Journal of The Korea Society of Computer and Information
Vol. 22 No. 4, pp. 17-24, April 2017

www.ksci.re.kr
https://doi.org/10.9708/jksci.2017.22.04.017

Development of Full Coverage Test Framework for NVMe Based Storage

Jung Kyu Park *, Jaeho Kim**

Abstract

In this paper, we propose an efficient dynamic workload balancing strategy which improves the performance of high-performance computing system. The key idea of this dynamic workload balancing strategy is to minimize execution time of each job and to maximize the system throughput by effectively using system resource such as CPU, memory. Also, this strategy dynamically allocates job by considering demanded memory size of executing job and workload status of each node. If an overload node occurs due to allocated job, the proposed scheme migrates job, executing in overload nodes, to another free nodes and reduces the waiting time and execution time of job by balancing workload of each node. Through simulation, we show that the proposed dynamic workload balancing strategy based on CPU, memory improves the performance of high-performance computing system compared to previous strategies.

▸ Keyword : Allocation, Workload, Migration, Load balancing, Simulation

I. Introduction

최근 빅 데이터, 딥 러닝, 다중 가상머신의 기술들로 인해 기존의 SATA 인터페이스 보다 더욱 빠른 입출력에 대한 요구가 발생하고 있다. 이를 위해 PCI-express 버스와 새로운 인터페이스인 NVMe가 등장하였다. NVMe는 다중의 원형 큐를 이용하여 하나의 큐를 이용하는 SATA 인터페이스보다 더욱 빠른 입출력을 제공하며 namespace, end-to-end data protection, reservation, data set management 등의 다양한 기능을 제공한다. 하지만 기존에 오랫동안 사용되었던 SATA 인터페이스와 달리 NVMe는 계속해서 새로운 기술들이 개발되고 있기 때문에 기능에 대한 체계적인 검증이 필요하다 [1].

이를 위해 다양한 NVMe 기능 검증 테스트 툴들이 존재한다. 대표적으로 tnvme라는 오픈소스 테스트 툴이 존재하지만 tnvme는 linux-2.6.35의 커널에서만 동작한다는 단점이 존재한다 [2]. 또한 tnvme는 새로운 테스트 시나리오를 추가하기 위해서 전체 프레임워크에 대한 이해가 필요하기 때문에 확장성이 한정적이라는 문제점이 존재한다. 그리고 테스트 결과를 성공, 실패 여부로만 표현하기 때문에 기능 검증에 대한 상세한 분석이 불가능하다. NVMe의 기능이 계속적으로 추가되는 상황에서 NVMe 테스트 툴은 확장성, 적용 가능성, 상세한 분석이 매우 중요하다 [3].

이런 이유로 본 논문에서는 확장성, 적용 가능성, 상세한 분석이 가능한 새로운 테스트 프레임워크를 제안한다. 본 논문에서 제안하는 프레임워크는 다수의 테스트 시나리오 쉘 스크립트들로 구성되어있으며 테스트 시나리오는 다수의 단일 명령어 스크립트로 이루어져있다. 단일 명령어 스크립트들은 NVMe 인터페이스 명령어를 제공하는 NVMe-cli (NVMe Management Command Line Interface)를 이용하여 개발하였다.

본 논문에서 제안하는 프레임워크는 사용자가 원하는 시나리오를 추가하기 위해 단순히 단일 명령어 스크립트들만을 조합하면 되기 때문에 확장성과 적용 가능성이 매우 뛰어나고 단일 명령어들로 이루어져 있다. 단일 명령어를 이용하여 컨트롤러에서 일어나는 변화를 직접 검증하고 검증 결과를 로그파일로 기록하기 때문에 상세한 분석이 가능하다.

• First Author: Jung Kyu Park, Corresponding Author: Jaeho Kim
*Jung Kyu Park (smartjkpark@swu.ac.kr), Dept. of Digital Media Design and Applications, Seoul Women's University
**Jaejo Kim (jh.kim@unist.ac.kr), School of Electrical and Computer Engineering, UNIST
• Received: 2017. 02. 23, Revised: 2017. 03. 24, Accepted: 2017. 04. 18.

논문의 구성은 다음과 같다. 2장에서는 본 논문의 주요 개념인 SATA, NVMe의 차이점과 NVMe가 제공하는 기능들을 살펴본다. 3장에서는 본 논문에서 제안하는 테스트 프레임워크의 구조를 살펴본다. 4장에서는 테스트 프레임워크를 구성하고 있는 다양한 테스트 시나리오들의 구현 방법에 대하여 설명하고 5장에서는 실제 NVMe의 기능들을 본 논문에서 제안하는 테스트 프레임워크로 테스트 한 결과를 분석하며 마지막으로 6장에서 결론을 맺는다.

II. Related Works

1. SSD

수십 년 동안 저장매체로 자리잡아왔던 하드 디스크는 CPU와 메인 메모리에 비해 굉장히 낮은 입출력 처리 속도로 인해 전체적인 시스템의 성능 하락의 주요 원인이었다 [4]. 또한 최근 웹 기반의 서비스들은 방대한 양의 데이터 처리량을 요구함으로써 새로운 저장장치 개발의 필요성이 대두되었다 [5]. 이러한 성능 요구를 충족시키기 위해 저장장치는 계속되어 진화해왔고 플래시 메모리를 기반으로 하는 SSD (Solid State Drives)가 등장하였다 [6,7]. SSD는 플래시 메모리 칩의 집적도가 향상되며 대용량 저장장치로 발전하면서 데스크톱, 노트북, 휴대폰뿐만 아니라 각종 서버에서도 사용되면서 고성능의 서비스를 제공하고 있다.

SSD는 일반 사용자뿐만 아니라 서버 시장에서도 빠르게 성장하고 있다. 그 이유는 우선 기존의 하드 디스크에 비해서 굉장히 빠른 성능을 제공하기 때문에 사용자들에게 고품질의 서비스를 제공할 수 있기 때문이다. 또한 기술이 발전함에 따라 플래시 메모리의 집적도가 높아지며 용량 대비 가격 또한 감소하고 있기 때문에 매우 다양한 분야에서 SSD가 활발히 사용되고 있다 [4,17,18].

2. SATA SSD, NVMe SSD

최근 빅 데이터, 딥러닝, 다중 가상머신 등의 기술들은 굉장히 높은 데이터 처리 속도를 요구한다. 하지만 기존에 존재하던 SSD는 SATA 인터페이스를 사용하기 때문에 물리적인 대역폭의 한계가 존재하며 그로 인해 성능의 한계를 지닌다. 본래 SATA 인터페이스는 케이블 또는 PCB (인쇄 회로 기판) 통한 점대점 연결을 위해 고안된 인터페이스로써 성능보다는 비용적인 측면에 초점을 지닌 인터페이스이기 때문에 성능의 한계를 지닐 수 밖에 없다 [1,8,9].

이를 위해 등장한 것이 PCIe 버스와 NVMe 인터페이스를 사용하는 NVMe SSD이다. SATA SSD와 NVMe SSD는 많은 차이를 지니고 있다. 기본적으로 인터페이스의 차이로써 PCIe 인터페이스는 SATA 인터페이스에 비해 더 높은 처리량을 제공한다. SATA가 최대 600MB/s의 데이터 처리속도를 제공하는 것에 반해 PCIe 인터페이스는 최대 1GB/s의 처리속도를 제공한다. 또한 SATA SSD와 NVMe SSD는 소프트웨어 계층과 하드웨어 계층에서도 큰 차이점을 보이며 이는 그림 1과 같다. 전형적인 SATA 인터페이스 기반의 SSD는 호스트 버스를 통해서 시스템에 연결되는 형태이다. SATA SSD의 I/O 요청은 AHCI (Advanced Host Controller Interface) 드라이버, 호스트 버스, AHCI 호스트 버스 어답터 (HBA)를 차례대로 거치며 SATA 컨트롤러에 도달하게 된다. 반면에 NVMe SSD의 I/O 요청은 PCIe 루트 포트를 통해 NVMe 컨트롤러에 바로 도달하게 되며 SATA 인터페이스에 반해 단순한 구조를 지니게 된다. 이러한 하드웨어 계층의 구조적인 차이로 인해 NVMe SSD는 SATA SSD보다 더욱 빠른 입출력 속도를 보인다 [10,11,12.13].

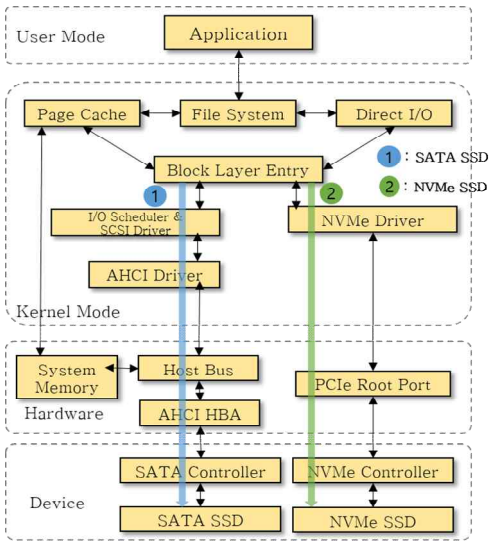


Fig. 1. Difference of Hardware Layer of SATA SSD and NVMe SSD

III. Motivation

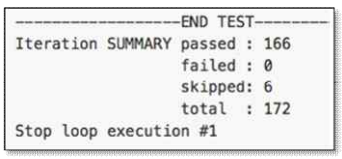
현재 NVMe는 SATA 인터페이스와 달리 개발되고 있는 단계이며 기존에 없던 새로운 기능들을 제공하기 때문에 이들에 대한 신뢰성 및 기능의 정상 동작 여부를 검증할 수 있는 프레임워크가 필요하다. 이를 위해 다수의 NVMe 기능 검증 테스트 툴들이 존재한다. 하지만 기존에 존재하던 NVMe 기능 검증 테스트 툴로써 대표적인 툴인 tnvme와 oakgate conformance 테스트 툴들은 확장성, 적용 가능성, 상세한 분석 가능성에 대하여 한계점을 지니고 있다 [2]. NVMe는 새로운 기능들이 계속해서 추가되고 있기 때문에 NVMe 테스트 툴의 확장성은 매우 중요하게 여겨진다. 하지만 오픈 소스인 tnvme는 모든 NVMe 명령어가 C++로 구현되어있으며 linux-2.6.35, Ubuntu 10.10 버전에서만 동작한다. Linux-2.6.35에는 NVMe 드라이버를 포함하고 있지 않기 때문에 자체적인 NVMe 드라이버인 dnvme를 제공한다.

이는 매우 구 버전의 커널이며 현재의 커널은 자체적으로 드

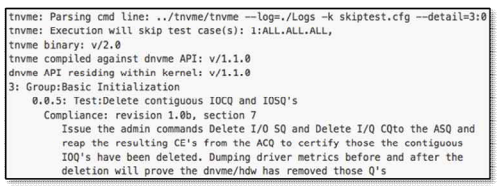
라이버를 포함하고 있기 때문에 최신 커널의 NVMe 드라이버에서 테스트를 할 수 없는 한계점을 지닌다. 또한 NVMe에 새로운 기능이 추가되는 경우에 사용자는 직접 새로운 명령어의 역할을 하는 소스 코드를 작성해야 하며 tnvme의 자체적인 NVMe 드라이버인 dnvme도 수정해야한다. 하지만 tnvme는 다양한 헤더파일과 클래스 관계를 지니고 있기 때문에 이를 분석하고 새로운 소스 코드를 추가하는 작업이 쉽지 않다 [14].

또한 새로운 테스트 시나리오를 개발하고 싶은 경우에도 tnvme와 dnvme의 모든 소스코드에 대한 이해도가 깊어야 한다. 이는 tnvme가 확장성과 적용 가능성에 매우 큰 문제를 지니고 있음을 의미한다. 또한 tnvme는 일련의 테스트들이 하나의 그룹을 이루고 있으며 하나의 그룹 안에 존재하는 테스트들은 서로 간에 의존성을 지니고 있다. 만약 테스트를 하던 도중 그룹 안에 하나의 테스트가 실패한 경우 의존성으로 인해 테스트를 처음부터 다시 수행해야 한다. 테스트 간의 의존성 문제를 해결하기 위하여 문서를 제공하고 있지만 매우 복잡하며 새로운 테스트 시나리오를 개발하고 싶은 경우에도 의존성 문제도 해결해야만 하는 한계점이 존재한다 [2].

그림 2는 tnvme의 실행결과이다. 그림 2(a)는 단순히 성공, 실패, 전체 테스트 횟수의 정보만 제공함으로써 테스트의 상세한 내용을 볼 수 없는 단점이 존재한다. 이는 NVMe의 기능을 테스트 할 수는 있지만 상세한 분석은 불가능함을 의미한다. 그림 2(b)는 테스트 결과의 상세한 부분을 보여주는 옵션을 통해 수행한 결과다. 이 또한 테스트의 수행결과를 상세히 분석하기에는 적합하지 않음을 알 수 있다.



(a) Test Result 1



(b) Test Result 2

Fig. 2. Test Results of tnvme

Oakgate conformance tool은 PCIe/NVMe, SAS, SATA, AHCI 인터페이스를 테스트하기 위한 테스트 툴로써 oakgate technology에서 제공하는 비용을 지불해야하는 테스트 툴이다. Oakgate conformance tool은 oakgate technology에서 제공하는 특정한 머신에서만 구동할 수 있기 때문에 해당 머신에 대한 비용 또한 지불해야하는 단점이 존재한다. 또한 특정 머신에서만 동작하기 때문에 테스트 환경의 제한 사항이 존재하며 오픈소스가 아니기 때문에 확장성의 문제가 존재한다. 테스트 수행결과는 테스트 항목, 테스트 시도 횟수, 성공 횟수, 실패 횟수, 진행 정도를 나타내며 성공한 테스트 항목은 초록색으로 나타내며 실패한 테스트 항목은 빨간색으로 나타내도록 구성되어 있다.

현재 oakgate conformance tool의 테스트 항목은 namespace, reservation, smart log, firmware, I/O, format 위주로 구성되며 총 104개의 테스트 시나리오가 존재한다. 하지만 NVMe에서 지원하는 end-to-end protection, dual port에 관한 테스트 시나리오는 포함하지 않고 있기 때문에 다양한 기능을 테스트할 수 없다. 이는 tnvme와 동일하게 테스트 시나리오의 적용 가능성이 부족함을 의미한다. 또한 다수의 테스트 시나리오가 오류를 지니고 있다. 예를 들어 20번 테스트 항목인 “Identify all NSID values”는 현재 컨트롤러에 생성된 namespace의 nsid를 확인하는 테스트인데 하나의 namespace가 생성되어 nsid가 1만 존재하지만 컨트롤러에서 지원할 수 있는 모든 nsid를 출력하는 오류를 지니고 있다.

이러한 문제는 50번 테스트 항목인 “Get SMART Log Page ALL Namespaces”에서도 동일하게 나타난다. 34번 테스트 항목인 “Firmware Activate All Slots”는 표 2의 펌웨어 활성화 옵션에서 아직 제공하지 않는 3번 옵션을 사용함으로써 테스트가 불가한 경우도 존재한다. 또한 86번 테스트 항목인 “Namespace Delete All Namespaces”에서는 테스트 이전에 존재하던 namespace를 제거하지 않고 컨트롤러가 지원하는 namespace 개수를 초과하여 생성을 시도하려는 오류를 지니고 있다.

이처럼 현재 Oakgate conformance tool은 내부적으로 다양한 오류를 지니고 있다. 하지만 오픈소스가 아니기 때문에 사용자가 오류를 직접 수정할 수 없으며 수정을 위해서는 okgate technology에 직접 수정을 요청해야 한다. 이처럼 기존의 NVMe 기능 테스트 툴들이 확장성, 적용 가능성, 상세한 분석에 대한 한계점을 지니고 있기 때문에 이를 해결할 수 있는 새로운 NVMe 기능 테스트 프레임워크가 필요하다.

IV. Architecture of Test Framework

기존의 NVMe 기능 테스트 툴은 확장성, 적용 가능성, 상세한 분석 가능성에 대한 한계점을 지니고 있고 본 논문에서는 이를 고려한 새로운 기능 검증 프레임워크를 제안한다. 본 논문에서 제안하는 기능 검증 프레임워크는 마이크로 스크립트와 매크로 스크립트로 구성되어있다. 마이크로 스크립트는 NVMe 인터페이스를 호스트 계층에서 사용할 수 있는 NVMe-cli (NVMe management command line interface)를 사용하여 NVMe가 제공하는 단일 기능을 수행하며 쉘 스크립트로 개발되었다. 매크로 스크립트는 기능 검증을 위한 테스트 시나리오를 포함하고 있으며 테스트 시나리오는 NVMe의 단일 기능을 수행하는 마이크로 스크립트의 조합으로 이루어진다.

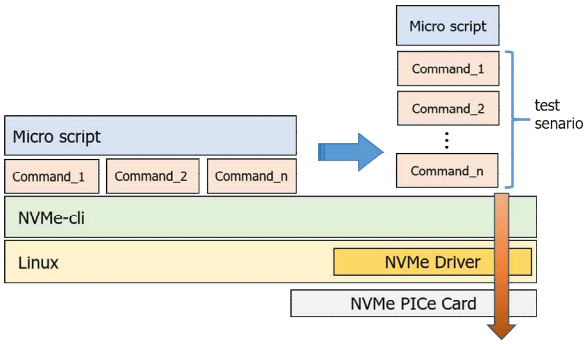


Fig. 3. Functional Verification Framework Structure Considering Scalability

그림 3에서 구성한 프레임워크의 이점은 다음과 같다. 먼저 프레임워크 내에 현재 NVMe가 제공하는 모든 기능들을 마이크로 스크립트로 구현했기 때문에 새로운 테스트 시나리오 개발은 단순히 다양한 마이크로 스크립트의 조합으로 이루어진다. 본 프레임워크를 제공하는 마이크로 스크립트 조합으로 테스트 시나리오를 개발할 수 있기 때문에 사용자가 원하는 테스트 시나리오는 모두 개발 가능하다. 이는 기존의 테스트 툴들의 문제점이었던 확장성과 적용 가능성에 대한 문제를 해결할 수 있는 새로운 구조이다.

 # Micro script
- NVMe-cli command
- micro command validity check

 # Macro script
- Test environment check
  a. Target device check
  b. Command support check
- Test environment initialization
  a. Deleting all namespace
  b. Creating/attaching namespace
- Test scenario (Set of micro script)
- Device fault check
  a. Target device check after NPOR
  b. I/O verify

Fig. 4. Structure of Micro and Macro Script

또한 논문에서 제안하는 마이크로 스크립트, 매크로 스크립트로 이루어진 프레임워크의 구조는 기존 툴들의 문제점이었던 상세한 분석 가능성도 해결할 수 있다. 마이크로 스크립트와 매크로 스크립트의 구조는 그림 4와 같고, 그림 5에서는 스크립트의 예를 표시하고 있다. 마이크로 스크립트에서는 단일 명령어의 기능 검증을 수행한다. 단일 명령어의 검증은 다음과 같이 이루어진다. 먼저 단일 명령어를 사용함으로써 NVMe-cli가 반환하는 값을 구별하여 단일 명령어의 성공 여부를 판단한다. 이에 그치지 않고 실제 컨트롤러에 해당 명령어의 수행결과가 반영되었는지도 검증한다. 예를 들어 NVMe의 create 명령어의 경우 그 결과로써 활성화되지 않은 namespace가 컨트롤러에 생성되는데 이를 확인하여 단일 명령어의 기능을 검증한다.

매크로 스크립트의 경우 테스트 시나리오로만 구성되지 않

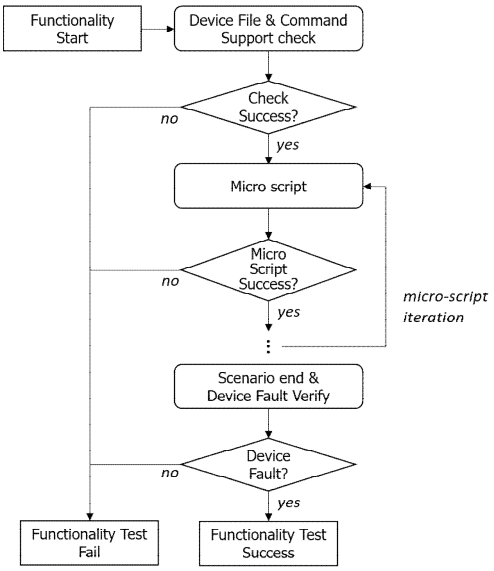


Fig. 5. Flowchart Of Macro Script Execution

고 특정한 구조를 지닌다. 크게 4 부분으로 테스트 환경 사전 검증, 테스트 환경 초기화, 테스트 시나리오, NVMe 장치파일 결함 검증으로 이루어진다. 테스트 환경 사전 검증은 테스트를 하기 위한 환경이 구성되어 있는지를 검증한다. 먼저 테스트 하고자하는 NVMe 장치파일의 존재여부를 확인하고 컨트롤러에서 해당 테스트에서 사용하고자하는 명령어를 지원하는지 여부를 확인한다. NVMe의 경우 단일 명령어의 지원여부를 컨트롤러 내부에 존재하는 자료구조의 한 변수로서 표현하기 때문에 각 명령어에 해당하는 변수를 조사하여 명령어 지원여부를 판단한다.

테스트 환경 초기화에서는 이전의 테스트에서 생성되었던 namespace들을 모두 삭제하고 다시 생성하여 테스트 환경을 초기화한다. 이후에 실제 단일 명령어들의 조합으로 이루어진 테스트 시나리오가 수행된다. 테스트 시나리오가 성공적으로 수행된 이후에는 NVMe 장치의 결함을 검증한다. NVMe 장치의 결함 검증은 테스트 시나리오를 작성하기 위해 수 많은 단일 명령어들을 반복적으로 수행해본 결과 가장 빈번하게 발생하는 장치 결함들의 목록으로 구성되어있다. 먼저 시스템에 전원 사이클을 주입했을 때 장치를 인식하지 못하는 경우, namespace와 NVMe 장치에 대한 정보를 컨트롤러 계층에서 확인했을 때 시스템이 멈추는 경우가 가장 빈번하게 발생하였다. 또한 장치에 결함이 생기지 않았음에도 I/O에 문제가 발생할 경우를 대비하여 I/O가 성공적으로 수행되는지 여부를 확인한다. 이를 그림 5에서 순서도로 표현하고 있다. 이처럼 마이크로, 매크로 스크립트에서 각각 기능과 장치에 대한 결함을 상세하게 검증하고 이를 로그 파일로도 기록하기 때문에 사용자는 NVMe 기능에 대한 상세한 분석이 가능하다.

V. Implementation

현재 테스트 프레임워크에 존재하는 모든 마이크로, 매크로 스크립트들은 쉘 스크립트로 구현하였다. 또한 마이크로 스크립트에서 NVMe 인터페이스의 단일 명령어를 수행하기 위해 NVMe 인터페이스를 호스트 계층에서 사용할 수 있도록 명령어를 제공하는 NVMe-cli를 사용하였다. 현재 테스트 프레임워크는 NVMe spec 1.2에서 제공하는 모든 인터페이스의 단일 명령어를 마이크로 스크립트로 제공하며 아래의 테스트 시나리오는 모두 마이크로 스크립트의 조합으로 구현하였다 [12,13].

1. Scripts for Pre-Verification

앞서 언급하였듯이 실제 테스트 시나리오를 수행하기 위해 다양한 테스트 환경 사전 검증을 수행해야한다. 이에 대한 검증 항목은 표 1과 같다.

Table 1. Pre-Verification Items for Test Environment

Verification items	Verification goal
Check device files	check device files for testing
oncs_dsm	check the 2nd bit of oncs variable
oncs_reservation	check the 5th bit of oncs variable
oacs_namespace	check the 3rd bit of oncs variable
oacs_firmware	check the 2nd bit of oncs variable
oacs_format	check the 1st bit of oncs variable

테스트를 하기 위한 장치파일이 현재 시스템에 존재하는지 확인하는 항목은 리눅스 시스템에서 NVMe 컨트롤러는 “/dev/nvme0“, namespace는 ”/dev/nvme0n1“ 장치파일로 인식되기 때문에 장치파일 확인을 통해 테스트 대상 장치파일의 존재여부를 확인한다 [15.16]. 또한 인터페이스 지원 여부를 확인하기 위해서 컨트롤러의 변수를 확인한다. 컨트롤러의 명령어 지원여부를 나타내는 oacs, oncs 변수는 nvme-cli의 ”id-ctrl“ 명령어로 확인가능하며 변수의 값을 비트로 구분하여 명령어 지원여부를 판단한다.

2. Scripts for Function Verificatoin

비록 모든 마이크로 스크립트에서 단일 명령어의 기능 검증을 수행하지만 테스트 시나리오로 인하여 장치 결함이 발생하는 경우가 존재한다. 가장 빈번하게 발생하는 장치 결함으로는 장치파일 인식 실패, I/O 불가능이며 이에 대한 검증 항목은 표 2와 같다.

Table 2. Device Defect Verification Items

Verification items	Test scenario
check device_1	Check whether the system's device file exists
check device_2	nvme-cli command “nvme list” Check whether the system's device file exists
check device_3	nvme-cli command “nvme list-ns” check for existence of namespace
I/O verification	check I/O availability of namespace

가장 빈번하게 발생하는 장치 결함으로써 리눅스 시스템에서 장치파일이 사라져 장치를 인식하지 못하는 경우와 장치 정보를 확인하는 nvme-cli 명령어인 “nvme list” 사용 시 장치를 인식하지 못하는 경우와 명령어 결과 시스템이 중단되는 경우가 있다. 또한 장치 인식은 가능하지만 namespace를 생성하여도 namespace를 인식하지 못하는 경우가 있기 때문에 이를 검증하는 항목을 추가하였다. 마지막으로 장치 인식, namespace 인식은 가능하지만 실제로 I/O는 불가능한 경우가 존재하기에 이를 검증하는 스크립트를 구현하였다.

Table 3. Function Verification Items for Namespace Management

Verification items	Test scenario
Namespace test 1	Repetitive create -> delete -> power cycle -> device check
Namespace test 2	Repetitive create -> detach -> attach -> device check
Namespace test 3	Repetitive create-> detach-> attach-> delete -> power cycle -> device check
Namespace test 4	Repetitive create-> attach-> detach-> delete -> power cycle -> device check
Namespace test 5	Repetitive create-> attach-> delete-> detach -> power cycle-> device check
Namespace test 6	Repetitive create-> delete-> attach-> detach -> power cycle-> device check
Namespace test 7	Repetitive create-> delete-> detach-> attach -> power cycle-> device check
Namespace test 8	Repetitive create-> detach-> delete-> attach -> power cycle-> device check
Namespace test 9,11,13	1~N times create -> Repetitive detach -> power cycle -> device check
Namespace test 10,12,14	1~N times create -> Repetitive attach -> power cycle -> device check
Namespace test 15	Repetitive create -> delete -> attach -> power cycle -> device check
Namespace test 16	Repetitive create-> delete -> detach -> power cycle-> device check

3. Scenario for Namespace Test

Namespace와 관련된 기능 검증 테스트 시나리오는 크게 두 부분으로 나누어 구현하였다. namespace management 인터페이스인 create, delete, attach, detach의 조합만으로 이루어진 테스트 시나리오와 namespace management 명령어와 format, reset, power cycle 인터페이스 조합으로 이루어진 테스트 시나리오로 구현하였다.

먼저 namespace management 검증 테스트 시나리오는 표 3과 같다. 기본적으로 모든 시나리오 이후에는 전원 사이클을 주입한 후에 장치의 결함 검증을 수행한다. 장치의 전원 사이클 주입은 Quarch Technology의 PCIe slot to M.2 Power Injection Fixture 장비를 사용하여 시스템의 전원을 끄지 않고 NVMe의 전원만 독립적으로 전원 사이클을 주입하여 효율적인 테스트가 될 수 있도록 하였다. Namespace Management의 경우 정상적인 경우와 비정상적인 경우 모두를 감안하여 다양

한 명령어 순서 조합의 테스트 스크립트를 구현하였다. 정상적인 경우는 create, attach, detach, delete 순서이다. 기본적으로 create으로 namespace를 생성한 후에 attach를 수행해야 namespace가 활성화 상태가 되며 정상적인 동작이 가능해진다. 하지만 create 후 attach를 하지 않는 경우 비활성화 상태가 되기 때문에 위의 순서가 정상적인 순서라고 할 수 있다. 비정상적인 경우는 이외의 순서를 의미한다.

예를 들어 namespace 1개를 create한 후에 해당 namespace를 계속해서 attach하거나, attach되지 않고 create을 통해 생성만 된 inactive한 namespace를 계속해서 detach, delete하는 비정상적인 경우도 모두 고려하였다. 표 4는 namespace와 관련된 다양한 명령어의 조합으로 이루어진 테스트 시나리오의 목록이다.

Table 4. Namespace Management and Combination of Other Commands

Verification items	Test scenario
Namespace test 20	namespace format -> format verify
Namespace test 21	N namespace format -> format verify
Namespace test 22	N namespace format -> power cycle -> namespace, device check
Namespace test 23	namespace create -> namespace size check by nsze
Namespace test 24	total device size check by tnvmcap
Namespace test 25	Generate more namespace than device capacity -> power cycle -> device check
Namespace test 26	list-ns -> power cycle -> list-ns -> list-ns check
Namespace test 27	N, N+1 namespace LBA overwrite -> I/O -> power cycle -> device check, I/O verify

Fig. 6. Results of Test Scenario

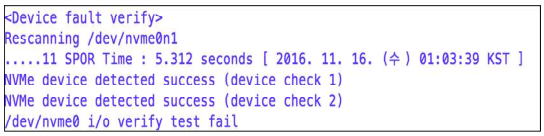


Fig. 7. Result of Devide Fault

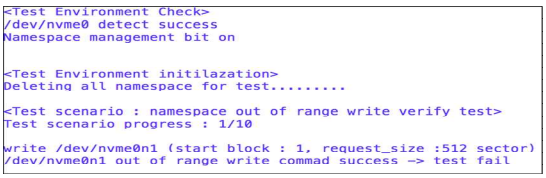


Fig. 8. Result of Function Fail

VI. Results

본 논문에서 제안한 NVMe 기능 검증 프레임워크를 평가하기 위해 실제 NVMe 장치를 대상으로 테스트를 수행하였다. 테스트에 사용된 장치는 현재 SKT 네트워크 연구소에서 개발 중인 M.2 NVMe SSD와 AIC NVMe SSD를 제공받아 테스트를 수행하였다. 각 장치에 사용된 펌웨어는 NVMe spec 1.2를 기반으로 개발되고 있는 최신의 펌웨어를 포팅하여 테스트 하였다. 테스트 환경은 Ubuntu 16.04의 운영체제, 리눅스 커널 4.2.0-42, NVMe-cli 0.9 버전을 사용하였다. 그림 7은 namespace의 create과 delete 기능을 검증하기 위한 하나의 매크로 스크립트 (테스트 시나리오)의 수행 결과다.

그림 6의 수행 결과는 크게 4 부분으로 나뉜다. 첫 번째는 테스트 환경 검증이다. 테스트의 환경검증은 테스트를 수행하고자하는 장치 파일 유무를 검사하고 NVMe 컨트롤러가 해당 시나리오에서 사용하고자하는 명령어 지원 여부를 확인한다. 두 번째는 테스트 환경 초기화다. 현재는 namespace 기능 검증을 위한 테스트 시나리오이기 때문에 테스트를 수행하기 전에 존재하던 모든 namespace를 지움으로써 테스트 환경을 초기화한다. 세 번째는 실제 테스트 시나리오가 수행되는 부분이다. 현재 테스트 시나리오를 표기하며 반복되는 테스트 시나리오 횟수를 표기함으로써 진행상황을 알 수 있도록 개발하였다. 또한 테스트 시나리오를 구성하고 있는 단일 명령어인 마이크로 스크립트의 수행결과를 상세히 나타낸다. 예를 들어 “create-ns: Success, created nsid:1은 namespace의 create 기능의 성공 혹은 실패 여부를 나타내고 생성된 namespace의 nsid까지 나타내고 있다. 모든 테스트 시나리오가 종료된 이후

에는 마지막으로 장치 결함을 검증하기 위해 총 4가지 테스트를 수행하고 결과를 나타낸다.

4가지 장치 결함을 검증하기 위한 테스트 항목은 표 2와 같다. 현재 테스트 시나리오는 성공적으로 수행된 경우이며 장치 결함이 발생한 경우 수행 결과는 그림 7과 같다. 그림 8은 표 2의 Namespace test 22의 수행 결과다. 해당 기능 검증은 테스트 시나리오는 성공적으로 수행되었지만 장치 파일에서 I/O가 실패하며 결함이 발생한 경우다.

그림 8은 테스트 시나리오를 수행하던 중 결함이 발생한 경우이다. 해당 테스트 시나리오는 namespace의 LBA 범위가 아닌 위치에 I/O를 수행하기 때문에 I/O가 수행되지 않아야 정상이다. 하지만 그림 8와 같이 namspace의 LBA 범위 외의 LBA 위치에 쓰기가 성공하였으므로 검증에 실패하였음을 수행결과로 나타낸다. 실제로 본 논문에서 제안한 기능 검증 프레임워크를 이용하여 발견한 기능 결함은 표 5와 같다. Namespace test 21, 22와 Namespace + I/O test 18의 경우 테스트 시나리오는 결함 없이 수행되었지만 장치 결함이 발생하였다. Namespace test 21, 22는 namespace의 I/O 검증이 실패하였다. 특정 LBA 영역에 쓰기를 한 후에 바로 동일한 영역을 읽어서 데이터를 비교하였지만 두 데이터가 서로 달랐다. 이는 테스트 시나리오로 인하여 데이터 무결성을 유지 못하였음을 알 수 있다.

Namespace + I/O test 18번의 경우에는 표 2의 “장치 확인_2” 검증에 실패함으로써 결함이 발생하였다고 판단했다. 테스트 시나리오는 결함 없이 수행되었지만 NVMe 전원이 꺼졌다 켜진 이후에 장치 파일을 인식하지 못하는 결과가 발생하였으며 이로 인해 컴퓨터 전체 시스템의 전원을 껐다 켜도 장치를 전혀 인식하지 못하고 공장 초기화를 필요로 하는 치명적인 결함을 보였다. Namespace test 23은 namespace를 섹터 단위로 생성한 이후 컨트롤러 변수인 nvmcap과 비교한다. nvmcap 변수는 namespace의 용량을 섹터 단위로 나타낸다. 해당 테스트 시나리오의 경우 생성한 섹터 단위와 생성 되어진 namespace의 섹터 단위를 나타내는 변수인 nvmcap이 일치하지 않음으로써 기능 검증에 실패하였다. Namespace test 25는 NVMe 장치 용량인 3.2 TB이상의 용량을 가진 namespace를 생성하는 테스트 시나리오이다. 해당 테스트 시나리오는 NVMe 장치 용량 이상의 namespace를 생성하고 전원 사이클 이후에 장치를 인식하지 못함으로써 기능 검증에 실패하였다. Namespace + I/O test 14, 16, 17, 20의 경우 NVMe가 지원하는 I/O 기능인 write, write-zeros를 특정 LBA 영역에 수행한 이후에 다시 동일한 LBA 영역을 읽어서 I/O 검증을 수행하지만 모두 쓰여진 데이터와 읽어진 데이터가 동일하지 않음으로써 기능 검증에 실패하였다.

Table 5. Test Results of M.2 MVMe SSD

Verification items	Test scenario
Namespace test 21	N namespace format -> format verify
Namespace test 22	N namespace를 format -> power cycle -> namespace, device check
Namespace test 23	namespace create -> namespace size check by nsze
Namespace test 25	Generate more namespace than device capacity -> power cycle -> device check
Namespace + I/O test 14	N namespace write -> power cycle -> I/O verify
Namespace + I/O test 16	N namespace write -> flush -> power cycle -> I/O verify
Namespace + I/O test 17	N namespace write-uncorrectable -> read -> I/O verify
Namespace + I/O test 18	N namespace write-uncorrectable -> read -> power cycle -> I/O verify
Namespace + I/O test 20	N namespace write-zeros -> read -> power cycle -> I/O verify
Namespace + I/O test 23	N namespace create -> namespace Attempts to I / O over a LBA range -> I/O verify

VII. Conclusions

본 논문에서는 기존의 NVMe 기능 검증 테스트 툴들이 확장성, 적용 가능성, 상세한 분석 가능성이 매우 취약함에 따라 이를 보완할 수 있는 프레임워크 구조를 제안하였다. 현재 NVMe가 제공하는 모든 인터페이스 명령어들에 관하여 마이크로 스크립트를 만들었고 새로운 기능 검증 테스트 시나리오를 만들기 위해서 단순히 프레임워크 내에 존재하는 마이크로 스크립트를 조합하여 개발할 수 있도록 개발하였다. 따라서 NVMe 기능이 새로 추가되는 경우에도 확장성이 매우 뛰어나며 사용자는 자신이 검증하고자 하는 테스트 시나리오를 손쉽게 개발할 수 있기 때문에 적용 가능성이 매우 뛰어나다. 또한 테스트 시나리오의 진행 상황을 상세하게 제공하며 로그 기록으로도 남기기 때문에 상세한 분석이 가능하다. 이는 실제 NVMe를 개발하는 개발자에게도 개발 비용과 시간을 줄일 수 있는 점으로 작용한다. 또한 기존의 툴들이 장비 혹은 테스트 환경과 같은 제약 사항이 많이 존재하지만 본 논문에서 제안하는 테스트 프레임워크는 장비와 테스트 환경과 같은 제약사항이 거의 존재하지 않으며 보다 유연한 테스트를 할 수 있도록 한다.

REFERENCES

[1] Enterprise SSD Interface Comparisons, http://www.seagate.com/files/www-content/product-content/_cross-product/en-us/docs/enterprise-interface-comparisons-tp625-1-1203us.pdf.

[2] NVMe Compliance Suite - High Level Test Architecture, https://github.com/nvmecompliance/tnvme/blob/master/Doc/testDependencyPreso.pdf. 2012.

[3] Nvme express, "NVMe spec 1.2", http://nvmexpress.org/wp-content/uploads/NVM_Express_1_2_Gold_20141209.pdf. 2014.

[4] C. Dirik and B. Jacob, "The Performance of PC Solid-State Disk as a Function of Bandwidth, Concurrency, Device Architecture, and System Organization," in Proc. of the ISCA09, Jun. 2009.

[5] Q. Xu, H. Siyamwala, M. Ghosh, M. Awasthi, T. Suri, Z. Guz, A. Shayesteh, and V. Balakrishnan, "Performance Characterization of Hyperscale Applications on NVMe SSDs," in Proc. of ACM SIGMETRICS, Jun. 2015.

[6] Y. Kim, S. Lee, K. Zhang. and J. Kim, "I/O Performance Optimization Techniques for Hybrid Hard Disk-Based Mobile Consumer Devices," IEEE Transactions on Consumer Electronics, pp. 1469-1476. Nov. 2007.

[7] Y. Cai, G. Yalsin, O. Mutlu, E. F. Haratsch, O. Unsal, A. Cristal, and Ken Mai, "Neighbor-Cell Assisted Error Correction for MLC NAND Flash Memories," in Proc. of the ACM SIGMETRICS, Jun. 2014.

[8] M. Zheng, J. Tucek, F. Qin, and M. Lillibirdge, "Understanding the Robustness of SSDs under Power Fault," in Proc. of the FAST, Feb. 2013.

[9] S. Park, E. Seo, J. Shin, S. Maeng, and J. Lee, "Exploiting Internal Parallelism of Flash-based SSDs," IEEE Computer Architecture Letters, vol. 9, no. 1, pp. 9-12, Jan. 2010.

[10] Q. Xu, H. Slyamwala, M. Ghosh, T. Suri, M. Awasthi, Z. Guz, A. Shayestech, and V. Balakrishnan, "Performance analysis of NVMe SSDs and their Implication on Real World Database," in Proc. of the SYSTOR, May. 2015.

[11] K. Eshghi and R. Micheloni, "SSD Architecture and PCI Express Interface," Inside Solide State Drives, vol. 37, pp. 19-45, 2013.

[12] D. Cobb and A. Huffman, "NVM Express and the PCI Express SSD Revolution", Interl Developer Forum, http://nvmexpress.org/wp-content/uploads/2013/04/IDF-2012-NVM-Express-and-the-PCI-Express-SSD

[13] H. Strass, "An Introduction to NVMe," http://www.seagate.com/files/www-content/product-content/ssd-fam/nvme-ssd/nytro-xf1440-ssd/_shared/docs/an-introduction-to-nvme-tp690-1-1605us.pdf

[14] M. Bjorling, J. Axboe, D. Nellans, P. Bonnet, "Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems," in Proc. SYSTOR, Jun. 2013.

[15] T. kim, D. Kang, D. Lee, and Y. Eom, "Improving Performance by Bridging the Semantic Gap between Multi-queue SSD and I/O Virtualization Framework," in Proc. MSST, May. 2015.

[16] Michael Rice, "Tuning Linux I/O Scheduler for SSDs", http://dev.nuodb.com/techblog/tuning-linux-io-scheduler-ssds. 2013.

[17] J. K. Park, "Correlated Locality Data Distribution Policy for Improving Performance in SSD," Journal of The Korea Society of Computer and Information, vol. 21, no. 2, pp. 1-7, Feb. 2016.

[18] H. Choi, and Y. Kim, "An Efficient Cache Management Scheme of Flash Translation Layer for Large Size Flash Memory Drives," Journal of The Korea Society of Computer and Information, vol. 20, no. 11, pp. 31-38, Nov. 2015.

Authors

Jung Kyu Park received the M.S. and Ph.D. degrees in computer engineering from Hongik University in 2002 and 2013, respectively. He has been a research professor at the Dankook University since 2014. In 2017, He joined the assistant professor of Department of Digital Media Design and Applications, Seoul Women's University. His research interests include operating system, new memory, embedded system and robotics theory and its application.

Jaeho Kim received the BS degree in information and communications engineering from Inje University, Gimhae, Korea, in 2004, and the MS and PhD degrees in computer science from the University of Seoul, Seoul, Korea, in 2009 and 2015, respectively. He is currently a postdoctoral researcher in the School of Electrical and Computer Engineering at UNIST (Ulsan National Institute of Science and Technology), Ulsan, Korea. His research interests include storage systems, operating systems, and computer architecture.