논문 2008-03-21

내장형 시스템을 위한 PMU (Performance Monitoring Unit) 기반 동적 XIP (eXecute In Place) 기법

(PMU (Performance Monitoring Unit)-Based Dynamic XIP (eXecute In Place) Technique for Embedded Systems)

김 도 훈, 박 찬 익*
(Dohun Kim, Chanik Park)

Abstract : These days, mobile embedded systems adopt flash memory capable of XIP feature since they can reduce memory usage, power consumption, and software load time. XIP provides direct access to ROM and flash memory for processors. However, using XIP incurs unnecessary degradation of applications' performance because direct access to ROM and flash memory shows more delay than that to main memory. In this paper, we propose a memory management framework, dynamic XIP, which can resolve the performance degradation of using XIP. Using a constrained RAM cache, dynamic XIP can dynamically change XIP region according to page access pattern to reduce performance degradation in execution time or energy consumption resulting from native XIP problem. The proposed framework consists of a page profiler gathering applications' memory access pattern using PMU and an XIP manager deciding that a page is accessed whether in main memory or in flash memory. The proposed framework is implemented and evaluated in Linux kernel. Our evaluation shows that our framework can reduce execution time at most 25% and energy consumption at most 22% compared with using XIP-only case adopted in general mobile embedded systems. Moreover, the evaluation shows that in execution time and energy consumption, our modified LRU algorithm with code page filters can reduce more than at most 90% and 80% respectively compared with applying just existing LRU algorithm to dynamic XIP.

Keywords : eXecute-In-Place, XIP, Dynamic XIP, PMU-based XIP

I. 서 론

모바일 단말과 같은 내장형 시스템에 탑재되는 응용은 다양한 기능이 요구됨에 따라 코드의 크기가 증가하고 있는 추세이다. 따라서 응용을 실행하기 위해서는 기존보다 많은 메모리가 요구되며, 이를 지원하기 위해서는 다이(Die) 크기와 소비전력이 증가하게 된다 [1].

* 교신저자 (Corresponding Author)
논문접수 : 2008. 8. 31., 채택확정 : 2008. 9. 23.
김도훈, 박찬익 : 포항공과대학교 컴퓨터공학과
※ 본 연구는 대학 IT 연구센터 육성지원사업의 연구결과로써 HY-SDR연구센터의 연구비 지원으로 수행되었음.

일반적으로 내장형 시스템에서 응용을 수행하는 방법은 코드 섀도우잉(Code Shadowing)과 XIP로 나뉜다. 코드 섀도우잉은 수행할 응용을 램(RAM)과 같은 메모리에 모두 미리 적재하거나 혹은 수행에 필요한 페이지만을 요구 페이징 기법에 따라 메모리에 적재하여 수행하는 방법이다. 즉, 응용의 코드 및 데이터 페이지에 대한 모든 접근을 메모리에서 해결하는 방식이다. XIP는 메모리를 거치지 않고도 CPU가 ROM, 플래시 메모리 등의 저장장치 데이터를 직접 접근할 수 있는 기법이다. 특히 플래시 메모리를 탑재한 모바일 내장형 시스템은 다이 크기, 소비전력, 메모리 사용량, 소프트웨어 적재 시간 등의 감소를 위해 XIP 방식을 사용하고 있다. 그러나 플래시 메모리는 현재 기술의 한계로 인해 램과 같은 메모리와는 달리 접근성능이 떨어지므로, XIP를

통한 데이터 접근은 추가 지연시간이 발생하게 된다. 따라서 소프트웨어의 수행시간이 증가한다. 특히 메모리 접근시간 증가는 하드웨어 특성상 CPU의 스톨 사이클(stall cycle) 증가를 의미하므로 응용의 CPU 및 주변 장치 사용시간이 증가하여 불필요한 추가 에너지 소모가 발생한다. 그림 1은 리눅스 환경에서 8MB 크기의 동영상을 Mplayer [2]로 플레이 했을 때 코드 섀도우잉과 XIP의 성능 차이를 보이고 있다. 그림에서 XIP의 성능이 수행시간이나 에너지 소모량 측면에서 코드 섀도우잉에 비해 약 17%정도 저하됨을 알 수 있다. 따라서 XIP 사용시 발생하는 응용의 성능저하를 감소시킬 수 있는 방안이 필요하다.

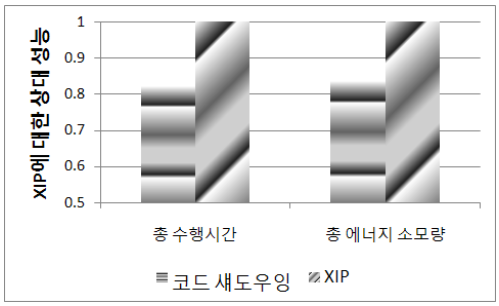

그림 1. 코드 섀도우잉과 XIP의 성능 차이
(리눅스 환경에서 Mplayer를 통해 8 MB 동영상을 플레이한 경우)
Fig. 1. Performance comparison between Code Shadowing and XIP
(This shows the performance when Mplayer plays an 8 MB movie in Linux)

PMU는 CPU 수준에서 발생하는 하드웨어 이벤트를 수집하여 소프트웨어의 성능을 측정할 수 있는 하드웨어 장치이다. 모바일 내장형 시스템에서 주로 사용되고 있는 ARM, XScale 등의 프로세서에서는 성능 분석을 위해 PMU를 제공하고 있다. 특히 PMU를 통해 측정할 수 있는 이벤트 중 캐시 미스는 CPU가 캐시 라인을 채우기 위해 메모리 접근을 시도하는 이벤트로써 CPU의 메모리 접근 수를 측정할 수 있다. 즉, PMU를 이용하여 응용의 메모리 접근 패턴을 프로파일링할 수 있다. 그러나 프로파일링시 PMU를 관리하기 위한 오버헤드가 발생하므로 성능감소를 최소화하면서도 높은 정확성을 가지도록 프로파일링 성능을 결정해야 한다.

본 논문에서 제안하는 동적 XIP는 XIP의 성능 향상을 위해 XIP로 접근되는 페이지 영역을 동적으로 변환하는 기법이다. 즉, XIP로 접근되는 응용의 코드 페이지들 중 일부 페이지를 접근 패턴에 따라 동적으로 선택하여 램에 캐싱함으로써 XIP로 인한 응용의 수행시간 및 에너지 소모량 증가를 감소시킨다. 일반적으로 XIP를 사용하는 내장형 시스템은 램 용량에 제한이 있어 캐시로 사용할 수 있는 램 용량이 제한적일 수 밖에 없다. 따라서 제한된 램 캐시를 이용하여 XIP의 수행시간 및 에너지 소모 관련 성능을 높이기 위해서는 페이지에 대한 접근 패턴 분석 및 분석에 기반한 램 캐시 관리가 동적 XIP의 중요한 문제이다.

본 논문은 동적 XIP 지원을 위한 메모리 관리 프레임워크를 제안하였다. 제안된 프레임워크는 PMU를 통해 응용의 메모리 접근 패턴을 분석하는 페이지 프로파일러와 분석된 패턴에 따라 CPU의 메모리 페이지에 대한 접근을 램의 영역 혹은 플래시 메모리 XIP 영역으로 동적으로 결정하는 XIP 관리자를 포함하고 있다.

본 논문은 제안된 프레임워크의 성능을 평가하기 위해 리눅스 커널을 확장하여 구현하였으며, 대표적인 내장형 응용들에 대한 실험을 통해 응용의 코드 크기에 비해 작은 캐시를 사용하고도 기존 XIP에 비해 수행시간 면에서는 최대 25%, 에너지 소모량에서는 최대 22%의 성능향상을 보였다. 또한 기존 LRU 캐시 알고리즘을 단순히 동적 XIP에 적용했을 때에 비해 제안된 프레임워크를 적용했을 때 수행시간 면에서 최대 90%, 에너지 소모량에서 최대 80%의 성능이 향상됨을 보였다.

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구에 대해 간략하게 설명한다. 3장에서는 본 논문에서 제안한 프레임워크에 대해 설명한다. 4장은 제안된 프레임워크의 성능 결과를 설명한다. 마지막으로 5장에서는 결론을 맺는다.

II. 관련 연구

모바일 내장형 시스템에서 XIP 기법은 플래시 메모리에 주로 사용되고 있으며, 그 중에서도 노어(NOR) 플래시 메모리에서 주로 사용되고 있다. 최근 가격대 성능비가 우수한 낸드(NAND) 플래시 메모리에서도 XIP를 지원하기 위해 특별한 하드웨어가 고안되었다 [3].

플래시 메모리에 저장된 응용을 XIP 기법으로 접근하기 위해서는 파일 시스템에서 이를 지원해야 한다. CELinux의 CramFS[4]는 파일 단위의 XIP를

지원하는 파일 시스템이다. 파일이 압축되어 있으면 CPU에서 파일 데이터의 페이지를 직접 접근할 수 없으므로 기존 CramFS와는 달리 XIP로 접근될 파일을 압축하지 않는 형태로 파일시스템에 저장한다. 따라서 기존 CramFS와는 달리 파일시스템의 크기가 커지는 단점을 가지고 있다. AXFS[5]는 페이지 단위로 XIP를 지원하는 파일 시스템이다. 즉, 하나의 파일 내에서도 XIP로 접근하는 페이지를 따로 설정할 수 있다. XIP로 설정되지 않은 페이지는 CramFS와 비슷한 방법으로 압축된 형태로 저장된다. 따라서 AXFS는 CramFS보다는 작은 크기의 파일시스템을 구성할 수 있다. 그러나 AXFS는 어떤 페이지를 XIP로 접근할지에 대한 기준을 제시하지 못하고 있다.

본 논문에서 제안된 내용과 같이 OneNAND상에서 동적으로 XIP를 지원하기 위한 연구도 진행되었다[6]. OneNAND 플래시 메모리는 XIP를 지원하는 낸드 플래시 메모리의 상용화 제품이다. 이 연구에서는 요구 페이징시 발생하는 페이지 복사 오버헤드를 최소화하기 위해 동적으로 XIP 기능을 적용하였다. 그러나 이 연구는 OneNAND라는 특수한 하드웨어 환경을 가정하고 있을 뿐만 아니라, 페이지 복사 오버헤드를 줄이기 위한 구체적인 기준을 제시하지 않고 있다.

본 논문은 기존 XIP 관련 연구와는 달리 다음과 같은 특징을 가지고 있다. 1) XIP를 동적으로 적용하기 위한 기준을 제시하고 있다. 2) 동적 XIP 기법을 실제 시스템에 구현하여 제안된 프레임워크의 유용성을 보였다. 3) 특정 하드웨어가 아닌 일반적인 환경에서 적용하여 사용할 수 있다.

Ⅲ. 제안된 동적 XIP 프레임워크

1. 기본 아이디어

본 논문은 XIP 적용시 저하되는 응용의 수행 성능을 개선하기 위해 접근 빈도가 높은 응용의 일부 페이지를 코드 섀도우잉으로 접근하는 방법을 사용한다. 즉, 응용의 일부 페이지는 램을 통해 접근하고, 나머지는 플래시 메모리의 XIP 영역에서 접근하는 방법이다. 그림 2는 그림 1에서 사용된 Mplayer의 코드 페이지 중 가장 많이 접근되는 코드 페이지를 5개 선택하여 코드 섀도우잉으로 접근하도록 정적으로 설정했을 때 측정된 성능이다. 즉, 부분적으로만 XIP를 적용한 경우에 측정된 성능이다. 선택된 코드 페이지 수는 Mplayer의 전체 코드 페이지 중 약 0.8%에 해당되는 수이다. 따라서 적은 수의 코드 페이지만을 코드 섀도우잉하여 XIP의 성능을 향상할 수 있음을 보였다.

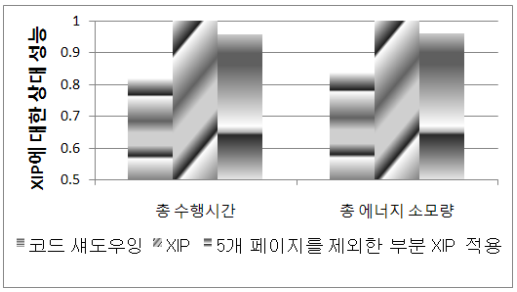

그림 2. 리눅스 Mplayer의 코드 페이지 중 5개만을 코드 섀도우잉한 경우 성능 비교

Fig. 2. Performance comparison between Fig. 1 and Mplayer with code-shadowed five pages

XIP를 사용하는 장점 중 하나는 메모리 사용량을 줄이는데 있다. 따라서 XIP 성능 향상을 위해 무제한적으로 모든 페이지에 대해 코드 섀도우잉을 하는 경우가 없어야 한다. 즉, 제한된 메모리 공간만을 사용하여 코드 섀도우잉을 지원해야 한다. 그러나 이 문제는 XIP로 수행중인 응용의 페이지 중 어떤 페이지를 코드 섀도우잉으로 수행할 것인지에 대한 문제를 야기할 수 있다. 그림 3은 리눅스용 모바일 응용인 Iout[7]을 수행했을 때, 각 코드 페이지에 대한 접근을 시간 순으로 표시한 것이다. 그림에서 알 수 있듯이 코드 페이지에 대한 접근은 시간대에 따라 달라질 수 있으므로, 어떤 페이지를 코드 섀도우잉할 것인지를 결정하는 것은 동적으로 결정해야 한다. 즉, 어떤 페이지가 현재 시간대에서 코드 섀도우잉으로 접근되더라도, 다음 시간대에서 접근 빈도수가 떨어지면, 다시 플래시 메모리의 XIP 영역에서 접근되도록 수정해야 한다.

2. 동적 XIP 프레임워크 구조

그림 4는 본 논문에서 제안된 동적 XIP 프레임워크 구조를 나타내고 있다. 제안된 프레임워크는 크게 코드 페이지 프로파일러와 XIP 관리자로 구성되어 있다. 코드 페이지 프로파일러는 PMU를 통해 응용의 각 코드 페이지에 대한 메모리 접근 정보를 수집한 후, 윈도우 범위 내에서 각 코드 페이지에 대한 접근 패턴을 분석한다. 그리고 분석된 내용을 XIP 관리자에 전달한다. XIP 관리자의 코드 페이지 필터는 분석된 코드 페이지의 패턴 정보를 재분석하여 코드 섀도우잉시 응용의 성능을 향상시킬 수

있는 코드 페이지를 걸러낸다.

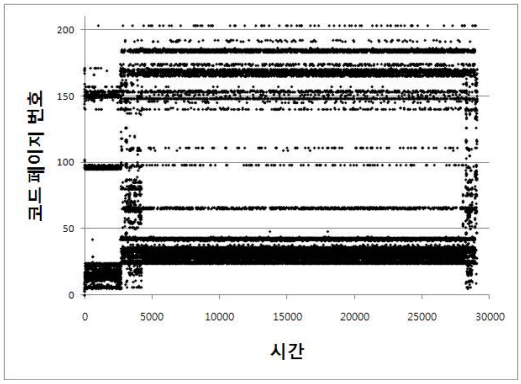

그림 3. 응용의 각 코드 페이지에 대해 시간에 따른 접근 모습(리눅스 lout 응용)

Fig. 3. Page access pattern as time passes (Linux lout case)

그리고 XIP를 위한 LRU 알고리즘은 응용에게 할당된 코드 섀도우잉용 캐시 메모리 공간에서 성능 향상에 기여할 수 없는 페이지를 선택한다. 이때 선택된 페이지는 다시 플래시 메모리의 XIP 영역에서 접근되도록 페이지 테이블의 내용을 수정한다. 만약 새로 접근된 페이지가 이미 코드 섀도우잉된 페이지들에 비해 응용의 성능을 향상시킬 수 없을 경우, 현재의 캐시 상태를 그대로 유지하게 된다.

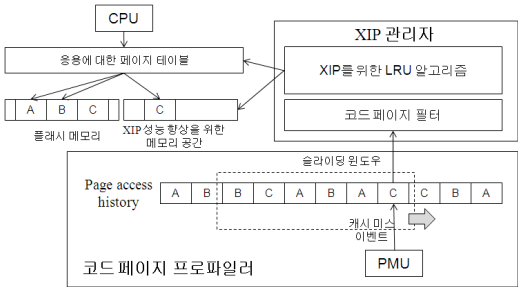

그림 4. 제안된 동적 XIP 프레임워크 구조

Fig. 4. The proposed framework for Dynamic XIP

3. 코드 페이지 프로파일러

3.1 PMU를 통한 메모리 접근 이벤트 수집

코드 페이지 프로파일러는 응용의 코드 페이지에 대한 접근 패턴을 분석하기 위해 PMU의 캐시 미스 이벤트를 수집한다. 일반적으로 PMU는 소프트웨어에서 이벤트를 수집할 기회를 제공하기 위해 PMU 인터럽트를 발생시킨다. 캐시 미스가 발생하는 순간마다 PMU 인터럽트를 발생시켜 이벤트를 수집하는 방법은 응용의 메모리 접근 정보를 가장 정확하게 프로파일링할 수 있는 방법이기는 하나, 많은 인터럽트 처리 오버헤드를 발생하므로 그대로 사용할 수 없다. 따라서 PMU는 인터럽트 오버헤드를 줄이기 위해 프로파일링 하고자 하는 특정 이벤트의 숫자를 누적하여 저장해 둘 수 있는 카운터 레지스터를 제공하고 있다. 따라서 특정 이벤트가 특정 숫자만큼 발생했을 때 PMU 인터럽트를 발생하도록 설정할 수 있다. 그러나 이 방법은 프로파일링 정확도가 떨어지는 단점을 가지고 있다. 따라서 프로파일링 오버헤드를 최소화하면서도 정확도를 높이도록 PMU 인터럽트를 발생시키는 이벤트 수를 결정해야 한다. 본 논문은 실험을 통해 적절한 PMU 인터럽트 빈도수를 결정하였고, 이를 각 실험에 적용하였다. 구체적인 내용은 4장에서 서술한다.

3.2 코드 페이지 접근 패턴 분석 방법

코드 페이지에 대한 접근 패턴 정보는 수집된 캐시 미스 이벤트 정보를 슬라이딩 윈도우 형태로 관리한다. 따라서 수집된 윈도우 내에서 특정 페이지에 대한 접근 수를 분석한다. 윈도우의 크기는 코드 페이지의 접근 패턴 분석에 영향을 주고, 이는 다시 분석 정보를 기반으로 동작하는 XIP 관리자의 성능에 영향을 줄 수 있다. 그러나 성능 향상을 위해 최적의 윈도우 크기를 결정하는 것은 온라인 알고리즘에서 풀기 어려운 문제이다. 따라서 본 논문은 윈도우 크기에 따른 성능 최적화 문제는 논외로 한다. 다만, 실험을 통해 윈도우 크기에 따른 성능 결과를 4장에 제시하였다.

표 1. 코드 페이지 필터를 위한 파라메터

Table 1. Parameters for code page filters

파라메터	설명
A_i	코드 페이지 i에 접근 수
T_R	램 접근 시간
T_F	플래시 메모리 접근 시간
T_{CP}	플래시에서 메모리로의 페이지 복사 시간
T_{RTB}	램 영역 페이지의 테이블 설정 시간
T_{FTB}	XIP 영역 페이지의 테이블 설정 시간
P_{CPU}	CPU 소비전력
P_{AR}	Active 상태의 램 소비전력
P_{IR}	Idle 상태의 램 소비전력
P_{AF}	Active 상태의 플래시 메모리 소비전력
P_{IF}	Idle 상태의 플래시 메모리 소비전력

4. XIP 관리자

4.1 수행시간 및 에너지 소모량 감소 가능 페이지 선택을 위한 코드 페이지 필터

코드 페이지 필터는 코드 섀도우잉시 응용의 성능 향상에 도움이 되는 새로운 코드 페이지를 선택하는 역할을 한다. 응용의 성능을 향상시킬 수 있는 코드 페이지를 선택하기 위해는 이를 구별할 수 있는 기준이 필요하다. 응용의 성능은 크게 수행시간과 에너지 소비량으로 나눌 수 있다. 따라서 코드 페이지를 선택하기 위한 기준도 두 가지로 나누어 존재해야 한다. 표 1은 수행시간과 에너지 소비량 면에서 페이지를 선택하기 위한 기준을 계산하는데 필요한 파라메터들을 나열한 것이다.

우선 수행시간 측면에서 어떤 코드 페이지(현재 XIP로 접근되는 페이지)가 이미 코드 섀도우잉된 페이지에 비해 응용의 성능을 향상시킬 수 있는지 검사할 수 있는 계산식을 설명한다. 현재 프로파일러를 통해 메모리 접근이 측정된 페이지를 i, 코드 섀도우잉된 페이지 중 성능향상 기여도가 가장 낮은 페이지를 j라 하자. 우선 아무 변화없이 현재 상태를 유지할 때 예상되는 관련 페이지들의 총 메모리 접근 시간(T^{norep})은 다음과 같다.

$$T^{norep} = A_j \times T_R + A_i \times T_F$$

다음으로 만약 페이지 j를 페이지 i로 교체할 경우 예상되는 관련 페이지들의 총 메모리 접근 시간(T^{rep})은 다음과 같다. 여기에서 오른쪽 세 개의 파라메터들은 요구 페이징에 대한 오버헤드를 의미한다.

$$T^{rep} = A_j \times T_F + A_i \times T_R + T_{CP} + T_{RTB} + T_{FTB}$$

따라서 다음과 같은 조건을 만족하면 페이지 j를 페이지 i로 교체한다. 즉, 페이지 i를 코드 섀도우잉할 경우 응용의 성능향상이 예상된다.

$$T^{norep} \geq T^{rep}$$

$$A_i - A_j \geq \frac{T_{CP} + T_{RTB} + T_{FTB}}{T_F - T_R} = TH_{access}$$

일반적으로 내장형 시스템에서 요구 페이징 오버헤드와 램 및 플래시 메모리 접근 시간은 알려져 있다. 따라서 TH_{access}는 특정값으로 고정할 수 있다. 이 수식의 의미는 페이지 i의 메모리 접근 수가 페이지 j의 접근 수 보다 TH_{access}이상 많을 경우 코드 섀도우잉을 시키면 응용의 수행시간 성능이 향상될 수 있음을 의미한다.

다음으로 에너지 소비량 측면에서 어떤 코드 페이지가 코드 섀도우잉된 페이지에 비해 응용의 성능을 향상시킬 수 있는지 검사할 수 있는 계산식을 살펴 본다. 앞의 기준 계산식과 마찬가지로 현재 프로파일러를 통해 메모리 접근이 측정된 페이지를 i, 코드 섀도우잉된 페이지 중 더 이상 성능향상을 기대할 수 없는 페이지를 j라 하자. 그러면, 아무 변화없이 현재 상태를 유지할 때 예상되는 관련 페이지들의 총 에너지 소비량(E^{norep})은 다음과 같다.

$$P^1 = P_{CPU} + P_{AR} + P_{IF}$$

$$P^2 = P_{CPU} + P_{IR} + P_{AF}$$

$$E^{norep} = A_j \times T_R \times P^2 + A_i \times T_F \times P^1$$

앞의 식에서 P^1과 P^2는 각각 램 접근시 소모되는 전력과 플래시 메모리 접근시 소모되는 전력을 의미한다.

요구 페이징를 처리하기 위해 필요한 에너지 소비량(E^{demand})을 계산하면 다음과 같다.

$$E^{demand} = T_{CP} \times (P_{CPU} + P_{AR} + P_{AF}) + T_{RTB} \times P^1 + T_{FTB} \times P^1$$

이제 만약 페이지 j를 페이지 i로 교체할 경우 예상되는 관련 페이지들의 총 에너지 소비량(E^{rep})은 다음과 같다.

$$E^{rep} = A_j \times T_F \times P^1 + A_i \times T_R \times P^2 + E^{demand}$$

따라서 다음과 같은 조건을 만족하면 페이지 j를 페이지 i로 교체한다. 즉, 페이지 i를 코드 섀도우잉할 경우 응용의 에너지 소모 측면에서 성능 향상이 예상된다.

$$E^{norep} \geq E^{rep}$$

$$A_i - A_j \geq \frac{E^{demand}}{T_F \times P^1 - T_R \times P^2} = TH_{energy}$$

4.2 XIP를 위한 LRU 알고리즘

XIP를 위한 LRU 알고리즘은 코드 섀도우잉을 위해 할당된 메모리 공간을 효율적으로 관리하기 위해 필요하다. 사용된 LRU 알고리즘은 페이지들을

LRU 리스트로 관리한다는 면에서는 기존의 LRU 알고리즘과 큰 차이가 없다. 그리고 본 논문에서는 응용의 성능향상에 기여도가 낮은 페이지로써 LRU 페이지를 선택하도록 하였다. 일반적으로 LRU 페이지는 향후 다시 접근할 가능성이 낮은 페이지이고, LRU 알고리즘에 드는 캐시 관리 오버헤드가 적기 때문이다. 사용된 LRU 알고리즘은 새로운 페이지에 대한 코드 섀도우잉 결정을 위해 코드 페이지 필터를 사용한다는 점에서 기존 LRU 알고리즘에 새로운 단계를 추가한 것으로 간주할 수 있다.

만약 새로운 페이지가 교체 대상 페이지보다 응용의 성능향상도가 낮다고 판단될 경우, 기존 LRU 알고리즘과 달리 페이지 교체를 하지 않는다. 따라서 XIP를 위한 LRU 알고리즘은 기존 알고리즘에 비해 낮은 캐시 적중율을 보인다. 그러나 캐시 적중율을 희생하는 대신 코드 페이지 필터를 통해 불필요한 페이지 교체수를 감소시키므로 응용 수준의 성능은 개선되는 효과를 보이게 된다.

4.3 일부 파라메터에 대한 고려사항

코드 페이지 필터에서 사용하는 파라메터 중 램과 플래시 메모리 접근 시간은 이상적으로는 모든 응용에 대해 동일하게 나타나야 한다. 그러나 실제 환경에서는 응용에 따라 다른 접근 시간을 보일 수 있다. 예를 들어 CPU에서 사용하는 시스템 버스는 캐시 뿐만 아니라 메모리 접근을 위해 사용되는 하드웨어이다. CPU-bound 응용의 경우 시스템 버스를 거의 사용하지 않지만, I/O-bound 응용의 시스템 버스를 자주 사용하게 되므로 메모리 접근 시간이 늘어날 수 있다. 따라서 각 응용에 적합한 파라메터 값을 선택해야 한다. 이는 PMU를 통해 온라인으로 일부 시간구간동안 측정하거나 혹은 모든 시간구간동안 측정하여 사용할 수 있다. 본 논문은 각 응용을 위한 파라메터 값을 모든 시간구간동안 측정하여 각 실험에 사용하였다. 각 응용 별로 측정된 파라메터는 4장에 서술하였다.

표 2. 실험에 사용된 응용
Table 2. Applications for workload

응용	설명(코드 페이지 수)
Mplayer	17MB mpeg4 동영상(627개)
lout	8KB 텍스트에서 PS 변환(225개)
Ghostscript	250KB PS 파일 데이터 접근(286개)
GSM	1MB 데이터를 GSM 인코딩(63개)

IV. 성능 평가

1. 실험환경

표 3. 각 응용별 메모리 접근시간 관련 파라메터
Table 3. Memory access time for each application

응용	측정된 파라메터	파라메터 값
Mplayer	T_R	0.54 마이크로초
	T_F	1.47 마이크로초
lout	T_R	0.58 마이크로초
	T_F	1.97 마이크로초
Ghostscript	T_R	0.66 마이크로초
	T_F	1.59 마이크로초
GSM	T_R	0.56 마이크로초
	T_F	1.79 마이크로초

본 논문은 하드웨어 환경으로써 인텔 PXA255 프로세서, 128MB SDRAM, 64MB 인텔 노어 플래시 메모리를 탑재한 임베디드 보드를 사용하였으며[8], 전력 측정을 위해 NI사의 PCI-4070 디지털 멀티미터를 사용하였다[9]. 그리고 모든 실험은 리눅스에서 멀티미디어 응용으로 사용되고 있는 Mplayer[2]와 MiBench[7]에서 제공하는 리눅스용 응용인 lout, Ghostscript, GSM을 사용하여 진행되었다(표 2). 그리고 4.3절에서 언급한 파라메터는 표3과 같다.

2. 코드 페이지 프로파일링 성능

그림 5, 6은 코드 페이지 프로파일링을 위해 선택한 PMU 인터럽트의 발생 설정에 따른 오버헤드와 프로파일링 정확도를 나타낸 것이다. 여기에서 샘플링 크기(Sampling Granularity)는 PMU 인터럽트를 발생하기 위한 누적된 캐시 미스 이벤트 수를 의미한다. 성능 대비 정확도가 비교적 안정되는 시점인 113개를 선택하여 다음 실험들에서 사용하였다.

3. 기존 LRU 알고리즘과의 성능 비교

그림 7,8은 기존 LRU 알고리즘을 동적 XIP 환경에서 코드 페이지 필터 없이 적용한 경우와 코드 페이지 필터를 사용하도록 수정한 경우의 성능을 나타내었다. 실험에서 모든 파라메터와 응용에 대해 동일한 경향을 보였으므로, 그 중에서 캐시 미스 수집 윈도우 크기가 500개이고, lout 응용을 수행했을 경우만 표시하였다.

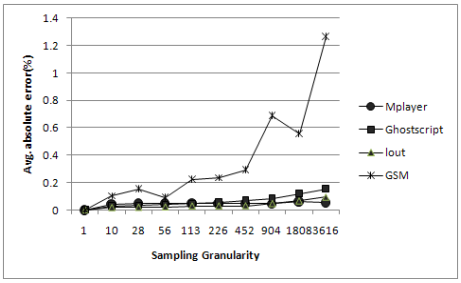

그림 5. 프로파일링 오버헤드 측정
Fig. 5. Profiling overhead

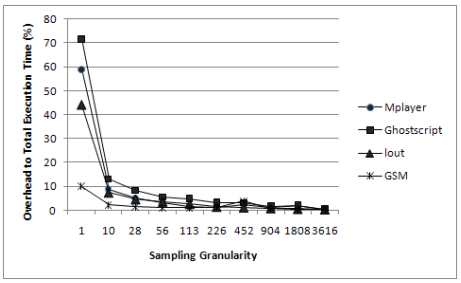

그림 6. 프로파일링 정확도 측정
Fig. 6. Profiling accuracy

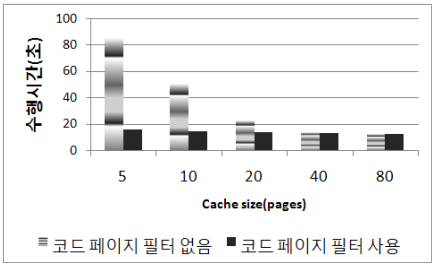

그림 7. 수행 시간 비교
Fig. 7. Execution time

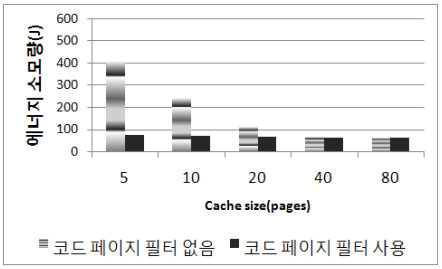

그림 8. 에너지 소모량 비교
Fig. 8. Energy consumption

4.2절에서 서술했듯이 본 실험의 결과는 코드 페이지 필터 기법이 응용의 캐시 적중률은 다소 감소시키나(그림 9), 응용의 성능을 크게 저하시키는 불필요한 요구 페이징 수를 줄이므로(그림 10) 실질적으로 응용의 수행시간을 최대 90%, 에너지 소모량을 최대 80% 향상시켰음을 보이고 있다.

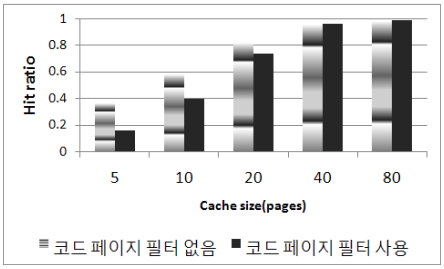

그림 9. 캐시 적중률 비교
Fig. 9. Cache hit ratio

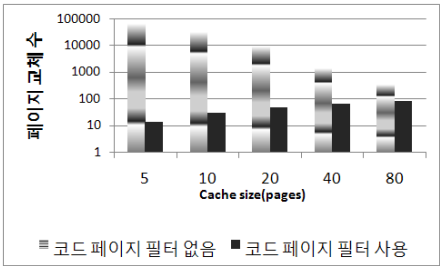

그림 10. 요구 페이징 수 비교
Fig. 10. The number of page replacements

4. 기존 XIP와 동적 XIP의 성능 비교

그림 11, 12, 13, 14는 각 응용에 대해 기존 XIP와 동적 XIP의 성능을 비교한 것이다. 동적 XIP는 수행시간 감소를 위한 코드 페이지 필터와 에너지 소모량 감소를 위한 코드 페이지 필터를 각각 적용했을 때 측정된 데이터이다. 표에서 수가 표시되지 않은 항목은 성능향상이 없었던 부분을 나타낸다. 그림 11과 13의 응용은 코드 페이지에 대한 프로파일링 결과 각각 초당 메모리 접근 수가 가장 많은 응용과 적은 응용이었다. 따라서 메모리 접근 수가 가장 많은 응용인 lout이 동적 XIP를 적용했을 때 약 22% 정도의 가장 높은 성능 향상을 보였다. 현재 실험에서는 캐시 미스 수집 윈도우 크기가 대체적으로 성능에 큰 영향을 보이지 않는 것으로 나타나고 있으나, 이에 대한 체계적인 추가 연구가 필요하다. 이는 향후 연구 내용으로 진행될 예정이다.

	수행시간					에너지 소모량				
윈도우	Cache size					Cache size				
크기	5	10	20	40	60	5	10	20	40	60
50	0.15	3.75	7.06	13.16	14.35	0.23	2.03	6.84	11.83	13.59
100	0.65	3.34	8.67	12.92	14.65	0.27	3.36	7.06	11.79	13.64
200	0.91	3.40	7.44	12.85	15.17	0.66	3.08	6.79	11.45	13.66
500	1.13	3.90	7.54	13.14	14.79	0.64	3.50	7.58	11.84	13.54
1000	0.58	3.75	8.03	12.83	14.91	0.10	2.45	7.60	11.92	13.57
2000	1.34	3.56	8.83	13.66	15.68	1.02	3.17	7.50	12.10	13.95
5000	0.84	4.45	8.61	12.94	15.58	0.58	2.49	7.43	12.07	13.46

그림 11. XIP와의 성능 비교: Mplayer
Fig. 11. Performance comparison with XIP: Mplayer

	수행시간					에너지 소모량				
윈도우	Cache size					Cache size				
크기	5	10	20	40	60	5	10	20	40	60
50	1.83	10.79	19.08	22.77	24.77	3.72	9.49	16.12	20.44	21.07
100	2.74	10.49	19.56	23.01	25.01	4.29	11.17	17.31	19.93	21.54
200	2.19	10.49	19.26	23.50	24.95	3.39	10.50	17.62	20.55	22.05
500	4.31	11.09	19.99	23.56	25.01	3.83	10.90	18.18	21.05	22.04
1000	4.43	11.33	19.44	23.80	25.01	6.19	10.48	16.85	20.26	22.85
2000	4.25	11.70	19.32	23.56	24.71	3.41	10.25	16.90	20.34	21.74
5000	4.07	11.51	18.05	22.47	25.50	3.96	9.97	15.66	19.53	21.93

그림 12. XIP와의 성능 비교: lout(메모리 접근이 가장 많은 응용)
Fig. 12. Performance comparison with XIP: lout (This application shows the most memory accesses)

	수행시간					에너지 소모량				
윈도우	Cache size					Cache size				
크기	5	10	20	40	60	5	10	20	40	60
50	—	0.88	2.60	4.87	6.60	—	—	1.77	4.38	5.62
100	—	0.93	3.20	4.87	6.65	—	0.00	1.85	4.02	5.39
200	—	0.93	3.44	5.22	6.70	—	0.71	2.35	3.86	5.01
500	—	0.97	3.05	5.42	6.70	—	0.15	1.99	4.09	6.05
1000	—	1.12	3.34	5.52	6.90	—	0.75	2.45	4.31	5.29
2000	—	0.83	3.30	5.57	6.65	—	0.25	2.30	4.45	5.71
5000	—	1.22	3.49	5.81	6.85	—	0.55	2.58	4.17	5.39

그림 13. XIP와의 성능 비교: Ghostscript
Fig. 13. Performance comparison with XIP: Ghostscript

	수행시간			에너지 소모량		
윈도우	Cache size			Cache size		
크기	4	6	8	4	6	8
50	—	0.52	0.87	—	0.03	0.29
100	—	0.64	0.87	—	0.14	0.36
200	—	0.64	1.10	—	0.14	0.41
500	0.06	0.76	1.10	—	0.39	0.46
1000	0.06	0.87	1.10	—	0.47	0.58
2000	0.17	0.87	1.22	—	0.24	0.43
5000	0.06	0.99	1.22	—	0.14	0.41

그림 14. XIP와의 성능 비교: GSM(메모리 접근이 거의 없었던 응용)
Fig. 14. Performance comparison with XIP: GSM (This application shows the least memory accesses)

V. 결론

본 논문은 XIP를 사용하는 내장형 시스템에서 소프트웨어의 성능인 수행시간 혹은 에너지 소모관련 성능을 향상시키기 위한 메모리 관리 프레임워크인 동적 XIP 기법을 제안하였다. 제안된 프레임워크는 PMU를 통해 응용의 메모리 접근 패턴을 분석하는 페이지 프로파일러와 분석된 패턴에 따라 CPU의 페이지에 대한 접근을 램 영역 혹은 플래시 메모리 XIP 영역으로 동적으로 결정하는 XIP 관리자로 구성된다. 본 논문은 제안된 프레임워크의 성능을 평가하기 위해 리눅스 커널을 확장하여 구현하였으며, 대표적인 내장형 응용들에 대한 실험을 통해 응용의 코드 크기에 비해 작은 메모리 캐시를 사용하고도 기존 XIP에 비해 수행시간 면에서 최대 25%, 에너지 소모량에서 최대 22%의 성능향상을 보였다. 또한 기존 LRU 캐시 알고리즘을 단순히 동적 XIP에 적용했을 때에 비해 제안된 캐시 알고리즘이 수행시간 면에서 최대 90%, 에너지 소모량면에서 최대 80%의 성능을 향상시킴을 보였다.

본 논문은 일반적인 리눅스 환경에서 사용하는 4KB 크기의 페이지를 사용하여 진행되었다. 따라서 임베디드 환경에서 사용되는 보다 작은 크기의 페이지를 사용하게 될 경우 제안된 동적 XIP의 효율성이 높아질 것으로 예상되며, 이는 향후 연구로써 진행될 예정이다.

참고 문헌

[1] Jared Hulbert and Justin Treon, "Creating optimized XIP systems", CELF Embedded Linux Conference Presentation, 2006.
[2] Linux movie player, http://www.mplayerhq .hu
[3] Chanik Park, Jaeyu Seo, Sunghwan Bae, Hyojun Kim, Shinhan Kim and Bumsoo Kim, "A low-cost memory architecture with NAND XIP for mobile embedded systems", Proceedings of the International Conference on Hardware-So ftware Codesign and System Synthesis, 2003.
[4] XIP in ceLinux, http://tree.celinuxforum. org/CelfPubWiki/ApplicationXIP
[5] Advanced XIP Filesystem for Linux, htt p://axfs.so urceforge.net

[6] Yongsoo Joo, Yongseok Choi, Chanik Park, Sung Woo Chung, Eui-Young Chung and Naehy k Chang, "Demand paging for OneNAND flash eXecute-In-Place", Proceedings of the Internat ional Conference on Hardware-Software Codesi gn and System Synthesis, 2006.
[7] MiBench Version 1.0, a free, commercially representative embedded benchmark suite, Mat thew R. Guthaus and et al, IEEE 4th Annual Workshop on Workload Characterization, 2001.
[8] PXA255 기반 임베디드 보드, http://www.han back.co.kr
[9] National Instruments Digital Multimeter PCI-4070 and Labview 7.0, http://www.ni.com

저 자 소 개

김 도 훈 (Dohun Kim)

1998년 충남대학교 컴퓨터공학과 학사.
2001년 포항공대 컴퓨터공학과 석사.
2008년 포항공대 컴퓨터공학과 박사.
현재 포항공대 컴퓨터공학과 박사후연구원.
관심분야: 임베디드 소프트웨어, 운영체제, 실시간 시스템
Email: hunkim@postech.ac.kr

박 찬 익 (Chanik Park)

1983년 서울대학교 전자공학과 학사.
1985년 한국과학기술원 전자공학과(컴퓨터공학) 석사.
1988년 한국과학기술원 전자공학(컴퓨터공학)박사.
1988년~현재, 포항공대 컴퓨터공학과 교수.
관심분야: 지능형 스토리지 시스템, 내장형 실시간 운영체제, 시스템 보안, 가상화
Email: cipark@postech.ac.kr