19

Disk Belleği: Daha Hızlı Çeviriler (TLB'ler)

Sanal belleği desteklemek için disk belleğini çekirdek mekanizma olarak kullanmak, ek performans yüklerine yol açabilir. Adres alanını küçük, sabit boyutlu birimlere (yani sayfalara) bölerek, sayfalama büyük miktarda haritalama bilgisi gerektirir. Bu eşleme bilgileri genellikle fiziksel bellekte depolandığından dolayı, disk belleği mantıksal olarak program tarafından oluşturulan her sanal adres için fazladan bir bellek araması gerektirir. Her talimat getirmeden veya açık yükleme veya depolamadan önce çeviri bilgileri için belleğe gitmek engelleyici derecede yavaştır. Ve böylece sorunumuz:

Püf Noktası:
Adres Çevirisini Nasıl Hızlandırırız

Adres çevirisini nasıl hızlandırabilir ve genellikle sayfalamanın gerektirdiği ekstra bellek başvurusundan nasıl kaçınabiliriz? Hangi donanım desteği gereklidir? Hangi işletim sistemi katılımı gereklidir?

İşleri hızlı hale getirmek istediğimizde, işletim sisteminin genellikle biraz yardıma ihtiyacı vardır. Ve yardım genellikle işletim sisteminin eski arkadaşından gelir: donanım. Adres çevirisini hızlandırmak için, (tarihsel nedenler için [CP78]) bir **Etkin sayfalar ön belleği (translation-lookaside buffer)** veya **TLB** [CG68, C95] olarak adlandırılan şeyi ekleyeceğiz. TLB, çipin **bellek yönetim biriminin (memory-managment unit MMU)** bir parçasıdır ve popüler sanaldan fiziksele adres çevirilerinin donanım **önbelleğidir (cache)**; bu nedenle, **adres-çeviri önbelleği** (**address-translation cache)** daha iyi bir isim olacaktır. Her sanal bellek kaynağı, donanım önce istenen çevirinin orada tutulup tutulmadığını görmek için TLB'yi kontrol eder; eğer öyleyse, çeviri sayfa tablosuna (tüm çevirileri içeren) başvurmak zorunda kalmadan (hızlıca) gerçekleştirilir. Muazzam performans etkilerinin nedeniyle, TLB'lerin gerçek anlamda sanal belleği mümkün kılmasıdır [C95].

```
1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)    // TLB Hit
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 Miss
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()
```

Şekil 19.1: **TLB Kontrol Akış Algoritması**

19.1 Temel TLB Algoritması

Şekil 19.1, basit bir **lineer sayfa tablosu (linear page table)** (yani, sayfa tablosu bir dizidir) ve **donanım tarafından yönetilen bir TLB (hardware-managed TLB)** (yani, donanım, sayfa tablosu erişimlerinin sorumluluğunun çoğunu üstlenir; aşağıda bu konuda daha fazla bilgi vereceğiz) varsayılarak donanımın sanal bir adres çevirisini nasıl işleyebileceğine dair kaba bir taslak göstermektedir.

Donanımın izlediği algoritma şu şekilde çalışır: ilk olarak, sanal adresteki sanal sayfa numarasını (VPN) ayıklayın (Şekil 19.1'deki Satır 1) ve TLB'nin bu VPN için çeviriye sahip olup olmadığını kontrol edin (Satır 2). Eğer öyleyse, bir **TLB isabetimiz** (**TLB hit**) var, bu da TLB'nin çeviriyi elinde tuttuğu anlamına geliyor. Başarı! Artık sayfa çerçevesi numarasını (PFN) ilgili TLB girişinden ayıklayabilir, bunu orijinal sanal adresten ofset üzerine birleştirebilir ve istenen fiziksel adresi (PA) oluşturabilir ve belleğe erişebiliriz. (Satır 5-7), koruma kontrollerinin başarısız olmadığını varsayarsak (Satır 4).

CPU çeviriyi TLB'de bulamazsa (**TLB ıska (TLB miss)**), yapacak daha çok işimiz var. Bu örnekte, donanım çeviriyi bulmak için sayfa tablosuna erişir (Satır 11–12) ve süreç tarafından oluşturulan sanal bellek başvurusunun geçerli ve erişilebilir olduğunu varsayarak (Satır 13, 15), TLB'yi çeviriyle güncelleştirir (Satır 18). Bu eylemler kümesi, öncelikle sayfa tablosuna (Satır 12) erişmek için gereken ek bellek başvurusu nedeniyle maliyetlidir. Son olarak, TLB güncellendikten sonra, donanım talimatı yeniden dener; bu kez, çeviri TLB'de bulunur ve bellek başvurusu hızlı bir şekilde işlenir.

TLB, tüm önbellekler gibi, yaygın durumda, çevirilerin önbellekte bulunduğu (yani, isabetlerdir) öncülü üzerine inşa edilmiştir. Eğer öyleyse, TLB işlem çekirdeğinin yakınında bulunduğundan ve oldukça hızlı olması için tasarlandığından çok az ek yük eklenir. Bir kaçırma meydana geldiğinde, yüksek sayfalama maliyeti ortaya çıkar; çeviriyi bulmak için sayfa tablosuna ve fazladan bir bellek başvurusu (veya daha karmaşık sayfa tablolarıyla daha fazla) sonuçlarına erişilmelidir. Bu sık sık gerçekleşirse, program muhtemelen belirgin şekilde daha yavaş çalışacaktır; Çoğu CPU talimatına göre bellek erişimleri oldukça maliyetlidir ve TLB kaçırmaları daha fazla bellek erişimine yol açar. Bu nedenle, TLB'nin elimizden geldiğince kaçırılmasını önlemek umudumuzdur.

19.2 Örnek: Bir Diziye Erişim

Bir TLB'nin çalışmasını netleştirmek için, basit bir sanal adres izlemesini inceleyelim ve bir TLB'nin performansını nasıl artırabileceğini görelim. Bu örnekte, bellekte sanal adres 100'den başlayarak 10 adet 4 baytlık tam sayıdan oluşan bir dizimiz olduğunu varsayalım. 16 baytlık sayfalara sahip küçük bir 8 bit sanal adres alanımız olduğunu varsayalım; Böylece, sanal bir adres 4 bitlik bir VPN'ye (16 sanal sayfa vardır) ve 4 bitlik bir ofsete (bu sayfaların her birinde 16 bayt vardır) ayrılır.

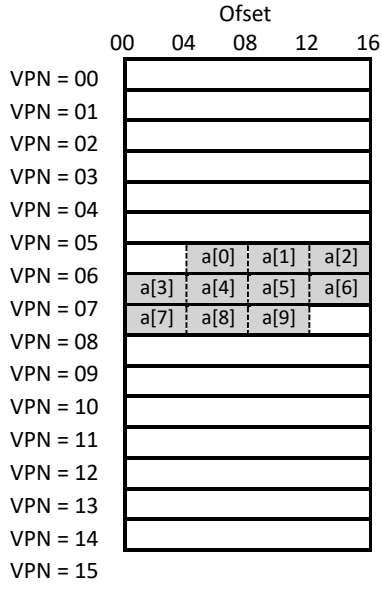
Şekil 19. 2 (sayfa 4), sistemin 16 adet 16 baytlık sayfasında belirtilen diziyi gösterir. Gördüğünüz üzere, dizinin ilk girişi (a[0]) şu şekilde başlar (VPN=06, offset=04); bu sayfaya yalnızca üç adet 4 baytlık tamsayı sığar. Dizi, sonraki dört girişin (a[3] ... a[6]) bulunduğu sonraki sayfaya devam eder. Son olarak, 10 girişli dizinin son üç girişi (a[7] ... a[9]) adres alanının bir sonraki sayfasında bulunur (VPN=08).

Şimdi, her dizi öğesine erişen basit bir döngü düşünelim, C'de şöyle görünecek bir şey:

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

Basitlik uğruna, döngünün ürettiği tek bellek erişiminin diziye ait olduğunu iddia edeceğiz (i ve sum değişkenlerini ve talimatların göz ardı ederek). İlk dizi öğesine (a[0]) erişildiğinde, CPU sanal adres 100'de bir yük görür. Donanım VPN'i bundan çıkarır (VPN=06) ve TLB'yi geçerli bir çeviri olup olmadığını kontrol etmek için kullanır. Bunun programın diziye ilk kez eriştiğini varsayarsak, sonuç bir TLB ıskası olacaktır.

Bir sonraki erişim a [1]' e ve burada bazı iyi haberler var: bir TLB isabeti! Dizinin ikinci öğesi birincisinin yanında paketlendiğinden, aynı sayfada yaşar; çünkü ilk sayfaya erişirken bu sayfaya zaten eriştik, dizinin öğesi, çeviri zaten yüklü.



Şekil 19.2: **Örnek: Küçük Bir Adres Alanındaki Bir Dizi**

TLB'ye. Ve dolayısıyla başarımızın nedeni. a[2]'ye erişim , benzer bir başarıya (başka bir isabet) karşılık gelir, çünkü o da a[0] ve a[1] ile aynı sayfada yaşar.

Ne yazık ki, program a[3] 'e eriştiğinde, başka bir TLB ıskası ile karşılaşıyoruz. Ancak, bir kez daha, sonraki girişler (a[4] ... a[6]) hepsi bellekte aynı sayfada bulunduğundan TLB'ye isabet eder .

Son olarak, a[7]' ye erişim son bir TLB kaybına neden olur. Donanım, bu sanal sayfanın fiziksel bellekteki konumunu bulmak için bir kez daha sayfa tablosuna başvurur ve TLB'yi buna göre günceller. Son iki erişim (a[8] ve a[9]) bu TLB güncellemesinin avantajlarından yararlanır; donanım çevirileri için TLB'ye baktığında, iki isabet daha elde edilir.

Diziye on erişimimiz sırasında TLB etkinliğini özetleyelim: **ıska (miss)**, vurdu, vurdu, **kaçırdı (miss)**, vurdu, vurdu, vurdu, **kaçırdı (miss)**, vurdu, vurdu. Böylece, isabet sayısının toplam erişim sayısına bölünmesiyle TLB **isabet oranımız (hit rate)** %70'tir. Bu çok yüksek olmasa da (fiilen %100 yaklaşan isabet oranları arzu ediyoruz), sıfır değil, bu bir sürpriz olabilir. Program diziye ilk kez erişiyor olsa da TLB **coğrafi yerellik (spatial locality)** nedeniyle performansı artırır. Dizinin öğeleri sayfalara sıkıca paketlenir (yani, **boşlukta (space)** birbirlerine yakındırlar) ve bu nedenle yalnızca sayfadaki bir öğeye ilk erişim TLB ıskası verir.

Ayrıca, sayfa boyutunun bu örnekte oynadığı role de dikkat edin.

İpucu: Mümkün Olduğunda Ön belleklemeyi Kullan

Ön bellekleme, bilgisayar sistemlerindeki en temel performans tekniklerinden biridir ve "ortak durumu hızlı" hale getirmek için tekrar tekrar kullanılır [HP06]. Donanım önbelleklerinin arkasındaki fikir, talimat ve veri başvurularındaki **locality** (**yöresellikten)** yararlanmaktır. Genellikle iki tür yerellik vardır: **temporal locality (zamanda yöresellik)** ve **coğrafi yöresellik (spatial locality)**. Zamansal yerellik ile ilgili fikir, yakın zamanda erişilen bir talimat veya veri öğesinin gelecekte yakında yeniden erişileceğidir. Bir döngüdeki döngü değişkenlerini veya yapılandırmalarını düşünün; zaman içinde tekrar tekrar erişilirler. Uzamsal yerellik ile ilgili fikir, bir program x adresindeki belleğe erişirse, x'in yakınındaki belleğe erişeceğidir. Burada bir tür diziden akış yaptığınızı, bir öğeye ve ardından bir sonrakine eriştiğinizi hayal edin. Tabii ki, bu özellikler programın tam doğasına bağlıdır ve bu nedenle zor ve hızlı yasalar değil, daha çok temel kurallar gibidir.

Yönergeler, veriler veya adres çevirileri için (TLB'mizde olduğu gibi) donanım önbellekleri, belleğin kopyalarını küçük, hızlı yonga üstü bellekte tutarak yerellikten yararlanır. Bir isteği yerine getirmek için (yavaş) bir belleğe gitmek zorunda kalmak yerine, işlemci önce yakındaki bir kopyanın önbellekte olup olmadığını kontrol edebilir; eğer varsa, işlemci hızlı bir şekilde erişebilir (yani, birkaç CPU döngüsünde) ve belleğe erişmek için gereken maliyetli zamanı harcamaktan kaçınabilir (birçok nanosaniye).

Merak ediyor olabilirsiniz: eğer önbellekler (TLB gibi) bu kadar büyükse, neden daha büyük önbellekler yapmıyoruz ve tüm verilerimizi içlerinde tutmuyoruz? Ne yazık ki, burası fiziğinki gibi daha temel yasalarla karşılaştığımız yerdir. Hızlı bir önbellek istiyorsanız, ışık hızı ve diğer fiziksel koşullar gibi konular alakalı hale geldiğinden küçük olması gerekir. Tanımı gereği herhangi bir büyük önbellek yavaştır ve bu nedenle amacı yener. Bu nedenle, küçük, hızlı önbelleklerle sıkışıp kaldık; Kalan soru, performansı artırmak için bunları en iyi şekilde nasıl kullanacağımızdır.

sadece iki kat daha büyük olsaydı (16 değil, 32 bayt), dizi erişimi daha az ıskadan mustarip olurdu. Tipik sayfa boyutları daha çok 4 KB olduğundan, bu tür yoğun, dizi tabanlı erişimler mükemmel TLB performansı elde eder ve erişim sayfası başına yalnızca tek bir kaçırma ile karşılaşır.

TLB performansıyla ilgili son bir nokta: Program, bu döngü tamamlandıktan kısa bir süre sonra diziye tekrar erişirse, gerekli olanı önbelleğe almak için yeterince büyük bir TLB'ye sahip olduğumuzu varsayarsak, muhtemelen daha da iyi bir sonuç görürüz. Çeviriler: hit, hit, hit, hit, hit, hit, hit, hit, hit, hit. Bu durumda, TLB isabet oranı, **zamansal yöresellik (temporal locality)**, yani bellek öğelerinin **zaman (time)** içinde hızlı bir şekilde yeniden referans alınması nedeniyle yüksek olacaktır. Herhangi bir önbellek gibi, TLB'ler de başarı için program uygunluğu olan hem coğrafi hem de zamansal yöreselliğe güvenir. İlgilenilen program böyle bir yerellik sergiliyorsa (ve birçok program bunu yapıyorsa), TLB isabet oranı muhtemelen yüksek olacaktır.

```
1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)    // TLB Hit
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 Miss
11     RaiseException(TLB_MISS)
```

Şekil 19.3: **TLB Kontrol Akış Algoritması (İşletim Sistemi İşlenen)**

19.1 TLB Iskasınıi Kim Yönetir?

Cevaplamamız gereken bir soru: TLB ıskasını kim idare eder? İki cevap mümkündür: donanım veya yazılım (OS). Eski günlerde, donanımın karmaşık komut kümeleri vardı (bazen karmaşık komut kümesi bilgisayarları için **CISC** olarak adlandırılır) ve donanım yazılımını oluşturan insanlar bu sinsi işletim sistemi insanlarına pek güvenmiyordu. Böylece, donanım TLB ıskası tamamen ele alacaktı. Bunu yapmak için, donanımın sayfa tablolarının hafızada tam olarak *nerede* bulunduğunu (Şekil 19.1'deki Satır 11'de kullanılan bir **sayfa tablosu temel kaydı (page-table base register)** aracılığıyla) ve *bunların tam biçimlerini* bilmesi gerekir; bir ıskada, donanım sayfa tablosunu "yürüyor", uygun sayfa tablosu girişini bulacak ve istenen çeviriyi ayıklayacak, TLB'yi çeviriyle güncelleyecek ve talimatı yeniden deneyecektir. **Donanım tarafından yönetilen TLB'lere (Hardware-managed TLBs)** sahip "eski" mimariye bir örnek, sabit bir **çok düzeyli sayfa tablosu (multi-level page table)** kullanan Intel x86 mimarisidir (ayrıntılar için bir sonraki bölüme bakın); geçerli sayfa tablosu CR3 kaydı [I09] tarafından işaret edilmektedir.

Daha modern mimariler (örneğin, MIPS R10k [H93] veya Sun'ınSPARC v9 [WG00], hem **RISC** hem de azaltılmış komut kümesi bilgisayarları), **yazılım tarafından yönetilen (TLB software-managed TLB)** olarak bilinen şeye sahiptir. TLB'yi kaçırdığında, donanım basitleştirmesi geçerli yönerge akışını duraklatan, ayrıcalık düzeyini çekirdeğe yükselten bir özel durum (Şekil 19.3'teki satır 11) oluşturur moduna geçer ve bir **tuzak işleyicisine (trap handler)** atlar. Tahmin edebileceğiniz gibi, bu tuzak işleyicisi, TLB ıskalarını işlemek amacıyla yazılan işletim sistemi içindeki koddur. Çalıştırıldığında, kod sayfa tablosundaki çeviriyi arar, TLB'yi güncellemek için özel "ayrıcalıklı" talimatları kullanır ve tuzaktan döner; bu noktada, donanım talimatı yeniden dener (TLB isabetiyle sonuçlanır). Birkaç önemli ayrıntıyı tartışalım. İlk olarak, tuzaktan dönüş talimatının, bir sistem çağrısına hizmet verirken daha önce gördüğümüz tuzaktan dönüş talimatından biraz farklı olması gerekir. İkinci durumda, tuzaktan dönüş, tıpkı bir tuzaktan geri dönüş gibi, işletim sistemine tuzaktan *sonraki* talimatta yürütmeye devam etmelidir. Prosedür çağrısı, prosedüre yapılan çağrıyı takiben hemen talimatına geri döner. İlk durumda, bir TLB yanlış kullanım tuzağından dönerken, donanımın tuzağa *neden* olan talimatla uygulamaya devam etmesi gerekir; bu yeniden deneme, böylece

ATARAFI: RISC VS. CISC

1980'lerde, bilgisayar mimarisi topluluğunda büyük bir savaş gerçekleşti. Bir tarafta **Karmaşık Komut Kümesi Hesaplama (Complex Instruction Set Computing)** anlamına gelen **CISC** kampı; diğer tarafta **Azaltılmış Komut Kümesi Hesaplama (Reduced Instruction Set Computing)** [PS81] için **RISC** vardı. RISC tarafı, Berkeley'deki David Patterson ve Stanford'daki John Hennessy (aynı zamanda bazı ünlü kitapların [HP06] ortak yazarları) tarafından öncülük edildi, ancak daha sonra John Cocke, RISC [CM00] üzerindeki en erken çalışması için Turing ödülü ile tanındı.

CISC komut setleri içlerinde çok fazla talimat bulundurma eğilimindedir ve her talimat nispeten güçlüdür. Örneğin, iki işaretçi ve bir uzunluk alan ve baytları kaynaktan hedefe kopyalayan bir dize kopyası görebilirsiniz. CISC'nin arkasındaki fikir, donanım dilinin kendisinin kullanımını kolaylaştırmak ve kodu daha kompakt hale getirmek için talimatların üst düzey ilkeller olması gerektiğiydi.

RISC talimat kümeleri bunun tam tersidir. RISC'nin arkasındaki önemli bir gözlem, komut kümelerinin gerçekten derleyici hedefleri olduğu ve tüm derleyicilerin gerçekten istedikleri, yüksek performanslı kod üretmek için kullanabilecekleri birkaç basit ilkel olmasıdır. Bu nedenle, RISC savunucuları, donanımdan (özellikle mikro kod) mümkün olduğunca çok şey koparalım ve kalanları basit, tek tip ve hızlı hale getirelim.

İlk günlerde, RISC çipleri gözle görülür derecede daha hızlı oldukları için büyük bir etki yarattı [BC91]; birçok makale yazıldı, birkaç şirket kuruldu (örneğin, MIPS ve Sun). Bununla birlikte, zaman ilerledikçe, Intel gibi CISC üreticileri, işlemcilerinin çekirdeğine birçok RISC tekniğini dahil ettiler, örneğin karmaşık talimatları mikro talimatlara dönüştüren erken boru hattı aşamaları ekleyerek daha sonra RISC benzeri bir şekilde işlenebilir. Bu yenilikler ve her çipte giderek artan sayıda transistör, CISC'nin rekabetçi kalmasını sağladı. Sonuç olarak, tartışma sona erdi ve bugün her iki işlemci türü de hızlı çalışacak şekilde yapılabilir.

talimatları tekrar çalıştırır, bu sefer bir TLB isabetiyle sonuçlanır. Bu nedenle, bir tuzağın veya özel durumun nasıl oluştuğuna bağlı olarak, donanımın, zamanı geldiğinde düzgün bir şekilde devam etmek için işletim sistemine ping yaparken farklı bir PC kaydetmesi gerekir.

İkincisi, TLB yanlış işleme kodunu çalıştırırken, işletim sisteminin sonsuz bir TLB kaçırma zincirinin oluşmasına neden olmamak için ekstra dikkatli olması gerekir. Birçok çözüm mevcuttur; örneğin, TLB eksik işleyicilerini fiziksel bellekte tutabilirsiniz (**eşlenmemiş (unmapped)** oldukları ve adres çevirisine tabi olmadıkları yerlerde) veya TLB'deki bazı girişleri kalıcı olarak geçerli çeviriler için ayırabilirsiniz, işleyici kodunun kendisi için bu kalıcı çeviri yuvalarından bazılarını kullanın; bu **kablolu (wired)** çeviriler her zaman TLB'de isabet eder.

Yazılım tarafından yönetilen yaklaşımın birincil avantajı *esnekliktir*: işletim sistemi, sayfayı uygulamak istediği herhangi bir veri yapısını kullanabilir.

Bir Tarafa: TLB VALID BIT /= PAGE TABLE VALID BIT

Yaygın bir hata, bir TLB'de bulunan geçerli bitleri bir sayfa tablosunda bulunanlarla karıştırmaktır. Sayfa tablosunda, bir sayfa tablosu girişi (PTE) geçersiz olarak işaretlendiğinde, bu, sayfanın işlem tarafından ayrılmadığı ve doğru çalışan bir program tarafından erişilmemesi gerektiği anlamına gelir. Geçersiz bir sayfaya erişildiğinde verilen olağan yanıt, işlemi öldürerek yanıt verecek olan işletim sistemine tuzak kurmaktır.

TLB geçerli biti, aksine, bir TLB girişinin içinde geçerli bir çeviri olup olmadığını ifade eder. Örneğin, bir sistem ön yüklendiğinde, her TLB girişi için ortak bir başlangıç durumu geçersiz olarak ayarlanmalıdır , çünkü burada henüz hiçbir adres çevirisi önbelleğe alınmaz. Sanal bellek etkinleştirildikten ve programlar çalışmaya ve sanal adres alanlarına erişmeye başladığında, TLB yavaşça doldurulur ve böylece geçerli girişler kısa sürede TLB'yi doldurur.

TLB geçerli biti, aşağıda daha ayrıntılı olarak tartışacağımız gibi, bir bağlam anahtarı oluştururken de oldukça kullanışlıdır. Tüm TLB girişlerini geçersiz olarak ayarlayarak, sistem çalıştırılmak üzere olan işlemin yanlışlıkla önceki bir işlemden sanaldan fiziksele çeviri kullanmamasını sağlayabilir.

Tablo, donanım değişikliği gerektirmeden. Diğer bir avantaj, TLB kontrol akışında görüldüğü gibi *basitliktir* (Şekil 19.1'deki satır 11, Şekil 19.1'deki 11-19 satırlarının aksine). Donanım çok fazla bir şey yapmaz: sadece bir istisna oluşturun ve işletim sistemi TLB ıska işleyicisinin gerisini yapmasına izin verin.

19.2 TLB İçeriği: İçinde Neler Var?

Donanım TLB'sinin içeriğine daha ayrıntılı olarak bakalım. Tipik bir TLB'nin 32, 64 veya 128 girişi olabilir ve olarak adlandırılan şey olabilir**.** Temel olarak, bu sadece belirli bir çevirinin TLB'nin herhangi bir yerinde olabileceği ve donanımın istenen çeviriyi bulmak için tüm TLB'yi paralel olarak arayacağı anlamına gelir. TLB girişi şöyle görünebilir:

VPN PFN diğer bitler

Hem VPN hem de PFN'nin her girişte bulunduğunu unutmayın, çünkü bir çeviri bu konumlardan herhangi birinde sona erebilir (donanım açısından, TLB **tamamen ilişkisel (fully associated)** önbellek olarak bilinir). Donanım, eşleşme olup olmadığını görmek için girişleri paralel olarak arar.

Daha ilginç olan "diğer bitler" dir. Örneğin, TLB'nin genellikle girişin geçerli bir çevirisi olup olmadığını belirten **geçerli (valid)** bir biti vardır. Ayrıca, bir sayfaya nasıl erişilebileceğini belirleyen **koruma (protection)** bitleri de yaygındır (sayfa tablosunda olduğu gibi). Örneğin, kod sayfaları okundu ve yürütüldü olarak işaretlenebilirken, yığın sayfaları *okundu ve yazıldı* olarak işaretlenebilir. Bir **Adres alanı tanımlayıcısı (Adress-space identifier)**, **kirli bit (dirty bit)** vb. dahil olmak üzere birkaç alan daha olabilir; Daha fazla bilgi için aşağıya bakın.

19.3 TLB Sorunu: Bağlam Anahtarları

TLB'lerde, süreçler arasında geçiş yaparken (ve dolayısıyla adres alanları) bazı yeni sorunlar ortaya çıkar. Özellikle, TLB yalnızca çalışmakta olan işlem için geçerli olan sanaldan fiziksele çeviriler içerir; bu çeviriler diğer süreçler için anlamlı değildir. Sonuç olarak, bir işlemden diğerine geçerken, donanım veya işletim sistemi (veya her ikisi) çalıştırılmak üzere olan işlemin, daha önce çalıştırılan bazı işlemlerden çevirileri yanlışlıkla kullanmadığından emin olun.

Bu durumu daha iyi anlamak için bir örneğe bakalım. Bir işlem (P1) çalışırken, TLB'nin kendisi için geçerli olan, yani P1'in sayfa tablosundan gelen çevirileri önbelleğe alıyor olabileceğini varsayar. Bu örnekte, P1'in 10. sanal sayfasının fiziksel çerçeve 100 ile eşlendiğini varsayalım. Bu örnekte, başka bir işlemin (P2) var olduğunu ve işletim sisteminin yakında bir bağlam anahtarı gerçekleştirmeye ve çalıştırmaya karar verebileceğini varsayalım. Burada P2'nin 10. sanal sayfasının fiziksel çerçeve 170 ile eşlendiğini varsayalım. Eğer girişler ise her iki süreç de TLB'deydi, TLB'nin içeriği şöyle olacaktı:

VPN	PFN	valid	Prot
10	100	1	rwx
—	—	0	—
10	170	1	rwx
—	—	0	—

Yukarıdaki TLB'de açıkça bir sorunumuz var: VPN 10, PFN 100 (P1) veya PFN 170 (P2) anlamına gelir, ancak donanım hangi girişin hangi işlem için olduğunu ayırt edemez. Bu nedenle, TLB'nin birden fazla süreçte sanallaştırmayı doğru ve etkili bir şekilde desteklemesi için biraz daha fazla çalışma yapmamız gerekiyor. Ve böylece, bir püf nokta:

Püf Nokta
BİR İÇERİK ANAHTARINDA TLB İÇERİĞİ NASIL YÖNETİLİR?

İşlemler arasında bağlam geçişi yaparken, TLB'deki son işlem için yapılan çeviriler, çalıştırılmak üzere olan işlem için anlamlı değildir. Bu sorunu çözmek için donanım veya işletim sistemi ne yapmalı?

Sorunun bir dizi olası çözümü vardır. Bir yaklaşım, TLB'yi bağlam anahtarlarında **yıkamak (flush)** ve böylece bir sonraki işlemi çalıştırmadan önce boşaltmaktır. Yazılım tabanlı bir sistemde, bu açık (ve ayrıcalıklı) bir donanım eğitimi ile gerçekleştirilebilir; donanım tarafından yönetilen bir TLB ile, sayfa tablosu temel kaydı değiştirildiğinde yıkama işlemi gerçekleştirilebilir (işletim sisteminin PTBR'yi bir bağlam anahtarında değiştirmesi gerektiğini unutmayın). Her iki durumda da yıkama işlemi tüm geçerli bitleri 0 olarak ayarlar ve esasen TLB'nin içeriğini temizler.

Her bağlam anahtarındaki TLB'yi temizlediğinden, artık çalışan bir çözümümüz var, çünkü bir süreç asla yanlışlıkla yanlış TLB'de yanlış çeviriyle karşılaşmaz.

Ancak, bir maliyet vardır: Bir işlem her çalıştığında, verilerine ve kod sayfalarına dokunduğunda TLB'nin kaçırdığı durumlara maruz kalmalıdır. İşletim sistemi işlemler arasında sık sık geçiş yapıyorsa, bu maliyet yüksek olabilir.

Bu ek yükü azaltmak için, bazı sistemler TLB'nin bağlam anahtarları arasında paylaşılabilmesi için donanım desteği ekler. Özellikle, bazı donanım sistemleri TLB'de bir **adres alanı tanımlayıcısı (adress-space identifier (ASID))** alanı sağlar. ASID'yi bir **işlem tanımlayıcısı (process identifier)** (PID) olarak düşünebilirsiniz, ancak genellikle daha az biti vardır (örneğin, ASID için 8 bit ve PID için 32 bit).

TLB örneğimizi yukarıdan alırsak ve ASID'ler eklersek, süreçlerin TLB'yi kolayca paylaşabileceği açıktır: aksi takdirde aynı çevirileri dağıtmak için yalnızca ASID alanına ihtiyaç vardır. İşte eklenen ASID alanına sahip bir TLB'nin tasviri:

VPN	PFN	valid	Prot	ASID
10	100	1	rwx	1
—	—	0	—	—
10	170	1	rwx	2
—	—	0	—	—

Böylece, adres alanı tanımlayıcıları ile TLB, farklı işlemlerden çevirileri aynı anda herhangi bir karışıklık olmadan tutabilir. Tabii ki, donanımın çevirileri gerçekleştirmek için şu anda hangi işlemin çalıştığını da bilmesi gerekir ve bu nedenle işletim sistemi, bir bağlam anahtarında, bazılarını geçerli işlemin ASID'sine ayrıcalıklı kayıt ayarlamalıdır.

Bir kenara bırakırsak, TLB'nin iki girişinin oldukça benzer olduğu başka bir durum düşünmüş olabilirsiniz. Bu örnekte, *aynı* fiziksel sayfaya işaret eden iki farklı VPN'ye sahip iki farklı işlem için iki giriş vardır:

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

Bu durum, örneğin, iki işlem bir sayfayı (örneğin bir kod sayfası) *paylaştığında* ortaya çıkabilir. Yukarıdaki örnekte, Süreç 1, fiziksel sayfa 101'i Süreç 2 ile birleştiriyor; P1 bu sayfayı adres alanının 10. sayfasına eşlerken, P2 bu sayfayı adres alanının 50. sayfasına eşler. Kod sayfalarının paylaşılması (ikili dosyalarda veya paylaşılan kitaplıklarda), kullanımdaki fiziksel sayfa sayısını azalttığı ve böylece bellek ek yüklerini azalttığı için yararlıdır.

19.4 Sorun: Değiştirme Politikası

Herhangi bir önbellekte ve dolayısıyla TLB'de olduğu gibi, göz önünde bulundurmamız gereken bir sorun ise **cache replacement (önbellek algoritmalarıdır)**. Özellikle TLB'ye yeni bir giriş yüklerken, eski bir girişle **değiştirmeliyiz (replace)**, ve böylece şöyle bir soru ortaya çıkar hangisi değiştirilmelidir?

Püf Nokta: TLB Algoritması Nasıl Tasarlanır

Yeni bir TLB girişi eklediğimizde hangi TLB girişi değiştirilmelidir? Amaç, elbette, **kaçırma oranını (miss rate)** en aza indirmek (veya **isabet oranını artırmak (hit rate)**) ve böylece performansı artırmaktır.

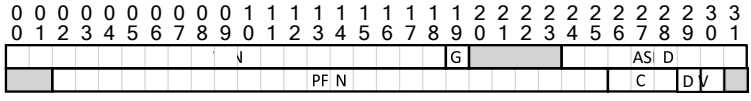
Sayfaları diske geçirme sorununu ele alırken bu tür politikaları biraz ayrıntılı olarak inceleyeceğiz; Burada sadece birkaç tipik politikayı vurgulayacağız. Yaygın bir yaklaşım, **en az kullanılan (least-recently-used (LRU))** girişini çıkarmaktır. LRU, yakın zamanda kullanılmamış bir girdinin yükseltme için iyi bir aday olma ihtimalinin yüksek olduğunu varsayarak bellek başvuru akışındaki yerellikten yararlanmaya çalışır. Bir başka tipik yaklaşım, bir TLB'yi rastgele bir TLB eşlemesini tahliye eden **rastgele (random)** bir politika kullanmaktır. Böyle bir politika, basitliği ve köşe durumu davranışlarından kaçınma yeteneği nedeniyle yararlıdır; Örneğin, LRU gibi "makul" bir politika, bir program n boyutunda bir TLB ile *n* + 1 sayfa üzerinde döngü yaptığında oldukça mantıksız davranır; Bu durumda, LRU her erişimi kaçırırken, rastgele çok daha iyisini yapar.

19.5 Gerçek Bir TLB Girişi

Son olarak, gerçek bir TLB'ye kısaca bakalım. Bu örnek, yazılım tarafından yönetilen TLB'leri kullanan modern bir sistem olan MIPS R4000 [H93]'ten; biraz basitleştirilmiş bir MIPS TLB girişi Şekil 19.4'te görülebilir.

MIPS R4000, 4 KB sayfalarla 32 bit adres alanını destekler. Bu nedenle, tipik sanal adresimizde 20 bit VPN ve 12 bit ofset bekleriz. Ancak, TLB'de görebileceğiniz gibi, VPN için sadece 19 bit vardır; Görünüşe göre, kullanıcı adresleri yalnızca adres alanının yarısından (geri kalanı çekirdek için ayrılmış) gelecek ve bu nedenle yalnızca 19 bit VPN gerekecektir. VPN, 24 bit'e kadar fiziksel kare numarasına (PFN) dönüşür ve bu nedenle 64 GB'a kadar (fiziksel) ana değere sahip sistemleri bellek destekleyebilir (2 adet 4 KB sayfa).

MIPS TLB'de birkaç ilginç bit daha vardır. Küresel bir dünya görüyoruz Bit (G), işlemler arasında küresel olarak paylaşılan sayfalar için kullanılır. Bu nedenle, global bit ayarlanırsa, ASID yok sayılır. Ayrıca, işletim sisteminin adres alanları arasında ayrım yapmak için kullanabileceği 8 bit *ASID'yi* de görüyoruz



Şekil 19.4: **A MIPS TLB Girişi**

İpucu: RAM her zaman RAM değildir (Culler Kanunu)

Rastgele bellek erişimli (random-access memory) veya RAM terimi, **RAM'in** herhangi bir bölümüne bir başkası kadar hızlı erişebileceğiniz anlamına gelir. RAM'i bu şekilde düşünmek genellikle iyi olsa daTLB gibi donanım / işletim sistemi özellikleri nedeniyle, belirli bir bellek sayfasına erişmek, özellikle de bu sayfa şu anda TLB'niz tarafından eşlenmiyorsa, maliyetli olabilir. Bu nedenle, uygulama ipucunu hatırlamak her zaman iyidir: RAM **her zaman RAM değildir**. Bazen adres alanınıza rastgele erişmek, erişilen sayfa sayısı TLB kapsamını aşarsa Partiküler olarak, ciddi perfor-mance cezalarına yol açabilir. Danışmanlarımızdan biri olan David Culler, TLB'yi birçok performans sorununun kaynağı olarak gösterme yolları kullandığından, bu yasayı onun onuruna adlandırıyoruz : **Culler Kanunu (Culler's Law)**.

(Yukarıda açıklanmıştır). Sizin için bir soru: işletim sistemi varsa ne yapmalı 256'dan fazla (2) süreçler aynı anda mı çalışıyor? Son olarak, 3 *Yapışma* (C) bitler, hangi a sayfası donanım tarafından önbelleğe alındı (bu notların kapsamının biraz ötesinde); sayfa yazıldığında işaretlenmiş *kirli* bir bit (bunun kullanımını daha sonra göreceğiz); *geçerli* bir bit bu, girişte geçerli bir çeviri olup olmadığını donanıma bildirir. Birden fazla sayfa boyutunu destekleyen bir *sayfa maskesi* alanı da vardır (gösterilmez); Daha büyük sayfalara sahip olmanın neden yararlı olabileceğini daha sonra göreceğiz. Son olarak, 64 bitin bazıları kullanılmaz (diyagramda gölgeli gri).

MIPS TLB'leri genellikle bu girişlerin 32 veya 64'üne sahiptir ve bunların çoğu kullanıcı işlemleri tarafından çalışırken kullanılır. Ancak, birkaçı işletim sistemi için ayrılmıştır. Donanıma işletim sistemi için kaç TLB yuvası ayıracağını söylemek için işletim sistemi tarafından *kablolu* bir kayıt ayarlanabilir; İşletim sistemi, TLB kaçırmanın sorunlu olacağı kritik zamanlarda erişmek istediği kod ve veriler için bu ayrılmış eşlemeleri kullanır (örneğin, TLB ıska işleyicisi).

MIPS TLB yazılım tarafından yönetildiğinden, TLB'yi güncellemek için girişimlerde bulunulması gerekir. MIPS bu tür dört talimat sağlar: TLBP, belirli bir çevirinin orada olup olmadığını görmek için TLB'yi araştırır; bir TLB girişinin içeriğini kayıtlara okuyan TLBR; Belirli bir TLB girişini yeniden yerleştiren TLBWI ve rastgele bir TLB girişinin yerini alan TLBWR. İşletim sistemi, TLB'nin içeriğini yönetmek için bu talimatları kullanır. Bu talimatların **ayrıcalıklı (priivileged)** olması elbette çok önemlidir; TLB'nin içeriğini değiştirebilseydi bir kullanıcı işleminin neler yapabileceğini hayal edin (ipucu: sadece makineyi ele geçirmek, kendi kötü amaçlı "işletim sistemini" çalıştırmak ve hatta Güneş'i ortadan kaldırmak da dahil olmak üzere herhangi bir şey hakkında).

19.6 Özet

Donanımın adres çevirisini daha hızlı yapmamıza nasıl yardımcı olabileceğini gördük. Adres çevirisi önbelleği olarak küçük, özel bir yonga üstü TLB sağlayarak, çoğu bellek referansının sahip *olmadan* ele alınacağını umuyoruz ana bellekteki sayfa tablosuna erişmek için. Böylece, ortak durumda,

programın performansı neredeyse bellek hiç sanallaştırılmıyormuş gibi olacak, bir işletim sistemi için mükemmel bir başarı ve modern sistemlerde disk belleği kullanımı için kesinlikle gerekli olacak .

Ancak, TLB'ler var olan her program için dünyayı pembe yapmaz. Özellikle, bir programın kısa sürede eriştiği sayfa sayısı, TLB'ye uyan sayfa sayısını aşarsa, program çok sayıda TLB kaçırma üretecek ve bu nedenle biraz daha yavaş çalışacaktır. Bu fenomeni **TLB kapsamını (TLB coverage)** aşmak olarak adlandırıyoruz ve bazı programlar için oldukça sorun olabilir. Bir sonraki bölümde tartışacağımız gibi, bir çözüm, daha büyük sayfa boyutları için destek eklemektir; Temel veri yapılarını, programın adres alanının daha büyük sayfalarla eşlenen bölgelerine eşleyerek, TLB'nin etkili kapsamı artırılabilir. Büyük sayfalar için destek genellikle hem büyük hem de rastgele erişilen belirli veri yapılarına sahip bir **veritabanı yönetim sistemi (database management system (DBMS))** gibi programlar tarafından kullanılır.

Bahsetmeye değer başka bir TLB sorunu: TLB erişimi, CPU boru hattında, özellikle de **fiziksel olarak dizinlenmiş önbellek (physically-indexed cache)** olarak adlandırılan bir darboğaz haline gelebilir. Böyle bir önbellekte, önbelleğe erişilmeden *önce* adres çevirisinin yapılması gerekir, bu da işleri biraz yavaşlatabilir. Bu potansiyel sorun nedeniyle, insanlar *sanal* adreslere sahip önbelleklere erişmenin her türlü akıllı yolunu aradılar, böylece önbellek isabeti durumunda pahalı çeviri adımından kaçındılar. Böyle **sanal olarak dizinlenmiş bir önbellek (virtually-indexed cache)**, bazı performans sorunlarını çözer, ancak donanım tasarımına da yeni sorunlar getirir. Daha fazla ayrıntı için Wiggins'in iyi anketine bakın [W03].

Kaynakça

[BC91] "Performance from Architecture: Comparing a RISC and a CISC with Similar Hard- ware Organization" by D. Bhandarkar and Douglas W. Clark. Communications of the ACM, September 1991. *A great and fair comparison between RISC and CISC. The bottom line: on similar hardware, RISC was about a factor of three better in performance.*

[CM00] "The evolution of RISC technology at IBM" by John Cocke, V. Markstein. IBM Journal of Research and Development, 44:1/2. *A summary of the ideas and work behind the IBM 801, which many consider the first true RISC microprocessor.*

[C95] "The Core of the Black Canyon Computer Corporation" by John Couleur. IEEE Annals of History of Computing, 17:4, 1995. *In this fascinating historical note, Couleur talks about how he invented the TLB in 1964 while working for GE, and the fortuitous collaboration that thus ensued with the Project MAC folks at MIT.*

[CG68] "Shared-access Data Processing System" by John F. Couleur, Edward L. Glaser. Patent 3412382, November 1968. *The patent that contains the idea for an associative memory to store address translations. The idea, according to Couleur, came in 1964.*

[CP78] "The architecture of the IBM System/370" by R.P. Case, A. Padegs. Communications of the ACM. 21:1, 73-96, January 1978. *Perhaps the first paper to use the term* ***translation lookaside buffer****. The name arises from the historical name for a cache, which was a* ***lookaside buffer*** *as called by those developing the Atlas system at the University of Manchester; a cache of address translations thus became a* ***translation lookaside buffer****. Even though the term lookaside buffer fell out of favor, TLB seems to have stuck, for whatever reason.*

[H93] "MIPS R4000 Microprocessor User's Manual". by Joe Heinrich. Prentice-Hall, June 1993. Available: http://cag.csail.mit.edu/raw/ . documents/R4400 Uman book Ed2.pdf *A manual, one that is surprisingly readable. Or is it?*

[HP06] "Computer Architecture: A Quantitative Approach" by John Hennessy and David Pat- terson. Morgan-Kaufmann, 2006. *A great book about computer architecture. We have a particular attachment to the classic first edition.*

[I09] "Intel 64 and IA-32 Architectures Software Developer's Manuals" by Intel, 2009. Avail- able: http://www.intel.com/products/processor/manuals. *In particular, pay attention to "Vol- ume 3A: System Programming Guide" Part 1* and "Volume 3B: System Programming Guide Part 2".

[PS81] "RISC-I: A Reduced Instruction Set VLSI Computer" by D.A. Patterson and C.H. Se- quin. ISCA '81, Minneapolis, May 1981. *The paper that introduced the term RISC, and started the avalanche of research into simplifying computer chips for performance.*

[SB92] "CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking" by Rafael H. Saavedra-Barrera. EECS Department, University of California, Berkeley. Technical Report No. UCB/CSD-92-684, February 1992.. *A great dissertation about how to predict execution time of applications by breaking them down into constituent pieces and knowing the cost of each piece. Probably the most interesting part that comes out of this work is the tool to measure details of the cache hierarchy (described in Chapter 5). Make sure to check out the wonderful diagrams therein.*

[W03] "A Survey on the Interaction Between Caching, Translation and Protection" by Adam Wiggins. University of New South Wales TR UNSW-CSE-TR-0321, August, 2003. *An excellent survey of how TLBs interact with other parts of the CPU pipeline, namely hardware caches.*

[WG00] "The SPARC Architecture Manual: Version 9" by David L. Weaver and Tom Germond. SPARC International, San Jose, California, September 2000. Available: `www.sparc.org/standards/SPARCV9.pdf`. *Another manual. I bet you were hoping for a more fun citation to end this chapter.*

Ödev (Ölçüm)

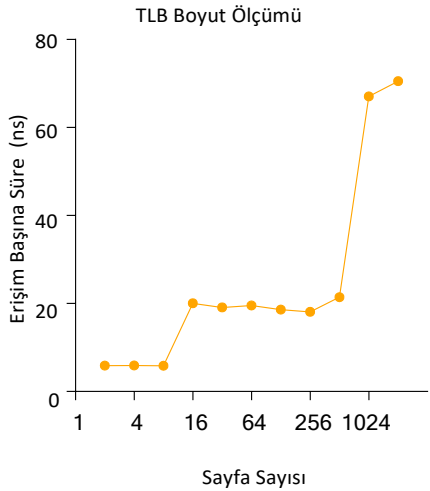
Bu ödevde, bir TLB'ye erişmenin boyutunu ve maliyetini ölçmelisiniz. Fikir, önbellek hiyerarşilerinin sayısız yönünü ölçmek için basit ama güzel bir yöntem geliştiren Saavedra-Barrera'nın [SB92] çalışmalarına dayanıyor ve hepsi de çok basit bir kullanıcı düzeyinde programla çalışıyor. Daha fazla ayrıntı için çalışmasını okuyun.

Temel fikir, büyük bir veri yapısı (örneğin, bir dizi) içindeki bazı sayfalara erişmek ve bu erişimleri zamanlamaktır. Örneğin, bir makinenin TLB boyutunun 4 olduğunu varsayalım (bu çok küçük olur, ancak bu tartışmanın amaçları için yararlı olur). 4 veya daha az sayfaya dokunan bir program yazarsanız, her erişim bir TLB isabeti olmalı ve bu nedenle nispeten hızlı olmalıdır. Ancak, 5 sayfaya veya daha fazlasına dokunduğunuzda, bir döngüde tekrar tekrar, her erişim aniden bir TLB ıskasına mal olur.

Bir dizide bir kez döngü oluşturacak temel kod şöyle görünmelidir:

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

Bu döngüde, a dizisinin sayfası başına bir tamsayı, NUMPAGES tarafından belirtilen sayfa sayısına kadar güncelleştirilir. Böyle bir döngüyü tekrar tekrar zamanlayarak (örneğin, bunun etrafındaki başka bir döngüde birkaç yüz milyon kez veya birkaç saniye boyunca çalışmak için kaç döngüye ihtiyaç duyulursa), her erişimin ne kadar süreceğini zamanlayabilirsiniz (ortalama olarak). NUMPAGES arttıkça maliyetteki sıçramaları arayarak, birinci seviye TLB'nin ne kadar büyük olduğunu kabaca belirleyebilir, ikinci seviye TLB'nin var olup olmadığını (ve varsa ne kadar büyük olduğunu) belirleyebilir ve genel olarak TLB'nin nasıl vurduğunu ve kaçırmasının nasıl performansı etkilediğini iyi anlayabilirsiniz.



Şekil 19.5: **TLB Boyutlarını ve Kaçırma Maliyetlerini Keşfetme**

Şekil 19.5 (sayfa 15), döngüde erişilen sayfa sayısı arttıkça erişim başına ortalama süreyi göstermektedir. Grafikte görebileceğiniz üzere, yalnızca birkaç sayfaya erişildiğinde (8 veya daha az), ortalama erişim süresi kabaca 5 nanosaniyedir. 16 veya daha fazla sayfaya erişildiğinde, erişim başına yaklaşık 20 nanosaniyeye ani bir sıçrama olur. Maliyette son bir sıçrama 1024 sayfa civarında gerçekleşir ve bu noktada her erişim yaklaşık 70 nanosaniye sürer. Bu veriden, iki seviyeli bir TLB hiyerarşisi olduğu sonucuna varabiliriz; ilki oldukça küçüktür (muhtemelen 8 ila 16 giriş tutar); ikincisi daha büyük ama daha yavaş (aşağı yukarı 512 giriş tutar). Birinci seviye TLB'deki isabetler ve ıskalar arasındaki genel fark oldukça büyük, kabaca on dört faktördür. TLB performansı önemlidir!

Soru

1. Zamanlama için bir zamanlayıcı kullanmanız gerekir (ör. gettimeofday()). Böyle bir zamanlayıcı ne kadar hassastır? Bir operasyonun tam olarak zamanlanması için ne kadar sürmesi gerekir? (bu, bir döngüde, başarılı bir şekilde zamanlamak için bir sayfa erişimini kaç kez tekrarlamanız gerekeceğini yardımcı olacaktır).

   Bir zamanlayıcının gettimeofday() gibi hassasiyeti çalıştırıldığı sisteme bağlı olarak değişebilir. Çoğu sistemde, yaklaşık 1 mikro saniye hassasiyete sahiptir, bu da zaman aralıklarını 1 mikro saniyeye kadar hata ile ölçebilmesini sağlar.

   Bir sayfaya erişimi zamanlamak için kaç kez tekrarlamamız gerektiğini belirlemek için, zamanlamaya çalıştığınız işlem beklenen süresini dikkate almamız gerekir. Eğer işlem 1 mikro saniyeden az sürerse, hassas bir ölçüm elde etmek için çok kez tekrarlamamız gerekir. Diğer taraftan, eğer işlem 1 mikro saniyeden daha uzun sürerse, tek bir yineleme ile zamanlamak mümkün olabilir.

   Sonuç olarak, zamanlamayı doğru şekilde yapmak için işlemi tekrarlamamız gereken sayı, zamanlayıcımızın hassasiyetine ve zamanlamaya çalıştığımız işlemin süresine bağlıdır. Güvenli bir şekilde, her zaman işlemi birkaç kez tekrarlamak ve ölçümlerin ortalamasını almak hata payını azaltmak için iyi bir fikirdir.

2. Her sayfaya erişim maliyetini kabaca ölçebilen tlb.c adlı programı yazın. Programa girdiler şöyle olmalıdır: dokunulacak sayfa sayısı ve deneme sayısı.

   Bu program, kullanıcı tarafından verilen sayfa sayısı ve deneme sayısı değerlerini alır ve dokunulacak sayfa sayısının ortalama erişim maliyetini hesaplar.

```c
#include <stdio.h>
#include <time.h>

int main(int argc, char *argv[]) {
  // Kullanıcı tarafından verilen sayfa sayısı ve deneme sayısı değerlerini al
  int pageCount, trialCount;
  printf("Lütfen dokunulacak sayfa sayısını girin: ");
  scanf("%d", &pageCount);
  printf("Lütfen deneme sayısını girin: ");
  scanf("%d", &trialCount);

  // Deneme sayısı kadar döngü oluştur
  int i;
  long totalCost = 0;
  for (i = 0; i < trialCount; i++) {
    // Rastgele sayı üret ve dokunulacak sayfayı belirle
    int page = rand() % pageCount;

    // Zaman ölç
    clock_t start = clock();
    // Dokunulacak sayfayı dokun
    accessPage(page);
    clock_t end = clock();

    // Erişim maliyetini hesapla ve toplamına ekle
    long cost = end - start;
    totalCost += cost;
  }

  // Ortalama erişim maliyetini hesapla ve ekrana yazdır
  double avgCost = (double) totalCost / trialCount;
  printf("Ortalama erişim maliyeti: %f\n", avgCost);

  return 0;
}
```

Bu programda, accessPage() fonksiyonunun ne yaptığı belirtilmemiştir. Bu fonksiyon, verilen sayfayı dokundurmak için kullanılır. Örneğin, bu fonksiyon bir sayfanın veri tabanından okunması veya bir dosyadan okunması gibi bir işleve sahip olabilir.

3. Bu programı çalıştırmak için, en sevdiğiniz betik dilinde bir senaryo yazın (bash?), erişilen sayfa sayısını 1'den birkaç bine kadar değiştirirken, belki de yineleme başına iki kat artar. Komut dosyasını farklı makinelerde çalıştırın ve veri toplayın. Güvenilir ölçümler elde etmek için kaç deneme gereklidir?
   Kullanıcı tarafından verilen sayfa sayısını 1'den birkaç bine kadar değiştirerek ve yineleme başına iki kat arttırarak çalışan bir bash senaryo aşağıdadır. Bu senaryo çalıştırırsa sonuçlar "results.txt" dosyasına kaydeder:

```bash
#!/bin/bash

# tlb.c programını derle
gcc tlb.c -o tlb

# Değiştirilecek sayfa sayısını belirle
page_count=1

# Çalıştırılacak deneme sayısını belirle
trial_count=1

# Sonuçları "results.txt" dosyasına kaydet
echo "page_count,trial_count,avg_cost" > results.txt

# Döngü oluştur ve sayfa sayısını artır
while [ $page_count -le 10000 ]
do
  # tlb programını çalıştır ve sonuçları dosyaya yazdır
  ./tlb $page_count $trial_count >> results.txt

  # Sayfa sayısını ve deneme sayısını artır
  page_count=$((page_count*2))
  trial_count=$((trial_count*2))
done
```

Bu senaryo, sayfa sayısını 1, 2, 4, 8, ..., 10000 olarak değiştirir ve yineleme başına iki kat arttırır. Bu, veri toplamak için yeterli olabilir, ancak güvenilir ölçümler elde etmek için kaç deneme gerektiğine dair bir kesin cevap vermek zor. Bu, dokunulacak sayfa sayısına, dokunma işleminin gerçekleştirildiği ortama ve diğer faktörlere bağlıdır. Örneğin, dokunulacak sayfa sayısı arttıkça daha fazla deneme yapılması gerekebilir. Benzer şekilde, dokunulacak sayfaların veri tabanından okunması veya dosyalardan okunması gibi daha zaman alıcı işlemler için daha fazla deneme yapılması gerekebilir. Bu nedenle, güvenilir ölçümler elde etmek için yeterli deneme sayısını belirlemek için deney-yanılma yöntemi kullanılabilir.

4. Ardından, yukarıdakine benzer bir grafik oluşturarak sonuçların grafiğini çıkarın. Ploticus veya hatta zplot gibi iyi bir araç kullanın. Görselleştirme genellikle verinin kavranılmasını çok daha kolay hale getirir; sizce neden böyle?
5. Dikkat edilmesi gereken bir şey derleyici optimizasyonudur. Derleyiciler, programın başka hiçbir bölümünün daha sonra kullanmadığı değerleri artıran döngüleri kaldırmak da dahil olmak üzere her türlü akıllı şeyi yapar. Derleyicinin yukarıdaki ana döngüyü TLB boyut tahmincinizden kaldırmadığından nasıl emin olabilirsiniz?

Derleyicinin bir döngüyü TLB boyut tahmincisinden kaldırmadığından emin olmak için birkaç yöntem kullanılabilir:

1. Döngünün TLB boyut tahmincisini kullanarak çalıştırılıp çıktıları incelenebilir. Eğer döngü TLB boyut tahmincisinden kaldırılmışsa, çıktılar beklenenden farklı olacaktır.
2. Derleyici ürettiği makine diline ait derlenmiş kodun incelenmesi yapılabilir. Eğer döngü TLB boyut tahmincisinden kaldırılmışsa, bu değişiklikler makine diline ait derlenmiş kodun içinde görülebilir.
3. Derleyici ürettiği makine diline ait derlenmiş kodun çalıştırılması sırasında TLB kullanımını izleyen bir araç kullanılabilir. Bu araç, TLB kullanımını izler ve eğer döngü TLB boyut tahmincisinden kaldırılmışsa, bu değişiklikleri tespit edebilir.

Bu yöntemlerden herhangi birini kullanarak, derleyicinin bir döngüyü TLB boyut tahmincisinden kaldırıp kaldırmadığını anlamaya çalışabiliriz.

6. Dikkat edilmesi gereken bir diğer şey, günümüzde çoğu sistemin birden fazla CPU ile gönderilmesi ve her CPU'nun elbette kendi TLB hiyerarşisine sahip olmasıdır. Gerçekten iyi ölçümler elde etmek için, zamanlayıcının bir CPU'dan diğerine sıçramasına izin vermek yerine kodunuzu yalnızca bir CPU'da çalıştırmanız gerekir. Bunu nasıl yapabilirsiniz? (ipucu: Bazı ipuçları için Google'da "bir iş parçacığını sabitleme" ye bakın) Bunu yapmazsanız ve kod bir CPU'dan diğerine geçerse ne olur?

Bir iş parçacığını sabitleme, bir iş parçacığının bir CPU'da çalıştırılmasını sağlar. Bu, iş parçacığının çalışma zamanı sırasında bir CPU'dan diğerine geçmemesine yardımcı olur ve böylece daha doğru TLB ölçümleri elde edilir.

Bir iş parçacığını sabitlemeyi nasıl yapabilirsiniz, çeşitli yöntemler kullanılabilir. Örneğin:

1. POSIX threads kütüphanesi kullanılarak pthread_setaffinity_np() fonksiyonu kullanılabilir. Bu fonksiyon, bir iş parçacığının belirli bir CPU'da çalışmasını sağlar.

2. Windows işletim sisteminde, SetThreadAffinityMask() fonksiyonu kullanılabilir. Bu fonksiyon da bir iş parçacığının belirli bir CPU'da çalışmasını sağlar.

Eğer bir iş parçacığını sabitlemez ve kod bir CPU'dan diğerine geçerse, TLB ölçümleri doğru olmayabilir. Bu, çünkü TLB hiyerarşisi her CPU için farklı olabilir ve iş parçacığı bir CPU'dan diğerine geçtiğinde TLB hiyerarşisi de değişebilir. Bu nedenle, TLB ölçümleri doğru değildir ve bu nedenle yanıltıcı sonuçlar elde edilebilir.

7. Ortaya çıkabilecek bir başka sorun da başlatma ile ilgilidir. Yukarıdaki diziyi erişmeden önce başlatmazsanız, talebe sıfırlama gibi ilk erişim maliyetleri nedeniyle diziye ilk kez eriştiğinizde çok pahalı olacaktır. Bu, kodunuzu ve zamanlamasını etkiler mi? Bu potansiyel maliyetleri dengelemek için ne yapabilirsiniz?

Evet, diziye erişimde başlatma yapılmaması kodumuzun performansını etkileyebilir. Başlatma, diziye erişim işleminin ilk kez gerçekleştiğinde dizinin bellekte yüklenmesi işlemini ifade eder. Eğer diziye erişim işlemini gerçekleştirmeden önce başlatma yapılmazsa, ilk erişim sırasında dizinin bellekte yüklenmesi gerekecek ve bu işlem pahalı olabilir.

Bu potansiyel maliyetleri dengelemek için, dizinin başlatılmasını hızlandırmak için birkaç yöntem kullanılabilir:

1. Dizinin önceden yüklenmiş bir kopyasını kullanmak. Bu, dizinin ilk kez yüklenmesi sırasında bellekte yer açılması ve dizinin elemanlarının yüklenmesi gibi işlemleri azaltacaktır.

2. Dizinin tüm elemanlarının aynı anda yüklenmesini sağlayan bir yükleme işlemi kullanmak. Bu, dizinin elemanlarının tek tek yüklenmesi yerine tüm elemanların aynı anda yüklenmesi işlemini gerçekleştirerek dizinin başlatma süresini azaltacaktır.

3. Dizinin elemanlarının büyük ölçüde sıkıştırılmış bir şekilde depolandığı bir veri yapısı kullanmak. Bu, dizinin elemanlarının daha az bellekte depolandığı anlamına gelecektir ve bu da dizinin başlatma süresini azaltacaktır.

Bu yöntemlerden herhangi birini kullanarak, dizinin başlatma süresini azaltarak potansiyel maliyetleri dengeleyebiliriz.