22

페이징: 더 빠른 변환 (TLB)

페이징은 상당한 성능 저하를 가져올 수 있다. 페이징은 프로세스 주소 공간을 작은 고정된 크기(즉, 페이지)로 나누고 각 페이지의 실제 위치(매핑 정보, mapping information)를 메모리에 저장한다. 매핑 정보를 저장하는 자료 구조를 페이지 테이블이라 한다. 매핑 정보 저장을 위해 큰 메모리 공간이 요구된다. 무엇보다 중요한 것은 가상 주소에서 물리 주소로의 주소 변환을 위해 메모리에 존재하는 매핑 정보를 읽어야 한다는 사실이다. 페이지 테이블 접근을 위한 메모리 읽기 작업은 엄청난 성능 저하를 유발한다. 모든 로드/스토어 (load/store) 명령어 실행이 추가적인 메모리 읽기를 수반하는 상황을 가정해 보라. 정말 머리 아픈 일이 아닐 수 없다. 우리의 질문은 다음과 같다.

핵심 질문: 주소 변환 속도를 어떻게 향상할까
주소 변환을 어떻게 빨리할 수 있을까? 페이징에서 발생하는 추가 메모리 참조를 어떻게 피할 수 있을까? 어떤 하드웨어가 추가로 필요하지? 운영체제가 어떤 식으로 개입해야 할까?

운영체제의 실행 속도를 개선하려면, 도움이 필요하다. 대부분의 경우 하드웨어로부터 도움을 받는다. 하드웨어는 운영체제의 오랜 친구! 주소 변환을 빠르게 하기 위해서 우리는 **변환-색인 버퍼**(**translation-lookaside buffer**) 또는 **TLB**라고 하는 부르는 것(역사적인 이유는 [CP78]을 참고)을 도입한다[CG68; Cou95]. TLB는 칩의 **메모리 관리부**(**memory-management unit**, **MMU**)의 일부다. 자주 참조되는 가상 주소-실주소 변환 정보를 저장하는 하드웨어 캐시이다. **주소-변환 캐시**(**address-translation cache**)가 좀 더 적합한 명칭이다. 가상 메모리 참조 시, 하드웨어는 먼저 TLB에 원하는 변환 정보가 있는지를 확인한다. 만약 있다면 페이지 테이블(모든 변환 정보를 갖고 있음)을 통하지 *않고* 변환을 (빠르게) 수행한다. 실질적으로 TLB는 페이징 성능을 엄청나게 향상시킨다. TLB를 도입함으로써, 페이징이 "사용 가능"한 가상 메모리 기법이 된다[Cou95].

22.1 TLB의 기본 알고리즘

그림 22.1은 가상 주소 변환이 이루어지는 과정을 대략적으로 나타내고 있다. 그림 22.1에서는 주소 변환부가 단순한 **선형 페이지 테이블**(**linear page table**, 즉, 배열로 이루어진 페이지 테이블)과 **하드웨어로 관리되는 TLB**(하드웨어가 페이지 테이블 접근에 대한 대부분의 책임을 관리한다. 아래에 더 자세히 설명한다)로 구성되어 있다.

```
1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)   // TLB 히트
4      if (CanAccess(TlbEntry.ProtectBits) == True)
5          Offset = VirtualAddress & OFFSET_MASK
6          PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7          AccessMemory(PhysAddr)
8      else
9          RaiseException(PROTECTION_FAULT)
10 else // TLB 미스
11     PTEAddr = PTBR + (VPN * sizeof(PTE))
12     PTE = AccessMemory(PTEAddr)
13     if (PTE.Valid == False)
14         RaiseException(SEGMENTATION_FAULT)
15     else if (CanAccess(PTE.ProtectBits) == False)
16         RaiseException(PROTECTION_FAULT)
17     else
18         TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
19         RetryInstruction()
```

〈그림 22.1〉 **TLB 제어 흐름 알고리즘**

하드웨어 부분의 알고리즘은 다음과 같이 동작한다. 먼저, 가상 주소(그림 22.1에서 1번 라인)에서 가상 페이지 번호(virtual page number, VPN)을 추출한 후, 해당 VPN의 TLB 존재 여부를 검사한다(라인 2). 만약 존재하면 **TLB 히트**이고 TLB가 변환 값을 갖고 있다는 것을 뜻한다. 성공! 이제 해당 TLB 항목에서 페이지 프레임 번호(page frame number, PFN)를 추출할 수 있다. 해당 페이지에 대한 접근 권한 검사가 성공하면(4번 라인), 그 정보를 원래 가상 주소의 오프셋과 합쳐서 원하는 물리 주소(PA)를 구성하고, 메모리에 접근할 수 있다(5-7번 라인).

TLB에 변환 정보가 존재하지 않는다면 (**TLB 미스**) 할 일이 많다. 이 예제에서는 하드웨어가 변환 정보를 찾기 위해서 페이지 테이블에 접근하며(11-12번 라인), 프로세스가 생성한 가상 메모리 참조가 유효하고 접근 가능하다면(13, 15번 라인), 해당 변환 정보를 TLB로 읽어들인다(18번 라인). 매우 시간이 많이 소요되는 작업이다. 페이지 테이블 접근을 위한 메모리 참조 때문이다(12번 라인). TLB가 갱신되면 하드웨어는 명령어를 재실행한다. 이번에는 TLB에 변환 정보가 존재함으로, 메모리 참조가 빠르게 처리된다.

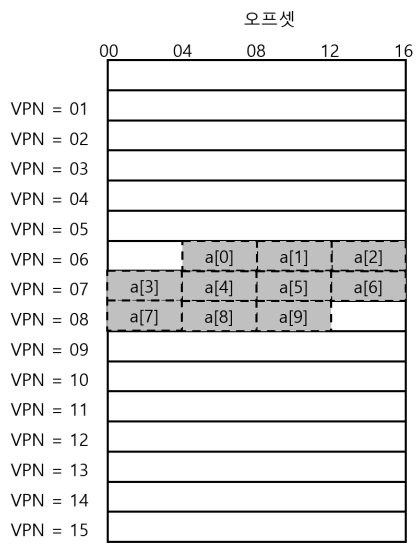
모든 캐시의 설계 철학처럼, TLB 역시 "주소 변환 정보가 대부분의 경우 캐시에 있다"(즉, 캐시에서 히트)라는 가정을 전제로 만들어졌다. TLB는 프로세싱 코어와 가까운 곳에 위치하고 있고, 매우 빠른 하드웨어로 구성되기 때문에, 주소 변환 작업은 그다지 부담스러운 작업이 아니다. 미스가 발생하는 경우, 페이징 비용이 커진다. 페이지 테이블을 접근하여 변환 정보를 찾아야 한다. 메모리 참조(또는 페이지 테이블이

복잡하다면 더 많이)가 추가된다. 이 상황이 자주 일어나면 프로그램은 상당히 느려지게 된다. 메모리 접근 연산은 다른 CPU연산(예, 더하기, 곱하기 등)에 비해 매우 시간이 오래 걸린다. TLB 미스가 많이 발생할수록 메모리 접근 횟수가 많아진다. TLB 미스가 발생하는 경우를 최대한 피해야 한다.

22.2 예제: 배열 접근

TLB 작동 과정을 좀 더 명확히 알아보자. 간단한 가상 주소 트레이스를 대상으로 TLB로 인한 성능 개선을 알아보자. 이 예제에서는 가상 주소 100번지부터 10개의 4바이트 크기의 정수 배열이 존재한다. 가상 주소 공간은 8비트이며, 페이지 크기는 16바이트이다. 가상 주소는 4비트 VPN(16개의 가상 페이지들을 표현)과 4비트 오프셋(각 페이지는 16바이트 크기를 가짐)으로 구성된다.

그림 22.2는 16개의 페이지로 구성된 가상 주소 공간을 도식적으로 표현하고 있다. 배열의 첫 항목(`a[0]`)은 (VPN=06, 오프셋=04)에서 시작한다. 세 개의 4바이트 정수가 페이지에 들어갈 수 있다. 배열은 다음 페이지(VPN=07)에서 계속된다. `a[3]` ... `a[6]`의 네 항목이 존재한다. 배열의 마지막 세 개 항목은 (`a[7]` ... `a[9]`)는 주소 공간 (VPN=08)에 시작하는 다음 페이지에 들어 있다.



〈그림 22.2〉 **예제: 작은 주소 공간 내의 배열**

배열 원소의 합을 구하는 간단한 코드를 살펴보자. C 코드는 다음과 같다.

```
int sum = 0;
for (i = 0; i < 10; i++) {
    sum += a[i];
}
```

메모리 접근을 단순화하기 위하여 변수 `i`, `sum` 그리고 명령어를 위한 메모리 접근은 무시한다. 정수 배열에 대한 메모리 접근만 보기로 하겠다. 첫 번째 배열의 항목 (a[0])이 접근된다. 가상 주소 100번이다. 하드웨어는 VPN을 추출한다. VPN이 06번이다. 하드웨어는 TLB에서 해당 VPN을 검색한다. TLB가 완전히 초기화되어 있다 가정하자. 첫 접근이다. TLB 미스가 발생한다. 미스가 발생하면 해당 VPN 06번에 대한 물리 페이지 번호를 찾아, TLB를 갱신한다.

다음은 a[1]을 읽는다. a[1]을 읽을 때에는 상황이 좋다. TLB 히트다! 배열의 두 번째 항목은 첫 번째 항목과 같은 페이지에 존재한다. 첫 번째 항목을 읽을 때, 이미 해당 페이지를 접근하였기 때문에, 필요한 변환 정보가 이미 TLB에 탑재되어 있다. 두 번째 항목을 읽을 때에는 TLB 히트가 발생한다. a[2]에 대한 접근도 같은 식으로 TLB 사용에 성공한다. 이 항목도 a[0], a[1]과 같은 페이지에 있기 때문이다.

불행히도 a[3]을 읽을 때에는 상황이 안 좋다. TLB 미스다. 하지만, 그 다음 항목들 (a[4] ... a[6])은 메모리 내에 같은 페이지에 있기 때문에 TLB 히트가 된다.

a[7]을 접근할 때 마지막 TLB 미스가 발생한다. 하드웨어는 페이지 테이블을 참고하여 가상 페이지에 해당하는 물리 메모리를 파악한 후에 TLB를 갱신한다. 마지막 두 개의 접근(a[8]와 a[9])은 이미 갱신된 TLB에서 주소 변환 정보를 얻는다. 두 번의 TLB 히트가 발생한다.

배열 원소를 읽는 TLB 동작을 정리해 보면 **미스**, 히트, 히트, **미스**, 히트, 히트, 히트, **미스**, 히트, 히트가 된다. 히트 횟수를 총 접근 횟수로 나누어 얻는 **TLB 히트 비율**은 70%가 된다. 그렇게 높은 수치는 아니지만 (히트율이 거의 100%이면 얼마나 좋을까?), 0은 아니니까 이 또한 놀랍다. 배열이 처음으로 접근되었지만, TLB는 **공간 지역성**(**spatial locality**)으로 인해서 성능을 개선할 수가 있다. 배열의 항목들이 페이지 내에서 서로 인접해 있기 때문에, 페이지에서 첫 번째 항목을 접근할 때만 TLB 미스가 발생한다.

이 예제에서 보는 바와 같이 페이지 크기는 TLB의 효용성에 매우 중요한 역할을 한다. 페이지 크기가 두 배가 되면(16이 아니라 32바이트), TLB 미스 횟수가 더 줄어든다. 일반적인 경우 페이지는 4KB이다. 예제와 같이, 정수 배열을 연속적으로 접근하는 프로그램 같은 경우, TLB 사용은 큰 성능 개선 효과를 가져올 것이다. 페이지 접근 시 한 번의 미스만 발생하기 때문이다.

TLB 성능에 관한 마지막 주요 사항이다. 만약 예제 프로그램이 루프 종료 후에도 배열을 사용한다면, 성능은 더욱 개선될 것이다. 모든 주소 변환 정보가 TLB에 탑재되어 있기 때문이다. TLB가 모든 주소 변환 정보를 저장할 정도로 충분히 크다면, 히트, 히트, 히트, 히트, 히트, 히트, 히트, 히트, 히트, 히트를 얻는다. 이 경우에서는, **시간 지역성**(**temporal locality**)으로 인해 TLB의 히트율이 높아진다. 시간 지역성이란 한번 참조된 메모리 영역이 짧은 **시간** 내에 재 참조되는 현상을 일컫는다. 다른 캐시와 마찬가지로 TLB의 성공 여부는 프로그램의 공간 지역성과 시간 지역성 존재 여부에 달려 있다. 만약 프로그램이 공간 혹은 시간 지역성을 보이는 경우, TLB 사용 효과가 더욱 두드러지게 나타날 것이다. 실제로 많은 프로그램들이 공간 지역성이나 시간 지역성을 띄고 있다.

팁: 가능하면 캐싱을 사용하자

캐싱은 컴퓨터 시스템에서 사용되는 가장 근본적인 성능 개선 기술들 중 하나이다. "일반적인 경우를 빠르게 (make the common case fast)" 하기 위해 오랜 시간 적용되어 온 방법이다[HP06]. 하드웨어 캐시 사용의 근본 취지는 명령과 데이터 참조에 있어서 **지역성(locality)**을 활용하는 것이다. 일반적으로 지역성에는 두 가지가 있다. **시간 지역성(temporal locality)**과 **공간 지역성(spatial locality)**이 그것이다. 시간 지역성은 최근에 접근된 명령어 또는 데이터는 곧 다시 접근될 확률이 높다는 사실에 근거한다. for나 while 등의 반복문을 생각해 보자. 반복문에 사용되는 변수나 명령어들은 일정 시간 동안 반복적으로 접근된다. 공간 지역성은 프로그램이 메모리 주소 x를 읽거나 쓰면, x와 인접한 메모리 주소를 접근할 확률이 높다는 사실에 근거한다. 배열을 순차적으로 읽는 프로그램이 공간 지역성을 갖는 대표적인 예이다. 어떤 프로그램이 시간 혹은 공간 지역성을 내재하느냐 하는 것은 공식이나 규칙으로 명확히 결정할 수 있는 성질의 것이 아니다. 그저 "대충" 판단할 수밖에 없다. 이를 영어로는 Rule of Thumb이라 한다. Rule Of Thumb의 어원은 각자 찾아보도록 하자.

모든 하드웨어 캐시의 목적은 필요한 메모리 내용을 매우 빠른 CPU 칩 내의 메모리에 위치시키고, 접근 지역성을 최대한 활용하는 것이다. 이 원리는 명령어 캐시, 데이터 캐시, 그리고 주소 변환 캐시 등(TLB) 모든 하드웨어 캐시에 적용된다. CPU가 메모리 내용을 참조할 때, (느린) 메모리를 직접 접근하는 것이 아니라, 우선 캐시에 사본이 있는지 먼저 확인한다. 만약 캐시에 원하는 내용이 존재하면 프로세서는 원하는 내용을 빠르게 접근할 수 있으며(즉, 몇 CPU 사이클 내에), 메모리를 접근하기 위해 비싼 시간(수 nsec)을 쓰지 않아도 된다.

이 시점에서 한 가지 의문점이 등장할 것이다. 캐시(TLB와 같은)가 그렇게 좋다면, 왜 초대용량 캐시를 제작하여 모든 내용을 저장하지 않을까? 단순한 질문이지만, 실질적으로는 매우 복잡한 문제이다. 간단히 답을 하겠다. 빠르게 할려면 크기가 작아야 한다. 크게 만들면 느려진다. 빠른 캐시를 원한다면 빛의 속도와 같은 문제나 다른 물리적 제약 조건들이 의미가 있을 정도로 작아야만 한다. 크기가 커지면 느려질 수밖에 없다. 캐시가 "캐시"역할을 하기 위해서는 작아야만 하는 것이다. 캐시에 관한 한, 우리가 풀어야 할 숙제는 "작은" 캐시를 어떻게 "잘" 사용하는가이다.

22.3 TLB 미스는 누가 처리할까

TLB 미스의 처리는 어디서 담당할까? 두 가지 방법이 있다. 하드웨어와 소프트웨어(운영체제)이다. 과거 하드웨어는 복잡한 명령어들로 구성되어 있었다. 이를 **CISC**(complex-instruction set computers)라고 통칭했다. 하드웨어 엔지니어들은 엉뚱한 방법쓰기를 좋아하는[1] 운영체제 개발자들의 특성 때문에 그들을 완전히 신뢰할 수가 없었다. 이런 연유로 TLB 미스를 하드웨어가 처리하도록 설계했다. 이를 위해서 하드웨어가 페이지 테이블에 대한 명확한 정보를 가지고 있어야 한다. 메모리 상 위치(그림 22.1의 11

1) 역자 주: 저자는 sneaky라는 표현을 썼음.

번째 줄의 **page-table base register**를 통해서)와 *정확한 형식을* 파악하고 있어야 한다. 미스 발생 시 하드웨어는 다음과 같은 일을 한다: (i) 페이지 테이블에서 원하는 페이지 테이블 엔트리를 찾고, (ii) 필요한 변환 정보를 추출하여, (iii) TLB를 갱신한 후, (iv) TLB 미스가 발생한 명령어를 재실행한다. 인텔 x86 CPU가 **하드웨어로 관리되는 TLB**의 대표적인 예다. x86 CPU는 **멀티 레벨 페이지 테이블**(**multi-level page table**)을 사용한다(다음 장에 자세한 설명이 나와 있다). CR3 레지스터가 페이지 테이블 주소를 갖고 있다 [Int09].

```
1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)      // TLB 히트
4      if (CanAccess(TlbEntry.ProtectBits) == True)
5          Offset   = VirtualAddress & OFFSET_MASK
6          PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7          Register = AccessMemory(PhysAddr)
8      else
9          RaiseException(PROTECTION_FAULT)
10 else                      // TLB 미스
11     RaiseException(TLB_MISS)
```

〈그림 22.3〉 **TLB 제어 흐름 알고리즘(운영체제가 관리)**

RISC(reduced instruction set computing)는 CISC보다 최근에 등장한 컴퓨터 구조이다. MIPS R10K[Hei93]나 Sun의 SPARC v9[WG00]이 대표적 예다. RISC 기반 컴퓨터는 **소프트웨어 관리 TLB**(**software-managed TLB**)를 사용한다. RISC 기반 컴퓨터에서 TLB 미스를 처리하는 과정은 다음과 같다. TLB에서 주소 찾는 것이 실패하면, 하드웨어는 예외(exception) 시그널을 발생시킨다(그림 22.3의 11번째 줄). 예외 시그널을 받은 운영체제는 명령어 실행을 잠정 중지하고, 실행 모드를 커널 모드로 변경하여, 커널 코드 실행을 준비한다. 실행 모드를 커널 모드로 변경하는 작업의 핵심은 커널 주소 공간을 접근할 수 있도록 특권 레벨(privilege level)로 상향 조정하는 것이다. 커널 모드로 변경이 되면 **트랩 핸들러**(**trap handler**)를 실행한다. 이때 실행되는 트랩 핸들러는 TLB 미스의 처리를 담당하는 운영체제 코드이다. 이 트랩 핸들러는 페이지 테이블을 검색하여 변환 정보를 찾고, TLB 접근이 가능한 "특권" 명령어(privileged instruction)를 사용하여 TLB를 갱신한 후에 리턴한다. 트랩 핸들러에서 리턴되면 하드웨어가 명령어를 재실행한다. 트랩 핸들러가 TLB를 갱신했으므로 이제는 TLB 히트가 날 것이다.

두 가지 중요한 사항을 다시 짚어보자. 첫 번째, TLB 미스를 처리하는 트랩 핸들러는 시스템 콜 호출 시 사용되는 트랩 핸들러와의 차이가 있다. 시스템 콜 호출의 경우는 트랩 핸들러에서 리턴 후 시스템 콜을 호출한 명령어의 "다음" 명령어를 실행한다. 일반적인 프로시저 콜과 동일하다. 프로시저 콜이 리턴되면, 프로시저를 호출한 다음 라인부터 실행이 시작된다. TLB의 경우는 다르다. TLB 미스 처리의 경우, 트랩에서 리턴하면 트랩을 발생시킨 명령을 "다시" 실행해야 하며, 재실행 시에는 TLB에서 히트가 발생한다. 트랩이 발생하면 운영체제는 트랩 핸들러가 종료되었을 때 다시 실행을 계속할 명령어 주소(Program Counter 값)를 저장한다. 중요한 사실을 알 수 있다. 운영체제는

트랩 발생의 원인에 따라 현재 명령어의 PC값 혹은 다음 명령어의 PC값을 저장해야 한다.

두 번째, TLB 미스 핸들러를 실행할 때, TLB 미스가 무한 반복되지 않도록 주의를 해야 한다. TLB 미스 핸들러를 접근하는 과정에서 TLB 미스가 발생하는 상황이다. 이를 위해 다양한 해법들이 존재한다. 예를 들면, TLB 미스 핸들러를 물리 메모리에 위치시키는 것도 한 방법이다. TLB 미스 핸들러의 주소는 핸들러의 '물리' 주소로 표시된다. 이 경우 해당 TLB 미스 핸들러는 **unmap**되어 있으며 주소 변환이 필요없다. 다른 방법으로 TLB의 일부를 핸들러 코드 주소를 저장하는 데 영구히 할당하는 것이다. 이렇게 되면 TLB 핸들러는 항상 TLB에서 히트된다. 이를 **연결**(**wired**) 변환이라 한다.

TLB를 소프트웨어로 관리하는 방식의 주된 장점은 *유연성이다*. 운영체제는 하드웨어 변경없이 페이지 테이블 구조를 자유로이 변경할 수 있다. 또 다른 장점은 단순함이다. TLB 제어 흐름에서 보는 것과 같이(그림 22.3의 11번째 줄과 그림 22.1의 11-19번째 줄을 비교) 미스가 발생하였을 때 하드웨어는 별로 할 일이 없다. 예외가 발생하면 운영체제의 TLB 미스 핸들러가 나머지 일을 처리한다.

22.4 TLB의 구성: 무엇이 있나?

하드웨어 TLB의 구성을 좀 더 상세히 살펴보자. 일반적인 TLB는 32, 64, 또는 128개의 엔트리를 가지며, **완전 연관**(**fully associative**) 방식으로 설계된다. 완전 연관 방식에서 변환 정보는 TLB 내에 어디든 위치할 수 있으며, 원하는 변환 정보를 찾는 검색은 TLB 전체에서 병렬적으로 수행된다. TLB의 구성은 아래와 같다.

```
VPN | PFN | 다른비트들
```

변환 정보 저장 위치에 제약이 없도록, 각 항목마다 가상 페이지 번호(VPN)와 물리 페이지 번호(PFN)가 있다. 하드웨어 측면에서 보자면, TLB는 **완전 연관** 캐시이다. 변환 주소를 찾을 때, 하드웨어는 TLB의 각 항목을 동시에 검색한다.

TLB 항목에서 VPN과 PFN을 제외한 "다른 비트들"에 대해서도 눈여겨 볼 필요가 있다. TLB는 일반적으로 **valid bit**를 갖고 있다. 이 비트는 특정 항목이 유효한 변환 정보를 갖고 있는지 여부를 나타낸다. **보호**(**protection**) 비트라는 것도 있다. 보호 비트들은 페이지가 어떻게 접근될 수 있는지를 나타낸다. 그 쓰임새는 페이지 테이블에서와 같다. 예를 들어, 코드 페이지들에 대한 변환은 *읽기와 실행이라고* 표기가 되어 있으며 힙 페이지들은 *읽기와 쓰기라고* 표기되어 있을 수 있다. 그 외에, **주소 공간 식별자**(**address-space identifier**), **더티 비트**(**dirty bit**) 등도 있다. 좀 더 자세한 정보는 다음을 참고하자.

22.5 TLB의 문제: 문맥 교환

TLB를 사용하게 되면 프로세스 간 (주소 공간들로 인해) 문맥 교환 시, 새로운 문제가 등장한다. 구체적으로 살펴보자. TLB에 있는 가상 주소와 실제 주소 간의 변환 정보는

여담: RISC 대 CISC

1980년대에 컴퓨터 구조 분야에 큰 전투가 있었다. 전투의 한 쪽은 **복잡 명령어 컴퓨터**(**complex-instruction set computing**)를 대표하는 **CISC** 진영이었고, 다른 반대편은 **축소 명령어 컴퓨터**(**reduced instruction set computing**)를 대표하는 **RISC** 진영이었다[PS81]. RISC 측에서는 버클리의 David Patterson과 스탠퍼드의 John Hennessy가 선봉장이었다 (이 둘은 잘 알려진 책의 공동 저자이기도 하다[HP06]). 하지만, 정작 John Cocke가 RISC에 대한 초기 연구를 진행한 공로를 인정받아 튜링상을 수상하였다[CM00].

CISC 명령어 집합은 많은 수의 명령어로 구성되어 있으며, 각 명령들은 강력한 기능을 가지는 것이 특징이다. 예를 들어, 두 개의 포인터와 길이를 인자로 하여 목적한 주소로 문자열 복사하는 것을 들 수가 있다. CISC의 설계 배경은 어셈블리어 명령어로 상위 수준 작업을 실행하고, 코딩을 쉽게 하며, 코드를 간단하게 하는 데 있다.

RISC 진영의 입장은 정반대이다. RISC 진영의 핵심 논리는 "어셈블리 명령어는 컴파일러를 위한 것이다."라는 사실이다. 컴파일러는 몇 가지의 단순한 어셈블리 명령어만 있으면 고성능의 코드를 생성해 낼 수 있다. RISC 지지자들은 명령어 집합을 작고 간단하게 설계하여, 즉 하드웨어를 최대한 단순화하여(특히 마이크로코드를), 빠른 컴퓨터를 만들 것을 주장하였다.

초기에 좋은 성능의 RISC 칩들이 발표되었다[BC91]. 이들은 기술 발전에 큰 영향을 미쳤다. 많은 논문이 발표되었고, 회사도 몇 개 생겨났다(예, MIPS와 Sun). 시간이 지남에 따라 인텔과 같은 CISC 제조사들이 프로세서 핵심에 RISC 기술을 도입하였다. 예를 들어, RISC와 유사한 형식의 마이크로 명령어들을 개발하고, CISC 명령어를 마이크로 명령어로 전처리하는 파이프라인 단계들을 도입하였다. 이러한 기술 혁신과 반도체 공정 집적도의 향상에 따라, 오늘날까지도 CISC 기반 프로세스가 경쟁력을 유지할 수 있었다. 지금은, RICS와 CISC 간의 장단점 논쟁이 거의 의미가 없어졌다. 두 종류의 프로세서 모두 빠르게 동작하도록 제작된다.

그것을 탑재시킨 프로세스에서만 유효하다. 다른 프로세스들에게는 의미가 없다. 새로운 프로세스에서는 이전에 실행하던 프로세스의 변환 정보를 사용하지 않도록 주의해야 한다.

예제를 하나 살펴보자. 하나의 프로세스 (P1)가 실행 중이라고 하자. 이 프로세스는 TLB가 자신에게 유효한 변환 정보를 캐싱하고 있다고 가정한다. 즉, P1의 페이지 테이블의 내용을 갖고 있다고 가정한다. 이 예제에서는 P1의 10번째 가상 페이지가 물리 프레임 100에 매핑되어 있다고 하자.

이 예제에서는 또 다른 프로세스 P2가 있다고 가정하고 운영체제가 곧 문맥 교환을 수행하기로 결정하여 이 프로세스를 실행시키려고 한다. P2의 10번째 가상 페이지는 물리 프레임 170에 매핑되어 있다고 가정하자. 두 프로세스의 항목들이 TLB에 존재한다고 하면 TLB의 내용은 다음과 같이 될 것이다.

위의 TLB에 문제가 있다. VPN 10에 대한 변환 정보가 두 개 존재하는 것이다. 10번 VPN이 PFN 100(P1) 과 PFN 170(P2)으로 변환될 수 있다. 하지만, 어떤 프로세스를

여담 : TLB valid bit ≠ 페이지 테이블 valid bit

흔한 실수 중 하나는 TLB의 valid bit와 페이지 테이블의 valid bit를 혼동하는 것이다. 페이지 테이블에서 어떤 페이지 테이블 항목(page-table entry, PTE)이 "무효"로 표시되어 있다면 해당 페이지는 프로세스에 할당되지 않았다는 것을 의미한다. 정상적인 프로그램은 해당 페이지를 접근하지 않는다. 유효하지 않은 페이지를 접근할 경우, 운영체제는 트랩을 발생시킨다. 일반적인 경우, 유효하지 않는 페이지를 접근한 프로세스는 kill 된다.

반면에, TLB의 valid bit는 TLB에 탑재되어 있는 해당 변환 정보가 유효한지를 나타내기 위해 사용된다. 예를 들어, 시스템이 시작되면 어떤 변환 정보도 아직 캐시되지 않았기 때문에 일반적으로 TLB의 모든 항목은 "무효"로 초기화되어 있다. 가상 메모리가 초기화되고, 프로그램들이 실행을 시작하여 자신의 가상 주소 공간을 접근하게 되면, TLB는 서서히 채워지게 된다. TLB는 곧 유효한 항목들로 채워지게 된다.

이제 곧 살펴보게 되겠지만, TLB의 valid bit는 문맥 전환할 때도 상당히 유용하다. 문맥 전환 시, 시스템은 모든 TLB 항목을 "무효"로 세팅하여, 새로운 프로세스가 이전 프로세스의 변환 정보를 사용하는 것을 원천적으로 차단한다.

VPN	PFN	valid	prot
10	100	1	rwx
--	---	0	---
10	170	1	rwx
--	---	0	---

위한 항목인지 알 길이 없다. TLB가 정확하고 효율적으로 멀티 프로세스 간의 가상화를 지원하기 위해서는 추가적 기능이 필요하다. 핵심은 다음과 같다.

핵심 질문 : 문맥 교환 시 TLB 내용을 어떻게 관리하는가

문맥 교환 시 실행될 프로세스에게는 이전 프로세스가 사용한 TLB 정보는 의미가 없다. 하드웨어 또는 운영체제는 이러한 문제를 해결하기 위해 무엇을 해야 하는가?

이 문제는 여러 가지 해법이 있을 수 있다. 한 방법은 문맥 교환을 수행할 때 다음 프로세스가 실행되기 전에 기존 TLB 내용을 비우는 것이다. 소프트웨어 기반의 시스템에서는 특별한(그리고 특권을 갖는) 하드웨어 명령어을 사용하여 이 목적을 달성할 수 있다. 하드웨어로 관리되는 TLB는 페이지 테이블 베이스 레지스터가 변경될 때 비우기를 시작할 수 있다(운영체제는 문맥 교환을 할 때 PTBR을 어쨌든 변경해야 한다). 둘 중 어느 경우든 비우는 작업은 모든 valid bit를 0으로 설정을 하는 것이다.

문맥 교환할 때마다 TLB를 비우면, 잘못된 변환 정보를 사용하는 상황을 방지할

수 있다. 하지만, 이 방식은 공짜가 아니다. 새로운 프로세스가 실행될 때, 데이터와 코드 페이지에 대한 접근으로 인한 TLB 미스가 발생하게 된다. 문맥 교체가 빈번히 발생한다면, 이 또한 성능에 큰 부담을 가져올 수 있다.

이 부담을 개선하기 위해 몇몇 시스템에서는 문맥 교환이 발생하더라도 TLB의 내용을 보존할 수 있는 하드웨어 기능을 추가하였다. TLB 내에 **주소 공간 식별자(address space identifier, ASID)** 필드를 추가하는 것이 그것이다. ASID는 **프로세스 식별자(process identifier, PID)**와 대략적으로 유사하다. 단, ASID는 좀 더 적은 비트를 갖고 있다(예, ASID는 8비트를 갖고 있지만 PID는 32비트를 갖고 있다).

위에서 사용한 TLB 예제에 ASID 정보를 추가하면 프로세스들이 TLB의 공간을 공유할 수가 있다. 다음은 ASID 필드를 추가한 TLB의 모습을 나타낸다.

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
--	---	0	---	-
10	170	1	rwx	2
--	---	0	---	-

주소 공간 식별자를 사용할 경우, 프로세스 별로 TLB 변환 정보를 구분할 수 있다. 올바른 주소 변환을 위해서 하드웨어는 현재 어떤 프로세스가 실행 중인지 파악하고 있어야 한다. 이를 위해, 문맥 전환 시, 운영체제는 새로운 ASID 값을 정해진 레지스터에 탑재한다.

추가적으로 TLB의 두 항목이 매우 유사한 경우를 생각해 볼 수도 있다. 이 예제에서는 두 개의 다른 VPN을 갖는 두 개의 다른 프로세스들의 두 항목이 동일한 물리 페이지를 가리키고 있다.

VPN	PFN	valid	prot	ASID
10	101	1	r-x	1
--	---	0	---	-
50	101	1	r-x	2
--	---	0	---	-

이러한 경우는 두 개의 프로세스들이 하나의 페이지(예, 코드 페이지)를 공유하고 있을 때 발생할 수가 있다. 이 예제에서는 프로세스 1이 물리 페이지 101을 프로세스 2와 공유하고 있다. P1은 이 페이지를 자신의 주소 공간의 10번째 페이지에 매핑하고 있으며 P2는 자신의 주소 공간의 50번째 페이지에 매핑하고 있다. 코드 페이지를 공유하는 것은 (바이너리에서든 또는 공유 라이브러리이든) 사용되는 물리 페이지의 수를 줄일 수 있기 때문에 유용하다. 그리고 공유 페이지를 사용하면 메모리 부하도 줄일 수 있다.

22.6 이슈: 교체 정책

모든 캐시가 그러하듯이 TLB에서도 **캐시 교체**(**cache replacement**) 정책이 매우 중요하다. TLB에 새로운 항목을 탑재할 때, 현재 존재하는 항목 중 하나를 **교체** 대상으로 선정해야 한다. 어느 것을 선택해야 할까?

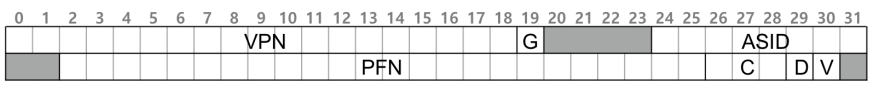
핵심 질문: TLB 교체 정책은 어떻게 설계하는가
TLB에 새로운 항목을 추가할 때 어떤 항목을 교체해야 할까? 목표는 **미스율**을 줄여 (또는 **히트 비율**을 증가시켜서) 성능을 개선하는 것이다.

이 내용은 디스크와 메모리 간의 페이지 스와핑 부분에서 상세히 다루도록 하고, 여기서는 몇 개의 일반적인 정책들만 요약하도록 하겠다. 한 가지 흔한 방법은 가장 오래 전부터 사용되었던 **최저 사용 빈도**(**least-recently-used**, **LRU**) 항목을 교체하는 것이다. LRU는 메모리 참조 패턴에서의 지역성을 최대한 활용하는 것이 목적이다. 사용되지 않은지오래된 항목일수록, 앞으로도 사용될 가능성이 작으며, 교체 대상으로 적합하다는 가정에 근거한다. 또 다른 일반적인 방법은 랜덤(random) 정책이다. 랜덤 정책에서는 교체 대상을 무작위로 정한다. 랜덤 교체 정책은 때때로 잘못된 결정을 내릴 수도 있다. 하지만, 랜덤 교체 정책은 구현이 간단하고 예상치 못한 예외 상황의 발생을 피할 수 있다는 장점이 있다. 예를 들어 LRU와 같은 "합리적인" 정책은 n 개의 변환 정보를 저장할 수 있는 TLB가 $n+1$ 페이지들에 대해서 반복문을 수행하는 프로그램에서 사용될 경우 최악의 TLB 미스를 생성하게 된다. 그 이유는 독자들이 한 번 상상해 보기 바란다. LRU는 모든 접근에 대해 미스를 유발한다. 그에 반해 랜덤의 경우에는 훨씬 더 잘 동작한다.

22.7 실제 TLB

마지막으로 실제 TLB가 어떻게 생겼는지 간략히 살펴보자. MIPS R4000을 예로 들겠다. MIPS R4000은 소프트웨어로 관리되는 TLB를 사용한다. 비교적 최근 시스템이다. MIPS의 간략화된 TLB 항목은 그림 22.4와 같다.

MIPS R4000은 32비트 주소 공간에 4KB 페이지를 지원한다. 일반적인 가상 주소 공간을 사용하기 때문에 VPN은 20비트 그리고 오프셋으로 12비트를 갖을 것이라 예측할 수 있지만, 아니다. TLB에서 보는 바와 같이 19비트가 VPN에 할당되어 있다. 전체 주소 공간의 절반만 사용자 주소 공간으로 할당되어 있기(나머지 반은 커널이 사용한다.)



〈그림 22.4〉 **MIPS의 TLB 항목**

때문에, VPN에 19비트가 할당되어 있다. 물리 프레임 번호(PFN)로 24비트가 할당되어 있다. 64 GB의 (물리) 주 메모리(2^{24}개의 4 KB 페이지들) 지원이 가능하다.

MIPS의 TLB에는 몇 가지 중요한 비트들이 더 있다. 전역(global) 비트 (G)다. 이 비트는 프로세스들 간에 공유되는 페이지들을 위해 사용된다. 전역 비트가 설정되어 있으면 ASID는 무시된다. 8비트 길이의 ASID 필드가 있다. 운영체제는 이 부분을 보고 주소 공간들을 서로 구분한다(위에서 설명한 것과 같이). 여기서 재미있는 질문을 할 수 있다. 만일 256(2^8)개 이상의 프로세스들이 동시에 실행된다면? 마지막으로 3비트 길이의 일관성(coherence, C) 비트를 볼 수 있다. 이 비트를 보고 페이지가 하드웨어에 어떻게 캐시되어 있는지를 판별한다 (이 부분은 이 글의 주제의 범위 밖의 내용이다). 더티(dirty) 비트는 페이지가 갱신되면 세팅되며(나중에 이것의 활용을 살펴볼 것이다), 유효(valid) 비트는 항목에 유효한 변환 정보가 존재하는지를 나타낸다. 그리고 페이지 마스크(page mask) 필드도 있는데 (표현은 안 되었음), 여러 개의 페이지 크기를 지원할 때 사용된다. 대용량 페이지 크기의 유용함에 대해서는 곧 배울 것이다. 마지막으로 회색 처리된 64번째 비트는 사용하지 않는다.

MIPS의 TLB들은 일반적으로 32개 또는 64개의 항목으로 구성된다. 대부분은 사용자 프로세스들이 사용한다. 몇 개는 운영체제를 위해서 예약이 되어 있다. 운영체제에 의해서 연결된(wired) 레지스터가 설정이 될 수도 있으며, 운영체제는 몇 개의 TLB를 예약할지를 하드웨어에게 알린다. 운영체제는 예약된 매핑 영역을 TLB 미스가 발생해서는 안 되는(예, TLB 미스 핸들러) 코드와 데이터를 위해서 사용한다.

MIPS의 TLB는 소프트웨어가 관리하기 때문에 TLB를 갱신하기 위한 명령어가 필요하다. MIPS는 네 개의 명령어를 제공한다. `TLBP`는 어떤 특정 변환 정보가 존재하는지 탐색하는 데 사용이 되며 `TLBR`은 TLB 항목의 내용을 레지스터로 읽어 드리는 데 사용된다. `TLBWI`는 특정 TLB 항목을 교체하며 `TLBWR`은 임의의 TLB 항목을 교체한다. 운영체제는 이 명령어들을 사용하여 TLB의 내용을 관리한다. 이 명령어들은 특별한 **실행권한(privileged)**을 갖고 있어야 한다. 만약 사용자 프로세스가 TLB 내용을 변경할 수 있다면 어떻게 될지 상상해 보라(힌트: 거의 무엇이든지 가능하다. 기계의 사용 권한을 바꿔서 악의적으로 "운영체제"를 사용할 수 있으며 또는 태양을 없애버릴 수도 있다).

22.8 요약

지금까지 주소 변환을 더 빠르게 처리하기 위한 하드웨어 기법에 대해 살펴보았다. 칩 상의 작은 전용 TLB를 주소 변환 캐시로 사용하여 대부분의 메모리 참조들은 메인 메모리 상의 페이지 테이블을 읽지 않고도 처리가 가능하게 되었다. TLB의 사용으로 일반적인 경우에 프로그램은 메모리 가상화 기능이 없는 것과 동일한 성능을 보일 것이기 때문에 TLB는 운영체제 입장에서 훌륭한 성과일 뿐만 아니라 현대 시스템에서 페이징을 사용하기 위한 필수 요소이다.

하지만, TLB가 모든 프로그램에서 항상 제대로 작동하는 것은 아니다. 특히, 프로그램이 짧은 시간 동안 접근하는 페이지들의 수가 TLB에 들어갈 수 있는 수보다 많다면

팁: RAM은 항상 RAM이 아니다(Culler's Law)

임의-접근 메모리(random-access memory) 또는 **RAM**이라는 용어는 RAM의 어떤 부분이든 다른 부분을 접근하는 것과 같은 속도로 접근할 수 있다는 것을 암시한다. 일반적으로 RAM이 그렇게 동작한다고 생각해도 괜찮다. 왜냐하면, 하드웨어/운영체제의 TLB와 같은 기능 때문이다. 특정 메모리의 페이지를 접근하는 것이 비쌀 수도 있다. 특히, 그 페이지가 현재 TLB에 매핑되어 있지 않다면 더욱 그렇다. 그러므로 **RAM은 항상 RAM이 아니다**라고 하는 구현의 팁을 기억하는 것이 도움이 된다. 때로는, 특히 TLB가 다룰 수 있는 것보다 많은 수의 페이지들을 접근할 때에 주소 공간을 임의 접근하는 것은 심각한 성능 저하를 가져올 수가 있다. 우리의 지도 교수 중의 한 분이셨던 David Culler는 TLB가 여러 성능 문제들의 원인이라고 지적하시곤 하셨다. 그를 위하여 이 법칙의 이름을 **Culler's Law**라고 부른다.

그 프로그램은 많은 수의 TLB 미스를 발생할 것이고 느리게 동작하게 될 것이다. 이러한 현상을 **TLB 범위(TLB coverage)**를 벗어난다고 하며 특정 프로그램들에서는 중요한 문제가 될 수도 있다. 다음 장에서 다룰 한 가지 해법은 더 큰 페이지 크기를 지원하도록 하는 것이다. 더 큰 페이지들을 사용할 경우 TLB의 유효 범위가 늘어날 수가 있다. 더 큰 페이지들을 위한 지원은 **데이터베이스 관리 시스템(database management system, DBMS)**과 같은 프로그램들에 의해서 주로 사용되며, 이러한 프로그램의 자료 구조들은 클 뿐만 아니라 임의적으로 접근된다.

TLB에 관한 또 다른 이슈는 CPU 파이프라인에서 TLB 접근은 병목이 될 수 있다. **물리적으로 인덱스된 캐시(physically-indexed cache)**가 사용될 경우 특히 문제가 된다. 이 캐시에서는 주소 변환이 캐시 접근 *전에* 이루어져야 하는데, 그런 경우에는 상당히 느려질 수가 있다. 이러한 문제 때문에 *가상* 주소로 캐시를 접근하는 다양한 방법들이 고안되었며, 캐시 히트가 발생한 경우에 비싼 변환을 하지 않도록 하였다. **가상적으로 인덱스된 캐시(virtually indexed cache)**는 일부 성능 문제를 해결하지만 새로운 하드웨어 설계 문제들을 수반한다. 자세한 내용은 Wiggins가 쓴 훌륭한 개요을 참고하자[Wig03].

참고 문헌

[BC91] **"Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware Organization"**
D. Bhandarkar and Douglas W. Clark
Communications of the ACM, September 1991
훌륭하고 공정하게 RISC와 CISC를 비교한 글이다. 핵심 요점은 하드웨어가 비슷하다면 RISC가 3배 정도 더 성능이 빠르다는 것이다.

[CP78] **"The architecture of the IBM System/370"**
R.P. Case and A. Padegs
Communications of the ACM. 21:1, 73-96, January 1978
*아마도 **변환 색인 버퍼**(**translation lookasdie buffer**)라는 용어를 처음 쓴 논문일 것이다. 멘체스터 대학에서 지도 시스템을 개발하던 사람들이 사용하던 **색인 버퍼**라고 하는 캐시의 역사적인 명칭에서 따왔다. 주소 변환 정보의 캐시는 이후에 **변환 색인 버퍼**로 바뀌게 되었다. 색인 버퍼라는 용어가 인기를 얻지 못하기는 하였지만, TLB라는 이름은 어떠한 이유에서 계속 사용되고 있다.*

[CM00] **"The evolution of RISC technology at IBM"**
John Cocke and V. Markstein
IBM Journal of Research and Development, 44:1/2
많은 사람들이 첫 번째의 진정한 RISC 마이크로프로세서라고 할 수 있는 IBM 801에 대한 개념과 기술들에 대한 정리이다.

[Cou95] **"The Core of the Black Canyon Computer Corporation"**
John Couleur
IEEE Annals of History of Computing, 17:4, 1995
대단히 흥미로운 이 역사적인 자료에서 Couleur는 1964년에 GE에서 일하면서 TLB를 어떻게 개발하였는지, 그리고 이후에 MIT의 프로젝트 MAC 사람들과 우연하게 협업을 하게 된 이야기를 소개하고 있다.

[CG68] **"Shared-access Data Processing System"**
John F. Couleur and Edward L. Glaser
Patent 3412382, November 1968
주소 변환 정보를 연관 메모리에 저장하는 개념을 다룬 특허이다. 이 개념은 Couleur에 따르면 1964년에 나왔다.

[Hei93] **"MIPS R4000 Microprocessor User's Manual"**
Joe Heinrich
Prentice-Hall, June 1993
URL: `http://cag.csail.mit.edu/raw/documents/R4400_Uman_book_Ed2.pdf`

[HP06] **"Computer Architecture: A Quantitative Approach"**
John Hennessy and David Patterson
Morgan-Kaufmann, 2006
컴퓨터 구조에 관한 훌륭한 책이다. 우리는 고전의 첫판에 대한 이상한 애착을 갖고 있다.

[Int09] **"Intel 64 and IA-32 Architectures Software Developer's Manuals"**
Intel, 2009
URL: `http://www.intel.com/products/processor/manuals`
특히, "Volume 3A: System Programming Guide Part 1"과 "Volume 3B: System Programming Guide Part 2"를 주의 깊게 읽어보자.

[PS81] **"RISC-I: A Reduced Instruction Set VLSI Computer"**
D.A. Patterson and C.H. Sequin
ISCA '81, Minneapolis, May 1981
RISC라는 용어를 소개한 논문으로 성능을 위해서 컴퓨터 칩들을 간단화하는 연구들의 산사태를 시작시켰다.

[Saa92] **"CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking"**
Rafael H. Saavedra-Barrera
EECS Department, University of California, Berkeley Technical Report No. UCB/CSD-92-684, February 1992
URL: `www.eecs.berkeley.edu/Pubs/TechRpts/1992/CSD-92-684.pdf`
응용 프로그램의 실행 시간을 예측하는 방법에 대한 훌륭한 학위논문으로서 그 시간을 구성 요소들로 세분화하고 각 요소들의 비용을 알아내는 방식을 사용하였다. 이 연구에서 가장 흥미로운 부분은 (제5장에서 설명하고 있는) 캐시 계층의 세부적 부분들을 측정할 수 있는 도구일 것이다. 거기에 있는 훌륭한 도표들을 꼭 살펴보기 바란다.

[WG00] **"The SPARC Architecture Manual: Version 9"**
David L. Weaver and Tom Germond
SPARC International, San Jose, California, September, 2000
URL: `http://www.sparc.org/standards/SPARCV9.pdf`

[Wig03] **"A Survey on the Interaction Between Caching, Translation and Protection"**
Adam Wiggins
University of New SouthWales TR UNSW-CSE-TR-0321, August, 2003
TLB들이 하드웨어 캐시라고 하는 CPU 파이프라인의 다른 부분들과 상호작용하는 것에 대한 훌륭한 개관이다.

숙제 (측정)

이 숙제에서는 TLB를 접근하는 크기와 비용을 측정할 것이다. 이 개념은 Saavedra-Barrera의 연구[Saa92]에 기초한 것으로 이들이 만든 간단하지만 캐시 계층의 여러 측면을 측정하는 아름다운 기법을 이용한다. 그리고 이 모든 것이 아주 간단한 사용자 수준의 프로그램을 통해 할 수 있다. 좀 더 자세히 알고 싶다면 이 연구를 읽어보도록 하자.

기본 개념은 커다란 자료 구조(예, 배열) 내에서 몇 개의 페이지들을 접근하고 그 접근들에 대한 시간을 측정하는 것이다. 예를 들어서 어떤 기계의 TLB 크기는 4라고 해보자(매우 작기는 하겠지만 이 논의의 목적에 충분히 부합한다). 프로그램을 작성하였는데, 그 프로그램이 네 개 또는 그보다 적은 페이지들을 접근한다면 각 접근들은 TLB 히트가 될 것이며 상대적으로 빠를 것이다. 하지만, 다섯 개 또는 그 이상을 반복문에서 계속 접근한다면 TLB 미스를 만들어서 각 접근은 비용이 갑자기 크게 늘어나게 된다.

다음의 간단한 코드에서 배열을 한 번만 접근하는 반복문을 표현하였다.

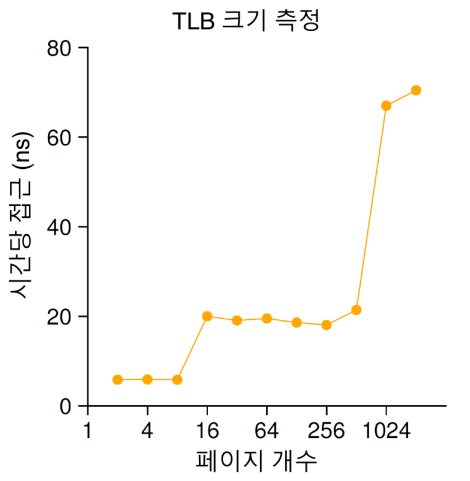
```
int jump = PAGESIZE / sizeof(int);
for (i = 0; i < NUMPAGES * jump; i += jump) {
    a[i] += 1;
}
```

이 반복문에서는 배열 `a`는 페이지당 하나의 정수 형 크기가 갱신이 되며 정해진 `NUMPAGES` 개만큼의 페이지들이 변경된다. 여러 차례 이 반복문이 수행되는 것(이 반복문을 반복하는 또 다른 반복문을 1억 번 수행하거나 또는 이 반복문이 수초 동안 실행될 수 있을 만큼의 반복을 수행한다)을 측정하는 것을 통해 각 접근이 얼마나 오랜 시간이 걸리는지 (평균적으로) 측정할 수 있다. `NUMPAGES`가 커짐에 따라서 비용이 뛰는 정도를 살펴보면 일 단계 TLB의 크기가 얼마나 큰지를 알 수 있으며 이 단계 TLB가 존재하는지를 판단할 수 있으며(존재한다면 얼마나 큰지도 함께), 그리고 TLB 히트와 미스가 성능에 어떤 영향을 줄 수 있는지에 대한 감을 잡을 수가 있다.

그림 22.5에 예시 그래프가 있다. 그래프에서 볼 수 있듯이, 몇 개의 페이지들을 접근할 때(8개 또는 그 이하) 평균 접근 시간은 약 5 nsec가 걸렸다. 16개 또는 그 이상의 페이지들이 접근되었을 때는 접근마다 약 20 nsec로 갑자기 비용이 늘어났다. 그 다음으로 비용이 늘어나는 구간은 1024개 이상의 페이지들을 접근할 때이며 각 접근은 약 70 nsec가 소요된다. 이 데이터에서 이 단계 TLB 계층이 있다고 결론지을 수 있으며 첫 번째는 상당히 작다(8개에서 16개의 항목을 담을 수 있는 것으로 보임). 두 번째는 좀 더 크지만 더 느리다(약 512개를 담을 수 있음). 일 단계 TLB의 히트들과 미스들을 비교해 보면 그 차이가 상당한 것을 알 수 있으며 약 14배 정도 차이나는 것을 볼 수 있다. TLB 성능은 중요하다!

문제

1. 시간을 재기위해서는 `gettimeofday()`를 통해서 만들 수 있는 타이머를 사용해야 할 것이다. 이 타이머는 얼마나 정확한가? 시간을 정확하게 재기 위해서는 타이머가



〈그림 22.5〉 **TLB 크기와 미스의 비용 파악하기**

얼마나 길어야 하는가?(반복문 내에서 몇 번이나 페이지를 접근해야 시간을 정확히 잴 수 있는지를 결정하는 데 도움을 줄 것이다)

2. `tlb.c` 라는 프로그램을 작성하여 각 페이지를 접근하는 데 드는 대략적인 비용을 측정해 보자. 프로그램의 입력 값은 접근해야 하는 페이지들의 수와 시도 횟수이다.

3. 선호하는 스크립트 언어(csh, python 등)를 사용하여 프로그램을 실행시키는 스크립트를 작성하여 접근하는 페이지들의 수를 1부터 수천까지 2의 배수로 증가시키도록 해 보자. 다른 기계들에서 스크립트를 실행하여 데이터를 수집하자. 신뢰할 수 있는 측정치를 얻기 위해서는 몇 번이나 시도해 봐야 하는가?

4. 그 다음으로 결과들을 사용하여 앞서 보인 그래프처럼 만들어 보자. `ploticus`와 같은 툴을 사용하면 된다. 시각화하는 것은 데이터를 좀 더 잘 이해하는 데 도움이 되는데, 왜 그런 것 같은가?

5. 한 가지 주의해야 할 것은 컴파일러 최적화이다. 컴파일러는 프로그램의 다른 부분들이 사용하지 않는 변수를 반복문을 통해 증가시키는 부분을 빼버리는 등의 온갖 교묘한 짓들을 한다. 컴파일러가 작성한 TLB 크기 추정기를 어떻게 수정하여야 컴파일러가 주 반복문을 제거하지 않도록 만들 수 있을까?

6. 또 한 가지 주의해야 할 것은 대부분의 시스템은 CPU가 여러 개이며, 각 CPU는 물론 각각의 TLB 계층을 갖고 있다. 정말 제대로 측정하기 위해서는 스케줄러가 하나의 CPU에서 또 다른 CPU로 이동시키지 못하도록 만들어서 하나의 CPU만을 사용해서 측정해야 할 것이다. 어떻게 그렇게 할 수 있을까? (힌트: 구글에서 "쓰레드 고정하기(pinning a thread)"를 검색해 보자) 이렇게 하지 않았을 때는 어떤 일이 발생할까? 그리고 코드가 한 CPU에서 다른 CPU로 이동할 때 어떤 일이 발생할까?

7. 또 다른 발생할 수 있는 문제는 초기화와 관련되어 있다. 접근하기 전에 배열 `a`를 초기화하지 않는다면, 처음 `a`의 배열을 접근할 경우 한 번도 접근한 적이 없기 때문에

매우 비싼 비용이 들 것이다. 이 부분이 시간을 재는 데에 영향을 줄까? 이러한 잠재적인 비용 요인의 균형을 잡기 위해서 무엇을 할 수 있을까?