creative commons

COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학석사 학위논문

RISC-V 아키텍처 기반 FPGA 플랫폼에서 하드웨어 디맨드 페이징의 구현

2021년 2월

서울대학교 대학원

컴퓨터공학부

공 정 훈

RISC-V 아키텍처 기반 FPGA 플랫폼에서 하드웨어 디맨드 페이징의 구현

지도교수 이 재 욱

이 논문을 공학석사 학위논문으로 제출함

2020년 10월

서울대학교 대학원

컴퓨터공학부

공 정 훈

공정훈의 석사 학위논문을 인준함

2020년 12월

위 원 장	김 지 홍	
부 위 원 장	이 재 욱	
위 원	유 승 주	

요약(국문초록)

전통적으로 OS는 디스크의 입출력(I/O)을 제어하며, 이를 위한 소프트웨어 스택을 가지고 있다. 이러한 소프트웨어 스택의 오버헤드는 과거 디스크가 느리던 때에는 무시할 수 있을 정도의 오버헤드로 작용하였다. 하지만 최근 초저지연 SSD의 등장으로 인해, 이러한 오버헤드가 상대적으로 커지게 되었다. 이를 해결하기 위해 유저 레벨 I/O 프레임워크가 제안되어 사용되고 있지만, OS가 페이지 폴트를 이용해 처리하는 디맨드 페이징에는 적용할 수 없다. 그러한 한계를 극복하기 위해 본 연구에서는 하드웨어 기반의 디맨드 페이징 기법이 제안되었지만, 해당 기법은 소프트웨어 에뮬레이션 및 시스템 시뮬레이션 단계에 머물러 실장 하드웨어에서의 구현 가능성이 불확실하다. 이 연구에서는 제안된 하드웨어 디맨드 페이징 기법을 실장 하드웨어에 구현하기 위한 구조를 제안하고, 이를 FPGA 기반의 시스템상에 구현해 기존 OS 기반의 디맨드 페이징 대비 성능 향상을 평가하였다. 구현한 시스템으로 평가한 결과, 페이지 폴트 처리에서 디스크 접근 외의 소프트웨어 시간 대부분을 제거했다. 또한, FIO read random 벤치마크의 평균 레이턴시를 최대 85.4% 감소시켰다.

주요어: 디맨드 페이징, 가상 메모리, 운영체제, CPU 아키텍처, 하드웨어 확장
학번: 2019-22365

Contents

Abstract **i**

Contents **iii**

List of Tables **iv**

List of Figures **vi**

Chapter 1 서론 **1**

Chapter 2 이론적 배경 **3**

2.1 메모리 매핑된 파일과 페이지 폴트 3

2.2 초저지연 스토리지와 페이지 폴트 오버헤드 4

Chapter 3 하드웨어 기반 디맨드 페이징 **7**

3.1 개요 7

3.2 LBA-augmented Page Table과 MMU 수정 8

3.3 Storage Management Unit (SMU) 10

3.4 운영체제 지원 10

3.4.1 메모리 및 자원 할당 10

3.4.2 LBA-augmented Page Table 생성 및 관리 12

3.4.3 OS의 자료구조와 동기화 12

3.5 기존 연구의 한계점 15

Chapter 4 하드웨어 기반 디맨드 페이징의 FPGA 구현 **16**

4.1 하드웨어 플랫폼 16

4.2 LBA-augmented Page Table 및 MMU 17

4.3 Storage Management Unit(SMU) 18

4.3.1 SMU 컨트롤러 19

4.3.2 Submission Unit 21

4.3.3 Completion Unit 23

4.3.4 Submission Unit과 Completion Unit 사이의 통신 . . . 25

4.4 운영체제 지원 26

Chapter 5 성능 평가 **27**

5.1 실험 환경 및 워크로드 27

5.2 페이지 폴트 성능 28

5.3 메모리보다 작은(small) 워크로드에서의 성능 평가 32

5.4 메모리보다 큰(large) 워크로드에서의 성능 평가 34

Chapter 6 관련 연구 **37**

6.1 SSD에서 메모리 매핑된 파일의 최적화 37

6.2 초저지연 SSD에서의 I/O 레이턴시 단축 기법 37

6.3 페이지 테이블 탐색 레이턴시 단축 38

Chapter 7 결론 **40**

Bibliography **41**

Abstract **47**

List of Tables

Table 4.1 구현에 사용한 FPGA 플랫폼의 상세 사양 17

Table 5.1 평가 환경 27

List of Figures

Figure 2.1 기존의 OS기반 페이지 폴트 처리 4
Figure 2.2 최근 컴퓨터 구성요소의 속도 경향성 5
Figure 2.3 FIO 벤치마크에서 I/O 수행 시간 분석 6

Figure 3.1 하드웨어 기반 디맨드 페이징 7
Figure 3.2 Present PTE와 LBA-augmented PTE의 구조 8
Figure 3.3 Kpted의 페이지 동기화 13

Figure 4.1 VC707 FPGA evaluation board 16
Figure 4.2 실제 구현에서 사용한 PTE의 레이아웃 18
Figure 4.3 Storage Management Unit(SMU) 19
Figure 4.4 SMU 컨트롤러의 구조 20
Figure 4.5 SMU Submission Unit의 구조 22
Figure 4.6 SMU Completion Unit의 구조 24

Figure 5.1 FIO mmap엔진 수행시의 타임라인 28
Figure 5.2 페이지 재충전이 없는 상황에서의 FIO 레이턴시 29
Figure 5.3 FIO 수행시의 페이지 폴트 처리 오버헤드 29
Figure 5.4 페이지 폴트 시간 분석 31
Figure 5.5 Small 워크로드에서 FIO의 평균 레이턴시(KPoold 없음) 32
Figure 5.6 Small 워크로드에서 FIO의 평균 레이턴시 32
Figure 5.7 Small 워크로드에서 FIO의 평균 대역폭(KPoold 없음) 33
Figure 5.8 Small 워크로드에서 FIO의 평균 대역폭 33

Figure 5.9 Large 워크로드에서 FIO의 평균 레이턴시(KPoold 없음) 35
Figure 5.10 Large 워크로드에서 FIO의 평균 레이턴시 35
Figure 5.11 Large 워크로드에서 FIO의 평균 대역폭(KPoold 없음) 36
Figure 5.12 Large 워크로드에서 FIO의 평균 대역폭 36

제 1 장 서론

최근, 삼성 Z-SSD [25] 및 인텔 Optane SSD [13]등 초저지연 스토리지들이 등장하면서, 메모리와 디스크의 성능 격차가 급격하게 줄어들고 있다. 과거에는 수십 밀리초 단위이던 디스크 접근 시간이 수 마이크로초 정도 수준까지 줄어들었고, 이에 따라 운영체제(OS) 스택이 디스크 접근 시간에서 차지하는 비중이 커지게 되었다. 이에 따라, 디스크 접근을 효율적으로 수행하기 위해서 기존 OS 기반 시스템 스택을 재고할 필요성이 제기되었다 [19, 32].

현대의 많은 컴퓨터 시스템은 페이징 기반의 가상 메모리(Virtual Memory, VM)를 사용한다. 디맨드 페이징은 이러한 시스템에서 디스크와 메모리 사이에 데이터를 주고받는 주요한 기법이다. 이 때, 메모리는 디스크의 캐시로써 사용 [23]된다. CPU가 메모리 load/store 명령어를 이용해 데이터에 접근할 때, 해당 데이터의 매핑 정보가 존재하지 않는다면 CPU는 예외(Exception)를 발생시켜 OS가 필요한 데이터를 디스크 혹은 OS의 페이지 캐시로부터 로드하게 한다. OS는 디스크 수행 시간동안 CPU 자원을 효율적으로 배분하기 위해서 문맥 교환(Context switch)을 수행해, 다른 프로세스가 동작할 수 있게 한다. 이는 디스크가 느리던 과거에는 효과적인 디자인으로 작용했지만, 초저지연 스토리지를 사용했을 때에는 어플리케이션의 레이턴시에 큰 영향을 줄 수도 있다. 페이지 폴트 소프트웨어를 처리하는 시간이 전체 페이지 폴트 처리 시간의 상당 부분을 차지하게 될 뿐만 아니라, 문맥 교환 과정에서 캐시, TLB, 분기예측기 등의 마이크로아키텍처 자원의 오염도 발생해 [29] 사용자 어플리케이션의 성능 저하를 유발한다. 페이지 폴트 처리에서 소프트웨어의 오버헤드를 줄이기 위해 하드웨어 기반의 디맨드 페이징 [18] 기법이 제안되었다. 이는 페이지 폴트 처리 시에 OS 커널의 개입을 없애고, 하드웨어를 이용해 페이지 폴트를 처리하는 기법이다.

해당 선행 연구에서는 총 세 가지의 하드웨어 확장 및 운영체제 지원을 제안하였다. 첫째로, 기존의 페이지 테이블 구조를 수정한 LBA-Augmented Page Table 구조를 생성하고, 이를 처리할 수 있는 Memory Management Unit(MMU)의 수정을 제안하였다. 둘째로, CPU에서 NVMe SSD에 바로 접근할 수 있게 하기 위해 Storage Management Unit(SMU)라는 새로운 하드웨어를 제안하였다. 마지막으로, 자원 할당 및 메타데이터 관리를 위해 OS 커널이 하드웨어 디맨드 페이징을 지원할 수 있게 하였다.

하지만, 해당 기법은 x86 시스템 상에서의 에뮬레이션과 시스템 시뮬레이션을 이용한 평가만 수행하여, 실제 하드웨어에서의 구현 가능성까지 확인할 수는 없었다. 본 논문에서는 하드웨어 페이징 기법을 실제 하드웨어에 구현하기 위한 구조를 제안하고, 제안한 구조를 FPGA 기반의 실장 RISC-V 하드웨어에 구현해 성능을 평가하였다. 이를 통해 하드웨어 페이징 기법이 페이지 폴트의 처리에서 OS 소프트웨어 오버헤드를 대부분 제거할 수 있으며, FIO random read의 평균 레이턴시(Latency, 지연 시간)를 OS 기반 디맨드 페이징 대비 최대 85.4% 감소시킨다는 점을 보였다.

제 2 장 이론적 배경

2.1 메모리 매핑된 파일과 페이지 폴트

메모리에 매핑된 파일(Memory-mapped file)은 디스크의 파일을 메모리에 매핑해, 메모리 접근(Load/Store)만을 통해 파일 읽기 및 쓰기를 수행할 수 있게 하는 기법이다. 이 기법은 여러 장점이 있다. 일반적으로 파일에 대한 접근은 프로그래머가 명시적으로 특정 파일의 특정 위치를 읽거나 쓰는 시스템 콜 혹은 라이브러리 콜(read, write 시스템 콜 등)을 사용해서 메모리에 로드하고, 사용한다. 하지만 메모리에 매핑된 파일의 경우 OS가 디스크와 메모리 사이의 파일 입출력을 수행하며, 프로그래머가 실제 디스크에서 데이터를 읽거나 쓰는 작업을 따로 수행할 필요가 없다. 또한, 물리적인 메모리 크기보다 큰 파일을 사용하는 경우에 있어서도 프로그래머가 파일에 사용되는 메모리 영역을 별도로 할당 및 해제할 필요 없이 OS가 자동적으로 파일의 필요한 부분을 메모리에 로드하고, 그렇지 않은 부분은 메모리에서 디스크로 내려보냄으로써, 사용자 입장에서의 관리가 용이하다. 가상 메모리에서의 디맨드 페이징(Demand paging)은 이를 지원하기 위한 메커니즘이다. 메모리에 매핑된 파일은 페이지 단위로 페이지 테이블에 매핑 정보가 저장된다. 이 페이지는 메모리상에 존재하고 매핑되어 있는(Resident page) 경우와, 그렇지 않은(Non-resident page) 두 가지의 경우가 있다. 유저 레벨 어플리케이션이 메모리에 매핑된 파일에 접근하면, CPU 내부의 Memory Management Unit(MMU)는 페이지 테이블의 페이지 테이블 엔트리(PTE)에 접근하여 요청한 페이지의 매핑 정보를 요청한다. 요청한 페이지의 매핑 정보가 존재한다면, CPU는 해당 메모리 영역에 접근하여 유저가 필요로 하는 데이터를 가져올 수 있다.

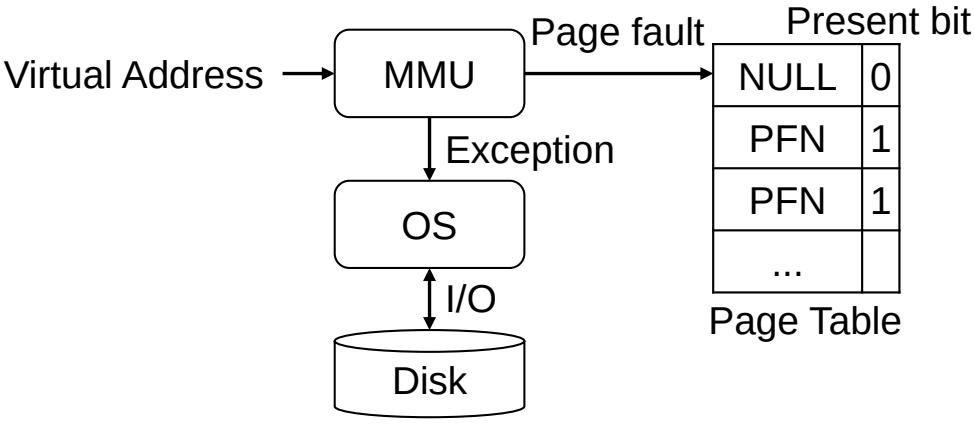

Figure 2.1: 기존의 OS기반 페이지 폴트 처리

반대로 요청한 매핑 정보가 존재하지 않는다면 페이지 폴트가 발생하게 된다. 페이지 폴트의 수행 과정은 그림 2.1과 같다. MMU는 예외(Exception)를 발생시켜 OS에게 해당 페이지의 로드를 요청한다. OS는 메모리의 페이지 캐시에서 데이터를 가져오거나, 디스크 I/O를 수행해 유저 어플리케이션이 요구하는 데이터를 메모리에 로드한 후 페이지 테이블을 업데이트한다. 그 후 문맥 교환(Context switch)을 통해 유저 어플리케이션 수행을 재개시킨다.

2.2 초저지연 스토리지와 페이지 폴트 오버헤드

그림 2.2는 컴퓨터 시스템 구성 요소의 성능 트렌드를 나타낸 것으로 [18], SSD의 성능은 삼성 Z-SSD [25], 인텔 Optane SSD [13]와 같은 초저지연 SSD의 등장에 힘입어 최근 가파르게 증가하고 있다. 이에 따라 페이지 폴트를 처리하는 데에 있어서 OS 소프트웨어 스택에서 소모되는 시간의 비중이 무시할 수 없을 정도로 커지게 되었다. OS 커널 기반의 페이지 폴트는 문맥 교환과 소프트웨어 수행을 필요로 하는데, 이 과정에서 CPU 캐시 및 TLB 등의 마이크로아키텍처 자원을 오염시킨다. 결과적으로, 사용자 코드의 처리 속도를 늦추게

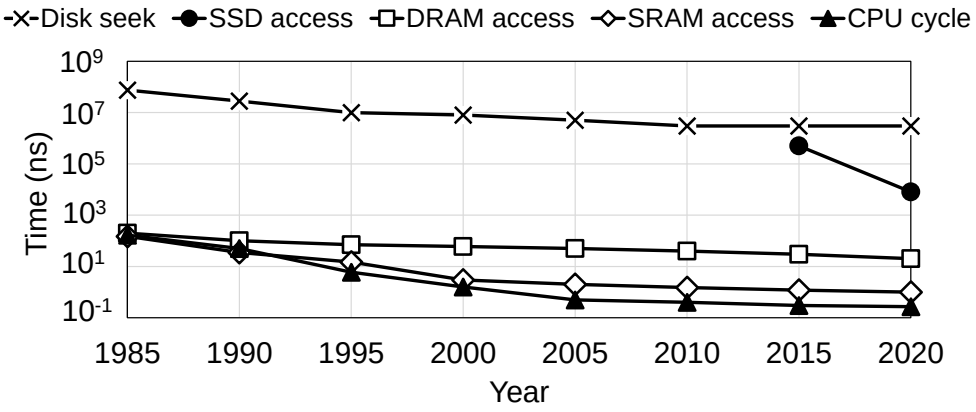

Figure 2.2: 최근 컴퓨터 구성요소의 속도 경향성

되는 부수적인 효과 또한 발생하게 된다.

페이지 폴트 처리에서 소프트웨어 스택이 차지하는 비율을 평가하기 위해, Xilinx VC707 FPGA 상에 구현한 100MHz RISC-V 리눅스 시스템과 인텔 Optane SSD를 이용, FIO 벤치마크 [10]의 mmap 엔진을 통해 OS의 페이지 폴트 레이턴시(지연 시간, latency)를 분석하였다. 해당 실험에서는 메모리보다 작은 크기의 워크로드를 이용, 4KB Random Read 벤치마크를 다음의 두 가지 경우에 대해 수행하였다. OSDP(OS-based Demand Paging)는 아무 수정 없이 FIO의 mmap 벤치마크를 수행한 경우이다. 이 결과는 유저 소프트웨어의 실행 시간과 페이지 폴트 처리 시간의 합을 나타낸다. Ideal은 *mmap()* 과정에서 *MAP_POPULATE* 플래그를 추가해 모든 데이터가 메모리에 존재하는 상태에서 벤치마크를 수행한 경우이다. 이 과정에서는 페이지 폴트 및 I/O가 발생하지 않아, 유저 소프트웨어의 실행 시간만을 볼 수 있다. 즉, OSDP에서 Ideal의 시간을 빼면 페이지 폴트 처리에 걸리는 시간만을 계산할 수 있다.

그림 2.3은 실험의 결과를 나타낸 것이다. OSDP의 평균 레이턴시는 921.922us, ideal의 평균 레이턴시는 109.372us이다. 페이지 폴트의 수행 시간은 OSDP의 레이턴시에서 ideal의 레이턴시를 뺀 값인 812.55us이다. 페이지 폴트 수행 시

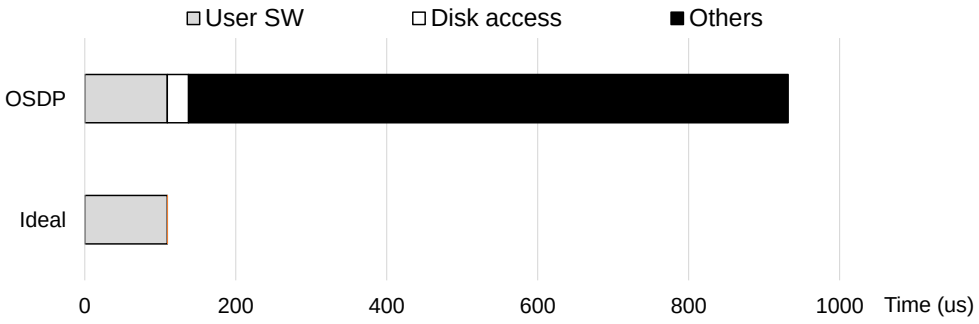

Figure 2.3: FIO 벤치마크에서 I/O 수행 시간 분석

간 812.55us 중 SSD I/O가 차지하는 시간(Disk access)은 28.16us이다. 전체 페이지 폴트 처리 시간에서 디스크 접근을 제외한 부분(Others)은 794.39us로, 페이지 폴트 처리 시간의 97.8%를 차지한다. 이 실험을 통해 페이지 폴트 처리에서 문맥 교환 및 소프트웨어 수행이 대부분의 시간을 차지하는 것을 확인하였다.

제 3 장 하드웨어 기반 디맨드 페이징

3.1 개요

A Case for Hardware-Based Demand Paging [18]은 페이지 폴트의 레이턴시를 줄이기 위해 그림 3.1과 같은 구조를 이용, OS의 개입 없이 하드웨어로 페이지 폴트 처리를 수행하는 기법을 제안한 연구이다. 이를 지원하기 위해 LBA-augemented Page Table이라는 자료구조와, Storage Management Unit(SMU)이라는 새로운 하드웨어, 그리고 CPU 내부 MMU의 수정을 제안하였다. 마지막으로, 자원 할당 및 메타데이터 관리를 위해 OS 커널이 하드웨어 디맨드 페이징을 지원할 수 있게 하였다.

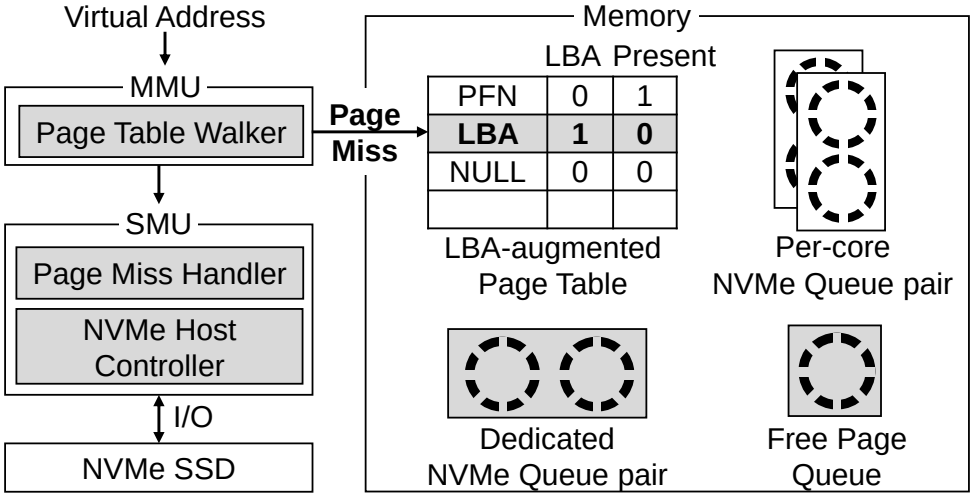

Figure 3.1: 하드웨어 기반 디맨드 페이징

해당 연구에서 제안하는 LBA-augmented Page Table은 메모리 상에 페이지가 존재하지 않을 때(Non-resident page) 페이지 테이블에 해당 데이터의 논리

블록 주소(Logical Block Address, LBA)를 담을 수 있다. 해당 페이지에 대한 접근이 발생하면 MMU는 페이지 테이블에서 LBA 정보를 추출해, 해당 정보를 어디서 읽어와야 하는지 알아낸다. SMU는 MMU로부터 LBA와 페이지 테이블에 대한 정보를 제공받아, MMU가 요청한 데이터를 디스크에서 메모리로 로드하는 하드웨어이다. SMU는 이러한 작업을 수행하기 위해 필요한 정보를 OS에서 제공받을 필요가 있는데, 이를 위해 OS와 소통할 수 있는 인터페이스를 제공한다. 마지막으로 OS는 하드웨어가 I/O를 독립적으로 수행할 수 있도록 필요한 자원을 할당하고, 이를 하드웨어에 알려줄 수 있게 수정된다. 그리고 하드웨어에서 처리한 페이지 폴트 정보를 기존 OS의 메타데이터와 동기화한다.

3.2 LBA-augmented Page Table과 MMU 수정

하드웨어 기반 디맨드 페이징에서는 페이지 폴트 시 하드웨어가 페이지 테이블 엔트리(PTE)를 보고 하드웨어적으로 처리할 수 있는 페이지 폴트인지 판단하고, 그렇다면 그것을 하드웨어로 처리할 수 있게 하기 위해 PTE의 구조를 변경하여 LBA 정보를 담을 수 있게 하였다. 이러한 구조를 포함하는 페이지 테이블을 LBA-augmented Page Table이라고 하였다. 또한 Multi-socket 플랫폼과 여러 개의 NVMe SSD를 동시에 지원하기 위해 Socket ID 및 Device ID 정보를 PTE에 기록해, 하드웨어에서 어떤 소켓의 어떤 NVMe로부터 데이터를 읽어야 하는지도 알 수 있다.

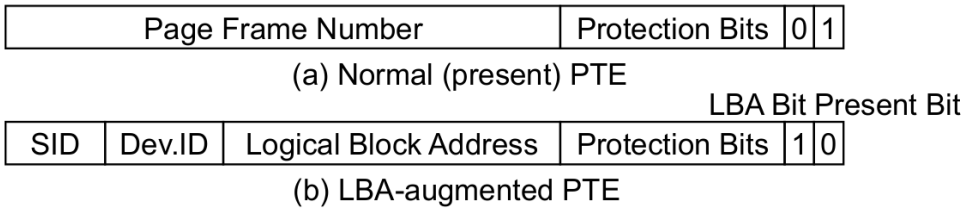

Figure 3.2: Present PTE와 LBA-augmented PTE의 구조

기존 페이지 테이블 구조에서 PTE의 Present bit는 해당 페이지가 메모리에 존재하는지(resident page), 존재하지 않는지(non-resident page)의 여부를 나타내는 비트이다. 해당 비트가 1이면 resident page, 그렇지 않다면 non-resident page이다. 하드웨어 기반 디맨드 페이징에서는 이에 더해 PTE상에서 사용하지 않는 비트를 이용해서 페이지 폴트가 발생 시, 그것이 하드웨어로 처리될 수 있는지 표시하는 비트를 추가하고, 이를 LBA bit라고 하였다. 또한, 가상 메모리에서의 메모리 보호(memory protection)를 지원하기 위해, 대응하는 페이지의 protection bit정보 또한 저장한다. 이는 resident page의 그것과 같은 위치에 저장되게 하였다.

LBA bit는 해당 페이지가 메모리 상에 존재하지 않으면서(Present bit가 0) PTE가 하드웨어적으로 처리될 수 있는 LBA 정보를 담고 있거나, 페이지가 메모리에 존재(Present bit가 1) 하면서 하드웨어로 처리한 정보를 OS 커널과 동기화해야할 필요가 있을 경우 1로 설정된다. 그림 3.2는 LBA-augmented Page Table에서 present PTE의 구조와 LBA-augmented PTE의 그것을 나타낸 것이다.

가상 주소 번역 과정에서 MMU는 PTE의 Present bit를 먼저 확인한다. Present bit가 1이라면 CPU가 요청한 가상 주소에 대응하는 물리 주소를 반환하고, 그렇지 않다면 MMU는 LBA bit를 확인한다. 그것이 0이면 MMU는 예외(Exception)를 발생시켜, OS에게 페이지 폴트의 처리를 요청하게 된다. 반면, LBA bit가 1이라면 MMU는 SMU에게 해당 페이지 폴트의 처리를 요청하게 된다. SMU가 요청한 페이지 폴트를 성공적으로 처리했다면, MMU는 CPU가 요청한 가상 주소에 대한 물리 주소를 반환하고, CPU는 사용자 어플리케이션을 이어서 실행한다. 만약, SMU가 요청한 페이지 폴트를 처리하는 데에 실패했다면, MMU는 예외를 발생시켜 OS에게 해당 페이지 폴트를 처리하도록 요청한다.

3.3 Storage Management Unit (SMU)

SMU는 MMU가 전달하는 페이지 폴트에 대해 페이지를 할당하고, NVMe I/O를 수행해 OS의 페이지 폴트 핸들러 기능을 대신하는 하드웨어이다. 이는 Page Miss Handler와 NVMe Host Controller의 두 가지 구성 요소로 이루어져 있다. Page miss handler는 MMU에서 보내는 중복된 페이지 폴트 처리 요청을 걸러내고, 페이지 폴트를 처리하는 데 필요한 메모리를 할당받는 하드웨어이다. NVMe Host Controller는 NVMe 프로토콜에 따라 커맨드를 생성하고, NVMe SSD와 직접 통신해 I/O를 지시하고 결과를 확인하는 하드웨어 구조이다. SMU는 이러한 동작을 수행하기 위해 OS로부터 메모리 공간 및 NVMe 자원을 할당받고, 이를 활용해 페이지 폴트를 처리한다.

3.4 운영체제 지원

하드웨어 기반 디맨드 페이징을 처리하기 위해서, OS 커널은 크게 세 가지 기능을 추가적으로 수행하게 된다. 첫째, 하드웨어를 위한 NVMe Submission Queue(SQ) /Completion Queue(CQ) 및 메모리 공간을 할당하고, 하드웨어에 알리는 기능을 수행한다. 둘째, 하드웨어가 인식할 수 있는 형태의 LBA-augmented 페이지 테이블을 생성한다. 셋째, 하드웨어에서 처리한 페이지 폴트 결과를 OS의 자료구조와 동기화한다.

3.4.1 메모리 및 자원 할당

NVMe 프로토콜 [21]은 메모리에 Submission Queue(SQ)와 Completion Queue (CQ)라는 원형 큐를 생성하고, 이를 이용해 통신한다. NVMe 드라이버는 SQ에 NVMe 명령에 대한 정보를 작성하고, NVMe SSD는 CQ에 처리한 명령에 대한 정보를 업데이트한다. 또한, NVMe SSD는 내부에 SQ/CQ doorbell 레지스터를 가지고 있다. SQ doorbell은 드라이버에서 NVMe에게 I/O 처리를

지시하는 용도로 사용되는 레지스터이고, CQ doorbell은 드라이버가 NVMe에서 업데이트한 완료 정보를 접수했음을 알리는 데에 사용하는 레지스터이다. SMU가 NVMe 프로토콜을 이용해 NVMe SSD I/O를 수행하기 위해서는 위의 자료구조와 하드웨어 레지스터에 접근할 필요가 있다. 이를 위해, OS는 SQ 및 CQ를 메모리상에 생성하고 SMU의 하드웨어 레지스터를 통해 그 주소를 전달한다. SQ 및 CQ doorbell 레지스터의 주소 또한 같은 방식으로 전달함으로써, SMU에서 곧바로 해당 레지스터에 접근할 수 있도록 한다. SMU가 페이지 폴트를 처리하기 위해서는, OS에서 페이지를 할당하고 그것을 SMU의 Free page pool에 삽입해 하드웨어가 사용할 수 있게 해 주어야 한다. 이러한 페이지 할당은 총 세 가지의 상황에서 발생한다. 첫째, OS에서 SMU를 초기화할 때 Free page pool에 페이지를 채워, SMU가 페이지 폴트를 처리할 수 있게 한다. 둘째로, SMU에서 페이지 폴트를 처리하던 중, Free page pool이 비어서 페이지 폴트를 처리할 수 없게 되었을 때이다. MMU는 예외를 발생시켜 해당 페이지 폴트의 처리를 요청한다. OS에서는 페이지 폴트를 처리하면서, 새로운 페이지를 Free page pool에 삽입해 다음 페이지 폴트는 하드웨어가 처리할 수 있게 한다. 마지막으로, 비동기적 할당이다. 본 연구에서는 페이지 할당을 빠르게 수행하기 위해 Kpoold라는 커널 스레드를 생성해 페이지를 할당하게 하였다. 이는 주기적으로 작동해 Free page pool의 Head/Tail 레지스터를 검사, Free page pool이 꽉 차있지 않다면 새로운 페이지를 할당해서 채워넣는다. 이를 통해 SMU에서 Free page pool이 비어 페이지 폴트를 처리하지 못하고 OS의 페이지 폴트 핸들러로 들어가는 빈도를 줄일 수 있다. Free page pool에 페이지를 추가로 할당 및 삽입하는 일련의 과정을 페이지 재충전(Free page refill)이라고 하였다.

3.4.2 LBA-augmented Page Table 생성 및 관리

하드웨어 디맨드 페이징을 선택적으로 사용자 어플리케이션에게 제공하기 위해, 메모리 매핑(*mmap()*) 시점에서 추가적인 플래그를 사용할 수 있게 하였다. OS 커널에서는 mmap 수행시 해당 플래그가 인식되면 일반적인 페이지 테이블이 아닌, LBA-augmented Page Table을 생성한다. 커널에서는 매핑되는 영역의 각각의 PTE에 대해 해당하는 페이지가 OS의 페이지 캐시에 있는지 확인하고, 있다면 PTE에 연결한다. 그렇지 않다면 파일 시스템을 탐색해 데이터가 존재하는 LBA 정보를 찾아서 PTE에 기록하고, PTE에 LBA 비트를 추가해 수정된 MMU가 이를 인식할 수 있게 한다.

이러한 동작은 *mmap()* 수행시 모든 페이지에 대해 PTE를 할당함으로써 메모리 사용량이 증가하는 단점이 있다. 하지만, 이러한 공간 오버헤드는 매핑된 공간 4KB당 8바이트로, 파일 크기의 0.2% 정도로 제한된다. 또한, *mmap()*수행 시간을 증가시킨다는 단점 또한 존재한다. 하지만, 이러한 오버헤드는 직접 데이터를 접근하는 오버헤드가 아니라 초기화 단계에서 작용하는 오버헤드이기 때문에, 사용자가 경험하는 레이턴시에 크게 영향을 주지 않는다. 또한, 비동기적으로 PTE를 채우는 [9] 기법을 적용하여 이러한 오버헤드를 경감시킬 수 있다.

3.4.3 OS의 자료구조와 동기화

기존의 OS 기반의 디맨드 페이징에서는 OS 커널이 페이지 폴트를 처리하고, 관련된 메타데이터를 함께 업데이트한다. 하지만 하드웨어 디맨드 페이징 기법에서는, OS 메타데이터가 페이지 폴트 처리와 동시에 업데이트 되지 않기 때문에, 동기화 문제가 발생할 수 있다. 이러한 동기화 문제를 해결하기 위해, 하드웨어 디맨드 페이징은 백그라운드 커널 스레드 Kpted를 사용한다. 이는 하드웨어로 처리된 페이지 폴트 정보를 수집하여, OS의 메타데이터를 업데이트

하는 역할을 담당한다. Kpted는 주기적으로 LBA-augmented Page Table들을 스캔한다. Kpted는 LBA-augmented page table에서 하드웨어로 처리되었지만 OS와 동기화되지 않은 PTE들을 탐색한다. 이는 LBA bit와 present bit가 모두 켜진 PTE이다. 이러한 PTE를 찾았다면, Kpted는 해당 PTE를 OS의 페이지 교체를 위한 LRU list에 삽입하고, 페이지 및 기타 OS 메타데이터를 업데이트한다. 그리고 해당 페이지를 OS 페이지 캐시에 삽입한다. 마지막으로, 해당 PTE의 LBA bit를 제거함으로써 해당 PTE가 OS와 동기화되었음을 표시한다.

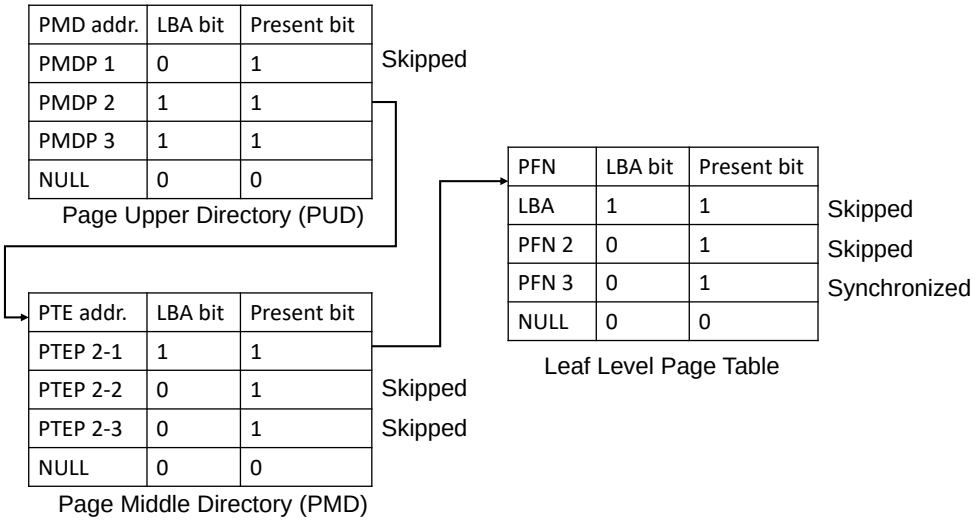

Figure 3.3: Kpted의 페이지 동기화

하지만 모든 PTE를 스캔하는 것은 시간이 오래 걸리고, 다른 프로세스들의 실행 성능을 저하시킬 수 있다. 메타데이터 업데이트가 필요한 PTE를 빠르게 탐색하기 위해, 하드웨어 디맨드 페이징 처리시 SMU에서는 해당 PTE 뿐만이 아니라, 페이지 테이블의 상위 디렉터리인 Page Upper Directory(PUD)와 Page Middle Directory(PMD)의 사용하지 않는 비트를 LBA bit로 활용해 해당 PUD/PMD의 하위 엔트리에서 동기화가 필요함을 표시한다. Kpted는 페이지 테이블의 PMD, PUD를 탐색하면서 PUD의 LBA bit가 0이라면 해당 PUD

에 대한 탐색을 건너뛰고 다음 PUD를 탐색한다. 그렇지 않다면, 해당 PUD의 하위 PMD들을 탐색한다. 마찬가지 방법으로 PMD의 LBA bit가 0이라면 해당 PMD의 탐색을 건너뛰고, 1이라면 해당 PMD의 하위 PTE들을 탐색하고, 그 중 LBA bit와 present bit가 1인 PTE들을 OS와 동기화한다. 그림 3.3은 Kpted가 LBA-augmented Page Table을 탐색하는 예시이다. PUD에서 PMDP 1은 LBA bit가 0이므로 내부를 탐색하지 않고 건너뛴다. PMDP 2와 PMDP 3은 LBA bit가 1이므로, 이를 0으로 바꾸고 내부를 탐색한다. PMD에서도 마찬가지 방법으로 PTEP 2-1의 내부만을 탐색하고 PTEP 2-2와 2-3은 건너뛴다. 마지막으로, PTEP 2-1이 가리키는 leaf-level page table에서 LBA bit 및 present bit가 모두 1인 PTE들의 메타데이터를 동기화하고 LBA bit를 제거한다. 이 예시에서는 PFN 3이 해당된다. Kpted는 동기화 중인 PUD와 PMD에 속하는 PTE가 SMU에 의해 업데이트 될 경우에 대비해, PUD와 PMD 내부 원소를 탐색하기 전에 LBA bit를 먼저 0으로 설정한다. 이는 동기화 중에 SMU가 PUD, PMD의 LBA bit를 1로 설정함으로써, SMU에 의해 업데이트되는 PTE가 다음 동기화 시에 빠짐없이 처리될 수 있게 하기 위함이다.

OS의 메타데이터는 비동기적으로 업데이트 되기 때문에, OS에서는 특정 작업을 수행하기 이전에 동기화를 실행할 필요가 있다. 본 연구에서는 *msync()*, *fsync()* 그리고 *munmap()* 시스템 콜은 원래의 동작을 수행하기 이전에 OS 메타데이터 동기화를 먼저 수행한다. 특히 메모리 매핑된 파일을 해제하는 *munmap()* 시스템 콜의 경우, SMU에서 처리한 모든 데이터를 동기화하지 않는다면 메모리 누수가 발생할 수 있다. 이러한 일을 방지하기 위해, 페이지 테이블 엔트리를 처리하기 이전에 OS는 SMU가 처리중인 모든 페이지 폴트가 완료될 때까지 기다린 후에 동기화 및 *munmap()* 작업을 수행한다.

3.5 기존 연구의 한계점

기존의 하드웨어 기반 디맨드 페이징 연구는 하드웨어로 페이지 폴트를 처리할 수 있는 기법을 제시했지만, x86 시스템상에서의 에뮬레이션과 시스템 시뮬레이터만을 이용해 평가를 진행하였다. 이러한 방식은 성능 향상의 가능성과 경향성을 볼 수는 있지만, 실장 시스템에서 구현이 가능한지의 여부와, 실제로 하드웨어로 구현되었을 때의 성능 향상이 어느 정도인지 정확하게 알기 어렵다는 한계가 있다. 이러한 한계를 극복하기 위해, 본 연구에서는 FPGA 플랫폼에서 동작하는 RISC-V CPU와 리눅스 4.15 커널을 사용하여 실장 시스템에 하드웨어 기반 디맨드 페이징을 구현하고, 성능을 평가하였다.

제 4 장 하드웨어 기반 디맨드 페이징의 FPGA 구현

4.1 하드웨어 플랫폼

본 연구에서는 Sifive에서 오픈소스로 제공하는 freedom [27] 플랫폼을 사용해 그림 4.1과 같은 Xilinx VC707 FPGA 보드에 하드웨어 디맨드 페이징 기능이 구현된 CPU를 프로그램하였다. 해당 플랫폼은 FPGA에 프로그램할 수 있는 RISC-V Rocket CPU, 메모리 컨트롤러, NVMe SSD를 부착할 수 있는 PCIe Root Complex, 리눅스 부팅을 위한 SD카드 리더, 사용자와의 인터페이싱을 위한 UART 인터페이스를 제공하는 플랫폼이다. 추가적으로, PCI Express 인터페이스를 이용하는 NVMe SSD를 장착하기 위해 Hitech Global사의 FMC to PCI Express 모듈을 사용하였다.

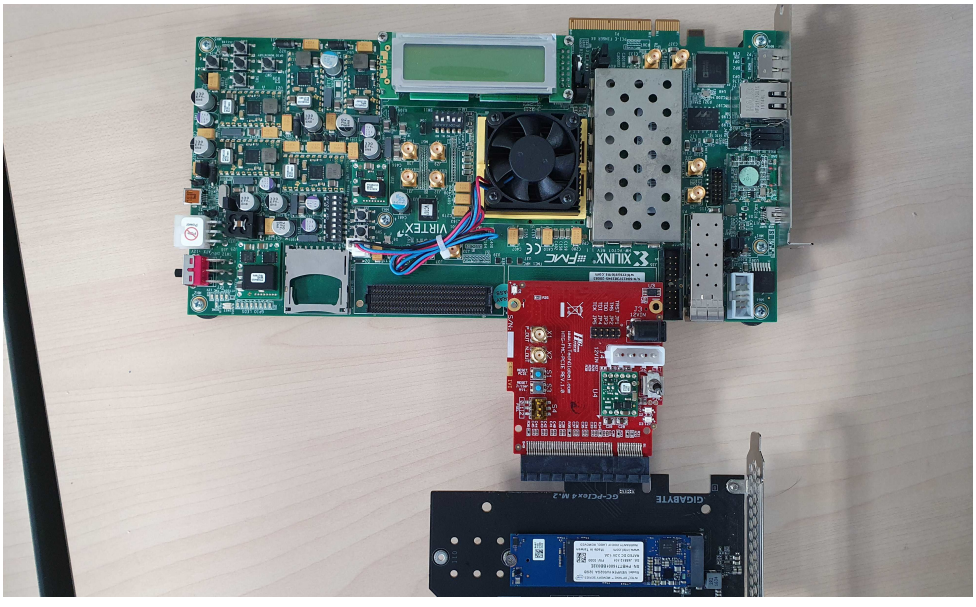

Figure 4.1: VC707 FPGA evaluation board

Freedom 플랫폼은 4코어 Rocket Core [8]를 CPU로 사용한다. 이는 오픈소스로 제공되는 Chisel3 기반의 5-stage in-order RISC-V CPU이다. 본 연구에서는 Rocket Core의 MMU를 수정해 하드웨어 디맨드 페이징을 감지할 수 있게 만들고, SMU를 Tilelink 기반의 MMIO 하드웨어로 구현하였다. Tilelink는 Rocket Core 내부의 구성 요소들이 통신하는 데에 사용되는 버스 프로토콜이다. 표 4.1은 FPGA 플랫폼에 구현한 시스템의 사양을 정리한 것이다.

		Note
하드웨어 플랫폼	Xilinx VC707 evaluation board	Sifive Freedom U500
CPU	4-Core Rocket @ 100MHz	MMU 수정 및 SMU 추가
커널 버전	Linux 4.15.0	하드웨어 디맨드 페이징 지원
메모리	1GB LPDDR3	
NVMe SSD	Intel optane SSD 32GB	M.2 PCIe 3.0 x4

Table 4.1: 구현에 사용한 FPGA 플랫폼의 상세 사양

Freedom-u-sdk [28]는 Sifive에서 제공하는 RISC-V 기반 ASIC 및 FPGA 플랫폼에서 부팅이 가능한 리눅스 커널 기반 OS를 빌드하고, 부팅에 필요한 디스크 이미지를 생성하는 오픈소스 소프트웨어이다. 이를 이용해 하드웨어 디맨드 페이징 기능을 지원하는 리눅스 4.15.0 커널 기반 OS 이미지를 SD카드에 프로그램하였다. 마지막으로, Freedom-u-sdk에서 생성한 SD카드를 이용해, RISC-V 리눅스 기반 OS를 구동하였다.

4.2 LBA-augmented Page Table 및 MMU

본 연구에서는 RISC-V 아키텍처의 페이지 테이블 구조를 고쳐 LBA-augmented Page Table을 구현하였다. RISC-V 아키텍처에서 사용되지 않는 PTE의 9번 bit를 LBA bit로 사용하였다. 그림 4.2는 실제 구현에서 사용한 PTE의 레이아웃을 나타낸 것이다. 단, 기존에 제안된 하드웨어 디맨드 페이징 기법에서 멀티 소켓

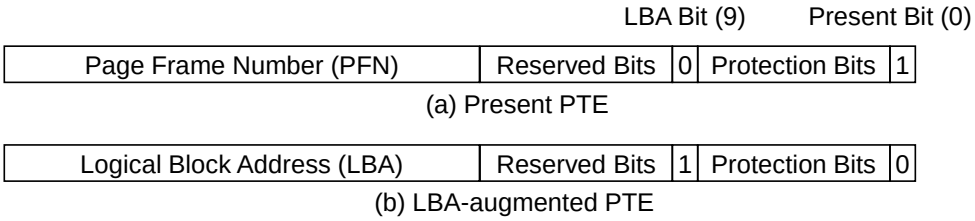

Figure 4.2: 실제 구현에서 사용한 PTE의 레이아웃

및 여러 개의 NVMe 디바이스를 지원하기 위한 Socket ID 및 Device ID 필드는 프로토타입 구현에서는 생략되었다. Rocket Core 내부의 MMU가 Non-resident page를 확인하였을 때 LBA bit가 1이면 MMIO를 통해 SMU에게 명령을 내리고 결과를 확인하는 로직을 구현하였다. SMU가 요청한 페이지 폴트의 처리에 성공하면 MMU는 메모리 번역 과정을 재개하고, 그렇지 않다면 원래의 하드웨어처럼 예외를 발생시켜 OS의 페이지 폴트 핸들러가 페이지 폴트를 처리하도록 하였다.

4.3 Storage Management Unit(SMU)

본 연구에서는 하드웨어 디맨드 페이징을 FPGA상에서 구현하기 위해, 이에 맞게 SMU의 구조를 수정하였다. 그림 4.3은 수정한 SMU의 전체적인 구조를 나타낸 것이다. 수정한 SMU는 OS 및 MMU와의 통신 그리고 중복된 페이지 폴트의 처리를 담당하는 SMU 컨트롤러, NVMe 커맨드를 생성하고 디스크 I/O 수행 명령을 내리는 Submission Unit, NVMe I/O가 완료되는 것을 확인하고 페이지 테이블을 업데이트하는 Completion Unit의 세 가지 구성 요소로 이루어져 있다.

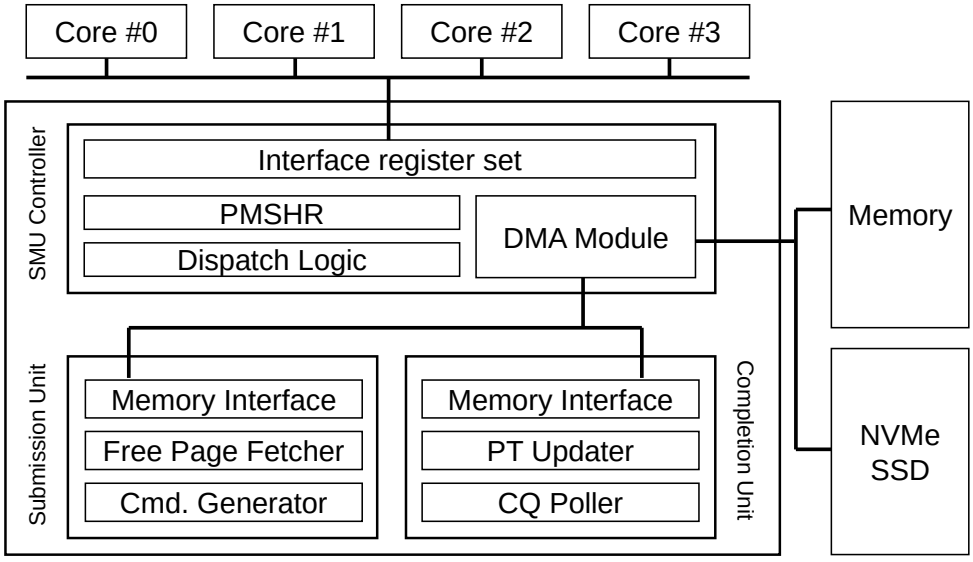

Figure 4.3: Storage Management Unit(SMU)

4.3.1 SMU 컨트롤러

SMU 컨트롤러는 OS, MMU 그리고 메인 메모리와의 통신을 담당한다. 또한, 중복된 요청이 들어왔을 때 이를 중재하고, 불필요한 디스크 접근을 막는 기능을 수행한다. SMU는 메모리 매핑된 I/O(MMIO)를 통해 사용자(OS 커널, MMU)와 통신하는 구조인데, 이를 위해 SMU 컨트롤러는 CPU에서 메모리 인터페이스를 통해 읽거나 쓸수 있는 인터페이스 레지스터를 가지고 있다. OS는 SMU를 사용하기 위해 메모리와 NVMe 자원을 할당하고 이를 인터페이스 레지스터를 업데이트함으로써 알린다. SMU는 전달받은 값을 활용해 페이지 할당 및 디스크 접근을 수행한다. MMU에서는 인터페이스 레지스터에 값을 쓰는 것으로 페이지 폴트 처리를 요청하고, SMU 컨트롤러는 이를 바탕으로 명령을 수행한다. OS와 MMU는 SMU가 업데이트한 인터페이스 레지스터의 값을 읽어서 처리 결과를 확인하고, 동기화를 수행한다. SMU 컨트롤러는 SMU 내부 구성요소의 메모리 접근을 간단하게 하기 위해 메모리 접근을 담당하는

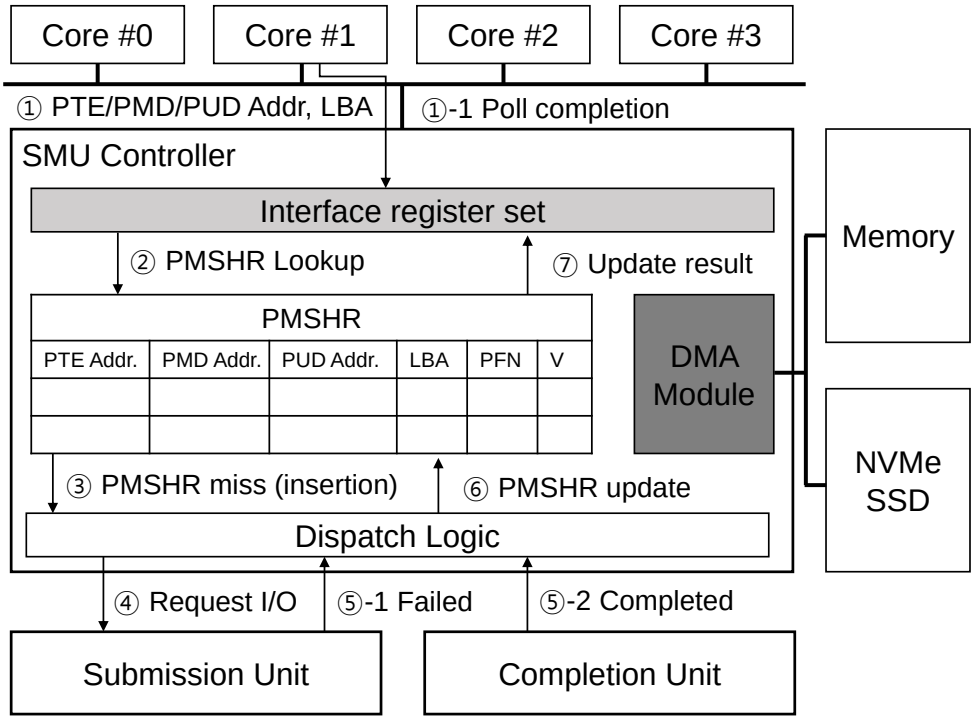

Figure 4.4: SMU 컨트롤러의 구조

DMA Module을 가지고 있다. 이는 Submission Unit 및 Completion Unit에서 요청하는 메모리 동작(읽기/쓰기)을 직접 수행하고, 그 결과를 요청한 모듈에 되돌려준다. SMU 컨트롤러는 내부에 현재 처리중인 페이지 폴트의 정보를 담는 Page Miss Status Holding Register(PMSHR)을 가지고 있다. 이는 Miss Status Holding Register(MSHR)과 비슷한 구조로, 중복된 요청을 걸러내 불필요한 디스크 I/O를 방지하는 역할을 수행한다. PMSHR의 각각의 엔트리는 현재 처리중인 페이지 폴트에 대한 정보를 담고 있다. 이는 PTE 주소를 Key로 사용하는 Content Addressable Memory(CAM) 구조로, 중복된 PTE에 대한 요청이 들어오는지 빠르게 판단할 수 있다.

그림 4.4는 SMU 컨트롤러가 명령을 수행하는 과정을 나타낸 것이다. (1)SMU는 I/O 수행 및 페이지 테이블 업데이트를 위해 MMU로부터 하드웨어 스레드

ID(HART ID), LBA, PTE의 주소, PMD의 주소, PUD의 주소를 전달받는다. 페이지 폴트 처리 요청이 들어오면, SMU는 (2)PMSHR에서 같은 페이지 폴트를 처리중인 것이 있는지 먼저 확인한다. 해당 PTE 주소에 해당하는 페이지 폴트가 이미 처리중이라면, 처리중인 요청이 끝날때까지 대기하고, 처리가 완료되면 해당 PTE를 요청한 MMU에 해당하는 하드웨어 레지스터에 해당 페이지 폴트가 처리된 결과를 업데이트한다. (3)요청받은 PTE 주소에 대해 처리중인 것이 없다면, 컨트롤러는 PMSHR의 HART ID번째 엔트리에 해당 페이지 폴트에 대한 정보를 삽입한다. (4)그 후, Submission Unit에 LBA와 PMSHR 엔트리 ID를 전달해 페이지 폴트를 처리하도록 명령한다. (5)Submission/Completion Unit에서 처리가 완료되면 SMU 컨트롤러는 (6)해당하는 PMSHR의 엔트리를 삭제하고, (7)해당하는 PTE를 요청한 MMU에 대응하는 하드웨어 레지스터의 값을 업데이트하여 페이지 폴트의 결과를 알린다.

4.3.2 Submission Unit

Submission Unit은 SMU에서 NVMe I/O를 수행하기 위해 메모리 공간을 할당하고, NVMe 커맨드를 작성하고, NVMe에 I/O를 요청한다. 이는 Memory Interface, Free Page Fetcher, Command Generator로 구성되어 있다.

Memory Interface는 Submission Unit에서 발생시키는 모든 메모리 동작을 SMU 컨트롤러의 DMA Module로 전달하고, 결과를 내부 구성요소로 전달하는 역할을 수행한다. Free Page Fetcher는 OS에서 할당해 준 페이지를 가져와서 I/O에 활용할 수 있게 하는 Submission Unit의 구성요소이다. OS는 Free page pool이라는 원형 큐를 생성하고, SMU가 사용할 수 있는 페이지의 PFN을 집어넣는다. SMU는 이 정보를 꺼내서 I/O에 활용한다. 통신은 인터페이스 레지스터를 통해 이루어진다. 인터페이스 레지스터는 Free page pool의 Base address, Head 및 Tail 값을 가지고 있다. OS는 큐 구조를 생성할 때 Base address를

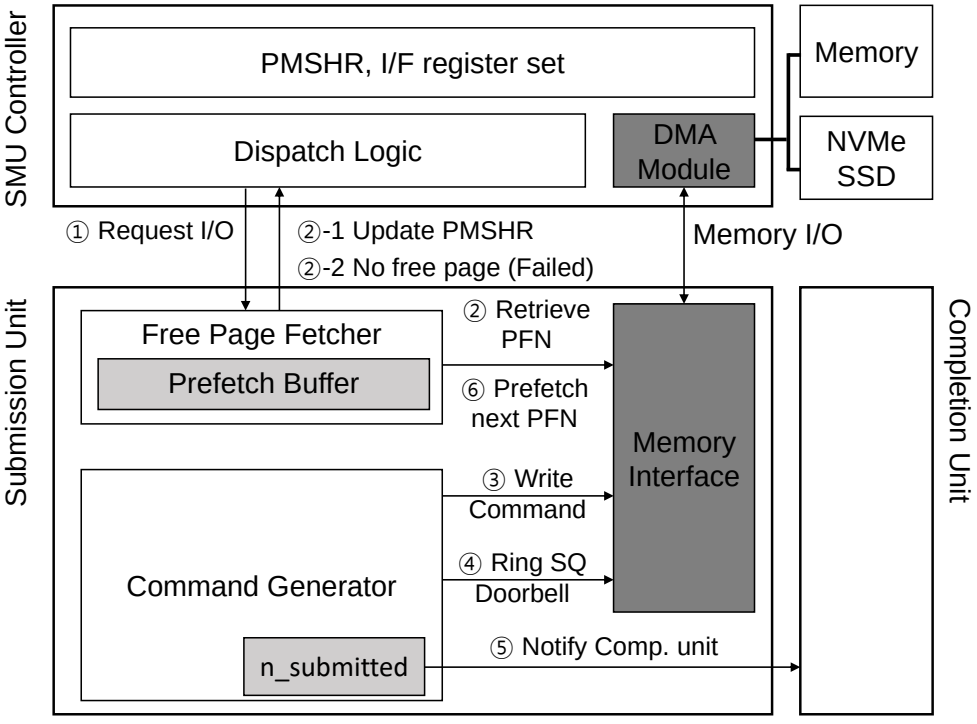

Figure 4.5: SMU Submission Unit의 구조

SMU의 하드웨어 레지스터에 기록한다. 또한 새로운 페이지를 할당시 페이지의 PFN을 큐에 집어넣고 Tail 레지스터의 값을 업데이트한다. SMU가 페이지를 꺼내서 사용할 때에는 Head 레지스터를 업데이트 함으로서 페이지를 사용했음을 OS에 알린다.

(1)새로운 페이지 폴트 처리 요청이 들어오면, (2)Free Page Fetcher는 Free page pool의 Head/Tail 레지스터를 검사, 메모리 페이지를 할당할 수 있는지를 먼저 확인한다. (2-1)페이지를 할당받는데 성공하면 페이지 테이블 업데이트를 위해 PMSHR의 PFN 필드를 업데이트 한다. (2-2)페이지를 할당받을 수 없다면, SMU 컨트롤러에 페이지 폴트의 처리가 실패했음을 알린다. 이후의 페이지 폴트 과정은 OS의 페이지 폴트 핸들러가 처리하게 된다. Free Page Fetcher가 페이지를 할당받았다면, Command Generator는 SMU 컨트롤러에서 전달받은

LBA 정보와 할당받은 페이지를 이용해 NVMe 커맨드를 생성한다. 그리고 어떤 PMSHR 엔트리에 해당하는 정보인지 표시하기 위해, NVMe submission 커맨드의 cmdid란에 PMSHR 엔트리 번호를 쓴다. (3)이후, SMU 컨트롤러의 DMA 모듈을 통해 메인 메모리에 커맨드를 복사하고, (4)NVMe SSD의 도어벨 레지스터에 값을 쓰는 것으로 NVMe를 동작시킨다. (5)이와 동시에 Completion Unit에 연결된 n_submitted 레지스터를 업데이트해 새로운 커맨드를 전달했음을 알린다.

Submission Unit은 명령을 처리한 후, 다음 명령을 빠르게 처리할 수 있게 하기 위해 메모리 페이지를 미리 가져오는 prefetch기능을 가지고 있다. 이 동작은 Completion Unit에 NVMe 커맨드를 전달했음을 알린 후에 Free page fetcher에 의해 이루어진다. Free page queue의 레지스터를 확인해 메모리 페이지를 할당받을 수 있다면 메모리 페이지를 할당받고, Free page fetcher 내부의 Prefetch register에 저장해 둔다. 이렇게 저장해 둔 값은 다음 번 명령을 내릴 때에 메모리 접근 없이 빠르게 NVMe 커맨드를 생성할 수 있게 하여, SMU의 동작을 더욱 가속시킨다. 다음에 사용할 페이지를 가져오는것까지 완료하였다면, Submission Unit은 SMU 컨트롤러로부터 다음 명령을 받기 위해 대기한다. 단, Free page pool이 비어 있어서 더 이상 페이지를 가져올 수 없다면, 미리 할당해 둔 페이지가 없다고 표시하고 대기한다.

4.3.3 Completion Unit

Completion Unit은 Submission Unit에서 내린 명령이 완료되었는지 확인하고, 처리 결과를 SMU 컨트롤러에 알려주는 기능을 수행한다. Completion Unit은 Memory Interface, CQ Poller 그리고 Page Table Updater로 구성되어 있다. Memory Interface는 Submission Unit의 그것과 마찬가지로 Completion Unit에서 발생시키는 모든 메모리 접근을 SMU 컨트롤러에 전달하고, 결과를 내

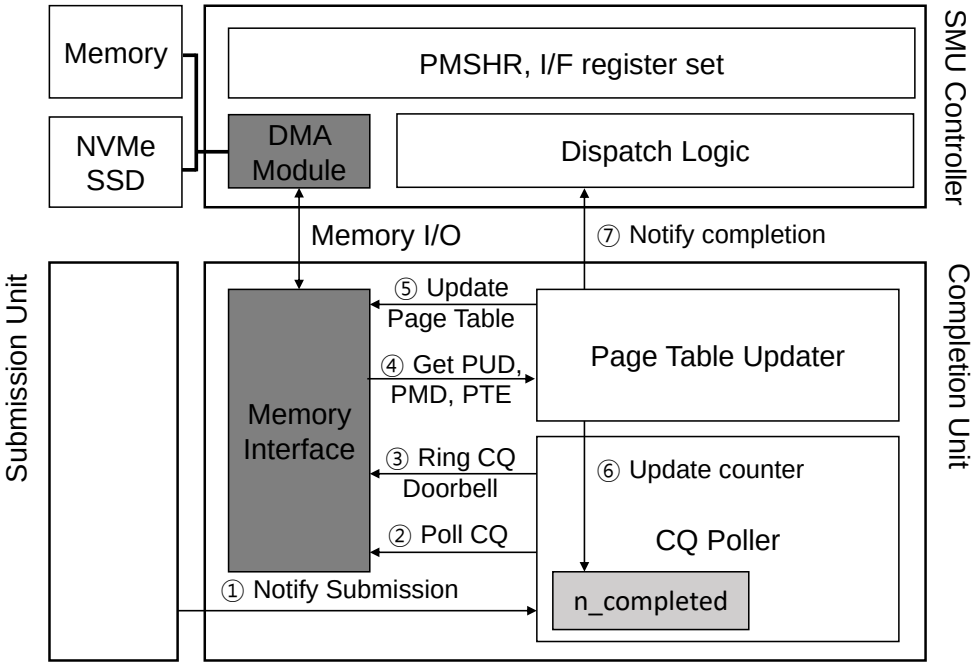

Figure 4.6: SMU Completion Unit의 구조

부 구성요소로 전달하는 역할을 수행한다. CQ Poller는 NVMe CQ에서 명령이 완료되는 것을 확인하고, NVMe에게 명령이 접수되었음을 알리는 하드웨어이다. Page Table Updater는 완료된 정보에 따라 페이지 테이블을 업데이트하고 SMU 컨트롤러에 NVMe I/O의 처리가 완료되었음을 알린다.

그림 4.6은 Completion Unit의 처리 과정을 나타낸 것이다. (1) CQ poller는 Submission Unit에서 내린 명령의 갯수(n_submitted 레지스터)와 Completion Unit에서 처리 완료한 명령의 갯수(n_completed)가 다르면, (2) CQ poller에서 NVMe completion queue (CQ)를 확인하여 NVMe 커맨드의 처리 결과를 폴링한다. (3) 처리가 완료된 것을 확인하면, Memory Interface를 이용해 NVMe SSD의 CQ doorbell 레지스터를 업데이트함으로써, NVMe에게 해당 completion 정보가 접수되었음을 알린다. 다음으로, Page Table Updater는 CQ entry

의 cmdid 필드를 확인해, 어떤 PMSHR 엔트리에 해당하는 커맨드가 처리되었는지 확인하고, (4)PMSHR의 값을 읽어 업데이트 해야할 PUD, PMD, PTE의 주소 및 디스크에서 가져 온 데이터가 담겨있는 Page Frame Number(PFN)을 가져온다. (5)Page Table Updater는 전달받은 주소에 해당하는 PUD와 PMD의 값을 읽어서, LBA bit를 추가한 값을 업데이트한다. 그리고, 할당받은 PFN을 이용해 PTE 값을 생성하고, Protection bit, LBA bit 및 Present bit을 추가해 PTE를 업데이트한다. (6) 이후, n_completed 레지스터의 값을 1 증가시켜 completion을 처리하였음을 표시한다. (7) 마지막으로, Completion Unit은 SMU 컨트롤러에게 어떤 PMSHR 엔트리에 해당하는 정보가 처리되었는지 알리고, 그 다음 NVMe 커맨드의 완료를 대기한다.

4.3.4 Submission Unit과 Completion Unit 사이의 통신

기존에 제시되었던 하드웨어 기반 디맨드 페이징 기법에서는 NVMe I/O 수행이 완료된 것을 확인하기 위해 메모리 상의 트랜잭션을 확인하는 스누핑(snooping) 기법을 사용한다. 하지만 MMIO로 구현된 SMU에서는 이 기법을 적용하기가 어려워, NVMe CQ를 폴링해서 완료를 확인하는 방식으로 변경되었다. 하지만 메모리 폴링은 지속적으로 메모리 동작을 발생시키고, 이로 인해 메모리를 사용하는 다른 하드웨어 구성 요소들의 성능에도 영향을 줄 수 있는 단점이 있다. 메모리 폴링이 다른 작업에 주는 영향을 최소화하기 위해, 본 연구에서는 필요한 만큼만 메모리 위치를 폴링하는 프로토콜을 구현하였다. Submission Unit은 n_submitted 레지스터를 가지고 있는데, 이 레지스터는 NVMe SSD에 I/O를 지시할 때 값이 1만큼 증가한다. Completion Unit은 n_completed 레지스터를 가지고 있고, 이 레지스터의 값은 NVMe SSD의 I/O 완료를 확인할 때마다 1 증가하게 된다. Completion Unit은 이 둘의 갯수가 다를 때에만 NVMe CQ를 폴링함으로써, 불필요한 메모리 동작을 줄이게 하였다. 이러한 프로토콜을

적용함으로써 불필요한 메모리 접근을 줄일 수 있을 뿐만 아니라, Submission Unit이 내린 모든 명령에 대해 빠짐없이 완료 상태를 확인할 수 있다. NVMe에서 완료한 명령은 완료된 순서대로 NVMe CQ에 기록되므로, 동시에 여러 개의 NVMe I/O가 완료되거나 Completion Unit의 처리 중에 I/O가 완료되더라도 Completion Unit에서는 이를 순차적으로 처리할 수 있다. 하나의 NVMe I/O를 확인하고 처리하는 동안 다른 것을 처리할 수 없는 구조이지만, 추후에 파이프라인(pipeline) 등의 기법을 통해 이를 가속할 수도 있을 것이다.

4.4 운영체제 지원

선행 연구인 하드웨어 디맨드 페이징 기법은 별도의 명령어(instruction)를 사용해 SMU와 OS가 통신하는 구조이다. 하지만 이 방법은 MMIO를 프로토콜로 사용하는 본 연구의 FPGA 구현에서는 사용할 수 없어, OS에서 SMU에 데이터를 전달할 때 MMIO를 사용하도록 수정하였다. SMU의 인터페이스 레지스터에 맞게 구조체를 구현하고, 이를 OS 커널의 가상 메모리 주소에 매핑해서 커널에서 메모리 I/O를 통해 접근할 수 있도록 구현하였다.

제 5 장 성능 평가

5.1 실험 환경 및 워크로드

본 연구에서는 4장에서 구현한 하드웨어 기반 디맨드 페이징의 성능을 평가하기 위해 FIO [10] 벤치마크를 사용하였다. 실험에 사용된 FPGA 플랫폼의 환경은 표 4.1과 동일하다. FIO 벤치마크는 사용자가 지정한 파일, 스레드 갯수, 접근 방식(I/O 엔진), 접근 패턴, 워크로드 크기에 따라 I/O를 수행하고, I/O 레이턴시 및 대역폭을 평가하는 벤치마크이다.

	No refill	Small	Large	Note
파일 크기	320MB	4GB	16GB	
접근 크기	64MB	64MB	2GB	스레드당
Free page pool 크기	320MB	4MB	4MB	
스레드 갯수	1	1, 2, 4	1, 2, 4	

Table 5.1: 평가 환경

본 연구에서는 메모리 매핑된 파일의 접근 성능을 평가하기 위해, FIO 3.19-27을 일부 수정하여 사용하였다. 수정한 부분은 다음과 같다. 하드웨어 기반 디맨드 페이징을 사용할 때, *mmap()* 함수에 플래그를 추가하여 OS가 LBA-augmented Page Table을 생성하게 하였다. 또한, 멀티스레드 환경의 경우 FIO 벤치마크는 각각의 스레드가 주어진 파일에 대해 *mmap()*을 수행하는데, 이는 각각의 스레드가 같은 파일을 바라보더라도, 서로 다른 페이지 테이블 영역을 접근하게 되어, 중복된 페이지 폴트를 제대로 처리하는지 확인하기가 어렵다. 이 문제를 해결하기 위해 *mmap()*을 모든 스레드에 대해 한번만 수행하고, 각각의

스레드가 같은 가상 주소를 바라보게 하였다.

본 연구에서는 하드웨어 기반 페이지 폴트 메커니즘 자체의 성능 이득과, 하드웨어로 해당 메커니즘을 오프로딩 했을 때의 성능 향상 폭을 세밀하게 측정하기 위해, 하드웨어 기반 디맨드 페이징의 세 가지 구현에 대해 성능을 평가하였다. 첫 번째 구현은, 하드웨어 디맨드 페이징의 기능을 OS의 페이지 폴트 핸들러를 이용해 소프트웨어적으로 똑같이 구현한 것이다 (Emulated). 두 번째는, 페이지 폴트 발생시 일반적인 페이지 폴트와 마찬가지로 트랩을 걸되, OS의 페이지 폴트 핸들러에서 해당 페이지 폴트가 하드웨어 기반으로 처리될 수 있다면, SMU에 명령을 내려서 페이지 폴트를 처리하는 구현이다(HWDP-trap). 마지막으로, MMU에서 OS 트랩 없이 SMU에 직접 명령을 내려 페이지 폴트를 처리하는 구현이다(HWDP). 표 5.1은 실험에서 사용한 FIO의 파라미터 설정을 정리한 것이다. 접근 패턴은 세 가지 실험 모두 4KB random read이다.

5.2 페이지 폴트 성능

FIO의 mmap엔진의 동작 과정은 그림 5.1과 같다. mmap엔진은 자기 자신이 가진 버퍼에 메모리 매핑된 파일의 내용을 *memcpy()* 함수를 이용해 복사하고, 그 성능을 측정한다. 만일 메모리에 접근하는 과정에서 페이지 폴트가 발생한다면, I/O 레이턴시는 (페이지 폴트 처리 시간 + 메모리 복사 시간)이 되고, 이는 그림 5.1에서의 t1 + t2와 같다.

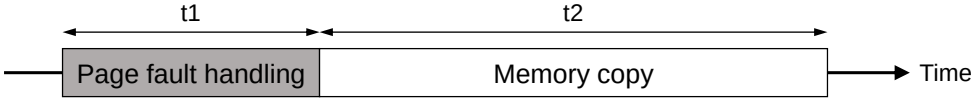

Figure 5.1: FIO mmap엔진 수행시의 타임라인

또한, 하드웨어 기반 디멘드 페이징을 사용하여 처리하는 과정에서 페이지 재충전이 발생한다면, 이 또한 레이턴시에 반영될 것이다. 따라서, 페이지 풀 크

기를 충분히 크게 설정한다면, 페이지 재충전의 레이턴시를 제외한 페이지 폴트 시간을 볼 수 있을 것이다. 메모리 복사 시간과 페이지 재충전 시간을 제외하고, 순수하게 페이지 폴트 처리에 걸리는 시간(그림 5.1에서의 t1)만을 측정하기 위해 벤치마크를 수행하였다.

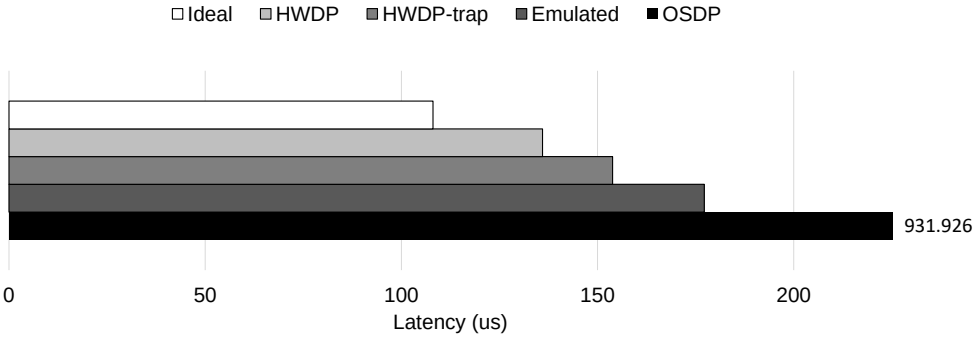

Figure 5.2: 페이지 재충전이 없는 상황에서의 FIO 레이턴시

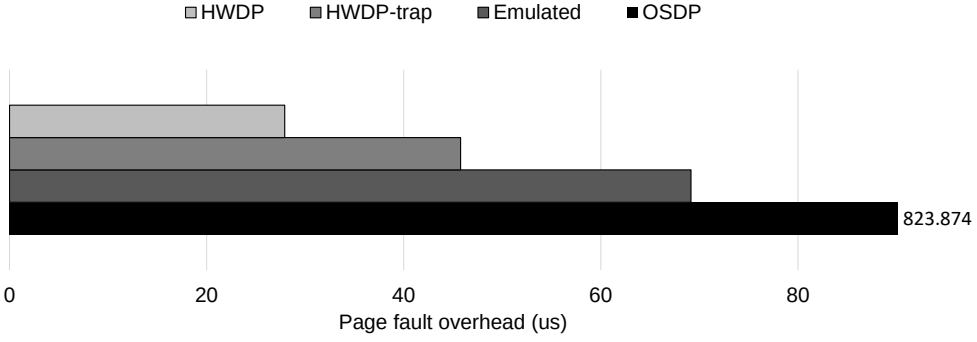

Figure 5.3: FIO 수행시의 페이지 폴트 처리 오버헤드

실험에 사용된 파라미터는 표 5.1의 No refill 항목과 같다. 64MB를 접근하는 데에 필요한 페이지의 갯수는 16384개로, 페이지 풀 크기를 이보다 크게 설정하면 페이지 폴트 처리 과정 중에는 페이지 재충전이 발생하지 않게 된다. 이를 위해 페이지 풀 크기를 충분히 크게(320MB, 81920개) 두었다. 파일 크기는 후술할 *MAP_POPULATE*를 이용한 평가를 수행하기 위해, 메인 메모리 사이즈보다 작은 320MB로 설정하였다. 또한, 백그라운드 스레드의 개입을 최

소화하기 위해, KPoold는 끈 상태로 실험을 수행하였다. 그림 5.2는 해당 실험 결과를 나타낸 것이다.

우선, 메모리 복사의 소프트웨어 오버헤드만을 측정하기 위해 실험을 수행하였다. 파일 크기, 접근 단위 크기, 접근 패턴 등은 모두 같게 설정하되, 파일을 램디스크에 로드하고, *mmap()* 수행시 *MAP_POPULATE* 플래그를 이용해 파일을 접근하는 과정에서 어떠한 페이지 폴트도 발생하지 않게 하고 벤치마크를 수행하였다(ideal). 이 방법을 이용하면 그림 5.1에서 t2의 시간만을 측정할 수 있다. 또한, 비교군으로 기존의 OS 기반 디멘드 페이징의 성능을 측정하였다(OSDP). 하드웨어 기반 디맨드 페이징을 평가하는데 사용한 것과 같은 FIO 벤치마크를 사용하되, *mmap()*시 LBA-augmented Page Table이 아닌 일반적인 페이지 테이블을 생성하도록 플래그를 수정하고 실행하였다.

실험군으로는 하드웨어 디맨드 페이징의 세 가지 구현을 사용하였다. Emulated는 OS 페이지 폴트 핸들러를 이용해 에뮬레이션 한 하드웨어 디맨드 페이징 구현이다. HWDP-trap은 페이지 폴트 발생시 OS 트랩을 걸되, 페이지 폴트 핸들러에서 SMU로 명령을 내리는 구현이다. HWDP는 MMU에서 SMU로 직접 명령을 내려 OS 트랩 없이 페이지 폴트를 처리하는 구현이다.

그림 5.3은 위에서 수행한 실험의 각 결과에서 어플리케이션 오버헤드(그림 5.1의 t2)를 제외한 값을 나타낸 것이다. 이는 그림 5.1의 t1과 같다. OSDP, Emulated, HWDP-trap, HWDP는 각각 823.874us, 69.1us, 45.8us, 27.9us의 레이턴시를 나타내었다. 각각의 레이턴시는 OSDP 대비 각각 11.9배(emulated), 18.0배(HWDP-trap), 29.5배(HWDP) 향상되었다.

그림 5.4는 위의 결과에서, 각각의 동작 방식을 더욱 자세하게 분석한 것이다. 각각의 페이지 폴트 시간을 Before(SW), Disk access, After(SW), Others로 분석하였다. Before(SW)는 커널의 페이지 폴트 핸들러 진입 시점부터 디바이스(SMU 혹은 NVMe SSD) 동작 직전까지, Disk access는 Emulated에서의

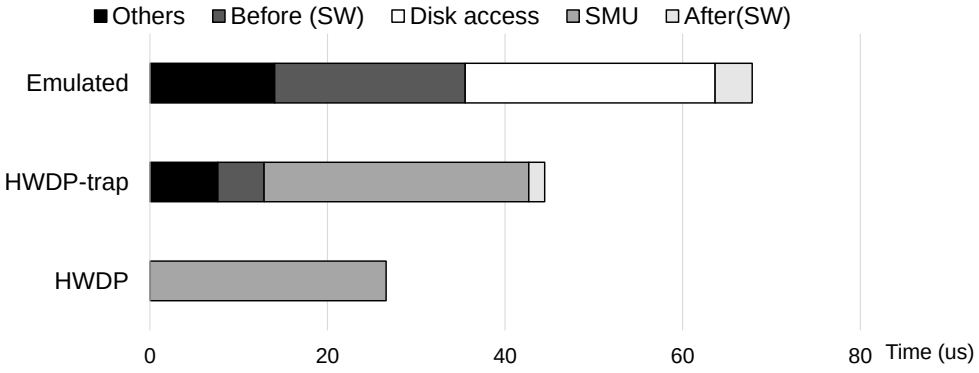

Figure 5.4: 페이지 폴트 시간 분석

NVMe SQ doorbell write부터 NVMe CQ doorbell write 직전까지의 시간이다. SMU는 HWDP 및 HWDP-trap에서 SMU 하드웨어의 동작 시간을 나타낸다. After(SW)는 커널의 페이지 폴트 핸들러에서 디바이스(SMU 혹은 NVMe SSD) 접근이 끝난 후부터 폴트 핸들러의 리턴 까지의 시간이다. Others는 그림 5.3의 각각의 시간에서 페이지 폴트를 처리하는 커널 소프트웨어 및 디바이스 시간을 뺀 것이다. 이 부분은 문맥 교환 시간 및 페이지 폴트 처리에 의한 자원 오염으로 인해 길어진 소프트웨어 수행 시간이 포함된다. Emulated의 경우 전체 시간 69.1us 중 28.1us가 디스크 접근에, 나머지인 41.0us가 디스크 접근 이외의 시간에 사용되었다. 이는 전체 페이지 폴트 처리 시간의 66.6%에 달하는 시간이다. HWDP-trap에서는 전체 시간인 45.8us 중 34.9%인 16.0us만이 하드웨어 접근 이외의 시간에 사용되었다. 마지막으로, 완전히 하드웨어로만 구현된 HWDP는 디스크 시간에서 오차범위 이내인 27.9us의 시간만으로 페이지 폴트 처리를 성공적으로 수행하는 것을 확인할 수 있었다.

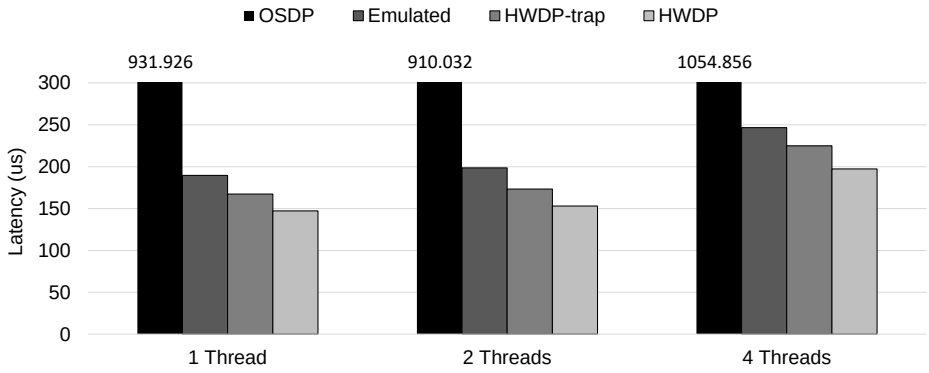

Figure 5.5: Small 워크로드에서 FIO의 평균 레이턴시(KPoold 없음)

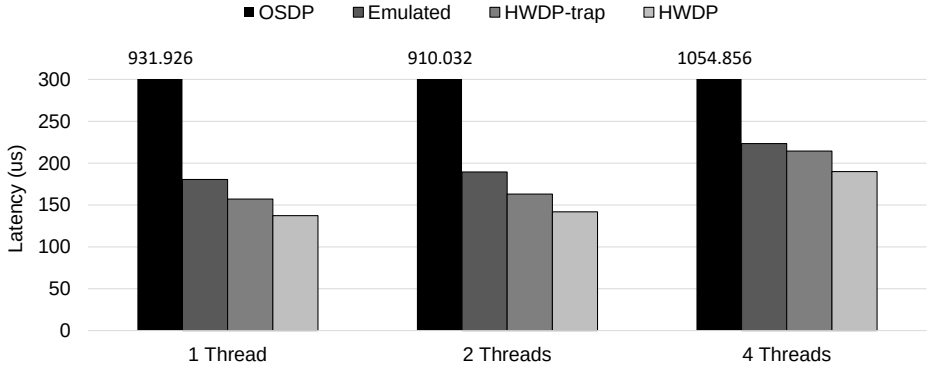

Figure 5.6: Small 워크로드에서 FIO의 평균 레이턴시

5.3 메모리보다 작은(small) 워크로드에서의 성능 평가

여기에서는 페이지 재충전이 발생하는 환경에 대해서 실험하였다. 이 실험에서는 DRAM보다 작은 사이즈에 대해서만 I/O를 수행해, 디스크로 페이지를 다시 내려보내지 않는 상황에 대해 실험하였다. 실험에 사용된 파라미터는 표 5.1의 Small 항목과 같다. 같은 실험을 페이지 재충전을 수행하는 커널 스레드인 KPoold를 껐을 때와 켰을 때, 두가지 조건에서 수행하였다.

그림 5.5는 KPoold를 끄고 수행했을 때의 평균 레이턴시이다. 수행 결과, HWDP는 OSDP 대비 최소 5.35배에서 6.33배, Emulated 대비 최소 1.24배

에서 1.29배의 성능 향상을 보였다. 4스레드 워크로드에서는 레이턴시가 크게 증가하는 것을 확인할 수 있는데, 어플리케이션 스레드와 백그라운드 스레드의 경쟁(contention)에 의한 것으로 보인다. 그림 5.6은 KPoold를 켜고 같은 FIO 워크로드를 수행한 레이턴시 결과이다. 수행 결과 HWDP는 OSDP 대비 5.55배에서 6.78배의 성능 향상을 보였다.

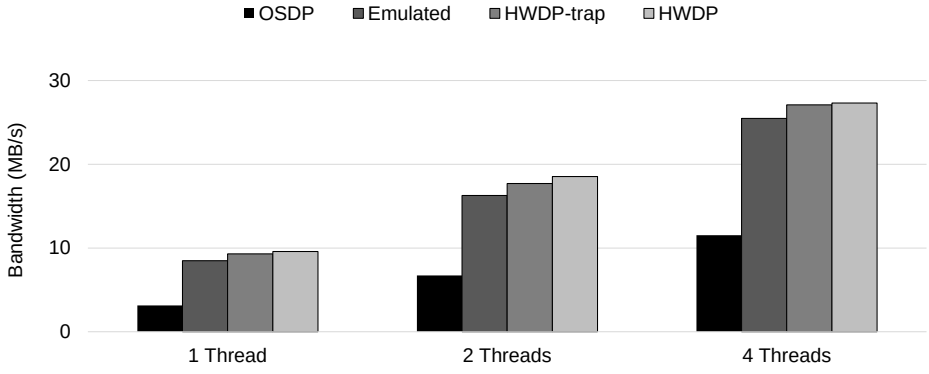

Figure 5.7: Small 워크로드에서 FIO의 평균 대역폭(KPoold 없음)

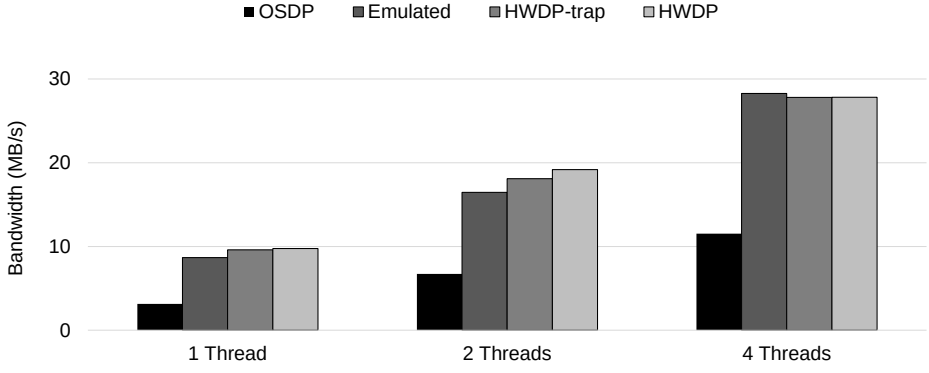

Figure 5.8: Small 워크로드에서 FIO의 평균 대역폭

그림 5.7과 그림 5.8은 같은 워크로드에서 디스크의 대역폭(bandwidth)을 측정한 결과이다. Kpoold가 없을 때에 HWDP의 디스크 대역폭은 OSDP 대비 최소 2.37배에서 3.05배, 소프트웨어로 에뮬레이션 한 Emulated 대비 최소 1.07배에서 1.12배 향상되었다. Kpoold는 백그라운드에서 페이지를 재충전함으로

써, MMU가 예외를 발생시키는 빈도를 줄인다. 그 결과, Kpoold가 없을 때 대비 최소 3.92%에서 최대 7.16%까지의 레이턴시 향상, 1.83%에서 3.45%의 대역폭 향상이 있었다. 스레드 수가 많을 때에는 성능 향상폭이 적은 것을 볼 수 있는데, 이는 Kpoold와 다른 백그라운드 스레드, 그리고 FIO 어플리케이션 스레드의 경쟁 때문인 것으로 분석된다. 그리고 4스레드 워크로드에서의 대역폭은 HWDP와 HWDP-trap 모두 emulated에 비해 감소하였다. 이는 백그라운드 스레드인 Kpoold가 더욱 빈번하게 작동하여 FIO 스레드와 경쟁했기 때문으로 분석된다.

5.4 메모리보다 큰(large) 워크로드에서의 성능 평가

마지막으로, DRAM보다 큰 워크로드에 대해 I/O를 수행해, 메모리에 올라온 데이터가 추방(eviction)되는 상황에서의 성능을 평가하였다. 실험에 사용된 파라미터는 표 5.1의 Large 항목과 같다. Small 워크로드와 마찬가지로, 같은 실험을 KPoold를 껐을 때와 켰을 때, 두가지 조건에서 수행하였다.

그림 5.9는 KPoold를 끄고, 그림 5.10은 KPoold를 켜고 레이턴시를 측정한 것이다. 수행 결과, HWDP는 OSDP 대비 최소 4.24배에서 6.12배, Emulated 대비 최소 1.09배에서 1.27배의 레이턴시 향상을 보였다.

그림 5.11은 KPoold를 끄고, 그림 5.12는 KPoold를 켜고 디스크 대역폭을 측정한 것이다. 수행 결과, HWDP의 대역폭은 OSDP 대비 최소 2.82배에서 3.42배, Emulated 대비 최소 1.14배에서 1.15배까지 향상되었다.

Large 워크로드에서는 page replacement로 인해 small 워크로드 수행시 대비 레이턴시가 증가하고 대역폭 향상폭이 낮아진 것을 확인할 수 있다. 특히, CPU 코어 갯수와 같은 수(4스레드)의 스레드로 동작시켰을 때 이러한 특징이 크게 나타났다. 이는 어플리케이션 스레드와 백그라운드 스레드들 사이에서 문맥 교환이 자주 일어나 레이턴시가 길어진 것으로 추정된다. 그럼에도 불구하고 하드웨어 디맨드 페이징은 OSDP 대비 최소 4.24배 빠른 레이턴시와 2.82배

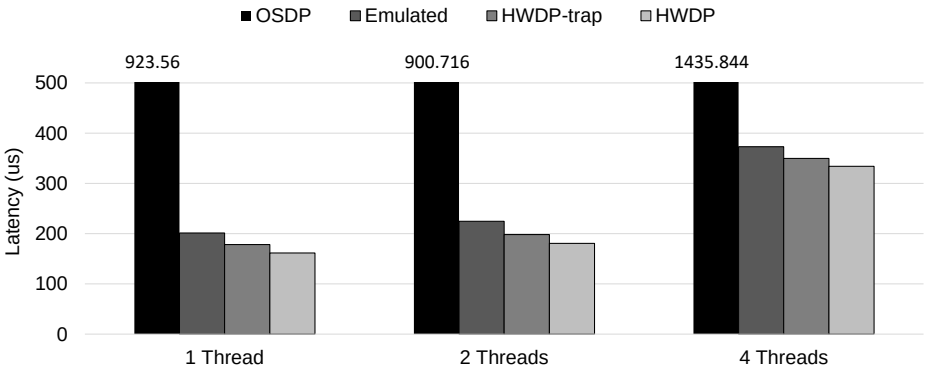

Figure 5.9: Large 워크로드에서 FIO의 평균 레이턴시(KPoold 없음)

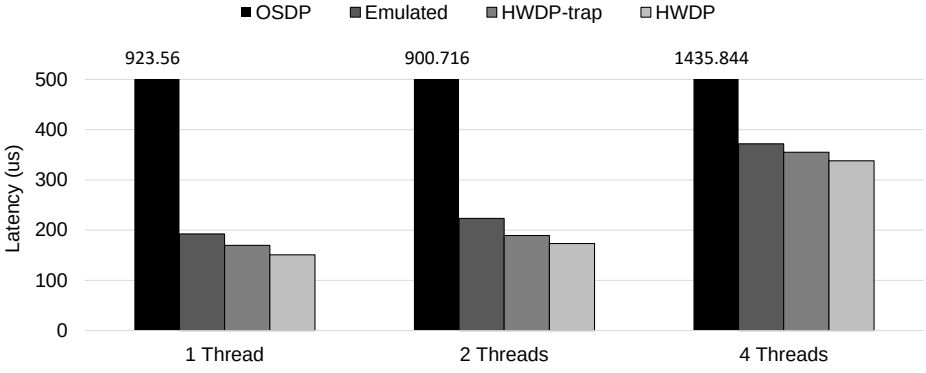

Figure 5.10: Large 워크로드에서 FIO의 평균 레이턴시

높은 대역폭을 보이면서 효과적으로 페이지 폴트를 처리하는 것을 확인할 수 있었다.

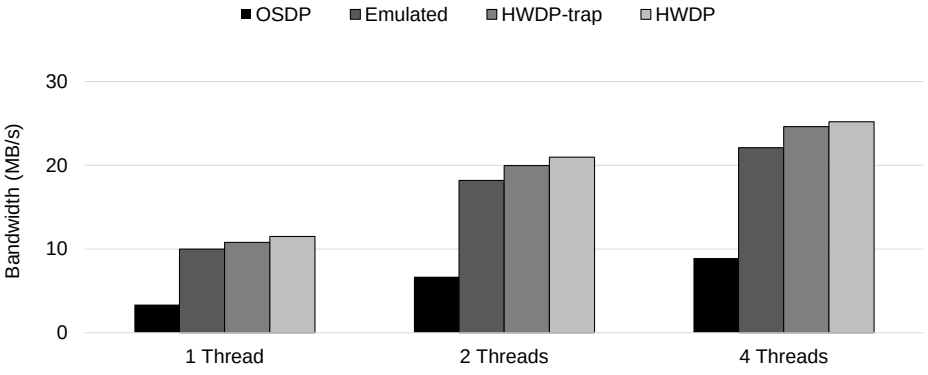

Figure 5.11: Large 워크로드에서 FIO의 평균 대역폭(KPoold 없음)

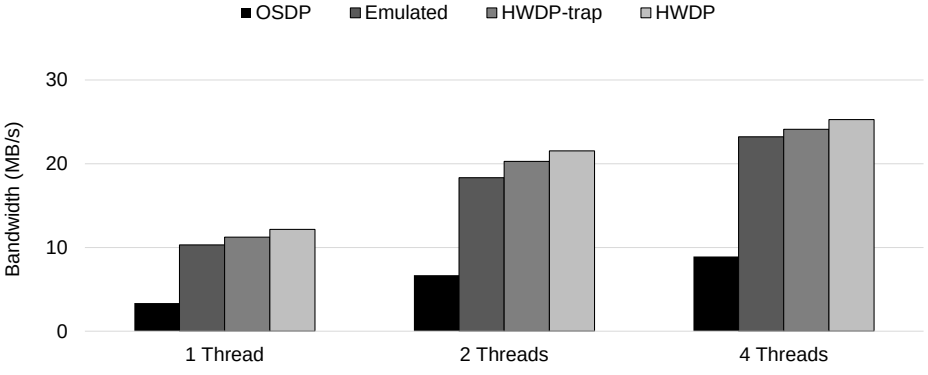

Figure 5.12: Large 워크로드에서 FIO의 평균 대역폭

제 6 장 관련 연구

6.1 SSD에서 메모리 매핑된 파일의 최적화

꾸준히 증가하는 메모리 수요로 인해, 메모리 매핑된 파일을 최적화하는 것은 중요한 연구 주제이다. SSDAlloc와 Chameleon [2, 3] 은 SSD를 이용해 메인 메모리의 용량을 확장하고자 하는 최초의 시도이다. 이 연구들은 가상 주소 (Virtual address) 영역이 DRAM뿐 아니라 SSD의 주소로도 번역될 수 있게 하였다. ZnG [31]는 GPU의 메모리를 용량이 큰 SSD로 대체하고 이에 효과적으로 접근하는 기법을 제시하였다. FlatFlash [1]는 바이트 단위로 접근 가능한 SSD와 *Promotion lookaside buffer*라는 구조를 활용하여 자주 사용하는 페이지와 그렇지 않은 페이지를 분리해 관리하는 페이지 프로모션 기법을 제시하였다. FlashMap [11]은 *Unified address*라는 구조를 이용해 메모리, 스토리지, 디바이스 사이의 3단계 주소 번역 오버헤드를 완화하였다.

이러한 기법들은 큰 메모리 공간을 효과적으로 확보하나, OS가 처리하는 오버헤드를 완전히 제거하지는 못 하였다. 이는 초저지연 SSD를 사용하는 경우에는 병목으로 작용할 수 있는 문제가 존재한다. 하지만, 하드웨어 디맨드 페이징 기법은 페이지 폴트 과정에서 OS의 개입을 완전히 배제함으로써, 효과적으로 성능 향상을 이끌어냈다.

6.2 초저지연 SSD에서의 I/O 레이턴시 단축 기법

초저지연 SSD의 접근 시간을 단축하기 위해 유저 I/O 드라이버 [12, 16, 22, 24], 파일 시스템 [15, 17] 및 복합적인 방법 [7] 등이 제시되었다. 이러한 기법들은 불필요한 커널 스택을 우회하고, 유저 수준에서 최소한의 드라이버만

이용해 SSD에 접근함으로써 아주 낮은 I/O 레이턴시만으로 데이터에 접근할 수 있다. 하지만, 이러한 기법들은 사용자가 SSD의 모든 사용 권한을 가지게 되거나 [12, 16, 22], 사용자 수준에서 안전한 인터페이스를 제공하기 위해 스토리지 인터페이스를 확장 [7, 24]해야 할 필요성이 존재한다. 오버헤드가 큰 커널 I/O 스택을 최적화 하는 연구들 [19, 26]도 존재하나, 이러한 연구들은 소프트웨어적 최적화라는 한계가 있다. DC-express [30]는 스토리지 프로토콜 수준의 오버헤드를 경감하는 연구이다. DevFS [14]는 SSD 내부에 파일 시스템을 구현하고, API를 제공해 최소한의 OS 수준 오버헤드만으로 SSD에 접근할 수 있게 하였다.FlashShare [32]는 SSD I/O를 수행하는 하드웨어를 제안하였지만, 해당 하드웨어는 I/O 제출 및 완료에만 국한되어 있다. 위의 연구들과는 다르게, 하드웨어 디맨드 페이징 기법은 CPU 아키텍처의 수정을 통해 페이지 폴트로 인한 예외 발생을 대부분 제거하고, 디맨드 페이징에서 초저지연 스토리지에 효과적으로 접근할 수 있게 하였다. 또한, 실장 하드웨어상에 구현하고 성능을 평가함으로써 구현 가능성을 검증하는 데에도 성공하였다. Flash Translation Layer(FTL)이 존재하지 않는 Open Channel SSD(OC-SSD)는 LBA로 추상화된 블럭 디바이스가 아닌, 낸드 플래시 등의 SSD 미디어에 직접 접근할 수 있는 디바이스이다. LightNVM [6]과 같은 서브시스템은 OS에서 OC-SSD 인터페이스를 사용할 수 있게 해주며, 본 연구의 LBA-augmented page table에 LBA 대신 데이터가 실제로 존재하는 미디어상의 위치를 기록함으로써 디스크 접근 레이턴시를 줄일 수 있을 것으로 예상된다.

6.3 페이지 테이블 탐색 레이턴시 단축

SPECTLB, ASAP, Tempo [4, 5, 20]는 페이지 테이블 탐색 및 가상 메모리 주소 번역을 가속화하는 연구들이다. SPECTLB [4]는 CPU의 TLB에서 접근될 페이지를 예측해서 로드함으로써, TLB miss를 줄이는 기법이다. ASAP [20]는

페이지 테이블의 마지막 두 단계에 해당하는 PTE를 prefetch해서 페이지 테이블 탐색 오버헤드를 감소시키는 기법이다. Translation-triggered prefetching [5]은 페이지 테이블 탐색을 진행하는 것과 동시에 메모리 컨트롤러가 PTE가 가리키는 데이터를 읽어오게 하는 기법이다. 이러한 연구들은 하드웨어 디맨드 페이징 기법과 함께 사용되었을 때 페이지 폴트의 레이턴시를 더욱 줄일 수 있을 것으로 예상된다.

제 7 장 결론

본 논문에서는 기존에 제안된 하드웨어 기반 디맨드 페이징 기법을 실장 하드웨어에 구현하기 위한 구조를 연구하고, 이를 RISC-V CPU 기반의 FPGA 플랫폼에 구현하였다. 또한, 이를 지원하기 위한 OS를 구현 및 구동하여 실장 시스템에서 성능 평가를 수행하였다. 이를 통해 제안한 기법이 실장 하드웨어에서 구현될 수 있음을 보이고, 제안한 구조를 이용해 디스크 접근 시간에 가까운 시간으로 페이지 폴트를 처리하고 데이터를 가져올 수 있음을 보였다.

Bibliography

[1] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. FlatFlash: Exploiting the byte-accessibility of SSDs within a unified memory-storage hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 971–985. ACM, 2019. ISBN 978-1-4503-6240-5.

[2] Anirudh Badam and Vivek S Pai. SSDAlloc: hybrid SSD/RAM memory management made easy. In *Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation*, pages 211–224. USENIX Association, 2011.

[3] Anirudh Badam, Vivek S Pai, and David W Nellans. Better flash access via shape-shifting virtual memory pages. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, pages 3:1–3:14. ACM, 2013.

[4] T. W. Barr, A. L. Cox, and S. Rixner. SpecTLB: A mechanism for speculative address translation. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, pages 307–317. IEEE Press, 2011.

[5] Abhishek Bhattacharjee. Translation-triggered prefetching. In *Proceedings of the Twenty-Second International Conference on Architectural*

Support for Programming Languages and Operating Systems, pages 63–76. ACM, 2017. ISBN 978-1-4503-4465-4.

[6] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. LightNVM: The linux open-channel SSD subsystem. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies*, pages 359–374. USENIX Association, 2017. ISBN 978-1-931971-36-2.

[7] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing safe, user space access to fast, solid state disks. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 387–400. ACM, 2012. ISBN 978-1-4503-0759-8.

[8] chipsalliance. rocket-chip. `hhttps://github.com/chipsalliance/rocket-chip`.

[9] Jungsik Choi, Jiwon Kim, and Hwansoo Han. Efficient memory mapped file I/O for in-memory file systems. In *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems*, page 5. USENIX Association, 2017.

[10] fio. Flexible I/O tester. `https://github.com/axboe/fio`, 2016.

[11] Jian Huang, Anirudh Badam, Moinuddin K. Qureshi, and Karsten Schwan. Unified address translation for memory-mapped SSDs with FlashMap. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 580–591. ACM, 2015. ISBN 978-1-4503-3402-0.

[12] Intel. Storage Performance Development Kit. `http://www.spdk.io/`.

[13] Intel. Intel Optane SSD DC P4800X/P4801X. `https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-ssd-dc-p4800x-p4801x-brief.pdf`, 2018.

[14] Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. Designing a true direct-access file system with DevFS. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, pages 241–256. USENIX Association, 2018. ISBN 978-1-931971-42-3.

[15] H. Kim and J. Kim. A user-space storage I/O framework for NVMe SSDs in mobile smart devices. *Proceedings of the IEEE Transactions on Consumer Electronics*, 63(1):28–35, 2017. ISSN 1558-4127.

[16] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. NVMeDirect: A user-space I/O framework for application-specific optimization on NVMe SSDs. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems*. USENIX Association, 2016.

[17] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 460–477. ACM, 2017. ISBN 978-1-4503-5085-3.

[18] G. Lee, W. Jin, W. Song, J. Gong, J. Bae, T. J. Ham, J. W. Lee, and J. Jeong. A case for hardware-based demand paging. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Archi-*

tecture (ISCA), pages 1103–1116, 2020. doi: 10.1109/ISCA45697.2020.00093.

[19] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. Asynchronous I/O stack: A low-latency kernel I/O stack for ultra-low latency SSDs. In *Proceedings of the 2019 USENIX Annual Technical Conference*, pages 603–616. USENIX Association, 2019. ISBN 978-1-939133-03-8.

[20] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. Prefetched address translation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1023–1036. ACM, 2019. ISBN 978-1-4503-6938-1.

[21] nvme. NVM express. `https://nvmexpress.org`.

[22] openmpdk. OpenMPDK. `https://github.com/OpenMPDK/uNVMe`.

[23] David A Patterson and John L Hennessy. *Computer organization and design*. Newnes, 2013.

[24] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. *ACM Trans. Comput. Syst.*, 33(4):11:1–11:30, November 2015. ISSN 0734-2071.

[25] Samsung. Samsung Z-SSD SZ985. `https://www.samsung.com/semiconductor/global.semi.static/Brochure_Samsung_S-ZZD_SZ985_1804.pdf`, 2018.

[26] Woong Shin, Qichen Chen, Myoungwon Oh, Hyeonsang Eom, and Heon Y. Yeom. OS I/O path optimizations for flash solid-state drives. In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 483–488. USENIX Association, 2014. ISBN 978-1-931971-10-2.

[27] Sifive. Sifive Freedom. `https://github.com/sifive/freedom`, .

[28] Sifive. Sifive Freedom-u-sdk. `https://github.com/sifive/freedom-u-sdk`, .

[29] Livio Soares and Michael Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, pages 33–46. USENIX Association, 2010.

[30] Dejan Vučinić, Qingbo Wang, Cyril Guyot, Robert Mateescu, Filip Blagojević, Luiz Franca-Neto, Damien Le Moal, Trevor Bunker, Jian Xu, Steven Swanson, and Zvonimir Bandić. DC express: Shortest latency protocol for reading phase change memory over PCI express. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, pages 309–315. USENIX Association, 2014. ISBN ISBN 978-1-931971-08-9.

[31] Jie Zhang and Myoungsoo Jung. Zng: architecting gpu multi-processors with new flash for scalable data analysis. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 1064–1075. IEEE, 2020.

[32] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim

Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, and Myoungsoo Jung. Flashshare: Punching through server storage stack from kernel to firmware for ultra-low latency ssds. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 477–492. USENIX Association, 2018. ISBN 978-1-939133-08-3.

Abstract

FPGA Implementation of Hardware-based Demand Paging on RISC-V Architecture

Jeonghun Gong
Department of Computer Science and Engineering
The Graduate School
Seoul National University

Operating System (OS) controls disk I/O with a dedicated software stack. The software stack's overhead was negligible when hard disks were used as primary storage, which has relatively long access latency. However, with the emergence of modern low-latency SSDs, OS software overhead became significant in I/O handling. To address this problem, user-level I/O frameworks have been proposed and utilized for I/O latency-sensitive applications. User-level I/O frameworks can minimize the software overhead of I/O operations; however, those approaches can't be applied to demand paging where an OS handles I/O operations via page faults. The emerging alternative to the user-level I/O frameworks is the hardware-based demand paging, which minimizes the software overhead for page faults. However, its effectiveness has only been evaluated in the simulation, and thus it is unclear whether the proposed scheme is effective in real hardware. This paper proposes a design

to implement hardware-based demand paging in real hardware and evaluates performance over traditional OS-based demand paging on an FPGA-based system. With proposed design and implementation, hardware-based demand paging eliminates most of the software overhead in demand paging on real hardware and improves the performance of FIO read random benchmark by up to 85.4%.

Keywords: demand paging, virtual memory, operating systems, CPU architecture, hardware extension

Student Number: 2019-22365