DRAM 인터페이스 기반 SSD 메모리 장치 관리 기법

한규화 ○ 신동군
성균관대학교
Hgh6877@skku.edu, Dongkun@skku.edu

Memory Management Techniques for DRAM Interface-based SSD

Kyuhwa Han○ Dongkun Shin
Sungkyunkwan University

요 약

대용량의 데이터를 분석하는 응용이 확산되면서 서버에서는 많은 메모리를 필요로 하고 있으며, 이를 해결하기 위해 플래시메모리를 사용하여 메모리를 확장하는 방법에 대한 연구들이 진행되고 있다. 본 연구에서는 DRAM과 플래시메모리로 구성된 UMD(Unified Memory Device)라는 장치를 관리하는 방법을 제안한다. UMD는 내부적으로 작은 크기의 DRAM과 대용량의 플래시 메모리로 구성되고, 데이터 전체는 플래시메모리에 저장되며, DRAM은 데이터의 일부를 캐싱한다. 본 연구에서는 이러한 UMD의 특성을 고려하여 데이터가 플래시메모리에만 존재하여 UMD 내부에서 데이터의 이동으로 인해 응답 시간이 길어질 경우에 이를 호스트로 전달하여 CPU가 다른 작업을 수행할 수 있도록 하는 내부 페이지 폴트 처리 기법을 제안한다. 본 연구에서는 MARSS-x86과 DRAMSim을 수정하여, UMD의 동작을 평가하는 시뮬레이터를 제작하였다. Pigz를 사용하여 본 연구의 기법과 기존의 처리 기법의 성능을 평가한 결과, 45MB 파일의 압축을 푸는데 걸리는 시간이 최대 17% 줄어드는 것을 확인하였다.

1. 서 론

딥러닝과 빅데이터 분석 같은 대용량의 데이터를 사용하는 응용이 확산됨에 따라 서버에서는 많은 양의 메모리를 요구하고 있다. 최근에는 서버에서 요구하는 많은 메모리를 플래시메모리를 사용하여 보완하는 연구들[1,2]이 진행되고 있는데, 이러한 연구들에서는 DRAM과 플래시메모리로 구성된 하이브리드 형태의 메모리 시스템을 제안한다. 이는 플래시메모리가 DRAM보다 용량 대비 가격이 낮고 소비전력이 작아 대용량의 메모리를 구성하기에 적합하지만, DRAM에 비해 긴 응답 시간을 가지고 있으며 GC(Garbage Collection) 등의 내부 동작으로 인한 성능 저하가 있기 때문이다.

NVMalloc[1]은 플래시메모리를 DRAM의 확장 공간으로 사용하기 위해 유저 프로그램에게 nvmalloc이라는 인터페이스를 제공하고, 이를 통해 할당한 메모리 영역을 SSD(Solid State Drive)에서 할당해 준다. 이 기법에서는 SSD와 DRAM으로의 데이터 배치를 호스트 소프트웨어가 담당한다. 반면, NVMM[2]은 플래시메모리를 메인 메모리로 사용하고 DRAM을 LLC(Last Level Cache)로 사용한다. 이 기법에서는 hybrid controller라는 별도의 하드웨어를 통해 DRAM과 플래시메모리 사이의 페이징을 관리한다. 하지만, 위 기법들에서는 호스트에서 별도의 소프트웨어 계층 혹은 하드웨어 장치를 통해 데이터의 이동과 배치를 담당하는 오버헤드가 존재한다.

한편, 최근에는 DDR(Dual Data Rate) 인터페이스를 사용하는 NV-DIMM 장치[3]들과 NVM-X[4] 등이 등장하고, DDR 인터페이스 기반의 SSD 하드웨어[5]에 대한 연구가 진행되었다. 위의 기법들에서는 새로운 스토리지 장치를 제안하였지만, 해당 장치의 특성에 맞는 소프트웨어 관리 기법이 구체적으로 다루어지지 않았다.

본 연구에서는 DRAM 인터페이스 기반 SSD 장치인 UMD (Unified Memory Device)의 구조를 제안하고, MMU(Memory Management Unit)에서 UMD를 효율적으로 사용할 수 있는 방법을 제시한다. 특히, UMD 내부에서 데이터의 이동으로 인해 응답 시간이 길어질 경우에, 이를 호스트로 전달하여 CPU가 다른 작업을 수행할 수 있도록 하는 내부 페이지 폴트 처리라는 기법을 제안한다.

2. DRAM 인터페이스 기반 SSD 관리 기법

2.1 Unified Memory Device의 구조

UMD는 DIMM 슬롯을 통해 호스트에 연결되며, 내부적으로 작은 크기의 DRAM과 대용량의 플래시메모리로 이루어져 있다. 그리고, 데이터 전체는 플래시메모리에 저장되며, DRAM에는 데이터의 일부가 캐싱된다. 본 연구에서는 UMD를 사용하는 시스템을 디자인하기 위해 두 가지의 요구 사항을 반영하였다.

첫째, 기존 운영체제의 메모리 관리 기법이나 페이지 매핑 테이블의 구조를 변경하지 않아야 한다. 이를 위해, 운영체제의 페이지 매핑 테이블은 UMD 내의 DRAM이나 플래시메모리의 물리 주소에 관한 매핑이 아닌, UMD 장치의 논리 주소에 관한 매핑을 관리하며, UMD의 내부 컨트롤러는 논리 주소와 DRAM, 플래시메모리의 물리 주소 사이의 L2P(Logical to

이 논문은 2013년도 정부(교육부)의 재원으로 한국연구재단의 기초연구사업 지원을 받아 수행된 것임(2013R1A1A2A10013598)

Physical) 매핑 테이블을 관리한다. 이는 UMD가 DRAM 공간을 캐시로 사용하며, 플래시 메모리의 덮어쓰기가 불가능한 특징으로 인해서 매 접근마다 물리 주소가 변경되기 때문이다.

둘째, UMD 접근 속도를 빠르게 하기 위해서, 호스트와 UMD간의 데이터 접근은 DDR 인터페이스를 통해서만 진행해야 한다. 이를 위해, UMD는 DRAM 영역을 내부 컨트롤러를 거치지 않고 호스트가 접근할 수 있도록 하며, 호스트가 원하는 데이터는 DRAM에 우선 캐싱된 후에 호스트에게 DDR 인터페이스를 통해 전송한다. 만약, 호스트가 요청한 데이터가 플래시메모리에만 존재하는 경우에 플래시메모리에서 DRAM으로 데이터를 옮기는 작업이 필요한데, 이를 내부 페이지 폴트라 부른다. 내부 페이지 폴트의 처리 방법에 대해서는 2.3절에서 설명한다.

2.2 UMD와 MMU간 인터페이스

기존의 운영체제는 프로세스 별로 메모리의 가상 주소 공간을 제공하며, 운영체제가 제공하는 프로세스 별 페이지 테이블을 통해 가상 주소와 DRAM의 물리 주소 간의 매핑을 관리한다. 또한 일부 매핑 정보를 TLB에 캐싱하여 페이지 테이블을 접근하지 않고 DRAM에 접근이 가능하도록 한다. 반면, UMD 장치를 사용하는 경우, O/S의 페이지 테이블은 DRAM의 물리 주소가 아닌 UMD 장치의 논리 주소에 관한 매핑을 관리한다.

그림 1은 UMD의 시스템 아키텍쳐를 보여준다. CPU는 가상 주소를 기반으로 MMU에게 메모리 접근을 요청한다. MMU는 운영체제의 페이지 매핑 테이블을 통해 가상 주소와 UMD의 논리 주소간의 매핑을 관리하고, TLB에는 가상 주소에 해당하는 UMD의 논리 주소가 아닌 DRAM의 물리 주소를 캐싱한다. 호스트는 TLB에 캐싱된 DRAM의 물리 주소를 사용하여, UMD의 내부 컨트롤러를 거치지 않고 내부 DRAM에 바로 접근할 수 있다. MMU는 UMD에게서 TLB에 캐싱할 DRAM의 물리 주소를 전달받을 필요가 있는데, 이는 MMU가 운영체제의 페이지 테이블 워킹 작업 중 UMD에게 요청하도록 페이지 테이블 워킹 작업을 수정하였다.

CPU의 접근이 TLB에서 히트이면 DRAM의 물리 주소를 기반으로 UMD에서 읽기/쓰기 명령을 처리해준다. 반면, TLB에서 미스가 발생한 경우, MMU는 가상 주소에 해당하는 DRAM의 물리 주소를 찾는 페이지 테이블 워킹 작업을 수행한다. 이 때, MMU는 운영체제가 제공하는 프로세스별 페이지 테이블을 탐색하여 CPU에게 전달 받은 가상 주소에 해당하는 UMD의 논리 주소를 찾는다. 그 후, UMD 내부의 L2P 매핑 테이블을 탐색하여 DRAM의 물리 주소를 찾는다. 페이지 테이블 워킹 작업의 결과로는 해당 가상 주소가 DRAM에 존재하는 경우, UMD 내의 플래시메모리에만 존재하는 경우, UMD에 존재하지 않는 경우가 있다.

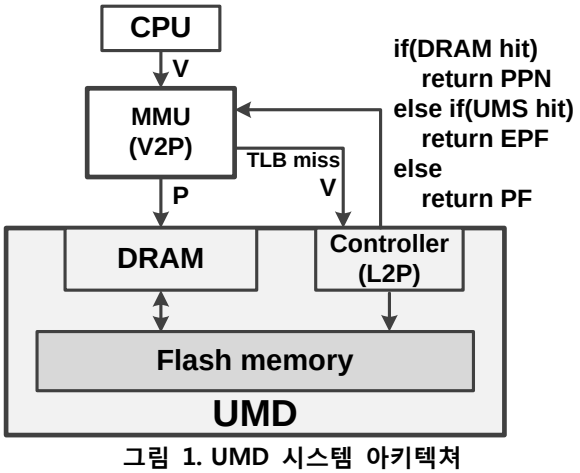

그림 1. UMD 시스템 아키텍쳐

가상 주소에 해당하는 데이터가 DRAM에 존재하는 경우, MMU는 DRAM의 물리 주소를 TLB에 캐싱하고 CPU에서 전달받은 읽기/쓰기 명령을 수행한다. 데이터가 UMD 내의 플래시메모리에만 존재하는 경우, 내부 페이지 폴트 처리를 통해 CPU가 다른 작업을 수행할 수 있도록 하는데, 이는 2.3절에서 설명한다. 데이터가 UMD에 존재하지 않는 경우, MMU는 운영체제에게 일반 페이지 폴트(PF) 예외를 발생시켜, 디스크에서 데이터를 UMD로 읽어오는 작업을 수행한다.

2.3 호스트 협력을 통한 내부 페이지 폴트 처리 기법

본 기법에서는 기존의 페이지 폴트 작업과 비슷하게 UMD의 내부 페이지 폴트 작업을 발생시킨 프로세스를 다른 프로세스로 컨텍스트 스위치(context switch)하여, 작업이 끝날 때까지 다른 프로세스의 작업을 수행한다. 이를 위해서는 UMD의 내부 페이지 폴트 발생을 MMU가 O/S에게 알리는 방법이 필요한데, 본 기법에서는 O/S에 EPF(Embedded Page Fault)라는 새로운 예외를 추가하여 이를 전달하였다.

반면, UMD는 DIMM 슬롯에 연결되는 장치이기 때문에, 호스트에게 기존 PF처럼 페이지 폴트의 처리가 끝났음을 인터럽트로 전달하기 위해서는 하드웨어의 수정이 필요하다. 본 기법에서는 기존의 하드웨어 구조를 수정하지 않기 위하여, EPF 예외를 전달할 때에 예상 소요시간을 함께 전달하고, 운영체제는 해당 시간 동안 프로세스를 대기 상태로 전환하는 방법을 사용한다. 이 때, 예상 소요 시간은 내부 페이지 폴트 처리 중 플래시메모리와 DRAM에 발생할 읽기/쓰기의 횟수를 미리 계산하고, 이를 기반으로 구한다. 만약, 내부 페이지 폴트를 처리하는 중에 GC가 발생할 것으로 판단되면, GC로 인해 발생할 읽기/쓰기의 횟수도 예상 소요 시간에 반영한다.

MMU는 페이지 테이블 워킹의 결과가 EPF이면 UMD에서 내부 페이지 폴트가 발생함을 인식하고, O/S에게 예상 소요시간 정보를 포함한 EPF 예외를 발생시킨다. O/S는 EPF 예외가 발생하면 해당 프로세스를 대기 상태로 전환이 가능한지 확인하고, 가능하다면 프로세스를 예상 소요시간 동안 대기 상태로 전환시킨 후에 다른 프로세스를 수행한다.

본 기법에서 제안하는 EPF 예외는 예외를 발생시킨 프로세스를 block 시키는 작업을 수행하는데, 운영체제는 프로세스를 block 시키지 못하는 몇 가지 경우가 존재한다. 예를 들어 프

로세스가 원자적 명령을 수행하는 중이거나 비선점 상태인 경우가 있다. EPF 예외는 프로세스를 block 시키지 못하는 경우에 UMD의 내부 페이지 폴트 처리가 끝날 때까지 기다린다.

3. 실 험

3.1 실험 환경

본 연구에서는 UMD를 시뮬레이션하기 위해 풀 시스템 시뮬레이터인 MARSS-x86[6]과 DRAMSim2[7]를 기반으로 시뮬레이터를 제작하였다. MARSS-x86은 QEMU[8]와 PTLSim[9]으로 구성되어 있으며, QEMU와 PTLSim이 사이클 단위로 번갈아 수행하면서 CPU와 주변 장치들에 대한 시뮬레이션을 진행하는 시뮬레이터이다. QEMU의 게스트로 동작하는 운영체제는 커널 버전 2.6.38의 우분투 11.04를 수정하여 사용하였다. 또한, QEMU의 하드웨어 설정은 in-order 코어 1개, 2GB의 UMD로 설정하였고, I-TLB와 D-TLB는 각각 32개의 엔트리를 저장하며, DRAM과 플래시메모리의 읽기/쓰기 지연시간을 각각 50ns/50ns와 25us/200us로 설정하였고, UMD 내의 DRAM의 크기를 바꿔가면서 성능 평가를 진행하였다. 그리고, DRAM 영역은 LRU (Least Recently Used) 방식으로 관리한다.

실험 시작 전 UMD의 DRAM은 캐싱된 데이터가 없는 상태로, 플래시메모리는 60% 영역에 대한 순차 쓰기를 하고 50% 영역에 대한 임의 쓰기를 수행하여, 실험 시작부터 GC가 발생하는 상태로 초기화하였다. 따라서, 모든 실험은 DRAM 영역이 채워지는 동안 내부 페이지 폴트가 발생하지 않으며, DRAM 영역이 꽉 찬 순간부터 내부 페이지 폴트가 발생한다.

테스트 케이스는 아무 작업 없이 내부 페이지 폴트의 처리를 기다리는 *no_EPF*, 호스트와 협력을 통해 내부 페이지 폴트를 처리하는 *all_EPF*, 내부 페이지 폴트의 소요시간이 50us 이상인 경우에만 예외를 발생시키는 *smart_EPF*가 있으며, 실험은 멀티-쓰레드로 압축 풀기를 수행하는 프로그램인 pigz[10]를 사용하여 45MB 파일의 압축을 푸는데 걸린 시간을 측정하였다. UMD 내의 DRAM의 크기가 100MB, 150MB인 경우에 대해 수행했는데, DRAM의 크기가 100MB인 경우, 소요 시간이 긴 내부 페이지 폴트가 많이 발생했으며, 150MB인 경우에는 소요 시간이 긴 내부 페이지 폴트가 거의 발생하지 않았다.

3.2 pigz 성능 평가

그림 2는 5개의 쓰레드로 45MB 파일의 압축을 푸는데 걸린 시간을 보여준다. UMD 내의 DRAM의 크기가 100MB인 경우, *no_EPF*, *all_EPF*, *smart_EPF*는 각각 3.9초, 3.25초, 3.3초가 걸리는 것을 확인할 수 있다. *All_EPF*와 *smart_EPF*의 경우, *no_EPF*보다 짧은 수행시간을 보이는데, 이는 내부 페이지 폴트 처리 중 다른 프로세스를 수행할 수 있기 때문이다. 한편, UMD 내의 DRAM의 크기가 150MB인 경우, pigz를 수행하는 동안 수행 시간이 긴 내부 페이지 폴트는 거의 발생하지 않아서 *no_EPF*와 *smart_EPF*의 경우 3.2초 정도의 시간이 소요된다. *All_EPF*의 경우, 짧은 응답시간을 가지는 내부 페이지 폴트에 대해 예외를 발생시켜 예외 처리 오버헤드로 인해 3.4초의 수

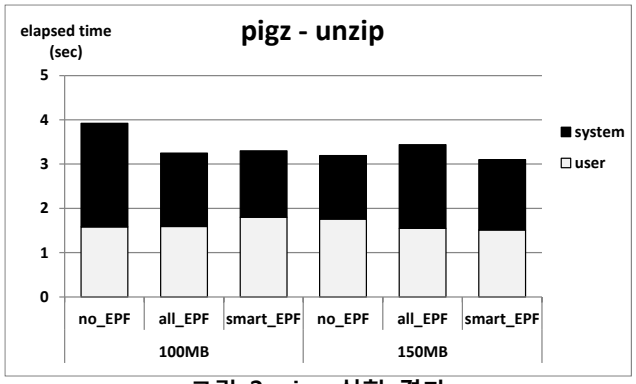

그림 2. pigz 실험 결과

행 시간을 보인다.

표 1은 DRAM의 크기가 150MB일 때 내부 페이지 폴트 횟수를 보여준다. *No_EPF, all_EPF, smart_EPF* 모두 UMD의 전체 접근 중 약 0.11%의 접근에서 내부 페이지 폴트가 발생했는데, 이 중 각각 0%, 100%, 27%가 EPF 예외로 호스트에게 전달되었다. *All_EPF*의 경우, 소요시간이 짧은 EPF 예외를 운영체제로 전달하여 *no_EPF*보다 낮을 성능을 보인다. 반면, *Smart_EPF*의 경우에는 소요시간이 짧은 EPF 예외를 운영체제로 전달하지 않아, *no_EPF*와 같은 수행시간을 가지는 것을 확인할 수 있다.

표 1. 내부 페이지 폴트 횟수

	EPF / Total UMD access	EPF Exception
no_EPF	0.11%	0%
all_EPF	0.11%	100%
smart_EPF	0.11%	27%

4. 결론 및 계획

본 연구에서는 DRAM과 플래시메모리로 구성된 장치인 UMD를 메모리 장치로 사용하는 시스템을 디자인하고, UMD 장치의 내부 페이지 폴트를 호스트 협력을 통해 처리하는 방법을 제안하였다. MARSS-x86에서 Pigz를 사용하여 성능을 평가한 결과, 호스트 협력을 통한 내부페이지 폴트의 처리가 기존 기법에 비해 최대 17%의 수행시간의 감소를 확인하였다. 향후에는 memcached 같은 많은 메모리를 요구하는 서버 환경에서 UMD 장치의 성능을 평가할 것이다.

참고문헌

[1] Wang, Chao, et al., "NVMalloc: Exposing an aggregate SSD store as a memory partition in extreme-scale machines," *proc. of the IPDPS*, 2012.
[2] B. Jacob et al., "A journaled, NAND-flash main-memory system." *Technical Report UMD-SCA-2010-12-01*, 2010.
[3] "Viking Technology, ArxCis-NV." http://www.vikingmodular.com/products/arxcis/arxcis.html, 2012.
[4] NVM-X, https://xitore.com/what-is-nvm-x/, 2014.
[5] Dong, K. I. M., et al. "Solid-State disk with Double Data Rate DRAM interface for high-performance PCs." IEICE *transaction on Information and Systems*, Vol. 92, No. pp. 727-731, 2009.
[6] Patel, Avadh, et al., "MARSS: a full system simulator for multicore x86 CPUs." *proc. of the DAC*, 2011.
[7] Rosenfeld, Paul, et al., "DRAMSim2: A cycle accurate memory system simulator," *proc. of the Computer Architecture Letters*, pp. 16-19, 2011.
[8] Bellard, Fabrice "QEMU, a fast and portable dynamic translator." *proc. of the ATEC*, 2005.
[9] Matt Yourst, "PTLsim: A Cycle Accurate Full System x86- 64 Microarchitectural Simulator, " *proc. of the ISPASS*, 2007.
[10] pigz, http://zlib.net/pigz/pigz.pdf.