ISSN 2383-630X(Print) / ISSN 2383-6296(Online)
Journal of KIISE, Vol 41, No 10, pp 832-837, 2014 10
http //dx doi org/10 5626/JOK 2014 41 10 832

브레이크포인트를 이용한 범용 워크로드 프리페칭 프레임워크

(Prefetching Framework for General Workloads Using Breakpoint)

고 광 진 † 유 준 희 †† 강 경 태 ††† 신 현 식 ††††
(Kwangjin Ko) (Junhee Ryu) (Kyungtae Kang) (Heonshik Shin)

요 약 프로그램의 로딩 속도는 프로그램이 요청하는 디스크 블록을 미리 읽어 들임으로써(프리페칭) 향상시킬 수 있다. 그러나 기존의 프리페칭 관련 기법들은 특정 프로그램에 최적화된 경우를 제외하면 상당한 오버헤드를 보여주었다. 특히 요청블록을 정확히 추적하는데 어려움이 있었다. 어떤 블록들은 여러 시퀀스(단위시간 내에 추적된 블록들)에 나타날 수 있고 두 접근 시퀀스가 동일 하더라도 버퍼 캐시에 의해서 접근 시간과 수집되는 블록 정보가 다를 수 있기 때문에 분석이 까다롭다. 본 논문에서는 소프트웨어적 접근 방법으로 새로운 범용 워크로드 프리페칭 기법을 제안한다. 제안하는 프리페칭 기법은 브레이크포인트를 프로그램의 적재 적소에 배치함으로써 요청 블록의 상관관계 정보를 수집하고, 이를 바탕으로 프리페칭을 수행한다. 상용 하드디스크를 이용한 실험 결과, 불필요한 오버헤드가 감소되었으며 기동 시간은 평균 30%, 로딩은 평균 15% 단축되었음을 확인하였다.

키워드: 브레이크포인트, 중단점, 프리페쳐, 범용 워크로드, 로딩시간 단축

Abstract Application loading speed can be improved by timely prefetching disk blocks likely to be needed by an application. However, existing prefetchers if they are not specialized to a particular application incur high overheads and are poor at identifying the blocks that will actually be required. There are many sequences in which blocks may be needed and, even if two access sequences are identical, block tracing and access timings can be affected significantly by the state of the buffer cache. We propose a new application independent software based prefetching technique, in which breakpoints are inserted at appropriate places in an application to collect the information on correlations between the blocks and to prefetch the potential blocks ahead of their schedule based on it. Experiments on an HDD based desktop PC demonstrated an average 30% reduction in application launch time and 15% in general I/O, while reducing the wasted overhead.

Keywords: breakpoint, prefetcher, general workloads, loading time reduction

· 이 논문은 2013년도 정부(미래창조과학부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임(NRF 2013R1A1A1059188, NRF 2012R1A1A2044653)

† 비 회 원 : 서울대학교 컴퓨터공학부
kjko@cslab snu ac kr
†† 비 회 원 : 삼성전자 네트워크사업부 책임연구원
jhryu@cslab snu ac kr
††† 종신회원 : 한양대학교 컴퓨터공학과 교수
ktkang@hanyang ac kr
†††† 종신회원 : 서울대학교 컴퓨터공학부 교수(Seoul National Univ)
shinhs@snu ac kr
(Corresponding author임)

논문접수 : 2014년 5월 9일
(Received 9 May 2014)
논문수정 : 2014년 8월 18일
(Revised 18 August 2014)
심사완료 : 2014년 8월 19일
(Accepted 19 August 2014)

정보과학회논문지 제41권 제10호(2014 10)

1 서 론

응용 프로그램의 긴 기동 및 중간 로딩 시간은 사용자의 몰입을 방해하고 사용자의 체감 성능에 큰 영향을 준다. 로딩 중 발생하는 디스크 입출력 요청은 임의 접근 패턴을 보이기 때문에 SSD보다는 기계적 구동장치를 사용하는 하드디스크를 사용할 때, 사용자는 긴 로딩 시간을 경험하게 된다[1]. SSD로 하드디스크를 대체할 수 있지만 단위 용량 당 비용이 높아 저용량의 플래시를 하드디스크의 캐시로 사용하는 경우가 많다[2,3].

디스크 블록 프리페칭은 가까운 시간 내에 읽힐 가능성이 높은 디스크 블록을 디스크 캐시에 미리 읽어서 프로그램이 메모리로부터 데이터를 읽도록 유도하여 디스크 입출력 시간을 단축시키는 기법이다[4]. (이 논문에서 '프리페칭'은 별도의 설명이 없는 한, 디스크 블록 프리페칭을 의미한다.) 프로그램의 기동과 관련된 블록만 프리페칭 대상으로 하는 경우에는 블록 모니터링 및 프리페칭 시점을 쉽게 알 수 있지만, 중간 로딩까지 프리페칭 대상을 확장한 경우에는 블록 간 상관관계 분석 및 프리페칭 시점을 파악하는데 많은 메모리와 계산 오버헤드가 요구되거나 적중률이 낮아 현실성이 낮다.

본 논문에서는 브레이크포인트를 소스코드에 삽입함으로써 블록 간 상관관계 분석의 오버헤드를 경미한 수준으로 낮추었으며, 디스크 블록 요청을 논리 블록 번호로 정렬 및 병합하고 NCQ (native command queuing)를 활용함으로써 하드웨어 재스케쥴링의 이점을 극대화하였다. 다양한 프로그램을 이용한 실험에서 제안한 기법을 적용하였을 경우 평균 기동 및 로딩 시간이 각각 30%, 15% 단축되었다.

본 논문의 구성은 다음과 같다. 2장에서는 기존에 제안된 프리페칭 관련 연구들을 살펴보고, 3장에서는 브레이크포인트의 동작에 대해서 설명한다. 4 5 장에서는 브레이크포인트를 이용한 프리페칭 프레임워크의 설계 및 구현에 대해 기술한다. 6장에서는 실험 환경 및 성능 평가를 다루며, 마지막으로 7장에서 결론을 맺는다.

2 관련 연구

2 1 기동 워크로드 프리페칭 기법

프로그램의 기동 및 부팅 시 발생하는 워크로드를 처리하기 위한 프리페칭 연구는 많이 수행되어 왔으며 윈도우나 리눅스, 맥 OS와 같은 대표적인 운영체제에서 사용되고 있다[5,6].

저장장치 측면에서는 하드디스크를 대상으로 하는 연구가 많다. 하드디스크는 기계적인 움직임이 많기 때문에 블록 요청을 최적화하여 헤드의 움직임을 최소화하면 접근 시간이 크게 단축된다[6]. Paralfetch [7]는 하드디스크뿐만 아니라 SSD를 지원하며 CPU 및 SSD의 플래시 메모리 수준에서 병렬성을 활용한다. 그러나 이러한 기법들은 모두 범용 워크로드에는 적용하기 어려운 문제가 있다.

2 2 범용 워크로드 프리페칭 기법

범용 워크로드 프리페칭은 적용 대상을 한정하지 않기 때문에 항상 블록 접근을 감시해야 하고, 블록들 간의 복잡한 상관관계를 분석해야 한다. 이러한 오버헤드를 줄이기 위한 기존 연구들이 많이 있다.

추측 실행[8]기법은 본 프로세스와 별도로 미리 실행하는 프로세스를 생성하여 관련 블록들이 프리페칭 되도록 하는 기법이다. 이 기법은 분기 예측 실패 시 오버헤드가 크고 입출력을 최적화하기 힘들다. 또한 실행 파일의 저장공간 오버헤드는 평균 3배의 증가를 보이고, 메모리 사용량은 두 배 가량 증가한다.

C Miner[9]와 Diskseen[4]은 요청 블록의 상관관계를 분석하여 프리페칭할 블록을 결정한다. C Miner는 상관관계 분석의 오버헤드를 줄이기 위해 서로 다른 두 블록이 항상 짧은 시간 안에 발생해야만 상관 관계 정보를 유지한다. 이 때문에 시간이 지날수록 상관관계 정보가 없어져 프리페칭 효과가 저하된다. Diskseen은 저장공간 오버헤드를 줄이기 위해 하나의 블록이 미리 정해진 개수의 다른 블록과 상관관계를 가질 수 있도록 제한한다. 이 기법은 디스크 1GB당 4MB의 메모리를 사용할 정도로 오버헤드가 크고 라이브러리 블록과 같이 많은 응용과 관계가 있는 경우 다루기가 어렵다.

BORG[10]는 하드디스크 상에서 많이 접근되는 블록들을 BOPT라는 특정 파티션에 복제함으로써 하드디스크의 헤드 움직임을 줄여 디스크 처리 성능을 높인다. 하지만 메모리상의 맵 테이블 오버헤드는 BOPT 파티션 크기의 0.25% 정도로 작지 않은 수준이다.

컴파일러 지원 프리페칭[11]은 컴파일러 수준에서 정적 분석을 통해 프리페치 코드를 삽입하여 페이지 폴트로 인한 디스크 처리 시간을 단축시킨다. 하지만 처리하는 데이터가 작은 경우 성능이 저하된다. Libprefetch[12]는 프리페칭 요청 코드를 개발자가 직접 소스코드에 삽입하는 기법이다. 개발자는 디스크 블록의 접근 패턴을 분석하여 프리페칭을 수행하는 콜백(callback) 함수를 작성해야 하는데, 효율적인 콜백 함수의 작성이 어렵다.

대부분의 응용 프로그램들은 코드나 데이터 영역보다 많은 수의 외부 리소스 파일들로 구성된다. 그러나 [11]은 코드나 데이터 영역에 대해서만 프리페칭을 지원하며, [12]는 외부 리소스 파일만 프리페칭이 가능하다. 본 연구에서 제안하는 방법은 메모리에 매핑되는 파일 영역과 외부 리소스 파일을 모두 포함하여 프리페칭을 수행한다. 분석의 기준점이 없는 C Miner에서 가까운 시

간 안에 발생한 블록 요청만 상관관계를 유지하는 단점은 제안하는 기법에서 브레이크포인트로 분석 기준을 제공함으로써 해결하였다. 그리고 Diskseen과 같이 블록 단위로 정해진 개수의 상관관계 정보를 갖지 않고, 브레이크포인트 간의 블록 입출력 정보를 기록함으로써 경미한 메모리 오버헤드를 갖는다.

앞서 살펴본 바와 같이 범용 워크로드의 프리페칭 기법들은 상당한 연산 및 메모리 오버헤드를 요구하고 예측 실패에 대한 자원 낭비가 크기 때문에 기존에 제안된 기법들은 실용성이 낮다고 할 수 있다.

3 브레이크포인트 원리 및 구현

브레이크포인트는 주로 디버깅에 사용되는 기법이다. 소스코드의 특정 위치에 브레이크포인트를 설정하고 프로그램을 시작하면 브레이크포인트가 설정된 지점에서 프로세스의 실행 흐름이 정지된다. 이때 디버거는 정지된 프로세스의 변수나 레지스터 값을 파악하고 수정할 수 있다. 소프트웨어 브레이크포인트 기법에서 디버거는 브레이크포인트를 설정하고자 하는 지점에 있는 명령의 명령코드(OpCode)를 0xCC로 변경시키고, 기존 명령코드는 디버거가 관리하는 별도의 테이블에 명령어의 주소와 함께 저장한다. x86계열의 CPU에서 0xCC 명령을 실행하면 3번 인터럽트(INT3)를 발생시킨다. 트랩 신호(SIGTRAP)는 커널을 거쳐 해당 명령을 실행시킨 디버거 프로세스에 전달된다. 디버거는 디버깅 관련 작업을 한 뒤 디버거 테이블을 조회하여 명령코드를 기존의 값으로 복원시키고 실행을 재개한다.

소스코드에 직접 브레이크포인트를 삽입하는 경우, C 언어에서는 *__asm__ __volatile__("int3");* 코드를 작성하면 명령코드 0xCC값을 갖는 명령어로 번역된다. 소스코드에 브레이크포인트를 직접 삽입할 때는 디버거와 달리 기존 명령을 수정하지 않으므로 별도의 명령코드 백업 및 복원 과정이 필요 없다.

4 브레이크포인트 기반 프리페칭 프레임워크의 설계

기존에 제안된 범용 워크로드 프리페처들의 가장 큰 약점은 워크로드 분석에 많은 오버헤드가 요구된다는 점이다. 본 논문에서는 프리페칭 시점을 파악하는 오버헤드를 낮추기 위해 브레이크포인트를 사용하였고, I/O 요청 수집 및 최적화에 경미한 오버헤드를 갖는 Paral fetch 컴포넌트를 적용하여 전체 분석 오버헤드를 상당히 낮출 수 있었다. 특히 브레이크포인트를 사용하면 소스코드 수준뿐만 아니라 바이너리 수준에서도 동일한 설계 구조로 프리페칭을 수행할 수 있다.

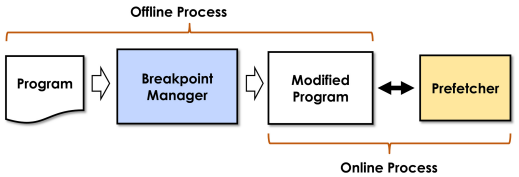


그림 1 브레이크포인트 프리페처 프레임워크
Fig. 1 Breakpoint prefetcher framework

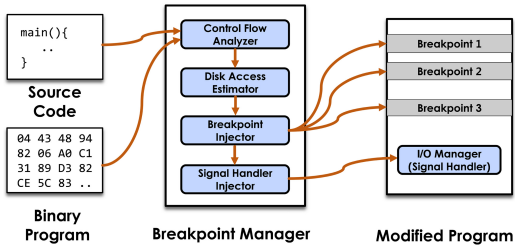


그림 2 프레임워크의 오프라인 프로세스
Fig. 2 Offline process of framework

제안하는 전체 프레임워크의 구성은 그림 1과 같다. 프리페칭의 실행 흐름은 크게 두 단계로 나뉜다. 첫 번째로 오프라인 프로세스에서는 정적 분석을 통해 바이너리 프로그램 및 소스코드에 브레이크포인트를 삽입한다. 두 번째 단계는 온라인 프로세스이며, 브레이크포인트가 삽입된 프로그램을 실행하고 프리페칭을 수행한다. 프리페처 컴포넌트는 커널 내부에 존재한다.

오프라인 프로세스의 구체적인 동작 방식은 그림 2와 같다. 브레이크포인트 매니저는 프리페칭할 목적 프로그램을 입력 받아 제어 흐름 분석(control flow analysis) 및 디스크 접근 예측(disk access estimation) 과정을 수행한다. 이 두 가지 과정을 통해 대량의 I/O 요청이 발생할 것으로 예상되는 지점을 추정할 수 있다. 브레이크포인트 삽입기(breakpoint injector)는 앞서 추정된 위치에 브레이크포인트를 삽입한다. 프로그램이 동작할 때 시그널 핸들러와 프리페처가 I/O 요청을 감시하면서 많은 I/O를 발생시키지 않는 브레이크포인트 지점은 다음 실행 시 무시할 수 있고, 많은 I/O가 발생하는 브레이크포인트 지점에서는 블록 정보를 기록하고 이후에 프리페칭을 수행토록 할 수 있다.

마지막으로 시그널 핸들러 삽입기(signal handler injector)는 I/O 매니저 역할을 하는 시그널 핸들러를 삽입한다. 바이너리를 입력으로 받은 경우 I/O 매니저는 바이너리 내부가 아닌 외부에 삽입할 수 있다. 디버거와 유사하게 프리페칭할 목적 프로그램을 감싸는 래퍼(wrapper) 프로그램을 만드는 것이다.

오프라인 프로세스는 유휴시간에 최초 1회 수행된다.

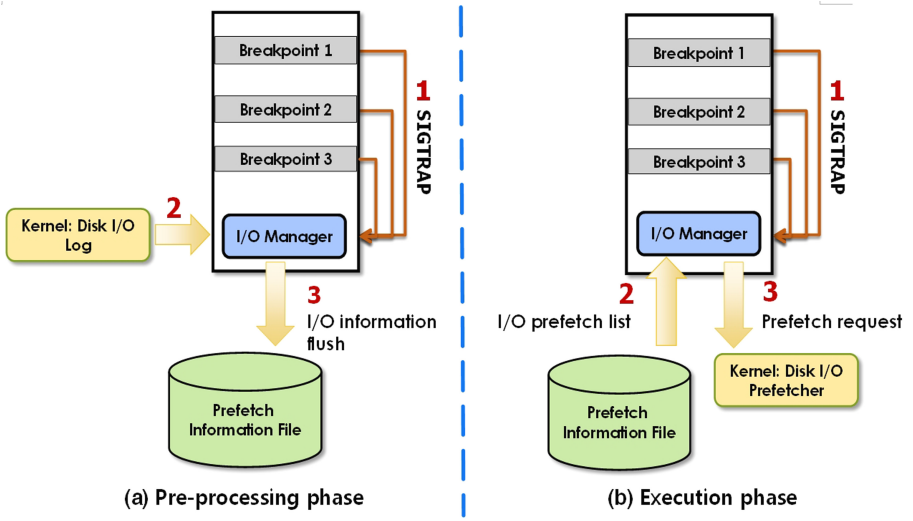


그림 3 프레임워크의 온라인 프로세스
Fig. 3 Online process of framework

그러나 프로그램의 패치 및 업데이트 등으로 프로그램 내용이 변경되는 경우 오프라인 프로세스를 재수행 할 필요가 있다.

온라인 프로세스의 구체적인 동작 과정은 그림 3과 같다. 프로그램 최초 실행시 그림 3(a)와 같이 브레이크포인트가 설정된 구간별로 요청된 블록 I/O정보를 로그에 기록한다. 이후 동일한 구간을 다시 실행할 때 그림 3(b)와 같이 I/O 매니저는 브레이크포인트 구간의 블록 요청 빈도수와 블록 간 상관관계 분석을 통해 필요 없는 브레이크포인트 지점은 제외시킨다. 대량의 I/O가 발생하는 브레이크포인트 지점에서는 프리페칭을 수행한다.

5 브레이크포인트 기반 프리페칭 프레임워크의 구현

제안하는 프레임워크는 크게 세 개의 구성요소로 이루어져있다. 첫 번째는 프리페쳐 커널 컴포넌트로 블록 I/O 요청을 최적화 및 기록하고 프리페칭 대상 프로그램을 대신하여 블록 I/O를 요청하는 기능을 담당한다. 본 논문에서는 Paralfetch 프리페쳐를 사용하였다. 두 번째로 브레이크포인트 매니저는 브레이크 포인트와 I/O 매니저를 프로그램에 삽입한다. 브레이크포인트의 삽입 위치는 정적 분석 또는 개발자가 직접 삽입하는 형태로 구현할 수 있는데, 본 논문에서는 직접 삽입 방법으로 구현하였다. 개발자는 자신이 개발하는 프로그램의 동작 흐름을 알고 있으므로 어느 부분에서 로딩을 수행하는지 알 수 있다. I/O 요청이 발생하는 코드의 처음 부분에 1줄짜리 코드로 간단히 브레이크포인트를 삽입할 수 있다.

동적인 I/O 요청으로 인해 동일한 브레이크포인트 위치에서 서로 다른 I/O 워크로드가 발생할 수 있다. 예를 들어 게임 응용프로그램에서 사용자가 스테이지를 선택하는 경우, I/O 매니저가 해당 변수 정보를 참조하면 동적인 워크로드도 효율적으로 프리페칭 할 수 있다.

마지막으로 I/O 매니저는 시그널 핸들러로 구현하였다. 브레이크포인트로 인해 시그널 핸들러가 호출되면 핸들러는 호출한 브레이크포인트가 유효한지 검사하여 프리페쳐 호출 여부를 결정한다. 유효성 검사는 브레이크포인트 구간 사이에서 발생하는 I/O 요청들의 수와 초당 요청 개수를 종합하여 판단한다.

응용 프로그램에서 프리페쳐를 직접 호출하지 않고 브레이크 포인트를 사용하여 I/O 매니저를 경유하도록 함으로써, 프리페쳐 호출 시점 분석을 I/O 매니저가 종합적으로 관리할 수 있다. 또한 바이너리 수준의 프리페칭에서도 동일한 설계구조를 재사용할 수 있는 이점이 있다. 바이너리 수준에서는 I/O 매니저를 ptrace 및 kprobe, gdb와 같은 디버깅 툴로도 구현할 수 있다.

본 논문에서는 커널 수준에서 구현된 프리페쳐 컴포넌트와의 인터페이스에 시스템 콜을 사용하였다. 또 다른 구현 방법은 후킹(hooking) 기법을 이용하여 트랩 신호를 받은 커널이 곧바로 프리페쳐를 호출할 수 있다.

본 연구에서는 제어 흐름 분석과 디스크 접근 예측 모듈의 인터페이스만 제공하며 이에 대한 실제적인 구현은 추후 연구로 남겨둔다.

6 실험 및 분석

본 실험은 리눅스 환경의 데스크톱 PC에서 수행하였다. 실험 PC는 Intel I3 2100 (3.1GHz) CPU, 4GB RAM, 640GB HDD, AMD Radeon HD4850 GPU로 구성되었고 Ubuntu 12.04 64bit OS가 설치되었다.

실험은 크게 두 부분으로 나뉜다. 첫 번째는 제안한 프레임워크에서 디스크 입출력이 효율적으로 수행되는지 살펴보는 실험이고, 두 번째는 프리페칭의 성능을 측정하기 위한 실험이다. 본 실험에서는 프로그램의 기동 시간과 로딩 시간을 cold, warm, prefetch 항목으로 평가하였다. cold는 요청 블록이 디스크캐시에서 모두 미스(miss)되는 최악의 경우이고, warm 에서는 모든 요청 블록이 히트(hit) 된다. 마지막으로 prefetch는 cold 상태에서 제안한 기법을 적용했을 때 소요되는 시간을 측정하였다.

프리페쳐는 동기적으로 동작하도록 설정하였고[7], 브레이크포인트는 로딩이 발생하는 지점의 선두에 설정하였다. 사용자의 조작 시간은 실험에서 제외하였다.

6 1 제안한 프레임워크의 동작 검증

제안한 프레임워크가 제대로 동작하는지 검증하기 위해 blktrace툴로 I/O 요청 크기와 디스크 상에서 서비스 중인 큐 깊이(queue depth) 값을 측정하였다. 실험 대상 프로그램으로 Speed Dreams 게임[13]을 사용하였다.

그림 4는 소스코드 수준 프리페칭을 적용했을 때 총 I/O요청 크기와 큐 깊이의 변화를 나타낸다. 그림 4(a)에서 프리페칭을 적용하지 않고 cold 상태로 실행했을 때 요청 크기는 300KB이내로 나타났다. 반면 그림 4(b)

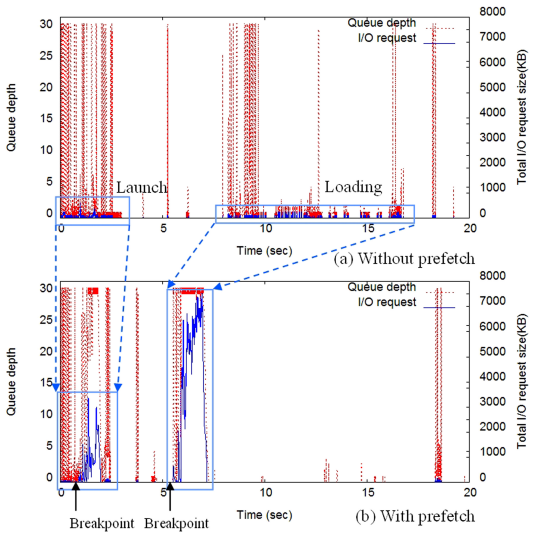


그림 4 I/O 요청 크기 및 Queue depth 변화

Fig. 4 Changes in I/O request size and queue depth

와 같이 프리페칭을 적용한 결과 작은 요청들이 병합되어 최대 8,000KB에 가까운 크기로 요청한 것을 볼 수 있다. 기동에서 브레이크포인트가 다소 늦게 실행되는 것은 메인 프로그램이 적재된 이후 브레이크포인트가 실행되기 때문이다. 큐 깊이는 프리페칭을 적용하지 않았을 때 5 이내의 값이 빈번한 반면, 프리페칭을 적용한 결과 대부분이 최대 큐 깊이인 32 값을 보여준다.

6 2 성능 평가

그림 5와 그림 6은 각각 제안한 프리페쳐를 기동과 로딩에 적용한 실험 결과이다. 실험 결과는 cold start 시간을 기준으로 정규화 하였다. 기동과 로딩에서 실험한 모든 응용 프로그램의 prefetch 시간이 cold start 시간보다 적게 걸리는 것을 보여주었다. 프리페쳐를 적용했을 때 평균적으로 cold start 대비 30%의 시간이 감소되었고 로딩의 경우 평균적으로 15% 감소되었다.

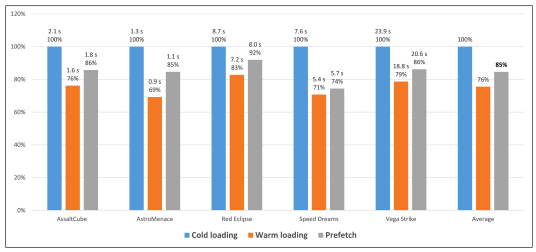


그림 5 로딩 프리페칭 실험 결과

Fig. 5 Experimental results of loading prefetching

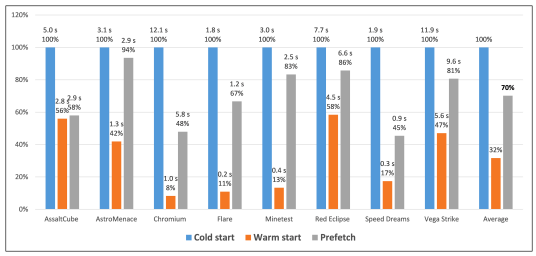


그림 6 기동 프리페칭 실험 결과

Fig. 6 Experimental results of start up prefetching

7 결론 및 앞으로 수행할 연구

본 논문에서는 프로그램 기동뿐만 아니라 실행 도중 발생하는 로딩까지 처리할 수 있는 브레이크포인트 기반 프리페칭 기법을 제안하였다. 본 논문의 가장 큰 의의는 프리페칭 대상 프로그램이 브레이크포인트를 사용하여 프리페쳐에게 힌트를 전달함으로써, 프리페칭 시점 분석에 대한 오버헤드를 상당히 낮춘 점이다. 또한 기존에 구현된 고성능 기동 프리페쳐를 재사용할 수 있도록 설계되었다.

실험결과 기동 시간은 평균 30% 단축되었고, 로딩 시간은 평균 15% 감소되었다.

향후 과제로는 소스코드의 제어흐름을 분석하여 브레이크포인트를 자동으로 삽입하는 기법에 대한 연구가 필요하다.

References

[1] Y. Joo, J. Ryu, S. Park, and K. G. Shin, "FAST: Quick Application Launch on Solid State Drives," *FAST*, pp. 259 272, Feb. 2009.

[2] R. Micheloni, L. Crippa, and M. Picca, "Hybrid Storage," *Inside Solid State Drives (SSDs)*, pp. 61 77, Springer, 2013.

[3] T. Coughlin, "Evolving Storage Technology in Con sumer Electronic Products," *IEEE Consum. Electron. Mag.*, Vol. 2, No. 2, pp. 59 63, Apr. 2013.

[4] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang, "DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch," *2007 USENIX Annu. Tech. Conf. Proc. USENIX Annu. Tech. Conf.*, pp. 261 274, 2007.

[5] M. E. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals Part2*, 6th Ed., Microsoft Press, 2012.

[6] K. Lichota. (2007, Sep. 14). Linux solution for prefet ching necessary data during application and system startup. [Online]. Available: http://code.google.com/p/prefetch/ (downloaded 2014, May. 7)

[7] Junhee Ryu, "Reducing Application Launch Time by Using Execution time Prefetching Techniques," Seoul National University, 2013. (in Korean)

[8] F. Chang and G. A. Gibson, "Automatic I/O Hint Generation Through Speculative Execution," *Pro ceedings of the Third Symposium on Operating Systems Design and Implementation*, pp. 1 14, 1999.

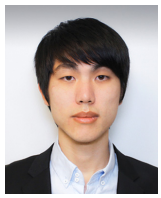
[9] Z. Li, Z. Chen, S. M. Srinivasan, and Y. Zhou, "C Miner: Mining Block Correlations in Storage Systems," *FAST*, pp. 173 186, 2004.

[10] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis, "BORG: Block reORGanization for Self optimizing Storage Systems," *Proccedings of the 7th Conference on File and Storage Technologies*, pp. 183 196, 2009.

[11] T. C. Mowry, A. K. Demke, and O. Krieger, "Automatic Compiler inserted I/O Prefetching for Out of core Applications," *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, pp. 3 17, 1996.

[12] S. VanDeBogart, C. Frost, and E. Kohler, "Reducing Seek Overhead with Application directed Prefet ching," *Proceedings of the 2009 Conference on USE NIX Annual Technical Conference*, pp. 299 312, 2009.

[13] Speed Dreams. [Online]. Available: http://www.speed dreams.org/ (downloaded 2014, May. 7)

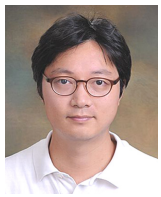


고 광 진

2011년 세종대학교 컴퓨터공학과(학사) 2014년 서울대학교 전기컴퓨터공학부(석사). 2014년~현재 다산네트웍스 개발본부 연구원. 관심분야는 운영체제, 센서 네트워크

유 준 희

2003년 한국항공대학교 컴퓨터공학과(학사). 2005년 서울대학교 전기컴퓨터공학부(석사). 2013년 서울대학교 전기컴퓨터공학부(박사). 2013년~현재 삼성전자 네트워크사업부 책임연구원. 관심분야는 운영체제, 가상화



강 경 태

1999년 서울대학교 수학계산통계학과군 전산과학전공(학사). 2001년 서울대학교 전기컴퓨터공학부(석사). 2007년 서울대학교 전기컴퓨터공학부(박사). 2008년 PostDoc. Res. Ass. Coordinated Science Lab., UIUC. 2010년 PostDoc. Res. Ass. Dept. Comupter Science, UIUC. 2011년~현재 한양대학교 컴퓨터공학과 조교수. 관심분야는 사이버 피지컬 시스템, 모바일 컴퓨팅

신 현 식

1973년 서울대학교 응용물리학과(학사) 1978년 미국 서던일리노이대학교 생물학(학사). 1980년 미국 텍사스대학교(오스틴) 의공학과(석사). 1985년 미국 텍사스대학교(오스틴) 전기컴퓨터공학과(박사) 1986년~현재 서울대학교 컴퓨터공학부 교수. 관심분야는 실시간 내장형 시스템, 모바일 컴퓨팅, 운영체제