데이타 배열을 사용하는 병렬 프로그램에서 그레인 크기를 이용한 데이타 선인출 기법

(A Data Prefetching Scheme Exploiting the Grain Size in Parallel Programs using Data Arrays)

정 인 범 † 이 준 원 ††

(In-Bum Jung) (Joon-Won Lee)

요 약 데이타 선인출 방법은 데이타 참조와 프로세서 계산의 중첩을 이용하여 주메모리 접근 지연시간을 줄여주는 효과적인 방법이다. 그러나 선인출된 데이타가 캐쉬 메모리에 있는 다른 유용한 데이타들을 대체시키거나 또한 선인출된 데이타가 사용되지 않는 무익한 선인출일 경우 프로그램의 성능은 저하된다. 이러한 현상은 향후 사용되는 데이타들에 대한 정확한 예측이 부족하므로 발생된다. 병렬 프로그램이 계산을 위하여 데이타 배열들을 사용할 때 그레인 크기는 향후 사용되는 데이타 지역의 범위를 나타내므로 데이타 선인출을 위한 유용한 정보이다. 이런 정보를 기반으로 본 논문에서는 병렬 프로그램의 그레인 크기를 이용한 새로운 데이타 선인출 방법을 제안한다. 모의시험에서 제안된 선인출 방법은 기존의 선인출 방법들보다 버스 트랜잭션을 감소시킬 뿐만 아니라 유용한 선인출의 증가로 시험된 병렬 프로그램들의 성능을 향상시킨다.

Abstract The data prefetching scheme is an effective technique to reduce the main memory access latency by exploiting the overlap of processor computations with data accesses. However, if the prefetched data replicate the useful existing data in the cache memory and they are not being used in computations. performances of programs are aggravated. This phenomenon results from the lack of correct predictions for data being used in the future. When parallel programs exploit the data arrays for computations, the grain size is useful information for data prefetching scheme because it implies the range of data using in computations. Based on this information, we suggest a new data prefetching scheme exploited by the grain size of the parallel program. Simulation results show that the suggested prefetching scheme improves the performance of the simulated parallel programs due to the reduction of bus transactions as well as useful prefetching operations.

1. 서 론

캐쉬 메모리는 빠른 성능의 프로세서와 상대적으로 느린 메모리의 성능 차이를 완화하기 위하여 사용되어진다. 따라서 캐쉬 실패는 주 메모리(main memory)에 접근하기 위한 지연 시간이라는 불이익을 발생하며 프로그램의 수행시간에 커다란 영향을 미친다. 이러한 캐쉬 메모리의 성능을 향상시키기 위한 방법들 중 캐쉬 선인출(prefetching) 기법이 있다. 캐쉬 선인출은 데이타의 참조와 프로세서의 계산 시간을 중첩하므로 주 메모리 접근시간을 줄이고자 하는 방법이다. 특히 공유 버스를 사용하는 다중 프로세서 시스템에서는 버스의 버스트(burst) 전송 모드를 사용할 경우 프로세서가 임의의 데이타를 메모리로부터 인출할 때 순차적 주소를 갖는 이웃 데이타들을 선인출하여 전송하여도 전송에 따르는 커다란 오버헤드 없이 캐쉬에 적재시킬 수 있다. 이렇게 선인출된 데이타에 대한 참조가 발생할 경우 캐쉬 적중(cache hit)이 되고 따라서 프로그램의 성능이 향상된

· 본 논문은 한국과학재단 특정 기초연구과제(과제번호 : 96-0101-05-01-3)의 지원을 받았습니다.

† 학생회원 : 한국과학기술원 전산학과
jib@camars.kaist.ac.kr
†† 종신회원 : 한국과학기술원 전산학과 교수
joon@cs.kaist.ac.kr

논문접수 : 1999년 1월 15일
심사완료 : 1999년 10월 25일

다. 그러나 제한된 크기를 갖는 캐쉬 메모리의 특징상 캐쉬에 선인출되는 데이타들은 이미 캐쉬에 적재되어있는 다른 유용한 데이타들을 대체(replacement)시킬 수 있고 또한 선인출된 데이타가 예상과는 달리 사용되지 않는 경우 캐쉬 메모리의 공간만을 차지하는 결과가 발생된다. 이러한 문제점들을 해결하기 위한 많은 선인출 방법들이 제안되었다. 그러나 미래를 예측하는 것은 어려운 일이므로 무익한 데이타를 완전히 제거한 선인출 방법은 제시되기 어려웠다.

공유메모리 다중 프로세서는 프로그램의 용이성 때문에 병렬 프로그램의 수행에 많이 사용되는 시스템이다. 이러한 시스템에서 프로세서들은 병렬 프로그램에서 생성된 프로세스들에게 할당되며 병렬 프로그램의 성김도(granularity) 정책에 따라 쪼개어진 그레인들을 처리하게 된다. 병렬 프로그램의 성김도는 프로그램의 총 계산량을 병렬처리에 참여하는 프로세서들에게 배분하는 단위이다. 이러한 성김도는 병렬 프로그램의 특성에 따라서 조밀(fine) 그레인에서 성긴(coarse) 그레인 범위에서 선택되어진다. 특히 프로그램이 실행 시간동안 고정된 그레인 크기를 사용하는 경우 그레인들을 구성하는 데이타 배열들의 주소 범위를 쉽게 예측할 수 있다. 본 논문에서는 이러한 예측 가능한 그레인들의 주소 범위를 이용하여 병렬 프로그램에서 사용되는 주요 배열들에 대한 효과적 선인출 방법을 제안한다. 제안된 방법은 선인출 후 사용되지 않는 무익한(useless) 데이타들을 감소시키므로 캐쉬 메모리의 공간을 낭비하지 않을 뿐만 아니라 버스 트랜잭션을 감소시키므로 프로그램의 성능을 향상시킨다.

본 논문의 구성은 다음과 같다. 2장에서는 본 논문과 관련된 연구에 대하여 설명한다. 3장에서는 본 논문에서 제안된 선인출 방법을 설명한다. 4장에서는 모의 시험 환경에 대하여 설명한다. 5장에서는 모의 시험을 통한 성능 평가를 한다. 마지막으로 6장에서 본 논문의 결론을 맺는다.

2. 관련연구

선인출에 대한 많은 연구가 하드웨어적 방법 또는 소프트웨어적 방법으로 이루어져왔다. 하드웨어적 방법은 규칙적인 메모리 접근 형태가 발견될 때 선인출을 시작하는 것으로 규칙적인 메모리 접근 형태를 발견하기 위한 복잡한 하드웨어가 필요하다. Smith[1]은 OBL(One Block Lookahead)이라는 하드웨어 선인출 방식을 제안했다. 이 연구에서는 3가지의 OBL 방식을 제안하였다. 첫째, 항상 선인출(always prefetching) 방식으로 메모리 참조가 발생될 때마다 이웃한 메모리 블록을 읽는 것이다. 이 방법은 무조건적 선인출에 의한 무익한 데이타 선인출이 증가되기 쉽다. 둘째, 태그 선인출(tagged prefetching) 방식으로 참조되는 메모리 블록이 처음 참조되는 경우에만 다음 블록을 선인출 하는 방법이다. 이렇게 하기 위해서는 캐쉬 블록당 상태를 나타내는 추가적 비트들이 필요하다. 세째, 캐쉬 실패시 선인출(prefetch on misses) 방식으로 캐쉬 실패가 발생되었을 때에만 다음 블록을 선인출하는 방법이다. 이 방법은 캐쉬 실패가 발생할 때 하나의 블록만을 선인출 하므로 순차적인 주소를 갖는 메모리 블록들을 참조할 때 비효율적이다. 이러한 OBL 방식의 확장으로 Jouppi[2]는 FIFO 스트림(stream) 버퍼들을 사용한 연속적 버퍼들을 선인출하는 방법을 제안했다. 제한된 FIFO 큐가 캐쉬 실패가 발생한 주소부터 순차적인 주소에 있는 메모리 블록들을 적재하는 방식이다. 이 방식은 OBL 방식의 확장이지만 FIFO 큐 크기에 의한 제한적 공간 국부성만을 이용할 수밖에 없으며 커다란 스트라이드(stride)단위로 참조되는 데이타 접근에 대해서는 적합하지 않다. Baer[3,4]는 메모리 참조의 스트라이드 폭을 이용하는 선인출 기법을 제안했다. 이 방법은 참조 예견 테이블(reference prediction table)이라는 하드웨어 테이블을 사용하여 참조되는 명령어들 및 데이타 주소들을 저장한다. 이 테이블에서 연속적으로 발생되는 주소 참조들을 비교 분석하여 스트라이드 형태의 메모리 참조가 발견될 때 이 정보를 미래에 사용될 수 있는 데이타들에 대한 선인출에 이용하는 방법이다. 그러나 이 방법은 불규칙적으로 발생되는 스트라이드 형태의 메모리 참조에서는 좋은 성능을 나타내지 않는다. Fu[5,6] 는 스트라이드 형태의 데이타에 대한 다른 방법의 선인출을 제안했다. 이 연구에서는 이전 메모리 참조들에 대한 커다란 이력 파일(history file)을 사용하여 스트라이드 폭에 대한 예측을 하여 선인출을 수행한다. 그러나 이 방법도 정확한 스트라이드 폭을 예측하기가 어렵고 이력 파일을 관리하기 위한 하드웨어 공간이 크며 불필요한 블록들이나 불규칙적인 메모리 참조에서는 불필요한 선인출을 수행하는 문제점을 가지고있다. Dahlgren[7]은 선인출된 데이타가 실제로 사용되는지를 판단해줄 수 있는 하드웨어에 근거한 적응 순차적 선인출(adaptive sequential prefetching)을 제안하였다. 이 방식은 캐쉬 실패가 발생했을 때 선인출이 일어나며 선인출된 데이타가 실제 사용될 경우 선인출되는 양을 늘리며 반면에 선인출된 데이타가 사용되지 않은 경우 선인출량을 감소 시켜가는 적응적 방식이다. 그러나 이 방법

은 캐쉬 실패가 발생될 때만 선인출을 수행할 수 있으며 프로그램들의 특성을 사용하지 않았고 적응적 방식을 사용하더라도 무익한 데이타 선인출을 제거하지는 못했다.

소프트웨어적 선인출 방법은 프로그램에 대한 분석을 통하여 프로그램 코드에 선인출 코드를 집어넣는 방식으로 프로그램의 크기를 증가시킨다. Porterfield[8]은 루프 구문을 기반으로 하는 프로그램에서 현재의 루프에서 다음 루프에서 사용되는 데이타들을 미리 읽는 선인출 방법으로 프로그램의 성능을 개선하였다. 그러나 모든 데이타를 미리 선인출 하므로 현재 사용되는 데이타들과의 주소 간섭에 의한 캐쉬 실패가 발생되며 이로 인한 유익한 데이타들이 대체되는 현상이 발생된다. Gornish[9]는 프로그램의 데이타 종속성과 제어 종속성에 근거하여 루프가 수행되기 이전에 데이타 배열들을 선인출할 수 있는 지점을 발견하는 알고리즘을 제안했다. 이 알고리즘의 주요 기능은 주 메모리에서 캐쉬 메모리로 데이타 블록이 이동되는 지점을 찾아내어 선인출 명령어를 집어넣는 것이다. 그러나 이 방법은 알고리즘을 프로그램 바인딩 시점에 적용하므로 보다 많은 선인출 기회를 얻지 못하게 했다. Mowry[10,11]은 프로그램에 대한 비 바인딩(nonbinding) 소프트웨어 선인출을 제안하여 바인딩 방식보다 선인출을 위한 제약점들을 줄였다. 이러한 방식은 선인출 데이타들에 대한 일관성 유지를 하드웨어가 한다는 가정을 가지고있다. 또한 프로그램의 국부성(locality) 분석과 루프 변형을 통하여 선인출 명령어를 삽입하는 컴파일러 알고리즘을 제안했다. 그러나 컴파일러가 복잡한 메모리 접근 형태를 나타내는 모든 프로그램들의 국부성을 분석하여 자동적으로 선인출 명령을 생성하는 것은 어려운 일이다.

선인출은 미래에 사용될 가능성이 있는 데이타들을 네트워크의 전송 대역을 이용하여 미리 캐쉬 메모리에 가져오므로 캐쉬의 성능을 높이는 것이 목적이다. 그러나 선인출에 관련된 기존 연구들에서는 실제로 사용되지 않는 데이타들이 선인출 하거나 캐쉬에 있는 유익한 기존 데이타들이 선인출된 데이타에 의해 대체되는 무익한 선인출을 완전히 제거할 수 없다. 이것은 미래를 정확하게 예측하기가 어려운 측면도 있지만 기존 방법들이 프로그램내의 모든 데이타들에 대한 선인출을 프로그래머와 무관하게 시도하므로 오히려 무익한 선인출을 유발시킴에 기인된다. 본 논문에서는 병렬 프로그램을 구성하는 프로그래머에게 데이타 선인출에 대한 기능성(functionality)을 제공하는 것에 관점을 맞추고있다. 즉, 프로그래머가 데이타 배열들을 선별 지정하여 하드웨어적 선인출을 요구할 수 있게 하고 또한 선인출의 범위를 병렬 프로그램의 그레인 크기를 이용하여 지정하므로 기존의 선인출시 발생되는 무익한 데이타들의 선인출을 줄이고자 한다.

3. 성김도에 기반한 순차적 데이타 선인출 기법

과학 계산용 병렬 프로그램들에서 사용되는 중요한 자료 구조들은 데이타 배열 구조들이다. 일반적으로 이런 배열 구조는 시스템 호출을 통하여 연속적인 메모리 할당을 운영체제로부터 제공 받으며 할당된 데이타 배열은 병렬 프로그램의 성김도 정책에 따라서 그레인들로 쪼개어져 각각의 프로세서들에게 할당된다. 프로세서들에게 할당된 그레인들에 대한 참조는 순차적인 주소 참조를 발생하게 되므로 선인출의 효과를 얻을 수 있는 부분이다. 또한 프로세서에게 할당되는 그레인 크기 및 그레인의 주소 공간은 프로그래머에게 알려질 수 있다. 따라서 본 논문에서는 프로그래머에게 알려지는 선인출을 위한 정보들을 바탕으로 무익한 데이타들의 선인출을 방지하는 효과적인 선인출 기법을 제안하고자 한다.

다음은 본 논문에서 제안하는 선인출 방법을 설명한다. 메모리 할당하는 시스템 호출을 운영체제는 2가지로 구별하여 사용자에게 제공한다. 하나는 기존에 사용하는 메모리 할당 시스템 호출이며 또 다른 하나는 선인출을 발생하는 메모리를 할당하는 시스템 호출이다. 메모리 할당 방식을 이렇게 2가지로 구분한 이유는 선인출 효과는 선인출 후 실제로 사용되는 데이타 배열을 순차적 주소가 보장되는 한 최대한 크게 선인출하는 것에 좌우되기 때문이다. 즉, 병렬 프로그램의 성김도 정책이 적용되는 프로그램의 주요 데이타 배열들이 할당된 메모리에 대해서만 선인출을 수행하고 그 이외의 동기화 변수, 전역 변수들은 사용되는 메모리 블록들의 순차적 주소 공간이 짧으므로 다중 워드로 구성되는 캐쉬 블록에 의해서도 충분하게 선인출 효과를 얻을 수 있기 때문이다. 다음은 이러한 2가지로 분류된 메모리 할당을 위한 시스템 호출들의 예를 보여주고 있다.

```
/* 예-1 : 일반적인 공유 메모리 할당용 시스템 호출 */
                    char *ip;
                    int  size;
                    ip = (char *) share_malloc(size);
/* 예-2 : 선인출을 위한 공유 메모리 할당용 시스템 호출 */
                    char *ip;
                    int  size, grain_size;
                    ip = (char *) prefetch_share_
malloc(size, grain_size);
```

예-2는 선인출을 위한 공유 메모리 할당 시스템 호출을 나타낸다. 이 예에서 정수 변수 grain_size는 하나의 프로세서에게 할당된 그레인 크기를 의미한다. 운영체제는 이 그레인 크기를 바탕으로 순차적 주소가 보장되는 최대의 캐쉬 블록 개수를 계산하여 선인출 정도로 사용한다. 또한 운영체제는 할당된 주소의 범위와 계산된 선인출 크기를 특정한 하드웨어에 저장한다. 이때 필요한 하드웨어로는 선인출 주소 관리 테이블(prefetching address management table)이 필요하다. 테이블의 한 항목은 하나의 데이타 배열에 대한 선인출 시작 주소, 종료 주소, 선인출 정도의 항목들로 구성된다. 이 테이블의 각 항목들은 하드웨어 레지스터 구조로 간단히 설계되어진다. 일반적으로 프로그램들에서 대용량의 선인출이 필요한 배열들은 많지 않다. 따라서 테이블의 크기가 작아도 효과적 선인출에 기여할 수 있다. 즉, 소 용량의 데이타 선인출은 다중 워드로 구성된 기존 하드웨어 캐쉬 블록으로도 충분히 이루어진다. 그림1은 이러한 하드웨어 구조를 하나의 프로세서 노드 측면에서 나타내고있다. 필요한 하드웨어적 공간은 테이블의 크기에 따라서 좌우된다. 그러나 많은 병렬 프로그램들이 사용하는 주요 데이타 배열들은 많지 않다. 만약 테이블 항목보다 더욱 많은 데이타 배열들을 사용자가 선인출 메모리로 요구할 때에는 운영체제는 선인출이 수행되지 않는 일반적인 공유 메모리를 할당한다. 이는 C 프로그램에서 레지스터 변수를 사용하여도 시스템에서 사용할 수 있는 레지스터 변수 개수를 초과할 경우 메모리에 변수를 할당하는 개념과 같다. 그림1의 회로에서 캐쉬 실패가 발생할 때 비교/선택 회로는 선인출 배열 주소

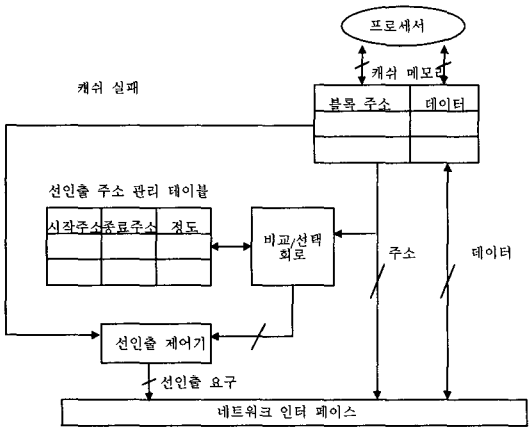

그림 1 선인출 기능이 추가된 프로세서 노드의 구조

테이블을 탐색하여 캐쉬 실패가 발생된 주소가 선인출을 수행할 수 있는 주소인지를 판단한다. 주소가 선인출 주소 범위 안에 있을 경우 선인출 정도(degree)값에 해당되는 메모리 블록 선인출을 선인출 제어기(prefetching controller)에게 요구한다. 선인출 제어기는 네트워크 인터페이스를 통한 후 메모리로부터 데이타들을 선인출하여 캐쉬 메모리에 적재시킨다.

4. 모의 시험 환경

4.1 공유 메모리 다중 프로세서 환경

모의 시험 환경은 공유 버스를 사용하는 공유 메모리 기반 다중 프로세서를 가정하였다. 병렬 프로그램들을 수행하는 기능 모의시험기로는 MINT(Mips INTerpreter)[12]를 사용하였고 기능 모의시험기에서 발생된 메모리 참조 관련 사건들은 바탕으로 공유 버스를 사용하는 다중 프로세서 시험기를 구성하였다. 프로세서 개수는 8개를 가정하였고 각 프로세서들은 메모리 참조를 제외하고는 하나의 싸이클(cycle)당 하나의 명령어를 수행하는 동일한 캐쉬 메모리 크기를 갖는 리스크(RISC) 프로세서로 가정하였다. 프로세서의 클록률(clock rate)은 250 Mhz로 메모리 지연시간은 약 80 ns(nano second), 공유 버스의 대역폭은 128 bits이고 버스트 모드로 전송될 때 메모리의 연속적 주소에 저장된 데이타들을 요청한 만큼 한번에 전송할 수 있음을 가정하였다.

4.2 캐쉬 변수 및 일관성 유지를 위한 시간 변수 환경

캐쉬의 크기는 64Kbytes이고 캐쉬 블록은 16 bytes를 갖는 2 웨이 세트 어소시어티브(2-way set associative) 캐쉬를 가정했다. 또한 캐쉬 일관성 프로토콜은 쓰기 무효화 프로토콜(write invalid protocol)을 사용했다[13]. 표1은 상기의 시스템 및 캐쉬에 대한 가정들을 기반으로 하여 1 싸이클의 버스 전송 시간과 1 싸이클의 주소 해석 시간을 포함한 캐쉬 프로토콜에서 기본적인 시간 변수들의 값을 나타내고 있다.

표 1 캐쉬 일관성 프로토콜을 위한 기본 시간 변수 값

사건	제어 동작	소요시간
공유 캐쉬 블록에 대한 쓰기	무효화 신호 처리	3 cycles
캐쉬 실패	다른 프로세서의 캐쉬로부터 데이타 전송	7 cycles
캐쉬 실패	메모리로부터 데이타 전송	20 cycles

4.3 시험용 병렬 프로그램

시험용 병렬 프로그램으로 과학 계산용 프로그램으로 널리 알려졌으며 연속적으로 데이타 배열들을 사용하는 BMM(Blocked Matrix Multiplication), FFT(Fast Fourier Transform), LU(LU-Decomposition), BS (Bitonic Sorting) 4가지의 병렬 프로그램을 선택하였다 [14,15]. 모든 프로그램은 C-Language로 작성되었고 SGI(Silicon Graphics Indigo)사에서 제공하는 병렬 매크로 팩키지(macro package)를 사용하였다.

• FFT 프로그램

FFT 프로그램은 Cooley-Tukey의 알고리즘을 사용한 퓨리어 변환을 수행하는 프로그램이다. 주요 데이타 구조로는 2개의 1차원 배열들이다. 루프가 반복될 때마다 2개의 배열이 교대로 입력 데이타 또는 결과를 저장하는 역할을 담당한다. 시험을 위하여 입력 데이타 항목은 65536 개로 총 데이타 배열들의 크기는 1 Mbytes이다. 프로세서에 할당된 그레인 크기는 최대 성긴 그레인 크기인 8192 개의 연속된 데이타 항목이다.

• BMM 프로그램

BMM 프로그램은 블록화 알고리즘을 적용한 행렬 곱셈이다. 행렬 곱셈을 위하여 3개의 2차원 배열들이 사용된다. 시험을 위하여 256 256 행렬 곱셈을 수행하였고 데이타 배열들의 크기 합은 1.5 Mbytes이다. 8개의 프로세서를 사용하므로 그레인 크기는 32 32 서브 블록을 사용했고 이 서브 블록은 블록 알고리즘의 블록화 요소(blocking factor)에 해당된다.

• LU 프로그램

LU는 하나의 행렬을 하 삼각 행렬과 상 삼각 행렬의 곱으로 분해하는 알고리즘이다. 본 논문에서는 캐쉬의 지역성을 이용하는 블록화 알고리즘을 적용하여 계산하는 방법을 사용했다. 주요 데이타 구조는 분해되는 2차원 행렬이다. 시험을 위하여 512 512 크기의 행렬을 사용하였고 크기는 2 Mbytes 이다. 그레인 크기는 64 64 서브 블록을 사용하였다.

• BS(Bitonic Sort) 프로그램

BS 프로그램은 입력키 값들을 바이토닉 순서화 (bitonic sequence)와 바이토닉 병합(bitonic merging)을 통하여 정렬하는 프로그램이다. 데이타 구조로는 두 개의 1 차원 배열들로 구성된다. 시험을 위한 입력 개수는 32768 개의 키값을 사용하였고 크기는 512 Kbytes 이다. 프로세서에게 할당되는 그레인 크기는 최대 성긴 그레인 크기인 4096 개의 연속된 데이타 항목이다.

표2는 상기의 병렬 프로그램들에서 선인출이 수행되는 데이타 배열 및 사용된 그레인 크기를 바탕으로 한 선인출 정도를 캐쉬 블록(16bytes)의 개수로 나타낸다.

표 2 선인출 요구 데이타 배열 및 선인출 정도

프로그램	선인출 되는 배열	그레인 크기	선인출 정도 (캐쉬 블록 개수)
FFT	2개의 1차원 데이타 배열	8192 개의 데이타 항목	4095 개
BMM	입력 데이타를 저장하고 있는 2개의 2차원 배열	32 x 32 서브 블록의 행 크기	15 개
LU	1개의 2차원 데이타 배열	64 x 64 서브 블록의 행 크기	31 개
BS	2개의 1차원 데이타 배열	4096 개의 데이타 항목	1023 개

5. 성능 평가

본 논문에서 제안한 선인출 방법(GSP: Granularity and Sequential Prefetching)은 선인출을 사용하는 메모리 지역을 사용자가 프로그램에서 지정하며 병렬 프로그램의 성김도에 따라 선인출 정도를 결정하므로 정확한 선인출 정도에 의한 무익한 선인출 데이타를 감소시킬 수 있다. 제안된 GSP와의 성능 비교를 위하여 본 논문에서 Smith[1]가 제안한 캐쉬 실패가 발생된 이웃 블록을 한개를 선인출하는 OBL(One Block Lookahead) 방식과 Dahlgren[7]이 제안한 ASP (Adaptive Sequential Prefetching: 적응적 순차 선인출)을 사용했다. 두 방식 모두 캐쉬 실패가 발생될 때 선인출이 시작되므로 GSP에서의 선인출 시작 동기와 같다. 특히 ASP는 무익한 선인출을 방지하기 위해서 선인출된 데이타들의 효용성을 검사하여 선인출 정도를 증가 또는 감소하였다. 따라서 본 논문의 GSP도 무익한 데이타들의 선인출을 방지하는 것이 주요 목적이므로 주요 비교 대상이 된다. 본 모의 시험에서는 ASP를 평가하기 위해서 선인출된 데이타의 사용 효율이 75% 이상에서는 선인출 정도를 증가시키고 50% 이하에서는 선인출 정도를 감소시키는 방법을 사용했다[7]. 실제로 ASP를 구성하기 위해서는 캐쉬 블록마다 2비트의 상태를 나타내는 하드웨어 및 선인출 효율 계산 회로 등 의 하드웨어 비용이 소요되며 GSP는 3장에서 설명한 작은 규모의 선인출 주소 관리 테이블이 필요하다.

그림2에서 그림6까지는 4개의 병렬 프로그램을 OBL, ASP, GSP 각각의 선인출 방법을 사용하여 수행할 때 발생된 실험 결과이다. 모든 값들은 OBL에서 측정된 값으로 정규화(normalization)하였다. 그림2는 4개의 프로그램들의 소요된 수행시간을 나타낸다. 모든 프로그램

에서 OBL 보다 ASP이 ASP 보다는 GSP가 좋은 성능을 나타내고있다. 이러한 결과의 원인으로는 그림3에 나타나는 각 방법들에 의하여 발생되는 캐쉬 실패율의 차이 때문이다. 즉 OBL에서는 선인출되는 정도가 한 블록에 불과하므로 선인출에 의한 캐쉬 효과가 작다. 반면에 ASP는 선인출되는 데이타들의 사용 여부에 따라 선인출되는 정도가 가변적으로 변하므로 OBL 보다는 많은 데이타들을 한번의 버스 트랜잭션으로 선인출 한다. 그러나 ASP는 배열로 구성된 그레인들에 대하여 그레인 경계에 도달할때 까지 점차적으로 선인출 정도를 증가시키므로 여러 번의 선인출이 필요하다. 또한 그레인 경계 지역에서는 다른 프로세서들에게 할당된 데이타들을 최대로 증가된 선인출 정도만큼 자신의 캐쉬 메모리에 선인출 하게 되므로 무익한 선인출이 발생하게 된다. 반면에 GSP는 사용자가 제공한 순차적 주소가 보장된 해당 데이타를 한번에 선인출 하므로 ASP보다 적은 버스 트랜잭션이 발생된다. 또한 그레인들의 경계지역을 맞추어 선인출을 수행하므로 선인출 데이타들은 대부분 유용한 데이타로 사용된다. 그림 5는 프로그램들의 수행 중 각각의 선인출 방법들에서 발생된 버스 트랜잭션 개수를 비교하고 있다. 또한 그림6은 각각의 선인출 방법들에서 유용한 선인출의 양과 선인출 후 사용되지 않거나 캐쉬 블록의 대체에 의한 무용한 선인출 양을 나타내고있다. 이들 그림들에서 나타나듯이 ASP는 상기에 지적한 원인들에 의하여 GSP 보다 많은 버스 트랜잭션 및 보다 많은 무익한 선인출을 발생하고 있음을 나타낸다.

모의 시험에서는 공유 버스를 사용하였고 선인출에 의한 버스 전송시 공유 버스의 버스트 모드를 사용하여 한번의 버스 트랜잭션에 의하여 많은 양의 데이타를 캐쉬에 적재시켰다. 이렇게 적재된 데이타들은 캐쉬 적중이 되어 버스 트랜잭션 요구를 전체적으로 감소시키므로 공유 버스를 사용하기 위하여 버스의 대기 큐에서 소모하는 프로세서들의 대기 시간에도 변화가 생긴다. 즉, 그림3의 BMM과 LU 프로그램에서 캐쉬 실패율에 있어서 ASP와 GSP는 차이는 근소하다. 그러나 그림4로부터 버스 큐에서 버스를 사용하기 위하여 대기하는 평균 프로세서 개수의 차이는 그림 3에 나타난 캐쉬 실패율 차이보다 더 크다는 것을 알수 있다. 이것은 GSP 방식이 ASP보다 많은 양의 유용한 데이타를 캐쉬에 적재시키고 따라서 제안된 버스의 용량을 보다 효과적으로 사용하게 하므로 발생된 것이다.

이러한 상기의 실험 결과들을 바탕으로 본 논문에서 제안된 GSP 방법은 병렬 프로그램들이 순차적 주소를 갖는 데이타들을 사용하는 경우 OBL, ASP 방법보다 적은 버스 트랜잭션 개수와 보다 많은 유용한 데이타들을 선인출 하므로 프로그램의 수행시간을 단축시킴을 알수 있다.

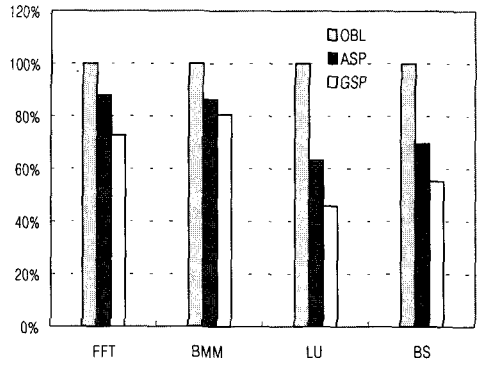

그림 2 프로그램들의 수행시간

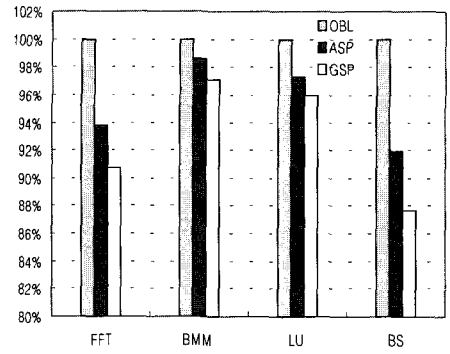

그림 3 프로그램들의 캐쉬 실패율

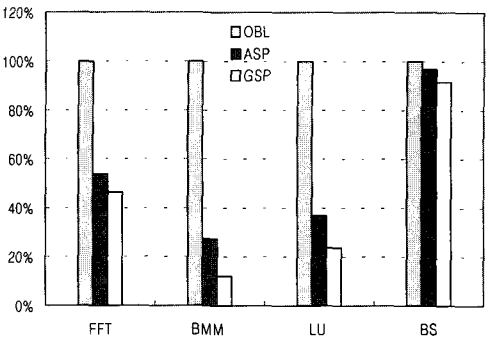

그림 4 버스에서 평균대기 프로세서

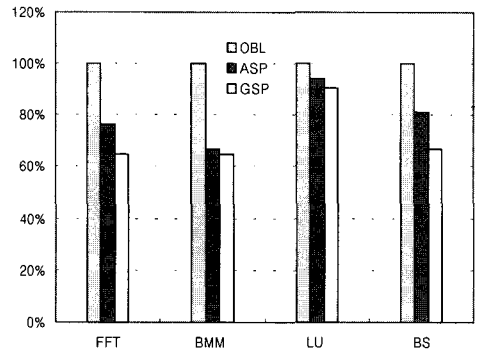

그림 5 버스 트랜잭션 개수

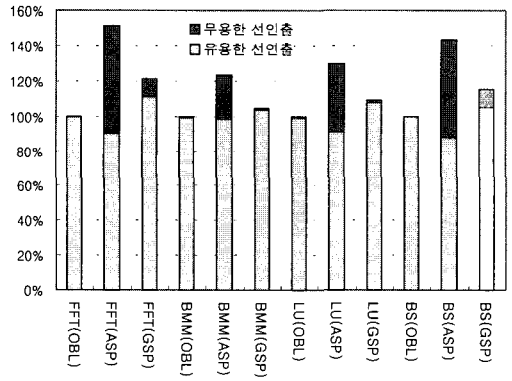

그림 6 선인출된 메모리 블록

6. 결 론

메모리로부터의 데이타 선인출 기법은 사용이 예측된 데이타들을 캐쉬 메모리에 미리 적재하므로 캐쉬 실패를 감소시키기 위하여 사용되는 방법이다. 그러나 선인출되는 데이타가 캐쉬 메모리내부의 다른 유용한 캐쉬 블록들을 대체하거나 또는 선인출 데이타가 예상과 다르게 사용되지 않을 경우 캐쉬 메모리의 공간만 차지하는 결과를 가져온다. 이러한 무익한 캐쉬 선인출을 감소시키기 위하여 본 논문에서는 병렬 프로그램의 성김도 정책을 이용한 데이타 선인출 방법을 제안했다. 프로세서에게 할당되는 그레인들의 주소공간과 그레인 크기는 프로그래머에게 알려지는 정보들이다. 본 논문에서 제안된 선인출 방법은 프로세서 노드에 하드웨어적으로 조그마한 테이블을 구성하고 프로그래머는 운영체제의 도움을 받아 선인출을 위한 정보들을 이 테이블에 저장한다. 이 테이블의 정보를 바탕으로 캐쉬 실패가 발생될때 병렬 프로그램의 그레인 크기를 기반으로 한 선인출을 수행하므로 무익한 데이타 선인출을 감소시킬 수 있었다.

모의 시험으로부터 본 논문에서 제안된 GSP 선인출 방법은 기존의 OBL 방법보다 19.4% ~ 65%의 성능향상을 나타내었고 또한 기존의 ASP 방법보다는 6.5% ~ 27.5%의 성능 향상을 나타내었다. 이런 결과의 원인들로는 제안된 GSP 방법은 기존의 OBL, ASP 방법보다 적은 버스 트랜잭션 회수와 보다 많은 유용한 데이타들을 선인출하기 때문이다. 그러나 본 논문에서 사용된 병렬 프로그램들은 수행 중 그레인 크기가 변하지 않는 특성들을 가지고 있다. 프로그램 수행 중 그레인 크기가 변화되는 특성을 갖는 프로그램들에서는 적응적으로 선인출 정도를 변화시키는 방법들이 연구되어야 한다.

참 고 문 헌

[1] A.Smith, "Cache memories," ACM Computing Surveys, vol.14,pp. 473-530, Sep. 1982.

[2] N. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," In Proceedings of the 17 th Annual International Symposium in Computer Architecture, pp.364-373, 1990.

[3] J. Baer and T. Chen, "An effective on-chip preloading scheme to reduce data access penalty," In Proceedings of Supercomputing '91, pp.176-186, 1991.

[4] J. Baer and T.Chen, "Reducing memory latency via non-blocking and prefetching caches," In Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, pp.51-61, Oct. 1992.

[5] J. Fu and J. Patel, "Data prefetching in multiprocessor vector cache memories," In Proceedings of the 18th Annual International Symposium on Computer Architecture, pp.54-63, 1991.

[6] J. Fu, J. Patel and B. Janssens, "Stride directed prefetching in scalar processors," In Proceedings of the 25th International Symposium on Microachitecture, pp.102-110, 1992.

[7] F. Dahlgren, M. Dubois and P. Stenstrom, "Fixed and Adaptive sequential prefetching in shared memory multiprocessors," In Proceedings of the International Conference on Parallel Processing, pp.56-63,1993.

[8] A. Porterfield, "Software methods for improvement of cache performance on supercomputer

applications," In Technical Report COMP TR-89-93, *Rice University*.
[9] E. Gornish, E. Granston and A. Veidenbaum, "Compiler-directed data prefetching in multiprocessors with memory hierarchies," In Proceedings of 1990 International Conference on Supercomputing, pp.354-368, 1990.
[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, 1991.
[11] T. Mowry, M. Lam and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," In proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, pp.62-73, 1992.
[12] J. E. Veenstra and R. J. Fowler, "MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors," In Proceeding of 2nd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), pp 201-207, Jan. 1994.
[13] J. Archibald and J-L. Baer, "Cache Coherence Protocols : Evaluation Using a Multiprocessors Simulation Model," ACM Transactions on Computer Systems, Vol. 4, No. 4, pp 273-298, Nov. 1986.
[14] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh and A. Gupta, "The SPLASH-2 Programs: *Characterization* and Methodological Considerations," In Proceedings of the 22th Annual International Symposium on Computer Architecture, pp 24-25, June 1995.
[15] Vipin Kumar, Ananth Grama, Anshul Gupta and George Karypis, "Introduction to Parallel Computing(Design and Analysis of Algorithms)," The Benjamin/Cummings Publishing Company, Inc., pp.169, pp.179, pp.380, 1994.

정 인 범

1985년 고려대학교 전자공학과 졸업(학사). 1994년 한국과학기술원 정보통신공학과 졸업(석사). 1995년 ~ 현재 한국과학기술원 전산학과 박사과정 재학중. 1985년 ~ 1995년 (주) 삼성전자, 컴퓨터 시스템 사업부 선임 연구원. 관심분야는 운영체제, 컴퓨터구조, 병렬처리, 멀티미디어 시스템.

이 준 원

1983년 서울대학교 계산통계학과 졸업(학사). 1990년 Georgia Tech. 전산학과(석사). 1991년 Georgia Tech. 전산학과(박사). 1983년 ~ 1986년 (주)유공 근무. 1991년 ~ 1992년 IBM 근무. 1992년 ~ 현재 한국과학기술원 전산학과 부교수. 관심분야는 운영체제, 병렬처리 등임.