운영체제 HW1

xxxxxxxx xxxxxxxx xxx

2022. 4. 14

1 실행 환경

이번 실험은 라즈베리 파이에서 진행했습니다. 자세한 내용은 다음과 같습니다.

Model	Raspberry Pi 3 Model B+
CPU	Broadcom BCM2837B0, Cortex-A53 64-bit SoC @ 1.4GHz
Memory	1GB LPDDR2 SDRAM
OS	Raspbian GNU/Linux 11 (bullseye) armv7l
Kernel	5.10.63-v7+

GNU Make 4.3
gcc version 10.2.1 20210110 (Raspbian 10.2.1-6+rpi1)

추가로 “Arm Cortex-A53 MPCore Processor Technical Reference Manual”, “Raspberry Pi, Wikipedia” 를 참고하여 CPU 캐시와 TLB 구조에 대해 조사해보았습니다. 이번 과제에서 관심있는 부분은 Data 영역이므로 Instruction Cache, Instruction TLB 는 생략하였습니다.

1.1 캐시

캐시는 2레벨 구조입니다.

L1 Data Cache	32KB, 4-way set associative
L2 Cache	512KB, 16-way set associative

1.2 TLB

TLB 또한 2레벨 구조입니다. 주소가 연속적일 때는 16KB, 64KB, 2MB, 1GB 등 다양한 크기의 페이지 정보도 저장할 수 있다고 합니다. 하지만 이번 실험에서는 리스트를 랜덤하게 방문하므로 연속적인 주소의 변환이 거의 발생하지 않을 것이므로, 4KB 페이지만 저장한다고 가정할 수 있습니다.

data micro TLB	10 entry, fully-associative
unified main TLB	512 entries, 4-way associative

2 여러 경우로 실행

다양한 map size, stride 조합을 실행하기 위해 Makefile 스크립트를 다음과 같이 수정했습니다.
2중 for 문 구조로, stride 4~4k, map size 4K~256M 의 경우를 실행합니다. 실행결과는 stride 별로 구분해서 csv 파일로 저장합니다.

```
4  run: test-tlb
5    for stride in 4 16 32 64 256 1k 4k ; do \
6      echo -n "stride $$stride\n"; \
7      echo 'size,ns,cycle' > outs/$$stride.csv; \
8      for i in 4k 8k 16k 32k 64k 128k 256k 512k 1M 2M 4M 6M 8M 16M 32M 64M 128M 256M ; do \
9        ./test-tlb -r $$i $$stride | tee -a outs/$$stride.csv; \
10     done ; \
11   done
```

또, CSV 파일에 적합하도록 test-tlb.c 소스코드 마지막의 부분을 다음과 같이 수정햇습니다.
콤마 외의 다른 불필요한 요소들을 제거하고 “map size, access time, cycle” 형식으로 출력했습니다.

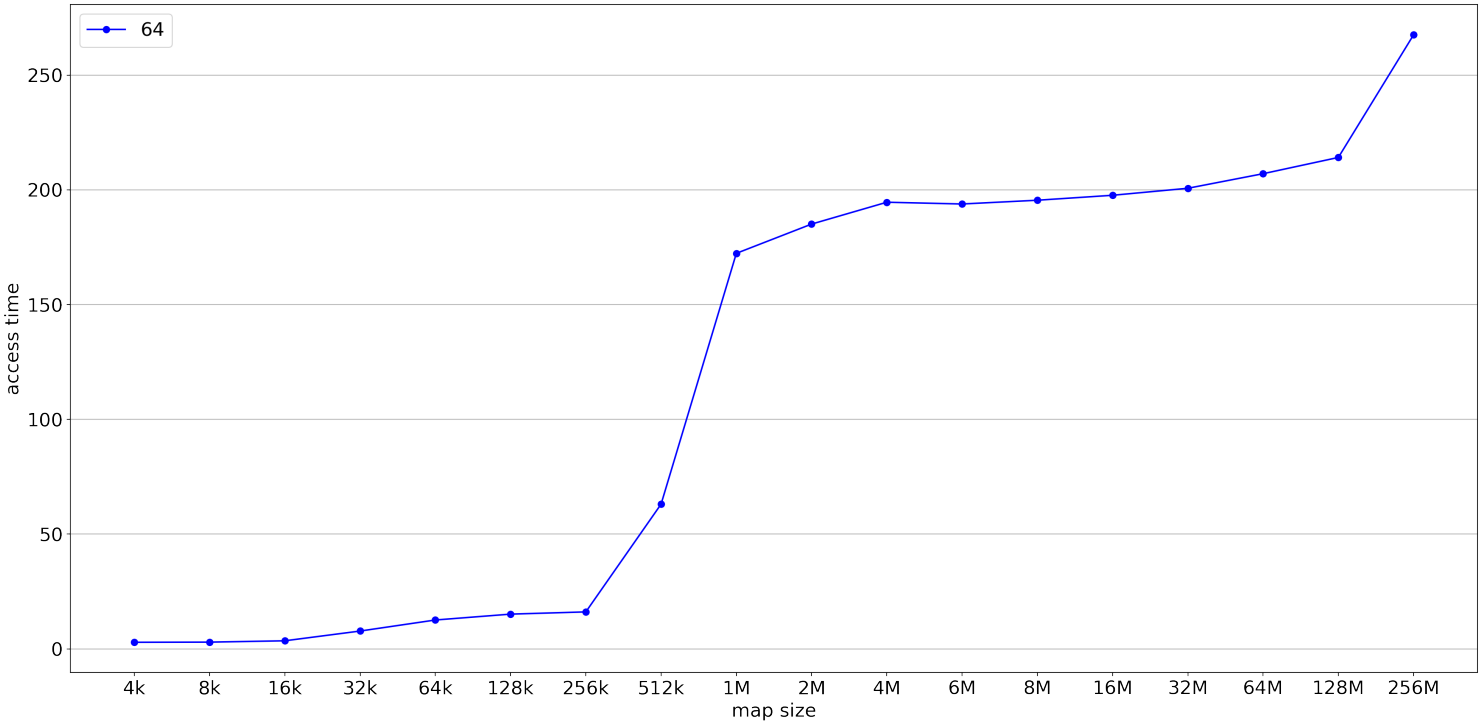
```
283 printf("%s,%.2f,%.1f\n",
284   argv[1], cycles, cycles*FREQ);
```

3 그래프

시각화는 Python의 matplotlib 라이브러리를 사용했습니다.

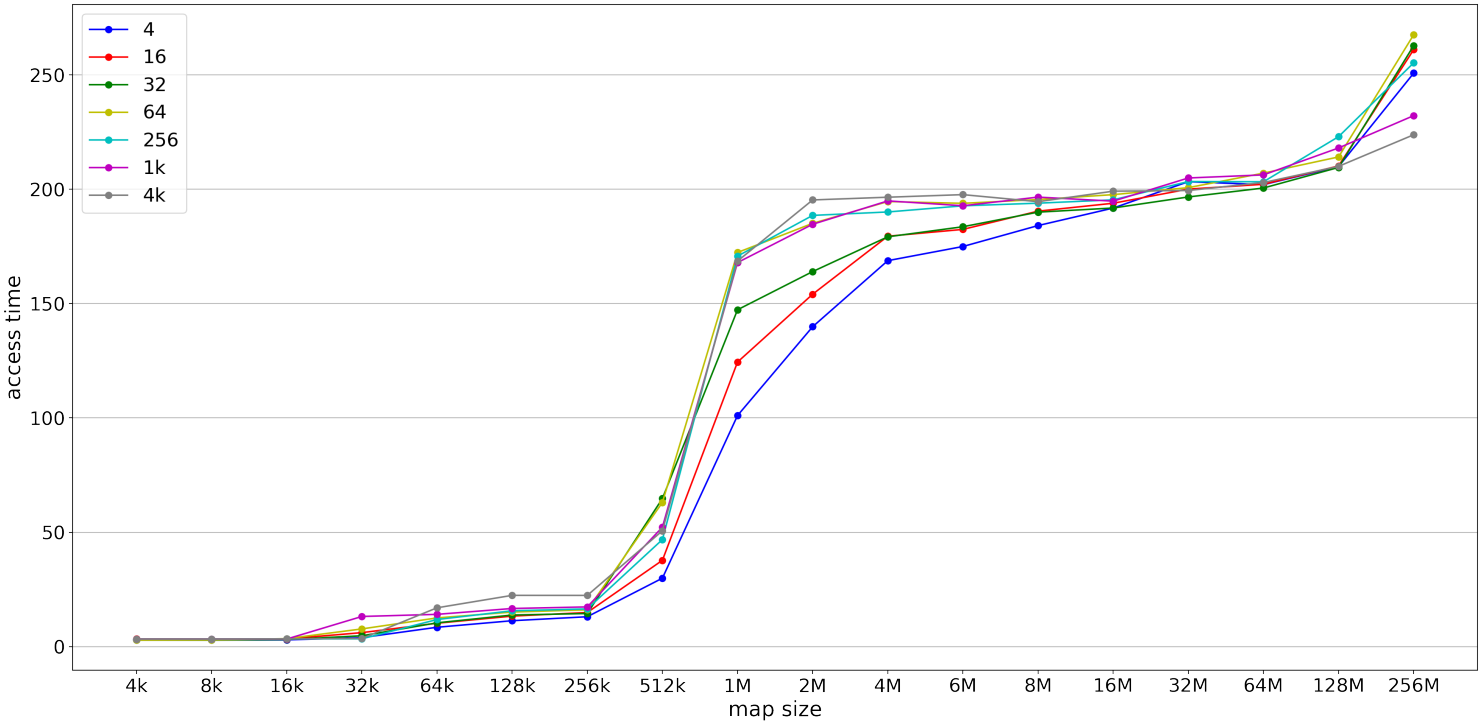
3.1 map size 를 여러가지로 변경

다음은 stride 64 경우의 map size의 변화에 따른 access time 그래프입니다.



3.2 stride를 여러가지로 변경 (메모리 캐싱 효과가 나타나게도 변경해 본다)

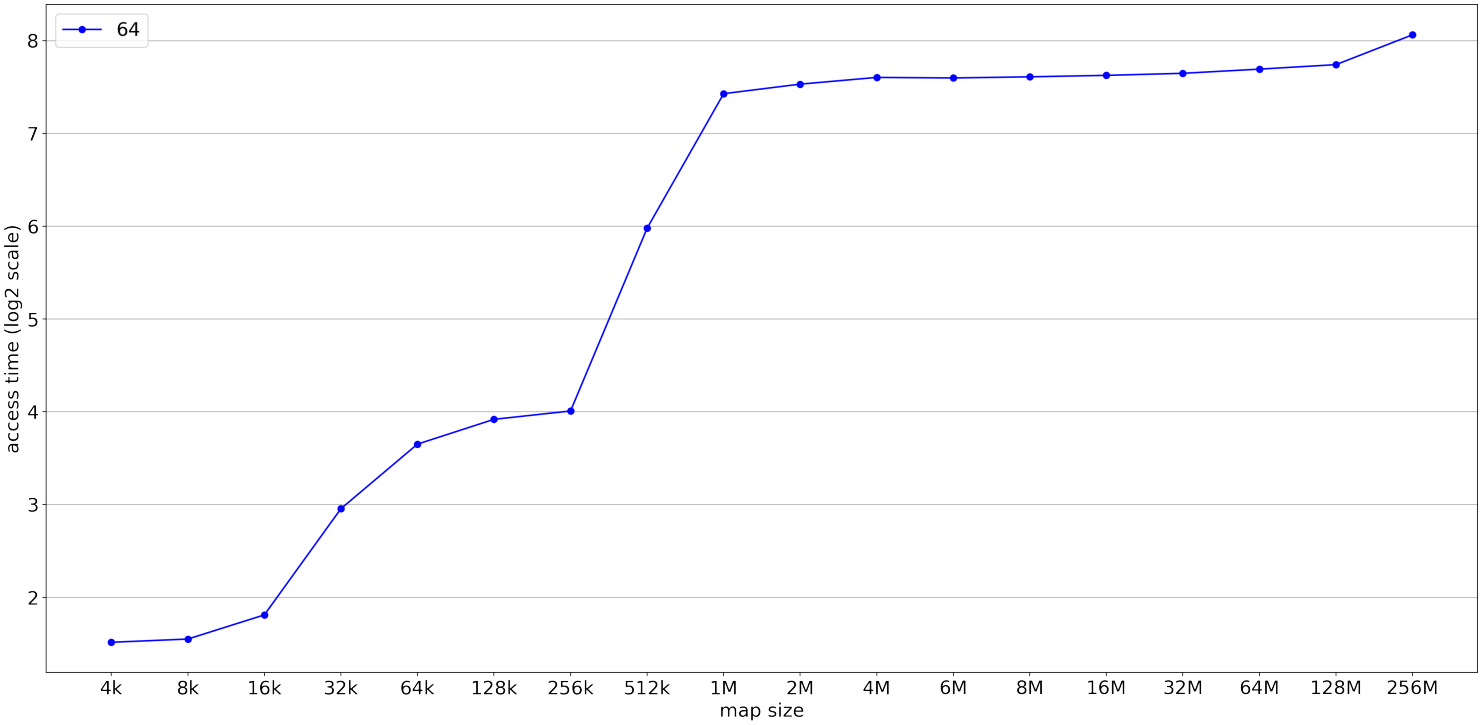
다음은 여러 stride를 동시에 나타낸 그래프입니다.



4 분석 내용을 설명

4.1 실행 환경의 TLB 구조

우선 access time이 작을때의 변화를 좀 더 뚜렷히 보기 위해 access time에 log_2 함수를 적용하여 그래프를 다시 그려보았습니다.



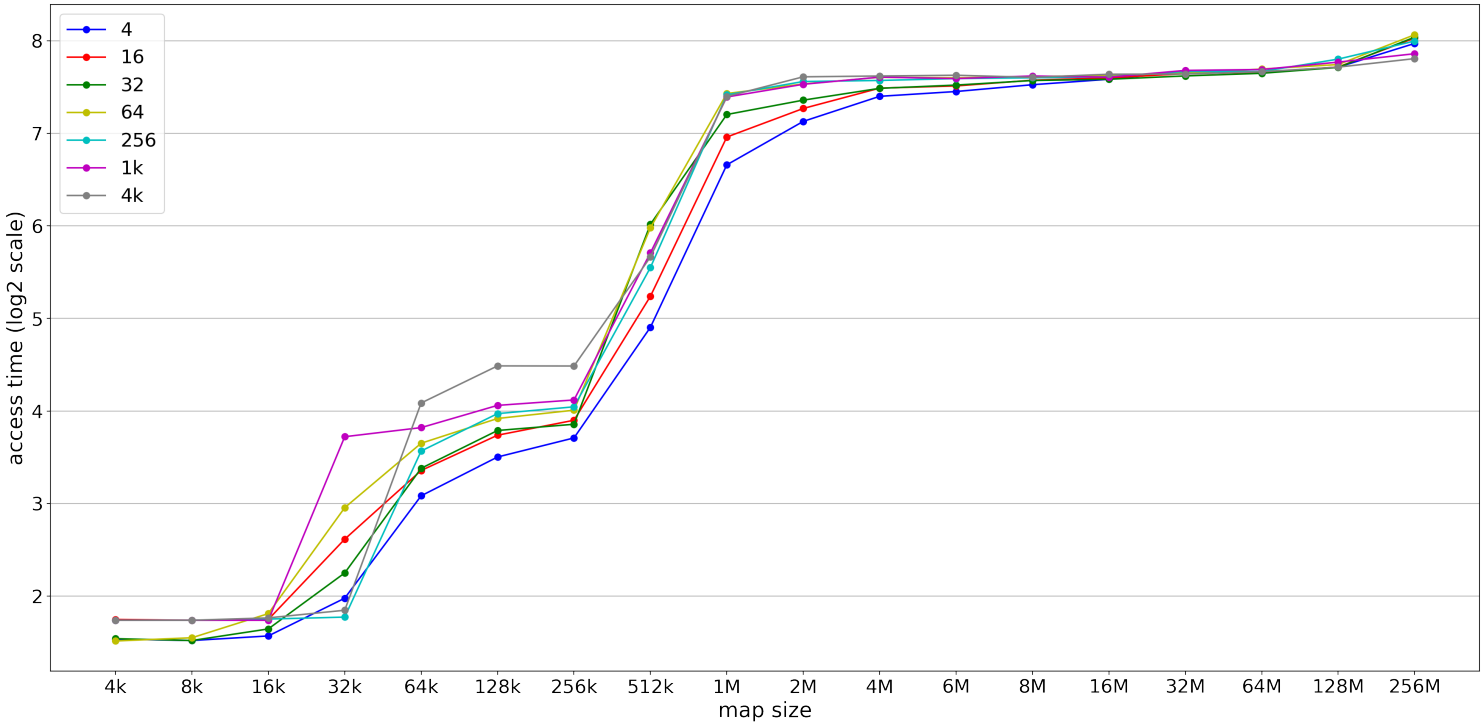
4KB 페이지만 사용한다고 가정했을 때, data micro TLB의 용량을 초과하는 4KB × 10 entry = 40KB 근처에서, Unified Main TLB의 용량을 초과하는 4KB × 512 entry = 2MB 근처에서 총 2번의 access time 상승이 발생할 것이라고 예상할 수 있습니다. 실제로 그래프를 보면 40KB, 2MB 근처에서 상승하는 걸 관찰할 수 있습니다. 2 레벨 계층 구조의 특징을 잘 보여주는 그래프입니다.

하지만 정확히 40KB, 2MB 에서 상승이 발생하지 않았습니다. 이에 대해서 제 개인적인 의견을 말해보겠습니다. 우선 40KB가 아닌 16KB → 32KB 구간부터 상승이 발생했습니다. 이는 OS, 백그라운드 프로세스들의 데이터가 차지하는 TLB 공간 때문에 40KB보다 이르게 상승이 발생했다고 해석할 수 있을 것 같습니다. 두번째 상승은 2MB가 아닌 256KB → 512KB 구간부터 발생합니다. 이는 Cache Miss 때문이라고 해석할 수 있을 것 같습니다. L2 Cache의 용량은 512KB 인데, 이를 초과하면서 Cache miss로 인해 발생하는 것 같습니다. 사실 두 번째 상승은 TLB Miss로 인한 상승도 뒤에 약간 있긴 하지만 Cache Miss에 의한 영향이 더 크다고 볼 수 있을 것 같습니다.

정리해보면, 첫번째 상승은 TLB Miss, Cache Miss가 복합적으로 반영된 것이고, 두번째 상승은 Cache Miss의 영향이 더 크게 반영되었으며, TLB Miss의 영향도 약간 반영되었다고 해석할 수 있습니다. 이번 실험 환경에서 access time의 증가는 TLB Miss로 인해 발생하기도 하지만, Cache Miss로 인해서 더 크게 발생한다는 결론을 내릴 수 있습니다.

4.2 실행 환경의 캐시라인의 크기

여러 stride 를 나타낸 그래프도 역시 log_2 스케일로 그려보았습니다.



우선 map size가 64k ~ 256k 일 때, stride 4k 의 access time이 크게 차이나는 것을 볼 수 있습니다. 이는 TLB Hit/Miss 로 인한 차이라고 해석할 수 있습니다. stride 4K일 때는 페이지 크기도 4KB이기 때문에 TLB에 저장된 페이지를 다시 방문하지 않기 때문에 항상 TLB Miss가 발생하고, 나머지 stride들은 TLB Hit이 발생하기 때문에 stride가 작을수록 access time이 빨라지는 것을 볼 수 있습니다.

stride가 4, 16, 32 일 때는 전체적으로 access time이 빠릅니다. Cache Hit/Miss로 인한 차이로 볼 수 있습니다. 어떤 노드를 방문했을 때 stride가 캐시라인보다 작다면 한 캐시라인에 여러 노드가 존재할 것입니다. 같은 캐시라인에 있는 노드들이 캐시에 저장되고, 이 캐시라인에 속한 노드들이 캐시에서 사라지기 전에 노드에 방문하는 경우, Cache Hit이 발생하고 access time이 감소합니다. 리스트를 순차적으로 방문한다면 $\frac{\text{cacheline}}{\text{stride}}$ 번의 access 중 $\frac{\text{cacheline}}{\text{stride}} - 1$ 번의 Cache Hit이 보장되겠지만, 이번 실험은 리스트를 랜덤으로 방문하므로 캐싱된 노드를 방문하기 전에 노드가 캐시에서 사라질 가능성이 있습니다. 실제로 map size가 8MB를 넘어가면서 캐싱된 것을 다시 참조할 확률이 점점 낮아지기 때문에 access time의 차이가 없어지는 것을 볼 수 있습니다.

map size 1MB ~ 2MB 구간에서는 Cache Hit/Miss 로 인한 차이를 뚜렷하게 확인할 수 있습니다. 우선 stride 64 ~ 4k 의 access time 차이가 거의 없습니다. 이는 스케일이 커짐에 따라 TLB Hit/Miss 차이가 무시할 수 있을 정도로 작아졌다고 해석할 수 있습니다. (log_2 스케일이 아닌 3.2의 그래프를 보면 TLB Hit/Miss로 인한 차이가 약간 있긴 합니다.) 또, stride 4, 16, 32의 경우 access time이 다른 stride들에 비해 꽤 큰 차이가 있습니다. 이는 Cache Hit으로 인해 발생한 차이라고 볼 수 있습니다.

즉, 이 구간에서는 TLB Hit/Miss로 인한 차이는 거의 나타나지 않으면서, Cache Hit/Miss 로 인한 차이는 뚜렷하게 관찰할 수 있습니다. stride 32까지는 Cache Hit으로 인한 access time의 차이가 발생하는데, stride 64 부터는 큰 차이가 없기 때문에 Cache Hit이 발생하지 않는다고 볼 수 있습니다. 즉, 캐시라인 크기가 64바이트라는 것을 확인할 수 있습니다.

A 시각화 코드

보고서에 사용한 그래프 시각화 코드입니다.

```python
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
plt.rcParams['figure.figsize'] = [20, 10]
plt.rcParams['figure.dpi'] = 200
plt.rcParams.update({'font.size': 18})

strides = (4, 16, 32, 64, 256, '1k', '4k')
colors = ('b', 'r', 'g', 'y', 'c', 'm', 'gray')

def draw(name, strides, log):
    plt.figure()
    for stride, color in zip(strides, colors):
        data = pd.read_csv(f'outs/{stride}.csv')
        data.drop('cycle', axis=1, inplace=True)
        sizes = data['size'].values
        nss = data['ns'].values
        if log:
            nss = np.log2(nss)
        plt.plot(sizes, nss, 'o-', color=color, label=stride)
    plt.grid(True, axis='y')
    plt.legend()
    plt.xlabel('map size')
    plt.ylabel('access time')
    plt.tight_layout()
    plt.savefig(f'outs/{name}.png')

draw('figure1', (64,), False)
draw('figure2', strides, False)
draw('figure3', (64,), True)
draw('figure4', strides, True)
```