로 증가하거나 감소하는 경우에 이를 예측하여 참조할 메모리 블록을 선인출하여 캐시로 적재하는 기법이다. 따라서 참조하는 메모리 주소의 간격이 일정하게 변화하는 경우에는 성능이 우수하나 메모리 참조주소가 불규칙할 경우에는 성능이 떨어지는 단점이 있다.

상관예측표(Correlation Prediction Table) 기법[29]은 순차적으로 참조하는 메모리 주소 간의 상관관계를 토대로 선인출 주소를 결정하는 방법이다. 따라서 이 기법은 메모리 참조 주소의 변화 간격이 일정하지 않아도 된다. 또한, 이 기법에서는 선인출하는 메모리 주소를 여러 개 발생시킬 수 있는 장점이 있으나, 상관표의 크기가 커지는 단점이 있다. 또한 유사한 방법인 마르코프 기법[12]은 순차적인 메모리 참조 주소 쌍의 발생 빈도를 바탕으로 선인출 해야 할 주소를 결정하는 방법이다. 따라서 이 기법은 선인출 주소가 매우 정확한 특징이 있으나 모든 경우를 테이블에 저장하기 위해서는 마르코프 표가 매우 크게 늘어나므로 구현 비용이 비싸고 오버헤드가 큰 단점이 있다.

동적필터 기법[13]은 간단한 선인출 기법, 예를 들면, OBL 기법에서 캐시의 오염이 발생하지 않도록 하여 궁극적으로 선인출의 성공가능성을 높인 기법이다. 즉, 선인출이 되어 캐시에 적재되었지만 이들 중 한번도 참조되지 않고 캐시에서 제거된 블록들은 그 주소를 표에 기록하여 놓고, 동일한 블록의 선인출 요구가 있을 때에 이들 블록을 더 이상 선인출 되지 않도록 한다. 따라서 이 기법은 알고리즘이 간단하고 표의 크기가 작아도 그 효과가 매우 큰 장점이 있다.

### 2.2 캐시 구조와 선인출

선인출 알고리즘의 연구와 함께 선인출의 효과를 높일 수 있는 캐시구조의 연구도 활발히 진행되었다. 전통적인 캐시 구조는 프로세서와 주메모리 간의 $L_1$, $L_2$의 캐시 모듈을 계층적으로 구성하고 있다. 이러한 전통적인 계층구조의 캐시 구조에서 선인출 기법은 $L_2$에서 캐시미스가 일어나는 경우에 캐시제어기는 캐시 미스를 유발한 메모리 참조 명령어가 요구하는 메모리 블록의 요구인출을 우선처리하고 이어서 요구하는 메모리 블록의 참조주소를 토대로 2.1절에서 다룬 선인출 알고리즘에 따라 선인출할 메모리블록의 주소를 예측하여 주메모리로 인출요구를 하게 된다.




(그림 1) 선인출 캐시구조

이 과정에서 선인출을 요구한 메모리 블록을 요구인출 블록에 이어서 사용될 블록이므로 선인출 블록은 $L_2$에 저장하기 보다는 $L_1$에 저장하는 것이 $L_1$의 캐시미스를 방지하여 계산의 연속적인 흐름을 단절시키지 않을 것이다. 그림 1(a)와 같이 선인출 블록은 $L_1$에 적재하도록 하고 요구인출블록은 $L_2$를 거쳐 $L_1$에 저장되도록 하였다[30].

그런데 그림 1(a)구조에서 선인출 블록은 요구인출블록과 비교하여 참조 가능성이 낮다. 참조가능성은 사용하는 선인출 알고리즘에 따라 차이가 있으나 가장 성능이 우수한 동적필터의 경우에도 선인출의 효과는 9%이하이다[13]. 따라서 선인출 기법을 사용하는 경우에는 그만큼 $L_1$의 캐시오염을 증가시켜 결국 캐시의 성능을 떨어뜨리는 원인이 된다. 더군다나 선인출의 효과를 높이기 위하여 선인출 블록을 여러 개 선택하여 $L_1$에 적재하도록 하면 그만큼 선인출의 효과는 높아지나 과도한 선인출로 인해 캐시오염이 급증하여 전체적인 성능이 저하되는 현상을 피할 수 없다.

이러한 문제를 해결하기 위하여 그림 1(b)와 같이 선인출 블록만을 별도로 적재하는 캐시를 두어 $L_1$에서의 캐시 오염을 방지하는 선인출 캐시구조가 제안되었다[12, 23]. 그림 1(b)에서 $L_P$는 선인출 블록을 적재하는 캐시를 $L_1$과 동일한 방법으로 동작하는 구조이다.

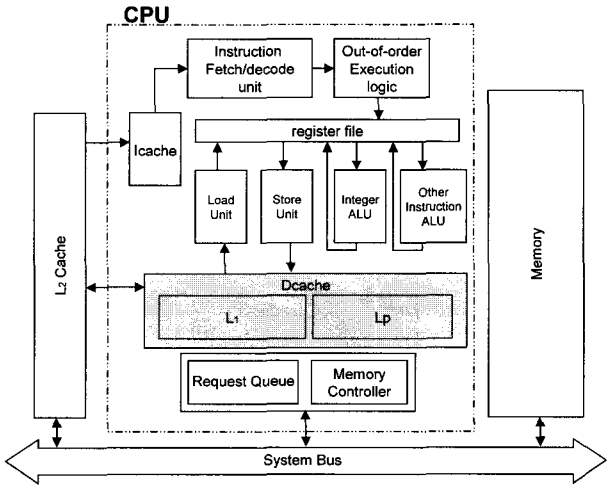
캐시의 크기는 캐시의 효과에 많은 영향을 끼친다. 예를 들어 그림 1(a)에서 $L_1$과 $L_2$의 크기를 키울수록 평균메모리 참조시간이 줄어들게 되어 이로 인하여 응용프로그램의 실행속도를 증가시킬 수 있다. 마찬가지로 그림 1(b)의 경우에도 $L_1$과 $L_P$ 및 $L_2$의 크기를 키우는 것은 전체적으로 프로세서의 성능을 높일 수 있게 한다.

근래에 개발된 프로세서들 [32, 33]의 경우에 $L_2$는 256KB~1MB크기로 프로세서의 외부에 별도로 구성하고 있다. $L_1$ 및 $L_P$의 경우에는 $L_2$보다 참조속도가 빨라야 하므로 칩 내에 포함시켜 제작하며 그 크기는 16KB~64KB 정도이다. 특히 단일칩 프로세서의 경우에는 하나의 칩 내에 계산에 필요한 여러 가지 장치를 포함하고 있으며 $L_1$ 또는 $L_P$를 위한 공간이 제한적이다. 즉, 칩을 구성하는 직접도가 획기적으로 개선되지 않는 한 $L_1$와 $L_P$ 을 위한 공간상의 제약은 피할 수 없을 것으로 보인다. 이러한 관점에서 이 논문에서는 $L_1$과 $L_P$를 구성할 수 있는 칩의 공간이 일정한 경우에 이 면적을 어떠한 비율로 $L_1$과 $L_P$용으로 구분하여 할당하는 것이 가장 효율적인가를 실험을 통하여 결정하였다.

## 3. 목표구조

이 논문에서 고려하는 프로세서 구조는 그림 2와 같이 명령어 인출 및 디코드 유니트(Instruction fetch/decode Unit)와 비순서 실행 제어기(Out-of-Order Execution Logic), 정수연산기(Integer ALU Unit), 기타 명령어연산기(Other Instruction ALU), Load/Store 유니트(Load and Store Unit), 메모리제어기(Memory Controller) 및 레벨 1 캐시(Level 1 Cache)로 구성된 단일칩 프로세서이다.

레벨 1 캐시는 기존의 레벨 1 캐시의 역할을 담당하는 $L_1$과 선인출 데이터를 저장하는 선인출 캐시인 $L_P$로 구성되어 있다. 또한 이 논문에서는 일정한 크기의 레벨 1 캐시 공간을 나누어 $L_1$과 $L_P$로 할당하여 사용하되 그 크기가 일정하다.




(그림 2) 목표 구조

명령어 인출 및 디코드 유니트는 명령어를 매 사이클마다 하나씩 인출하여 디코딩하는 장치이다. 비순서 실행제어기는 Load/Store 명령어를 수행하는 동안 명령어 인출 및 디코드 유니트의 동작을 멈추지 않도록 나중에 인출한 명령어를 먼저 수행하는 데 필요한 역할을 수행한다. 정수연산기와 기타명령어 연산기는 매 사이클마다 하나의 연산 작업을 수행한다. 여기서 기타 명령어 연산기는 분기제어 명령어 등을 처리하는 등을 처리하는 연산기이다.

Load/Store 명령어를 수행하는 유니트는 $L_1$ 또는 $L_P$의

데이터를 레지스터파일에 적재하는 데 1사이클을 필요로 한다. 만일 $L_1$ 또는 $L_P$의 캐시 미스로 인하여 $L_2$로부터 데이터를 참조하는 경우에는 20사이클이 소요된다. 이중에서 $L_2$를 검색하는 데 필요한 시간은 15사이클, 데이터를 복사하는 데 걸리는 시간은 3사이클 소요되며, $L_1$ 또는 $L_P$를 갱신하는 데 걸리는 시간은 1사이클이다.

$L_1$과 $L_P$를 할당할 레벨 1 캐시를 위한 공간은 최신 프로세서의 경우에 16KB~64KB 수준이다. 예를 들어 인텔의 Pentium[33]은 16KB이며 모토로라의 PowerPC60X는 32KB, PC604는 64KB이다[32]. 이 논문에서는 $L_1$과 $L_P$를 위한 레벨 1 캐시의 크기가 16KB, 32KB 및 64KB인 경우를 가정하고 $L_1$과 $L_P$의 크기를 결정하였다. $L_2$의 크기는 인텔의 Pentium 프로세서의 $L_2$캐시크기와 같이 매우 큰 512KB로 결정하였다.

$L_1$과 $L_P$에 적재되는 메모리블록의 크기는 32바이트이며 $L_2$의 페이지 크기는 128바이트로 구성하였다. 즉, $L_2$는 한 번에 128바이트를 메모리 장치로부터 읽어 들일 수 있으며 $L_1$은 $L_2$로부터 32바이트를 읽어 들인다. 즉, 메모리로부터 $L_2$에 적재된 하나의 페이지는 4개의 $L_1$ 메모리 블록을 구성하고 있다. $L_P$는 메모리 블록의 크기가 $L_1$과 같이 32바이트이며 $L_2$를 경유하지 않고 직접메모리로부터 하나의 메모리블록을 읽어 들인다. $L_P$의 메모리블록크기가 $L_2$의 페이지크기에 비하여 작은 것은 메모리 참조시간을 줄여서 선인출 시기를 앞당기기 위함이다. 또한, $L_1$, $L_P$의 캐시구조는 완전사상구조이며 $L_2$는 직접 사상 구조를 채용하는 것으로 간주하였다. $L_2$에서 캐시미스가 발생하면 메모리 제어기는 178사이클 후에 128바이트를 $L_2$에 적재하게 된다.

선인출제어기는 메모리 제어기 내에 포함되어 있으며 $L_2$ 미스시에 요구하는 메모리블록의 주소와 선인출할 메모리 블록의 주소를 큐(Request Queue)에 순서대로 적재한다. 선인출제어기에 적용되는 선인출 알고리즘은 OBL[11], RPT[4], CPT[29], 동적필터 기법[13] 등이 가능하다. 이 논문에서는 RPT기법이 CPT기법의 일종이므로 OBL, CPT, 동적필터 기법의 세 가지를 적용하는 것으로 간주하고 이들을 대상으로 바람직한 선인출 알고리즘이 어떤 것인지를 실험을 통하여 결정하였다. $L_1$과 $L_2$의 교체정책은 기존의 프로세서와 마찬가지로 LRU를 적용하였다. 그러나 $L_P$의 경우에는 $L_1$과 용도가 다른 선인출 데이터용으로 사용되므로 FIFO, LRU, RAMDOM의 세 가지 중의 하나를 실험을 통하여 결정하도록 하였다.

## 4 실험 및 고찰

### 4.1 실험 환경

이 논문에서는 목표로 하는 범용 프로세서용 캐시 구조의 성능 분석을 위하여 범용벤치마크인 SPEC[24]과 미디어 벤치마크[9]에 대한 명령어 추적을 생성하고 명령어 추적 중에서 Load/Store 명령어가 참조하는 메모리 주소를 토대로 선인출 주소를 예측하고 이들 선인출 주소의 메모리 블록을 $L_P$에 적재하는 시뮬레이터를 구현하였다. 벤치마크의 명령어 추적은 ATOM 시뮬레이터[7]를 사용하여 얻었다.

선인출 예측의 정확성은 추적 구동 방식의 시뮬레이터인 Dinero III[8]을 변형하여 CASIM을구현하였다. 즉, 캐시 시뮬레이터는 레벨 1 캐시 크기를 일정하게 유지하고 $L_1$과 $L_P$의 합을 일정하게 유지하고 $L_1$과 $L_P$의 크기를 변화시키면서 벤치마크별, 선인출 기법별로 캐시미스 수, 히트 수, 총 실행 사이클의 수 등을 측정하였다.

$L_1$과 $L_P$의 합은 기존의 프로세서들[32, 33]과 같이 각각 16KB, 32KB, 64KB인 경우로 가정하였다. 또한 이 과정에서 $L_1$과 $L_P$캐시의 크기를 각각 2KB씩 늘이거나 줄이면서 실험하였고, 64KB의 경우에는 $L_1$과 $L_P$캐시의 크기 비를 2의 배수로 변경하면서 실험하였다.

$L_P$의 교체정책으로는 처음 들어온 블록을 가장먼저 제거하는 방법(FIFO), 최근에 가장 참조되지 않는 블록을 교체하는 방법(LRU), 블록을 전혀 관리하지 않고 계산하지 않는 임의에 블록을 교체하는 방법(Random)을 실험하였다. $L_1$과 $L_2$는 기존의 프로세서구조와 마찬가지로 LRU를 사용하는 것을 간주하였다.

이 논문에서는 사용한 벤치마크는 표 1과 같다. 각 벤치마크 프로그램별로 캐시 분할 비율과 선인출 알고리즘 및 교체정책별로 총 실행 사이클을 측정하였다.

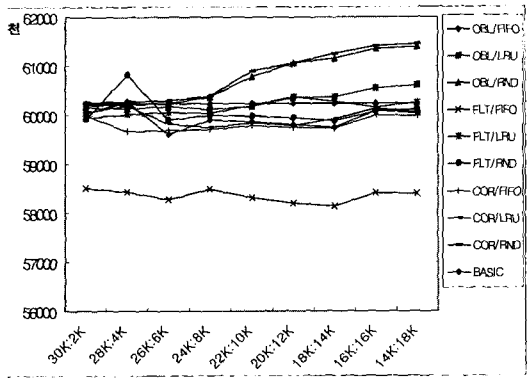
〈표 1〉 실험에 사용된 벤치마크

| 구분 | 벤치마크 | 설명 |
|---|---|---|
| Meida Bench | EPIC | EPIC image Compression |
| | DJPEG | EPIC image Decompression |
| | CJPEG | JPEG image Compression |
| | DJPEG | JPEG image Decompression |
| | MPEG2ENC | MPEG2 Compression |
| | MPEG2DEC | MPEG2 Decompression |
| SPEC Bench | Gcc | C Programming Language Compiler |
| | Gzip | GNU Compression |
| | Perl | Perl Programming Language |
| | TimberWolfSC | Place and Route Simulator |

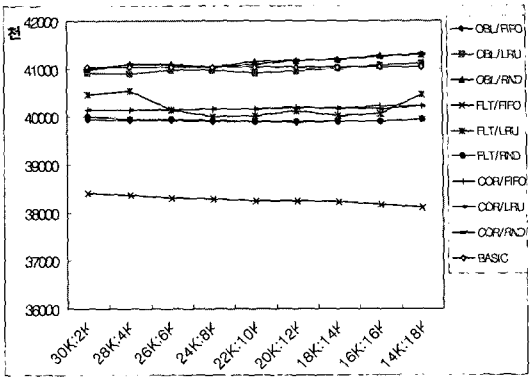
### 4.2 실험 결과 및 분석

벤치마크별로 실험한 결과는 선인출 기능이 없는 기존의 프로세서 구조를 대상으로 $L_2$ 캐시의 크기를 512KB로 고정하고, 레벨 1 캐시의 크기가 각각 16KB, 32KB, 64KB 일 경우에 총 실행 사이클을 측정하였다.

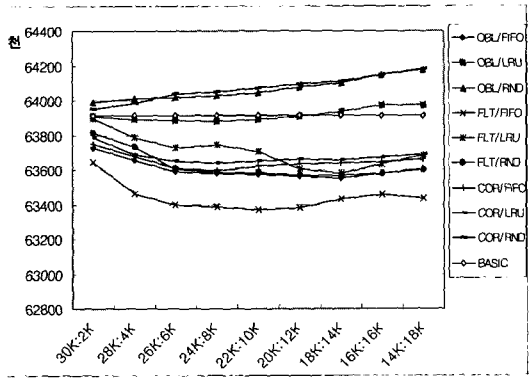
그림 3은 레벨 1 캐시의 크기가 32KB일 때 $L_1$과 $L_P$의 크기를 2KB단위로 변화 시켰을 때 측정한 결과이다. 그림 3



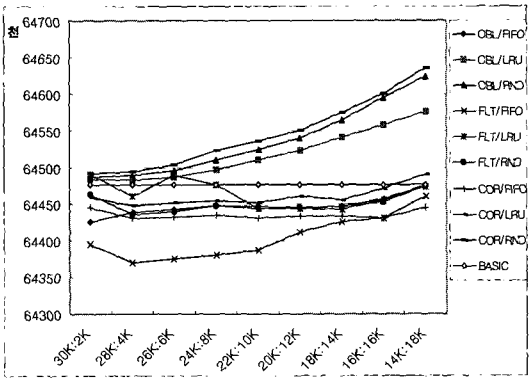

(a) epic



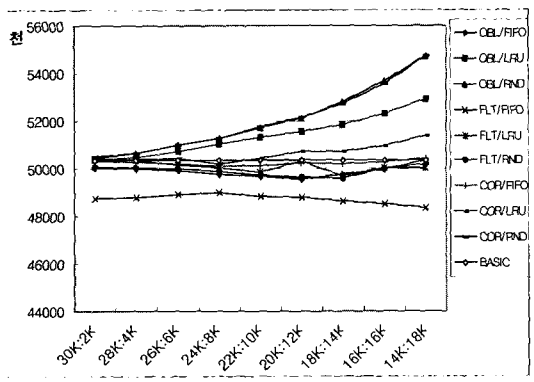

(b) unepic



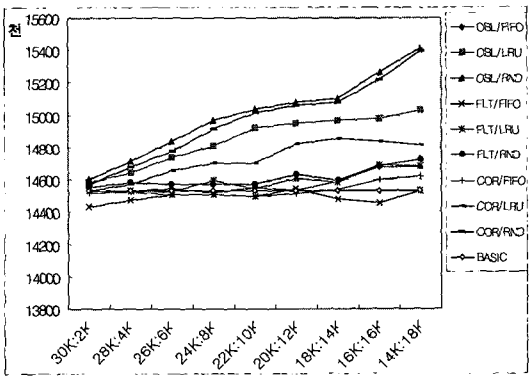

(c) mpeg2enc



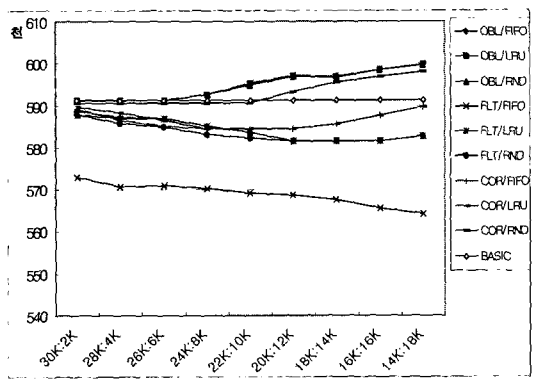

(d) mpeg2dec



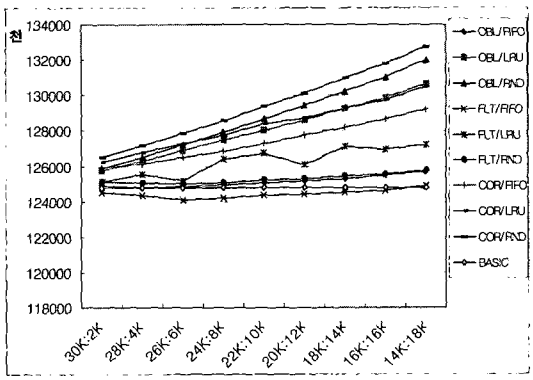

(e) cjpeg



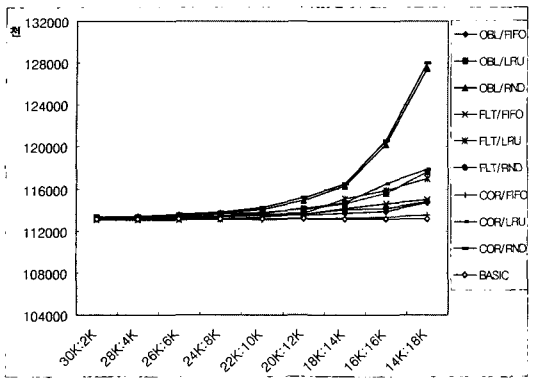

(f) djpeg



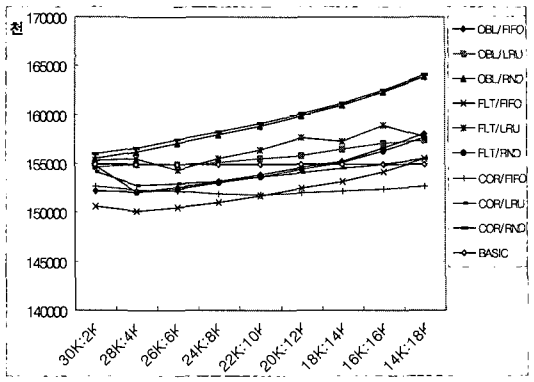

(g) gcc




(h) gzip




(i) perl

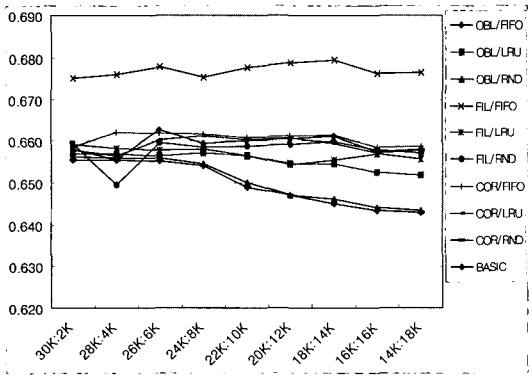



(j) timber

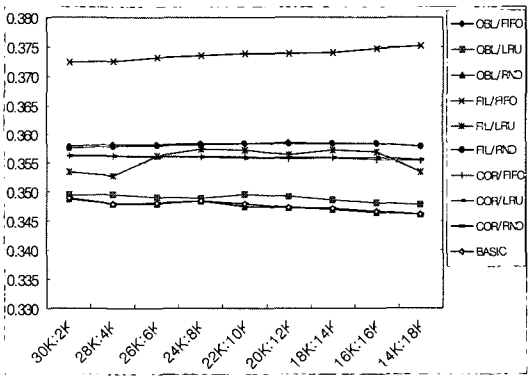
(그림 3) 레벨 1 캐시가 32KB인 경우의 벤치마크별 총 사이클 수

에서 BASIC은 레벨 1 캐시를 모두 $L_1$으로 사용하고 선인출 기법을 적용하지 않을 때이다. 총 실행 사이클이 적은 것이 결국 성능이 좋은 것이므로 epic 등 대부분의 벤치마크에서 선인출 기법으로 동적필터 기법을 사용하고 $L_P$의 교체정책으로 FIFO를 사용하는 것이 좋은 성능이 가장 우수하였음을 보여주었다. 그러나 perl과 timber, mpeg2dec의 경우에는 동적필터 기법 대신 상관표 기법을 사용하는 경우가 성능이 좋아지는 경우도 있었다. 이는 이들 벤치마크의 특성이 데이터의 인출 규칙성이 떨어지기 때문인 것으로 판단된다.

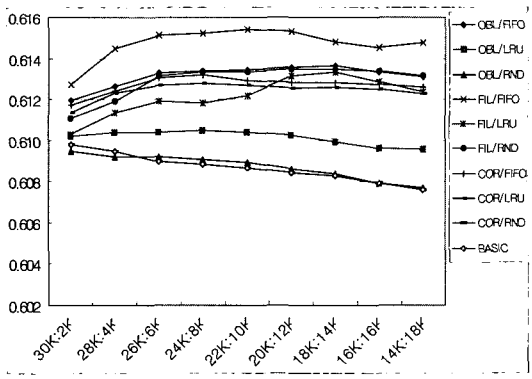
그림 4는 레벨 1 캐시의 크기가 32KB인 경우에 단위 사이클 당 실행된 명령어수(Instruction Per Cycle)를 측정한 것으로 그림 3과 동일한 패턴을 보여준다. 즉, 그림 3에서와 같이 대부분의 벤치마크에서 동적필터 기법과 $L_P$의 캐시교체 정책으로 FIFO를 사용하는 것이 IPC가 가장 컷으나 perl과 timber 및 mpeg2dec에서는 상관표 기법을 사용하는 것이 IPC가 큰 것을 보여주었다. 실험결과를 토대로 $L_P$의 크기와 선인출 전략 및 캐시교체 정책을 결정하기 위하여 모든 벤치마크에 대한 IPC의 평균치를 계산하여 레벨 1 캐시의 크기별로 IPC평균치가 가장 큰 경우를 선정하고 이때 사용된 선인출 전략과 캐시교체 정책을 해당 레벨 1 캐시의 크기에 따른 캐시 분할비, 선인출 전략 및 캐시교체 정책으로 결정하였다.



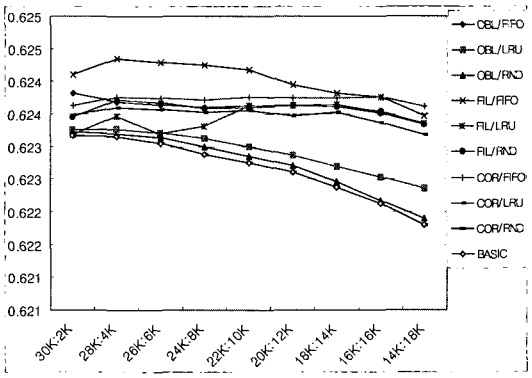

(a) epic



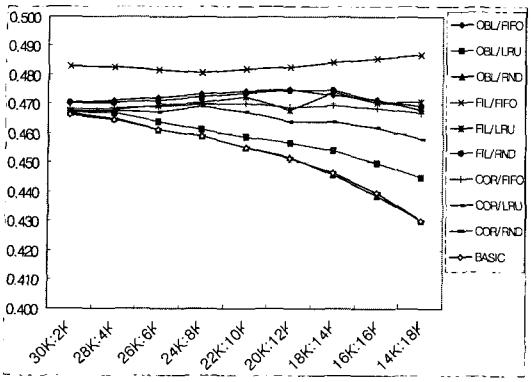

(b) unepic



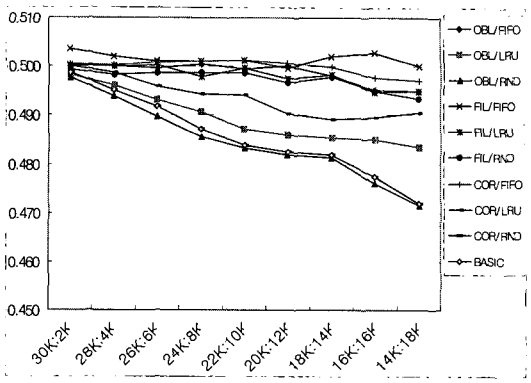

(c) mpeg2enc



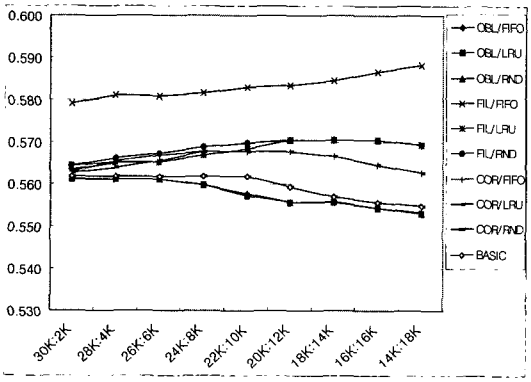

(d) mpeg2dec



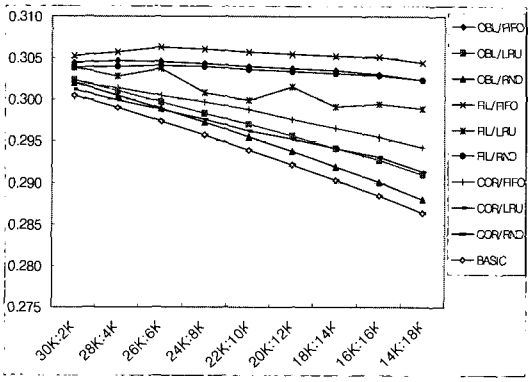

(e) cjpeg



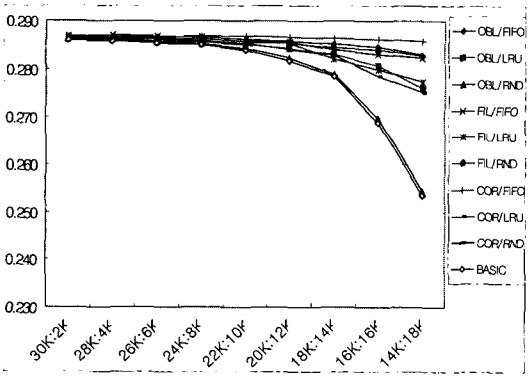

(f) djpeg



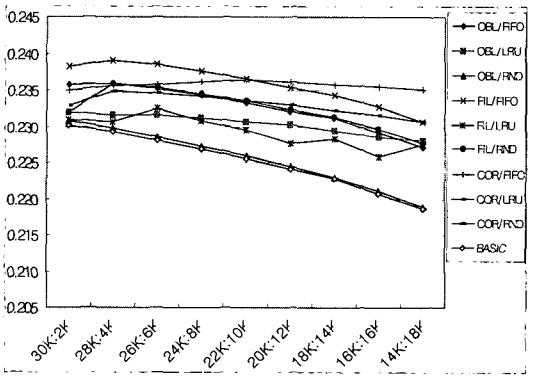

(g) gcc




(h) gzip




(i) perl



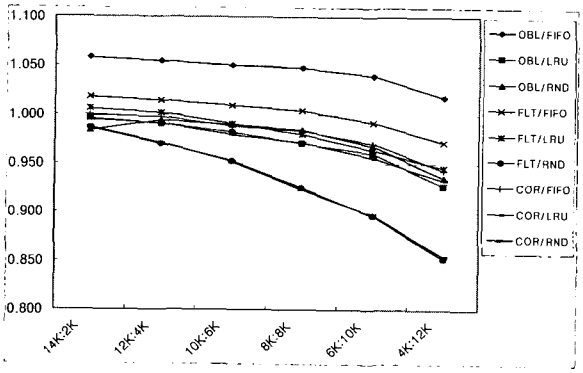

(j) timber

(그림 4) 레벨 1 캐시가 32KB인 경우의 벤치마크별 IPC

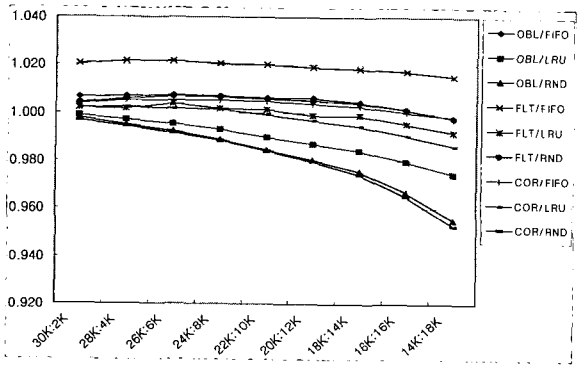
그림 5는 레벨 1 캐시의 크기가 각각 16KB 32KB 및 64KB인 경우에 $L_1$과 $L_P$의 크기를 변경하면서 선인출 전략과 캐시교체 전략별로 계산한 것이다. 그림 5(a)는 레벨 1 캐시의 크기가 16KB인 경우에 $L_1$과 $L_P$를 각각 14KB와 2KB로 구분하고 OBL과 FIFO를 선인출 기법과 $L_P$의 캐시 교체 정책으로 사용할 경우에 IPC가 가장 큰 것을 알 수 있었다.

그림 5(a)에서 $L_P$의 크기가 클수록 IPC가 낮아지는 경향을 보인다. 이는 선인출 캐시를 크게 하는 것이 성능향상에 별로 도움이 되지 않으며 요구인출캐시의 크기는 일정수를 유지해야 함을 의미한다. 그림 5(b)는 레벨 1 캐시의 크기가 32KB인 경우에 $L_1$과 $L_P$를 각각 28KB와 4KB로 구분하고 동적필터기법과 FIFO를 선인출 기법과 $L_P$의 캐시교체정책으로 사용하는 경우에 IPC가 가장 큰 것을 알 수 있다. 또한 $L_P$의 크기가 증가함에 따라 IPC의 값이 점차 낮아진다. 이는 레벨 1 캐시의 크기가 16KB인 경우와 비슷하나 낮아지는 비율이 완만하다. 이는 요구인출캐시의 크기가 일정크기 이상 유지되기 때문인 것으로 판단된다. 그림 5(c)는 레벨 1 캐시의 크기가 64KB인 경우에 $L_1$과 $L_P$를 각각 48KB와 16KB로 구분하고 그림 5(b)와 마찬가지로 동적필터 기

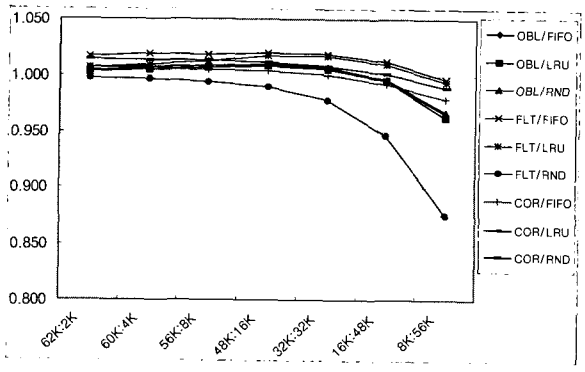
법과 FIFO를 선인출 기법과 $L_P$의 캐시교체정책으로 사용하는 경우에 IPC가 가장 좋다는 큰 것을 알 수 있다. 또한 $L_P$의 크기가 변화함에 따라 IPC값이 작아지는 경향은 레벨 1 캐시가 16KB이나 32KB인 경우와 동일하다. 그러나 64KB인 경우에는 그 정도가 완만하며, 선인출 기법 캐시교체 정책 별로 차이가 적어짐을 보여준다. 이는 요구인출 캐시와 선인출 캐시의 크기가 충분하기 때문인 것으로 판단된다.




(a) 16KB




(b) 32KB




(c) 64KB

(그림 5) 레벨 1 캐시 크기별 IPC 향상평균치

## 5. 결 론

컴퓨터의 응용 분야가 넓어지면서 다양한 미디어 응용 프로그램을 처리해야한다. 그러므로 미디어 응용프로그램을 고려하여 범용 프로세서가 설계되어야 한다.

이 논문에서는 분할된 캐시 구조를 이용하여 범용 프로그램과 미디어 프로그램 모두가 성능이 극대화될 수 있는 캐시구조를 제안하였다. 제안한 캐시구조는 레벨 1 캐시를 요구인출캐시와 선인출 캐시로 구분하고, 요구인출 데이터와 선인출 데이터를 각각 별개의 캐시($L_1$와 $L_P$)에 적재하는 구조이다.

미디어와 범용 벤치마크들을 대상으로 실험한 결과 이 논문에서 제안하는 캐시구조가 성능 개선의 효과가 있음을 알 수 있었다. 즉, 레벨 1 캐시의 크기를 16KB로 구성할 수 있는 경우에 요구인출 캐시와 선인출 캐시를 각각 12KB와 4KB로 분할하고 선인출 기법으로 OBL을, 선인출 캐시의 교체정책으로 FIFO를 사용하는 것이 가장 효과적이다. 이 구조는 $L_1$캐시를 분할하지 않고, 선인출 기법을 적용하지 않았을 때와 비교하면 범용 벤치마크의 경우에 4.9%, 미디어 벤치마크에 대하여 6.7%의 성능향상 효과가 있었다.

레벨 1 캐시의 크기를 32KB 로 구성할 수 있는 경우에는 요구인출 캐시와 선인출 캐시를 각각 28KB와 4KB로 분할하고 선인출 기법으로는 동적필터 기법을, 그리고 선인출 캐시의 교체정책으로 FIFO를 사용하는 것이 가장 효과적이었다. 이 구조는 기존의 구조와 비교하여 범용 벤치마크의 경우에 1.8%, 미디어 벤치마크에 대하여 2.5%의 성능향상 효과가 있었다.

마찬가지로 레벨 1 캐시의 크기를 64KB로 구성하는 경우에 요구인출 캐시와 선인출 캐시를 각각 48KB와 16KB로 분할하고 선인출 기법으로는 동적필터 기법을, 그리고 선인출 캐시의 교체정책으로 FIFO를 사용하는 것이 가장 효과적이었다. 이 구조는 기존의 구조에 비하여 범용 벤치마크의 경우에 2.2%, 미디어 벤치마크에 대하여 1.8%의 성능향상 효과가 있었다. 따라서 제안된 선인출 기법과 분할 캐시 구조를 사용하면 범용 벤치마크뿐 아니라 미디어 응용에서도 빠르게 동작하는 프로세서의 설계가 가능할 것이다.

## 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. Bear and W. H. Wang, "Architectural Choices for Multi-level Cache Hierachies," *Proceedings of 16th international Conference on Parallel Processing*, pp.258-256, 1987.

[3] S. P. VanderWiel and D.J. Lilja, When Caches Aren't Enough: Data Prefetching Techniques. *IEEE Computers*, 23-30, May 1995.

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

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

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

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

[9] 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.

[10] F. Harmsze, A. Timmer and J. van Meerbergen, Memory Arbitration and Cache Management in Stream-Based Systems, Proceedings of the Date 2000, pp.257-262, March 2000.

[11] A. J. Smith, "Cache Memories", *ACM Computing Surveys*, Vol. 14, pp.473-530, September 1982.

[12] D. Joseph and D. Grunwald, "Prefetching Using Markov Predictors," Proceedings 24th Inl. Symp. *Computer Architecture*, pp.252-263, June 1997.

[13] X. Zhang, H. S. Lee, A hardware-based cache pollution filtering mechanism for aggressive prefetches, *Proceedings. 2003 International Conference on Parallel Processing* , pp.286 - 293, 6-9, October 2003.

[14] A. Leung, K. Palem and C. Ungureanu, *Run-time versus Compile-time Instruction Scheduling in Superscalar (RISC) Processors: Performance and Tradeoffs, Technical report 699*, New York University, July 1995.

[15] C. Basoglu, W. Lee and J. S. O'Donnell, "The MAP1000A VLIW mediaprocessor," *IEEE Micro*, Vol. 20, No. 2, pp.48-59, March 2000.

[16] R. B. Lee, "Subword Parallelism with MAX-2," *IEEE Micro*, Vol. 16, No. 4, pp.51-59, August, 1996.

[17] C. Young, N. Gloy and M. D. Smith, "A comparative analysis of schemes for correlated branch prediction," *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp.22-24, June 1995.

[18] H. S. Stone, *High-Performance Computer Architecture*, Addison Wesley, 1993.

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

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

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

[22] F. Harmsze, A. Timmer and J. van Meerbergen, "Memory Arbitration and Cache Management in Stream-Based Systems," *Proceedings of the DATE 2000*, pp.257-262, March 2000.

[23] T. Horel and G. Lauterbach, "UltraSPARC-III : Designing Third-generation 64-bit Performance," *IEEE Micro*, Vol. 19, No. 3, pp.73-85, May 1999.

[25] J. Hennessy, D. Citron, D. Patterson and G. Sohi, "The use and abuse of SPEC: An ISCA panel," *IEEE Micro*, Vol. 23 , pp.73-77, July-August 2003.

[26] H. J. Moon, J. N. Jeon, S. I. Kim, "Design of A Media Processor Equipped with Dual Cache," *Journal of KISS*, Vol. 29, No. 9, pp.573-581, October 2002.

[27] H. J. Moon, *A Cache Managing Strategy for Fast Media Data Access*, Ph. D. *Thesis,* Dept. of Computer Science, Chungbuk National University, February 2003.

[28] N. B. Gaddis, J. R. Butler, A. Kumar, W. J. Queen, A 56-entry instruction reorder buffer, Solid-State Circuits Conference, *Digest of Technical Papers. 43rd ISSCC.*, 1996 IEEE International , pp.212-213, 447, February 1996.

[29] Y. Solihin, J. Lee, J. Torrellas, "Correlation prefetching with a user-level memory thread," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 14, pp.563-580, June 2003.

[30] N. Mitchell, "Philips TriMedia: A Digital Convergence Platform," *Wescon'97*, pp.56-60, 1997.

[31] Z. Hu, M. Martonosi and S. Kaxiras, "TCP: Tag Correlating Prefetchers," *Proceedings of 9th International Symposium on High-Performance Computer Architecture*, pp.137-147, 2003. .

[32] M. Denamn, "PowerPC 604," *Hot Chips VI*, pp.193-200, 1994.

[33] *Pentium Processor User's Manual*, Vol.1 Pentium Processor Databook, Intel, 1993.

주 영 관

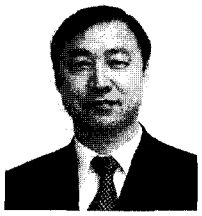
e-mail : rainbow@chungbuk.ac.kr

1999년 2월 청주대학교 컴퓨터정보공학과(공학사)

2004년 2월 충북대학교 전자계산학과(이학석사)

2004년 3월~현재 충북대학교 전자계산학과 박사과정

관심분야 : 데이터 선인출, 병렬 컴퓨터구조, 모바일 컴퓨팅, 이기종 분산처리



김 석 일

e-mail : ksi@chungbuk.ac.kr

1975년 서울대학교 전기공학과(공학사)

1975년~1990년 국방과학연구소 선임연구원

1989년 미국 North Carolina주립대학(공학박사)

1990년~2004년 충북대학교 전기전자컴퓨터공학부 학부장

2004년 9월~현재 유비쿼터스바이오정보기술연구센터 연구소장

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