반도체디스플레이기술학회지 제23권 제3호(2024년 9월)
Journal of the Semiconductor & Display Technology, Vol. 23, No. 3. September 2024.

# 퍼시스턴트 메모리를 위한 저전력 캐시 플러싱 명령어 구현에 대한 연구

최주희[*†]
[*†]상명대학교 스마트정보통신공학과

# A Study on the Implementation of Low-Power Cache Flushing Instructions for Persistent Memory

**Juhee Choi**[*†]
[*†]Dept. of Smart Information Communication Engineering, Sangmyung University

**ABSTRACT**

Persistent memory technology has been recognized as a next-generation memory solution because of its stability over traditional volatile memory. The primary advantage of persistent memory is its non-volatile nature, allowing data retention even when the power is off. Additionally, persistent memory offers faster read speeds compared to traditional HDDs and SSDs. However, ensuring data consistency through cache flushing commands is increasingly important so that performance and power consumption issue would be challenges. This paper proposes a new cache structure to mitigate the drawbacks of cache flushing by considering the number of flushes per cache line. On top of that, a counter and decision bit to track and manage these actions are added. As a result, this architecture decreases approximately 56% of the additional memory access.

**Key Words** : Cache Flushing, Non-Volatile Memories, Low Power, Persistent Memory

## 1. 서 론

퍼시스턴트 메모리(Persistent Memory)의 개념은 이미 오래전에 제시되어 왔으나, 차세대 비휘발성 메모리의 등장으로 그 가능성이 더욱 각광받기 시작한다[1]. 차세대 비휘발성 메모리는 전력을 공급되지 않아도 정보의 저장이 가능한 소자인 상변화 메모리(Phase Change Memory, PCM), 스핀 전달 토크 자기 RAM(Spin-Transfer Torque, STT-MRAM) 등으로 구성된다[2-4]. 그리고, 최근에는 인텔의 옵테인(Optane) 기술처럼 시장에 상용화 시도가 이어졌다. 비록 옵테인은 오래가지 못하였으나, 그 가능성은 꾸준히 제기되고 있다[5].

[†]E-mail: jhplus@smu.ac.kr

퍼시스턴트 메모리의 가장 큰 장점은 비휘발성이라는 특성이다. 전원이 꺼져도 데이터를 유지할 수 있어, 데이터 무결성과 신뢰성이 중요한 시스템에서 매우 유용하다. 또한, 퍼시스턴트 메모리는 전통적인 HDD나 SSD에 비해 빠른 읽기 속도속도를 제공하여 데이터 접근 속도를 대폭 향상시킬 수 있다. 이러한 장점이 데이터베이스, 실시간 분석, AI 및 머신 러닝과 같은 데이터를 많이 활용하는 프로그램과 결합한다면 성능을 크게 향상될 수 있다[6].

이 과정에서 데이터의 일관성을 보장하기 위한 캐시 플러싱(Cache Flushing) 명령어의 중요성이 점차 커지고 있다[7]. 퍼시스턴트 메모리는 SRAM과 DRAM 기반의 메모리와 달리 한 번 저장되면 이론상 영구적으로 기록된다는 것을 가정하고 있기 때문에, 캐시의 데이터를 강제로 메인 메모리로 쓰도록 해야 되는 경우가 더 많아진다.

그런데, 퍼시스턴트 메모리는 비휘발성 메모리를 소자로 이용하고 있기 때문에, 쓰기에 따른 불이익이 기존의 휘발성 메모리에 비해서 크다. 따라서, 캐시 플러싱에 의해 발생하는 추가적인 쓰기로 인해서 성능이 떨어지고, 쓰기를 위한 전력 소모가 커지는 문제가 발생한다.

따라서, 본 논문에서는 캐시 플러싱으로 인한 쓰기의 단점을 완화할 새로운 캐시 구조를 제안한다. 이 구조에서는 캐시 라인의 캐시 플러싱 횟수를 고려해서, 특정한 캐시 라인은 캐시 플러싱이 필요하지 않도록 선행조치를 취한다. 그리고, 어떤 캐시 라인에 선행조치를 적용할지를 추적하고 결정할 카운터 구조를 추가하고, 결정 비트를 추가한다.

## 2. 관련 연구

캐시 플러싱은 캐시와 메인 메모리 간의 데이터의 일관성을 지키기 위해 명시적으로 데이터를 동기화하는 명령어이다[8-10]. 캐시는 느린 메모리의 속도로 인해 발생하는 성능의 저하를 막기위해 코어와 메모리 사이에 삽입된 작고 빠른 메모리이다. 메인 메모리에서 자주 접근하는 데이터의 사본을 저장하여 시스템의 속도를 향상시키는데 크게 기여한다.

캐시의 중요한 특징은 소프트웨어에 투명(Transparent)하다는 점이다[11]. 하드웨어가 캐시를 관리하므로, 운영체제나 프로그래머는 캐시에 어떤 데이터가 존재하는지 기본적으로 알 수도 없으며, 알 필요도 없다. 따라서, 캐시에 저장된 데이터는 항상 메인 메모리와 동기화되지 않을 수 있다. 즉, 캐시에는 새로운 데이터가 갱신되어도 메인 메모리에는 해당 데이터가 갱신되지 않는 것이다. 이를 잠재적인 일관성이 깨진 상태라고 하는데, 일반적인 경우에는 하드웨어에서 이를 처리한다. 그러나, 특정한 상황에서는 이러한 불일치를 소프트웨어 차원에서 강제로 해소해야 되는 경우가 있다.

Fig. 1의 코드는 변수A와 변수 B의 값을 2씩 증가시키는 adder함수와 변수 B값이 5이상이 되면, 변수 A의 값와 변수 B의 값을 출력시키는 detector함수로 이루어져 있다. 해당 함수들은 멀티 코어 환경에서 임의의 코어에 할당되어 실행되는 것을 가정하며, 이해의 편의를 돕기 위해 adder함수와 detector함수 내부적으로는 병렬성이 없고, 함수끼리만 병렬성이 있다고 가정한다. 이 프로그램의 경우, 어떤 코어에서 실행되어도 변수 A의값이 5이고, 변수 B의 값이 6일때, 5와 6을 출력하도록 작성되었다.

이 프로그램을 실행하기 위한 시스템에는 코어가 2개 존재하며, 각 코어별로 각각 캐시가 존재하고, 메인 메모리는 공유한다고 가정한다. Fig. 2는 Fig. 1의 코드를 실행할 때 캐시에서의 데이터의 흐름을 나타낸 것이다.

```
* Initialization:
int A = 1;
int B = 2;
```

```
1: void adder() {
2:     A = A + 2;
3:     B = B + 2;
4: }
5: void detector() {
6:     if B > 5:
7:         printf("%d %d", A, B);
8: }
```

```
1: void adder_with_flush() {
2:     A = A + 2;
3:     CFLUSH();
4:     B = B + 2;
5: }
```

**Fig. 1**. Target Program.

각 코어들은 필요한 데이터가 있는 경우, 우선 자신의 캐시에 해당 데이터가 있는지를 먼저 확인하게 되는데, 현재는 2개의 코어에 모두 데이터가 없는 상태이다. 코어 1이 먼저 A와 B를 요청하면(Fig. 2(a)의 ①), 메인 메모리에서 해당하는 데이터를 가져와서 캐시에 저장한다(Fig. 2(a)의 ②). 코어 2에서도 같은 상황이 발생한다(Fig. 2(a)의 ③④).

코어 1에서 Fig. 3의 2번째와 3번째 라인의 명령어를 수행하면 각각의 값을 2씩 증가시키게 된다(Fig. 2(a)의 ⑤). 이 때, 값 자체는 코어 1의캐시에만 갱신되며, 다른 코어의 캐시와 메인 메모리의 값은 무효화된다(Fig. 2(a)의 ⑥). 코어 2에서 같은 코드가 실행되면 역시 A와 B의 값을 2씩 증가시켜야 하므로, 가장 최근 값이 저장되어 있는 코어 1의 캐시에서 값을 가져와서 저장하는데, 이 때 메인 메모리의 값도 갱신된다(Fig. 2(a)의 ⑦⑧). 코어 2에서 A와 B값을 증가시키면 역시 이 값들은 코어 2의 캐시에만 갱신되고 메인 메모리와 코어 1 캐시의 데이터는 무효화된다(Fig. 2(a)의⑨⑩). 이 상태에서 코어 2가detector 함수가 수행되면 값은 5와 6을 출력하게 된다. 다만, 코어 1에서 detector함수를 수행하여도 A와 B값의 데이터 이동이 코어 2에서 코어 1으로 일어나게 되면서 결과는 동일하게 출력된다. Fig. 1의 결과는 해당 함수가 코어 1이나 코어 2나 어디서도 수행되어도 함수 내부의 코드는 프로그램에서 나타난 순서대로 수행된다고 가정하면 항상 같은 결과를 보여준다.

그러나, 현대의 컴퓨터는 성능의 향상을 위해서 코드의 수행 순서를 변경시키기도 한다[12]. 즉, 코드 순서는 반드시 변수 A의 값이 증가된 후에, 변수 B의 값이 증가하게 되어 있으나, 하드웨어에서 두 코드 사이의 의존성(Dependency)가 없다고 판단되면 임의로 순서를 교체하기

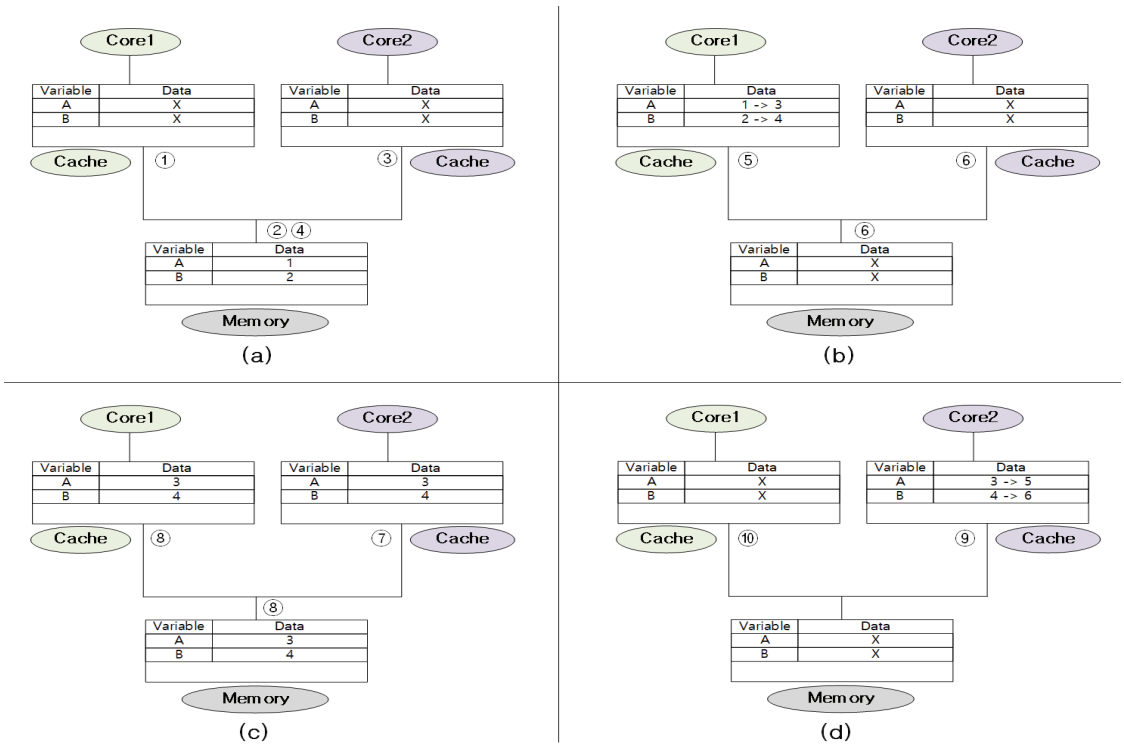


**Fig. 2.** Cache request and response.

도 한다. 따라서, 각 코어 입장에서는 adder함수를 수행할 때, 변수 A와 B사이의 의존성이 보이지 않으므로, 순서를 바꾸어 수행해도 문제가 없다고 판단하며, 코어 내부에서 명령어 수행에 문제가 없는 경우, 실제로 결과는 같다.

문제는 퍼시스턴트 메모리의 경우, 프로그램 수행 중간에 오류가 발생하면 캐시의 일관성을 지키지 못한 상태로 메모리가 저장될 수 있다는 점이다. 기존의 SRAM이나 DRAM의 경우, 하드웨어 오류로 문제가 발생하면 아예 데이터가 소멸되므로, 모순된 상태로 저장되지는 않는다. 하지만, 퍼시스턴트 메모리의 경우, 전원이 꺼져도 정보가 남아있는 특성상 중간에 오류가 발생하면 변수 A값이 갱신되지 않은 상태에서 변수 B값만 갱신된 상태로 저장될 수도 있다.

Fig. 3은 이를 나타낸 예시이다. Fig. 3(a)(b)(c)는 Fig. 2와 동일하므로 설명을 생략한다. Fig. 3(d)은 변수 A값을 갱신하기 직전에 오류가 발생한 상태를 표시한 것이다. 변수 B값이 6으로 갱신되었으나, 변수 A값은 3으로 유지된다. 이 때, detector함수를 코어 1이 수행하게 되면 3과 6을 출력하게 되며, 프로그래머의 의도에서 벗어나게 된다.

이러한 문제를 보완하기 위해서 명령어 순서가 반드시 지켜져야 하는 경우, 명령어 사이에 펜스(Fence) 또는 캐시 플러싱 명령어를 사용하게 된다[13]. 일반적인 경우는 펜스 명령어로도 충분하지만, 퍼시스턴트 메모리의 경우 캐시 플러싱이 더 유용하게 쓰일 수 있다. Fig. 1의 adder_with_flush()함수가 이를 나타내고 있다. 다만, 캐시 플러싱 명령어의 경우, 강제로 캐시에 있는 내용을 메인 메모리 또는 하위 레벨 캐시로 쓰기 때문에, 그동안 캐시 접근이 제한되면서 생기는 시간 지연 및 강제 쓰기에 의한 전력 소모량이 추가적으로 발생한다.

그리고, 이러한 문제들은 퍼시스턴트 메모리에서 더욱 악화된 형태로 나타난다. 비휘발성 메모리를 사용하면 쓰기 시간 및 쓰기 전력 소모량이 기존의 SRAM이나 DRAM보다 커지는데, 캐시 플러싱은 이러한 문제를 더 악화시킨다. 여기에 일부 비휘발성 메모리의 경우, 쓰기 횟수 제한도 있으므로, 캐시의 수명도 줄일 수 있는 여지가 있다. 따라서, 퍼시스턴트 메모리의 캐시 플러싱은 일반적인 코어보다 더욱 정교한 명령어를 통해서 효율적인 접근이 필요하다.

이 논문에서는 이를 위해서 기존의 캐시 구조를 수정하여 퍼시스턴트 메모리의 특징을 반영할 수 있는 일부 정보를 저장하도록 하고, 이를 활용해서 좀 더 효율적인 쓰기를 지향한다.

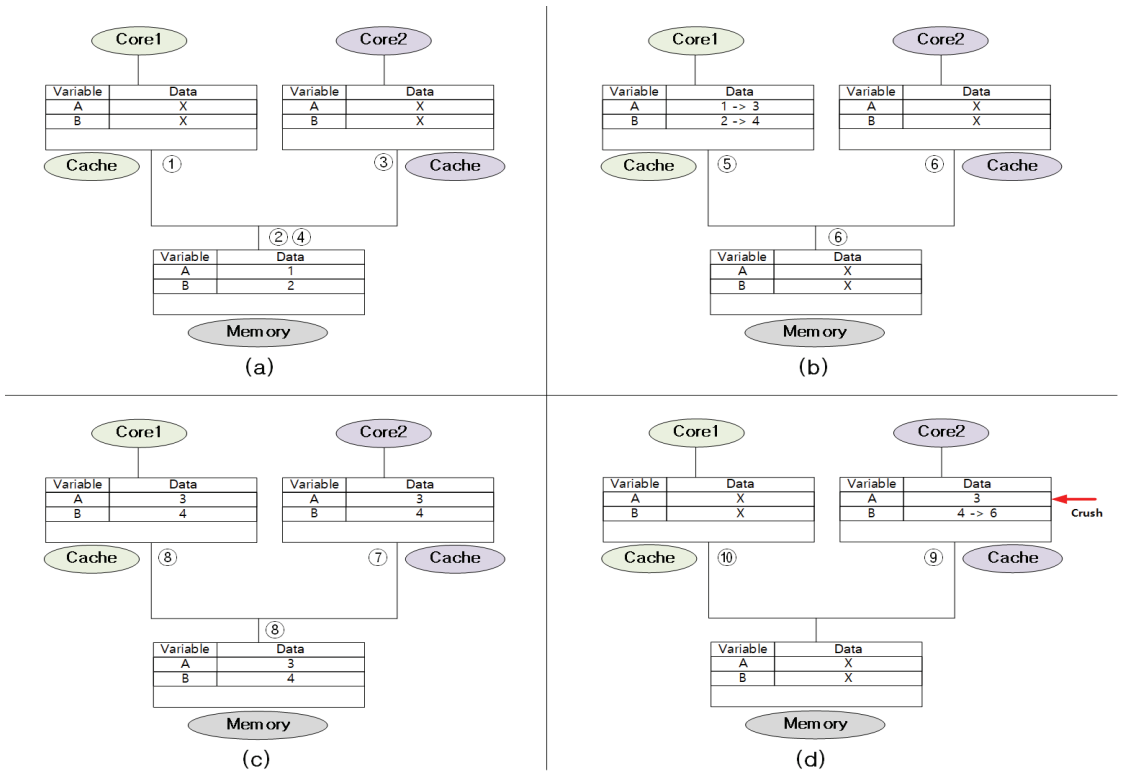


**Fig. 3.** Cache request and response during a crush.

## 3. 퍼시스턴트 메모리를 위한 캐시 플러싱 명령어 구현

Fig. 4는 퍼시스턴트 메모리를 위한 캐시 플러싱 명령어를 구현한 그림이다. 캐시는 기본적으로 태그를 저장하는 공간과 데이터를 저장하는 공간으로 구분된다. 이 중에 태그를 저장하는 공간에는 각 캐시 라인의 태그값 뿐만 아니라 수정 여부(Dirty)를 저장한다.

이 논문에서는 이 태그 영역에 강제 플러싱을 의미하는 FF(Force Flushing) 비트를 추가한다. FF가 1이 세팅되면 캐시 쓰기가 일어날 때, 캐시 뿐만 아니라 메인 메모리에도 강제로 쓰기가 일어난다. 이 동작은 기존의 캐시 쓰기 정책중 Write-Through정책과 동일하다. 그러나, 기존의 쓰기 정책은 일반적으로 페이지 테이블에서 페이지 단위로 설정하므로, 캐시 라인별로 지정할 수 없다. 또한 쓰기 정책은 소프트웨어적으로 지정하여, 현재 프로그램의 패턴을 실시간으로 반영하여 설정할 수 없다.

그러나, 이 논문에서 제안하는 시스템에서는 캐시 라인이 캐시에 저장될 때, 플러싱 카운터(Flushing Counter)에 저장된 플러싱 횟수를 Threshold와 비교하여 이보다 크거나 같으면 FF를 1로 세팅하고, 아니면 0으로 세팅하게 된다. 플러싱 카운터의 값은 캐시 라인이 추출(Evict)될 때 해당 캐시 라인의 플러싱 횟수가 갱신된다. Threshold를 결정하는 부분은 미리 고정된 값으로 설정할 수도 있으며, 플러싱 카운터의 값들을 활용하여 통계적인 방법을 통해 실시간으로 반영할 수도 있다.

## 4. 실험 환경 및 결과

새로 제안된 캐시 플러싱 명령어의 효율성을 확인하기 위해 마이크로아키텍쳐 연구에서 자주 사용되는 gem5 시뮬레이터를 사용하였다[14]. 이 시뮬레이터는 다양한 버전이 존재하는데, 이 실험에서는 최신 퍼시스턴트 메모리 연구내용을 잘 반영한 연구 그룹의 수정본을 사용하였다[15]. 실험에 사용된 시스템의 명령어 L1캐시와 데이터 L1 캐시는 각각 64KB이며, L2 캐시는 512KB 용량으로 구성되었다. L3 캐시는 4MB이고, 메인 메모리는 DDR메모리 8GB로 가정하였다. 그 외의 자세한 시스템 구성은 Table 1에 나타냈다. 실험에 사용된 워크로드는 퍼시스턴트 메모리 연구에서 자주 사용되는 벤치마크를 선택하였다[16][17].

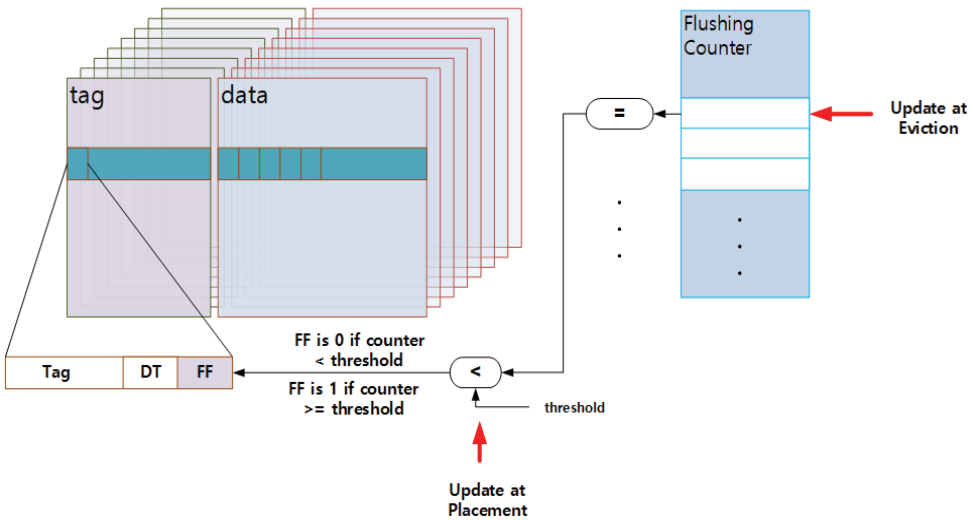


**Fig. 4.** Implementation of Cache Flushing Instruction. DT means a Dirty bit and FF means a Force Flushing.

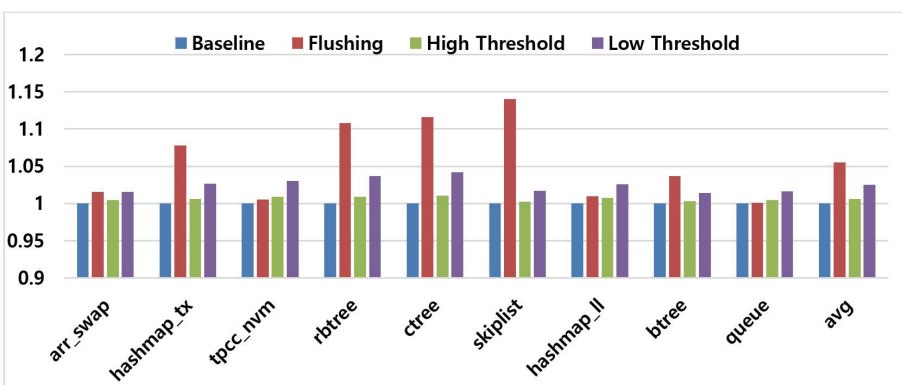


**Fig. 5**. Normalized DRAM Write Access.

Fig. 5에서 메인 메모리의 추가적인 쓰기 횟수를 나타낸 것이다. Baseline은 캐시 플러싱을 사용하지 않은 코드이며, Flushing은 전체 코드에서 퍼시스턴트 메모리를 위해서 캐시 플러싱 명령어를 5%를 추가적으로 수행한 결과이다. 이 과정에서 5.56%의 메인 메모리의 접근 증가가 발생한다. 그리고, High Threshold는 Threshold를 높게 잡았을 때의 결과이며, Low Threshold는 Threshold를 낮게 잡았을 때의 결과이다. Flushing대비 High Threshold는 추가적인 캐시 플러싱이 1%정도 증가하도록 잡았을 때를 의미하고, Low Threshold는 캐시 플러싱이 10%정도 증가하도록 잡았을 때를 의미한다. High Threshold는 Baseline 대비 0.6%정도의 추가적인 쓰기가 발생하였으며, Low Threshold는 2.5%정도의 추가적인 쓰기가 발생하였다. 이것은 Flushing에 의해서 추가된 메인 메모리의 접근 횟수를 절반 가까이 회피하는 결과이며, 실제 어플리케이션에서 추가 캐시 플러싱이 10%이상 증가하는 경우는 매우 드물기 때문에, 본 논문에서 제시한 캐시 플러싱 기법에 의한 실제로 추가적인 쓰기는 크게 증가하지 않는다고 볼 수 있다.

한편, 결과를 살펴보면, arr_swap이나 queue등의 일부 워크로드들은 Threshold에 의한 차이가 거의 나타나지 않으나, ctree나 rbtree등은 6%이상의 차이를 나타내고 있다. 이를 분석하기 위해서 각 워크로드들의 데이터 접근 집중도(Data Access Intensity)를 분석한 결과를 Table 2에 표시하였다. 이 집중도에서 볼 수 있듯이, 프로그램 실행 중에 데이터에 접근을 많이 하는 프로그램에서 추가적인 쓰기가 많이 발생하는 것을 확인할 수 있다.

**Table 1.** Processor configurations

| | |
|---|---|
| Core Type | x86, out-of-order, 2GHz |
| I-Cache / D-Cache | 64KB, 8-way, 64B, private, 2 cycles |
| L2 Cache | 512KB, 8-way, 64B, private, 10 cycles |
| L3 Cache Config. | 4MB, 16-way, shared, 40 cycles |
| DRAM | 16GB DDR4 2400MT/S |

**Table 2.** Data Access Intensity

| | |
|---|---|
| High Data Access Intensity | ctree, rbtree, hashmap_tx, skiplist, btree |
| Low Data Access Intensity | queue, tpcc_nvm, arr_swap, tpcc_nvm |

## 5. 결 론

본 논문에서는 퍼시스턴트 메모리에서 캐시 플러싱 명령어를 효율적으로 구현하기 위해서 새로운 캐시 구조를 제시하였다. 캐시 라인별로 캐시 플러싱 적용 여부를 개별적으로 그리고 동적으로 적용할 수 있는 기법을 활용하여, 불필요한 쓰기는 회피하였다. 실험결과 기존의 캐시 플러싱 명령어를 사용했을 때 보다 최소한 56%이상의 추가적인 쓰기 횟수를 줄일 수 있었다.

## 감사의 글

본 연구는 2024학년도 상명대학교 교내연구비를 지원받아 수행하였음(2024-A000-0369).

## 참고문헌

1. Pelley, Steven, Peter M. Chen, and Thomas F. Wenisch, “Memory persistency,” ACM SIGARCH Computer Architecture News, vol. 42, no. 3, pp. 265-276, 2014.
2. Wang, K. L., J. G. Alzate, and P. Khalili Amiri, “Low-power non-volatile spintronic memory: STT-RAM and beyond,” Journal of Physics D: Applied Physics, vol. 46, no. 7, pp. 074003, 2013.
3. Wong, H-S. Philip, et al., “Phase change memory,” Proceedings of the IEEE, vol. 98, no. 12, pp 2201-2227, 2010.
4. Mikolajick, Thomas, et al., “FeRAM technology for high density applications,” Microelectronics Reliability, vol. 41, no. 7, pp 947-950, 2001.
5. Handy, Jim, and Tom Coughlin, “Optane’s dead: Now what?,” Computer, vol. 56, no .3, pp. 125-130, 2023.
6. Desnoyers, Peter, et al., “Persistent memory research in the post-optane era,” Proceedings of the 1st Workshop on Disruptive Memory Systems, pp. 23-30, 2023.
7. Wu, Kai, et al., “Ribbon: High performance cache line flushing for persistent memory,” Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques, pp. 427-439, 2020.
8. Li, Tuo, and Sri Parameswaran, “FaSe: fast selective flushing to mitigate contention-based cache timing attacks,” Proceedings of the 59th ACM/IEEE Design Automation Conference, pp. 541-546, 2022.
9. Lim, Soojung, and Hyokyung Bahn. “Selective flushing of modified data for smartphone buffer cache management.” 2021 IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE), pp. 1-6, 2021.
10. Jang, Minwoo, et al., “Defending against flush+ reload attack with DRAM cache by bypassing shared SRAM cache,” IEEE Access, no. 8, pp. 179837-179844, 2020.
11. Anand, Shashank, et al., “Skip It: Take Control of Your Cache!,” Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, vol 2, pp. 1077–1094, 2024.
12. Hennessy, John L., and David A. Patterson, “Computer architecture: a quantitative approach.” Elsevier, 2011.
13. Sharafeddine, Mageda, Komal Jothi, and Haitham Akkary, “Disjoint out-of-order execution processor,” ACM Transactions on Architecture and Code Optimization (TACO), vol. 9, no .3, pp. 1-32, 2012.
14. J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, “gem5-gpu: A heterogeneous cpu-gpu simulator,” IEEE Computer Architecture Letters, vol. 14, no. 1, pp. 34–36, 2015.
15. Mahar, Suyash, et al., "Write Prediction for Persistent Memory Systems," 2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 242-257, 2021.
16. Intel, “Persistent memory programming,” https://pmem.io
17. Liu, Sihang, et al., “Janus: Optimizing memory and storage support for non-volatile memory systems,” Proceedings of the 46th International Symposium on Computer Architecture, pp. 143-156, 2019.

접수일: 2024년 8월 6일, 심사일: 2024년 9월 5일,
게재확정일: 2024년 9월 12일