Sayfalama: Daha Hızlı Çeviriler (TLB'ler)

Disk belleğini sanal belleği desteklemek için çekirdek mekanizma olarak kullanmak, yüksek performans giderlerine yol açabilir. Adres alanını küçük parçalara bölerek, sabit boyutlu birimler (yani sayfalar), sayfalama büyük miktarda haritalama bilgisi gerektirir. Çünkü bu haritalama bilgileri genellikle fiziksel bellekte depolanır , sayfalama mantıksal olarak program tarafından oluşturulan her sanal adres için fazladan bir bellek araması gerektirir. Her talimat getirme, açık yükleme veya mağaza işlemlerinden önce çeviri bilgisi için hafızaya gitmek engelleyici şekilde yavaş. Ve böylece sorunumuz:

DÖNÜM NOKTASI:
Adres Dönüşümü Nasıl Hızlandırılır?

Adreslemeyi nasıl hızlandırırız ve genellikle sayfalamanın gerekliliği gibi görünen ekstra bellek referansını nasıl engelleriz? Hangi donanımsal destek önerilir? Hangi işletim sistemi katılmalıdır?

Bir şeyleri hızlı yapmak istediğimizde, işletim sistemi genellikle yardıma ihtiyaç duyar. Ve sıklıkla yardım işletim sisteminin eski dostundan gelir: donanım. Adres çevirisini hızlandırmak için (tarihsel nedenlerle [CP78]) **çeviri görünümlü arabellek(translation-lookaside buffer)** veya **TLB** olarak adlandırılan şeyi ekleyeceğiz. Bir TLB çipin **bellek yönetimi biriminin (memory-management unit - MMU)** bir parçasıdır ve basitçe bir popüler sanaldan fiziksele adres çevirilerinin donanım önbelleğidir; böylece, daha iyi bir ad, bir **adres çeviri önbelleği(address-translation cache)** olacaktır. Her bir sanal bellek referansında, donanım ilk önce istenen çevirinin tutulduğuna bakmak üzere TLB'ye bakar; eğer öyle ise çeviri hızlıca sayfa tablosuna (tüm çevirileri içeren) başvurmak zorunda kalmadan yapılır. Muazzam performans etkilerinin nedeni olarak, TLB'ler gerçek anlamda sanal bellek mümkün [C95].

19.1 TLB Basit Algoritması

Şekil 19.1, basit bir doğrusal sayfa tablosu (yani, sayfa tablosu bir dizidir) ve donanım tarafından yönetilen bir TLB (yani, sayfa tablosunun sorumluluğunun çoğunu donanım üstlenir) varsayarak, donanımın bir sanal adres çevirisini nasıl işleyebileceğinin kabaca bir taslağını gösterir (erişim; bununla ilgili daha fazla bilgiyi aşağıda açıklayacağız).

```
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 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

Şekil 19.1, basit bir **doğrusal tablosu(linear page table)(**yani sayfa tablosu bir dizi) ve **donanım tarafından yönetilen (hardware-managed) TLB** (yani donanım, sayfa tablosu erişimlerin üzerindekilerden beklentileri üstlenir; bu konuda sunulanlar hakkında daha fazla bilgi kullanımı) varsayarak sanal adres çevriminin nasıl gerçekleştirilebileceğini gösteren bir tüketim gösterir.

Donanımın izlediği algoritma şu şekilde çalışır: önce, sanal adresten sanal sayfa numarasını (VPN) çıkarın (Şekil 19.1'deki Satır 1) ve TLB'nin bu VPN için çeviriyi elinde tutup tutmadığını kontrol edin (Satır 2). Varsa, bir **TLB isabetimiz(TLB hit)** var, bu da TLB'nin çeviriyi elinde tuttuğu anlamına gelir. Başarı! Artık sayfa çerçeve numarasını (PFN) ilgili TLB girişinden çıkarabilir, bunu orijinal sanal adresten ofset üzerinde birleştirebilir ve istenen fiziksel adresi (PA) oluşturabilir ve belleğe (Satırlar) erişebiliriz. 5–7), koruma kontrollerinin başarısız olmadığı varsayılarak (Satır 4).

CPU çeviriyi TLB'de bulamazsa (bir **TLB eksikliği(TLB miss)**), yapacak daha çok işimiz var. Bu örnekte, donanım çeviriyi bulmak için sayfa tablosuna erişir (Satır 11–12) ve işlem tarafından oluşturulan sanal bellek referansının geçerli ve erişilebilir olduğunu

varsayarak (Satır 13, 15), TLB'yi şu şekilde günceller: çeviri (Satır 18). Bu eylemler dizisi, öncelikle sayfa tablosuna (Satır 12) erişmek için gereken ekstra bellek referansı nedeniyle maliyetlidir. Son olarak, TLB güncellendiğinde, donanım talimatı yeniden dener; bu sefer çeviri TLB'de bulunur ve bellek referansı hızla işlenir

TLB, tüm önbellekler gibi, genel durumda çevirilerin önbellekte bulunduğu (yani, isabetler olduğu) öncülü üzerine inşa edilmiştir. Eğer öyleyse, TLB işlemci çekirdeğinin yakınında bulunduğundan ve oldukça hızlı olacak şekilde tasarlandığından, biraz yük eklenir. Bir hata oluştuğunda, yüksek sayfalama maliyeti ortaya çıkar; çeviriyi bulmak için sayfa tablosuna erişilmeli ve bir
ekstra bellek referansı (veya daha karmaşık sayfa tablolarıyla daha fazla) sonuçları. Bu sık sık meydana gelirse, program büyük olasılıkla fark edilir şekilde daha yavaş çalışacaktır; çoğu CPU talimatına göre bellek erişimleri oldukça maliyetlidir ve TLB kayıplar daha fazla bellek erişimine yol açar. Bu nedenle, elimizden geldiğince TLB ıskalamalarından kaçınmayı umuyoruz.

19.2 Örnek: Bir Diziye Erişim

Bir TLB'nin işleyişini netleştirmek için basit bir sanal adres izlemeyi inceleyelim ve bir TLB'nin performansını nasıl artırabileceğini görelim. Bu örnekte, bellekte sanal adres 100'den başlayan 10 adet 4 baytlık bir tamsayı dizimiz olduğunu varsayalım. Ayrıca, 16 baytlık sayfaları olan küçük bir 8 bitlik sanal adres alanımız olduğunu varsayalım; bu nedenle, bir sanal adres 4-bit VPN (16 sanal sayfa vardır) ve 4-bit ofset (bu sayfaların her birinde 16 bayt vardır) olarak bölünür.

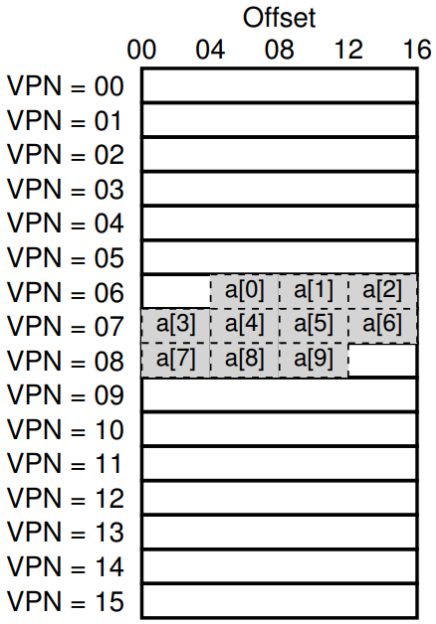
Şekil 19.2(sayfa 4), sistemin 16 adet 16 baytlık sayfasında düzenlenen diziyi göstermektedir. Gördüğünüz gibi dizinin ilk girişi (a[0]) (VPN=06, off set=04) ile başlıyor; o sayfaya yalnızca üç adet 4 baytlık tamsayı sığar. Dizi, sonraki dört girişin (a[3] ... a[6]) bulunduğu sonraki sayfaya (VPN=07) devam eder. Son olarak, 10 girişli dizinin (a[7] ... a[9]) son üç girişi, adres alanının (VPN=08) bir sonraki sayfasında bulunur.

Şimdi her bir 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 adına, döngünün oluşturduğu tek belleğin diziye erişim sağladığını varsayacağız (i ve sum değişkenlerinin yanı sıra talimatların kendisini de göz ardı ederek). İlk dizi öğesine (a[0]) erişildiğinde, CPU sanal adres 100'e bir yük görecektir. Donanım bundan VPN'i çıkarır (VPN=06) ve bunu TLB'yi geçerli bir çeviri için kontrol etmek için kullanır. Bunun programın diziye ilk kez eriştiği varsayılırsa, sonuç bir TLB hatası olacaktır.

Bir sonraki erişim a[1]'e olacak ve burada bazı iyi haberler var: bir TLB vuruşu! Dizinin ikinci öğesi birinci öğenin yanında paketlendiği için aynı sayfada yer alır; dizinin ilk öğesine erişirken bu sayfaya zaten eriştiğimiz için, çeviri zaten



Şekil 19.2: Örnek: Küçücük Adres Alanında Bir Dizi

TLB'ye yüklenmiştir. Ve dolayısıyla başarımızın nedeni. a[2] tr'ye erişim, benzer bir başarıyı (başka bir isabet) sayar, çünkü o da a[0] ve a[1] ile aynı sayfada yer alır.

Ne yazık ki, program a[3]'e eriştiğinde, başka bir TLB hatasıyla karşılaşıyoruz. Ancak, bir kez daha, sonraki girişler (a[4] ... a[6]), tümü bellekte aynı sayfada bulunduğu için TLB'de bulunacaktır.

Son olarak, a[7]'ye erişim, son bir TLB'nin kaçırılmasına neden olur. Donanım, bu sanal sayfanın fiziksel bellekteki yerini 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 avantajlarını alır; ne zaman donanım, çevirileri için TLB'ye bakar, iki isabet daha elde edilir.

Diziye on erişimimiz sırasındaki TLB etkinliğini özetleyelim: **ıskala(miss)**, vur, vur, **ıskala(miss),** vur, vur, vur, **ıskala(miss),** vur, vur. Böylece, isabet sayısının toplam erişim sayısına bölünmesiyle elde edilen TLB **isabet oranımız(hit rate)** %70'tir. Bu çok yüksek olmasa da (aslında %100'e yaklaşan isabet oranlarını arzu ediyoruz), sıfır değildir, bu sürpriz olabilir. Buna rağmen program diziye ilk kez eriştiğinde, TLB performansı uzamsal **mekansal yerellik konumdan(spatial locality)** yararlanır. Dizinin öğeleri, sayfalar halinde sıkıca paketlenir (yani, uzayda birbirlerine yakındırlar) ve bu nedenle yalnızca bir sayfadaki bir öğeye ilk erişim, bir TLB eksikliğine neden olur.

Ayrıca bu örnekte sayfa boyutunun oynadığı role dikkat edin. Sayfa boyutu iki kat daha büyük olsaydı (16 değil 32 bayt), dizi erişimi daha da az kayıp yaşardı. Tipik sayfa boyutları daha çok 4 KB gibi olduğundan, bu tür yoğun, dizi tabanlı erişimler mükemmel TLB performansı sağlar, erişim sayfası başına yalnızca tek bir hatayla karşılaşılıyor.

TLB performansıyla ilgili son bir nokta: eğer program, bu döngü tamamlandıktan kısa bir süre sonra diziye tekrar erişirse, gerekli çevirileri önbelleğe alacak kadar büyük bir TLB'ye sahip olduğumuzu varsayarsak, muhtemelen daha da iyi bir sonuç görürüz: vur, vur, vur, vur, vur, vur, vur, vur, vur, vur. Bu durumda TLB isabet oranı, zamansal konum

İPUCU: MÜMKÜN OLDUĞUNDA ÖNBELLEKLEMEYİ KULLANIN

Önbelleğe alma, bilgisayar sistemlerindeki en temel performans tekniklerinden biridir ve "ortak durumu hızlı" hale getirmek için tekrar tekrar kullanılır [HP06]. Donanım önbelleklerinin arkasındaki fikir avantaj sağlamaktır. talimat ve veri referanslarında **yerellik(locality).** Genellikle iki tür yerellik vardır: **zamansal yerellik(temporal locality)** ve **mekansal yerellik(spatial locality).**

Zamansal yerellik ile , yakın zamanda erişilen bir yönerge veya veri öğesine gelecekte yakında yeniden erişilmesi muhtemeldir. Döngü değişkenlerini veya bir döngüdeki yönergeleri düşünün; zaman içinde tekrar tekrar erişilirler. Uzamsal yerellik ile , bir program x adresindeki belleğe erişirse, muhtemelen yakında x yakınındaki belleğe erişecektir. Burada bir kanaldan akış yaptığınızı düşünün. Bir tür dizi, bir öğeye ve ardından diğerine erişiyor. Tabii ki, bu özellikler programın tam doğasına bağlıdır ve bu nedenle katı ve hızlı yasalar değil, daha çok pratik kurallar gibidir.

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

Merak ediyor olabilirsiniz: Önbellekler (TLB gibi) bu kadar harikaysa, neden daha büyük önbellekler oluşturup tüm verilerimizi içlerinde tutmuyoruz? Ne yazık ki, fiziğinkiler gibi daha temel yasalarla karşılaştığımız yer burasıdır. Hızlı bir önbellek istiyorsanız, ışık hızı ve diğer fiziksel kısıtlamalar önemli hale geldiğinden, küçük olması gerekir. Tanımı gereği herhangi bir büyük önbellek yavaştır ve bu nedenle amacı bozar. Bu nedenle, küçük, hızlı önbelleklerle sıkışıp kaldık; Geriye kalan soru, performansı artırmak için bunları en iyi nasıl kullanacağımızdır.

nedeniyle yüksek olacaktır, yani hızlı bellek öğelerinin zaman içinde yeniden referanslandırılması. Herhangi bir önbellek gibi, TLB'ler de başarı için program özellikleri olan hem uzamsal hem de zamansal konuma güvenir. İlgili program bu tür yerellik sergiliyorsa (ve birçok program gösteriyorsa), TLB isabet oranı muhtemelen yüksek olacaktır.

Bu döngünün tamamlanmasından hemen sonra program, diziyi tekrar erişirse, muhtemelen daha iyi sonuç görürüz, gereken çevirileri tamponlanmak için yeterince büyük bir TLB'ye sahip olursak: hit, hit, hit, hit, hit, hit, hit, hit, hit, hit. Bu durumda, TLB hit oranı **zamanlı yereliliğe(temporal locality)**(yani bellek öğelerinin **zaman(time)** içinde hızlı bir şekilde yeniden referans alınması) kadar yüksek olacaktır. Herhangi bir önbellek gibi, TLB'ler hem mekânsal hem de zamanlı yerelliklere dayanır, bu da program özellikleridir. İlgilenilen program böyle bir yereliliğe sahipse (ve birçok program bunu yapar), TLB hit 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 (OS Tarafından Yönetilen)

19.3 TLB Iskalamayı Kim Halleder?

Cevaplamamız gereken bir soru: Bir TLB özlediğini kim halleder? İki cevap mümkündür: donanım veya yazılım (OS). Eski günlerde, donanımın karmaşık talimat setleri vardı (bazen karmaşık talimat seti bilgisayarları için **CISC** olarak adlandırılır) ve donanımı oluşturan insanlar, bu sinsi işletim sistemi insanlarına pek güvenmiyordu. Böylece donanım, TLB eksikliğini tamamen giderebilir. Bunu yapmak için, donanımın sayfa tablolarının bellekte tam olarak nerede bulunduğunu (Şekil 19.1'de Satır 11'de

kullanılan bir **sayfa tablosu temel kaydı(pagetable base register)** aracılığıyla) ve bunların tam formatını bilmesi gerekir; ıskalama durumunda, donanım sayfa tablosunu "yürütür", doğru sayfa tablosu girişini bulur ve istenen çeviriyi çıkarır, TLB'yi çeviriyle günceller ve talimatı yeniden dener. **Donanım tarafından yönetilen TLB'lere(hardware-managed TLBs)** sahip "eski" bir mimari örneği, sabit birçok **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 edilir.

Daha modern mimariler (örneğin, MIPS R10k [H93] veya Sun'ın SPARC v9 [WG00], her ikisi de **RISC** veya azaltılmış komut setli bilgisayarlar) **yazılım tarafından yönetilen TLB(software-managed TLB)** olarak bilinen şeye sahiptir. Bir TLB hatasında, donanım basitçe bir istisna oluşturur (Şekil 19.3'teki 11. satır), mevcut talimat akışını duraklatır, ayrıcalık seviyesini çekirdek moduna yükseltir ve bir tuzak işleyiciye atlar. Tahmin edebileceğiniz gibi, bu **tuzak işleyici(trap handler),** TLB kayıplarını işlemek amacıyla yazılmış işletim sistemi içindeki koddur. Kod çalıştırıldığında sayfa tablosundaki çeviriyi arayacak, TLB'yi güncellemek için özel "ayrıcalıklı" yönergeleri kullanacak ve tuzaktan geri dönecektir; bu noktada, donanım talimatı yeniden dener (bir TLB isabetiyle sonuçlanır).

Birkaç önemli ayrıntıyı tartışalım. Birincisi, tuzaktan dönüş talimatının, daha önce bir sistem çağrısına hizmet verirken gördüğümüz tuzaktan dönüş komutundan biraz farklı olması gerekir. İkinci durumda, tıpkı bir prosedür çağrısından dönüşün prosedüre yapılan çağrının hemen ardından talimata geri dönmesi gibi, tuzaktan geri dönüş OS'ye tuzaktan sonra talimatta yürütmeye devam etmelidir. Önceki durumda, bir TLB yanlış işleme tuzağından dönerken, donanım tuzağa neden olan talimatta yürütmeye devam etmelidir; bu yeniden deneme, talimatın tekrar çalışmasına izin verir, bu sefer bir TLB isabetiyle sonuçlanır. Bu nedenle, bir tuzağın veya istisnanın nasıl oluşturulduğuna bağlı olarak, işletim sistemine bindirme sırasında donanımın zamanı geldiğinde düzgün bir şekilde

ASIDE: RISC VS. CISC

1980'lerde bilgisayar mimarisi camiasında büyük bir savaş yaşandı. Bir tarafta, **Karmaşık Komut Seti Hesaplamanın(Complex Instruction Set Computing)** kısaltması olan **CISC** kampı vardı; diğer tarafta, **Azaltılmış Komut Seti Hesaplama(Reduced Instruction Set Computing) [**PS81] için **RISC** vardı. RISC tarafı, Berkeley'den David Patterson ve Stanford'dan John Hennessy (aynı zamanda bazı ünlü kitapların ortak yazarları [HP06]) tarafından yönetildi, ancak daha sonra John Cocke, RISC üzerine ilk çalışması nedeniyle bir Turing ödülü ile tanındı [CM00]. CISC komut setlerinde çok sayıda talimat bulunur 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, montaj dilinin kendisinin kullanımını kolaylaştırmak ve kodu daha derli toplu hale getirmek için talimatların üst düzey ilkel olması gerektiğiydi.

CISC komut setlerinde çok sayıda talimat bulunur 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, montaj dilinin kendisinin kullanımını kolaylaştırmak ve kodu daha derli toplu hale getirmek için talimatların üst düzey ilkel olması gerektiğiydi. RISC komut setleri tam tersidir. RISC'nin ardındaki önemli bir gözlem, komut setlerinin gerçekten derleyici hedefleri olduğu ve derleyicilerin gerçekten istediği tek şeyin, yüksek performanslı kod oluşturmak için kullanabilecekleri birkaç basit ilkel olmasıdır. Bu nedenle, RISC savunucuları, donanımdan mümkün olduğu kadar çok şeyi (özellikle mikro kodu) söküp atalım ve geriye kalanları basit, tekdüze ve hızlı hale getirelim. İlk günlerde, RISC çipleri fark edilir derecede daha hızlı oldukları için büyük bir etki yarattı [BC91]; birçok makale yazıldı; birkaç şirket kuruldu (örneğin, MIPS ve Sun). Ancak zaman geçtikçe CISC üreticileri Intel gibi birçok RISC tekniğini işlemcilerinin çekirdeğine dahil ettiler; örneğin, karmaşık yönergeleri daha sonra RISC benzeri bir şekilde işlenebilen mikro yönergelere dönüştüren erken boru hattı aşamalarını ekleyerek. Bu yeniliklere ek olarak her çipte artan sayıda transistör, CISC'nin rekabetçi kalmasını sağladı. Sonuç olarak, tartışma sona erdi ve bugün her iki işlemci türü de hızlı çalışacak şekilde yapılabilir.

devam etmesi için farklı bir PC'yi kaydetmesi gerekir.

İkincisi, TLB hata işleme kodunu çalıştırırken, işletim sisteminin sonsuz sayıda TLB hatalarının oluşmasına neden olmamak için ekstra dikkatli olması gerekir. Birçok çözüm mevcuttur; örneğin, TLB eksik işleyicilerini fiziksel bellekte tutabilirsiniz (burada **eşlenmemişler (unmapped)** ve adres çevirisine tabi değiller) veya TLB'deki bazı girişleri kalıcı olarak geçerli çeviriler için ayırabilir ve bu kalıcı çeviri yuvalarından bazılarını işleyici için kullanabilirsiniz. kodun kendisi; bu **kablolu(wired)** çeviriler her zaman TLB'de görülür.

Yazılımla yönetilen yaklaşımın birincil avantajı esnekliktir: İşletim sistemi, donanım

KENARA: TLB GEÇERLİ BİT ≠SAYFA TABLOSU GEÇERLİ BİT

Bir TLB'de bulunan geçerli bitleri bir sayfa tablosunda bulunanlarla karıştırmak yaygın bir hatadır. Bir sayfa tablosunda, bir sayfa tablosu girişi (PTE) geçersiz olarak işaretlendiğinde, bu, sayfanın işlem tarafından tahsis edilmediği ve düzgün çalışan bir program tarafından erişilmemesi gerektiği anlamına gelir. Geçersiz bir sayfaya erişildiğinde olağan yanıt, işlemi sonlandırarak yanıt verecek olan işletim sistemine tuzak kurmaktır.

Bir TLB geçerli biti, aksine, basitçe bir TLB girişinin içinde geçerli bir çeviriye sahip olup olmadığını ifade eder. Örneğin, bir sistem önyüklendiğinde, her TLB girişi için ortak bir başlangıç durumu geçersiz olarak ayarlanmalıdır, çünkü burada henüz hiçbir adres çevirisi önbelleğe alınmamıştır. Sanal bellek etkinleştirildikten ve programlar çalışmaya ve sanal adres alanlarına erişmeye başladığında, TLB yavaş yavaş doldurulur ve bu nedenle geçerli girişler kısa süre sonra TLB'yi doldurur.

TLB geçerli biti, aşağıda daha ayrıntılı olarak tartışacağımız gibi, bir içerik anahtarı gerçekleştirirken de oldukça kullanışlıdır. Sistem, tüm TLB girişlerini geçersiz olarak ayarlayarak, çalıştırılmak üzere olan işlemin yanlışlıkla önceki bir süreçten sanaldan fiziksele çeviriyi kullanmamasını sağlayabilir.

değişikliği gerektirmeden sayfa tablosunu uygulamak için istediği herhangi bir veri yapısını kullanabilir. Diğer bir avantaj ise basitliktir; TLB kontrol akışında görebileceğiniz gibi (Şekil 19.1'deki 11–19 satırlarının tersine, Şekil 19.3'teki 11. satır), donanımın bir kayıp durumunda çok fazla bir şey yapması gerekmez; bir istisna oluşturur ve OS TLB kayıp işleyicisi gerisini halleder.

19.4 TLB İçeriği: İçinde Ne Var?

Donanım TLB'sinin içeriğine daha detaylı bakalım. Tipik bir TLB'nin 32, 64 veya 128 girişi olabilir ve **tamamen çağrışımsal(fully associative)** olarak adlandırılan şey olabilir. Temel olarak bu, herhangi bir çevirinin TLB'de herhangi bir yerde olabileceği ve donanımın istenen çeviriyi bulmak için tüm TLB'yi paralel olarak arayacağı anlamına gelir. Tipik bir TLB girişi şöyle görünebilir:

VPN | PFN | diğer bitler

Bir çeviri bu konumlardan herhangi birinde sona erebileceğinden (donanım açısından TLB, **tam olarak ilişkilendirilebilir(fully associative)** bir önbellek olarak bilinir) her girişte hem VPN hem de PFN'nin bulunduğunu unutmayın. Donanım, bir eşleşme olup olmadığını görmek için girişleri paralel olarak arar.

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

19.5 TLB Sorunu: Bağlam Anahtarları

TLB'lerde, işlemler (ve dolayısıyla adres alanları) arasında geçiş yaparken bazı yeni sorunlar ortaya çıkar. Spesifik olarak, TLB, yalnızca o anda çalışan 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 önceden çalıştırılan bazı işlemlerden çevirileri yanlışlıkla kullanmadığından emin olmak için dikkatli olmalıdır.

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 aldığını varsayar. Bu örnek için, P1'in 10. sanal sayfasının fiziksel çerçeve 100'e eşlendiğini varsayalım.

Bu örnekte, başka bir işlemin (P2) var olduğunu ve işletim sisteminin yakında bir bağlam anahtarı gerçekleştirmeye ve onu çalıştırmaya karar verebileceğini varsayalım. Burada P2'nin 10. sanal sayfasının fiziksel çerçeve 170 ile eşlendiğini varsayalım. Her iki işlem için girişler TLB'de olsaydı, TLB'nin içeriği şöyle olurdu:

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) olarak çevrilir, ancak donanım hangi girişin hangi işlem için olduğunu ayırt edemez. Bu nedenle, TLB'nin çoklu süreçlerde sanallaştırmayı doğru ve verimli bir şekilde desteklemesi için biraz daha çalışmamız gerekiyor. Ve böylece, bir dönüm noktası:

Önemli Nokta:

BİR BAĞLAM ANAHTARINDA TLB İÇERİĞİ NASIL YÖNETİLİR

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

Bu sorunun bir dizi olası çözümü vardır. Yaklaşımlardan biri, TLB'yi içerik anahtarlarında basitçe **yıkamak(flush)** ve böylece bir sonraki işlemi çalıştırmadan önce onu boşaltmak. Yazılım tabanlı bir sistemde bu, açık (ve ayrıcalıklı) bir donanım talimatıyla gerçekleştirilebilir; donanım tarafından yönetilen bir TLB ile, sayfa tablosu temel kaydı değiştirildiğinde temizleme etkinleştirilebilir (işletim sisteminin bir bağlam

anahtarında PTBR'yi her halükarda değiştirmesi gerektiğini unutmayın). Her iki durumda da, temizleme işlemi basitçe tüm geçerli bitleri 0'a ayarlar ve temel olarak TLB'nin içeriğini temizler.

TLB'yi her bağlam anahtarında temizleyerek, artık çalışan bir çözüme sahibiz, çünkü bir süreç hiçbir zaman yanlışlıkla TLB'de yanlış çevirilerle karşılaşmaz. Ancak bunun bir bedeli vardır: Bir işlem her çalıştırıldığında, veri ve kod sayfalarına dokunurken TLB'nin gözden kaçmasına neden olmalıdır. İşletim sistemi, işlemler arasında sık sık geçiş yapıyorsa, bu maliyet yüksek olabilir.

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

Yukarıdan örnek TLB'mizi alıp ASID'leri eklersek, süreçlerin TLB'yi kolayca paylaşabileceği açıktır: aksi takdirde aynı olan çevirileri ayırt etmek için yalnızca ASID alanı gerekir. ASID alanı eklenmiş bir TLB'nin tasviri aşağıdadır:

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

Böylece, adres alanı tanımlayıcıları ile TLB, herhangi bir karışıklık olmadan aynı anda farklı işlemlerden çevirileri tutabilir. Elbette donanımın çevirileri gerçekleştirmek için o anda hangi işlemin çalıştığını bilmesi gerekir ve bu nedenle işletim sisteminin bir bağlam anahtarında mevcut işlemin ASID'sine bazı ayrıcalıklı kayıtlar ayarlaması gerekir.

Bir yana, TLB'nin iki girişinin oldukça benzer olduğu başka bir durum da düşünmüş olabilirsiniz. Bu örnekte, aynı fiziksel sayfayı işaret eden iki farklı VPN'li 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 paylaştığında (örneğin bir kod sayfası) ortaya çıkabilir. Yukarıdaki örnekte, İşlem 1, fiziksel sayfa 101'i İşlem 2 ile paylaşmaktadır; P1, bu sayfayı adres alanının 10. sayfasına, P2 ise adres alanının 50. sayfasına eşler. Kod sayfalarının (ikili dosyalarda veya paylaşılan kitaplıklarda) paylaşılması, kullanımdaki fiziksel sayfaların sayısını azalttığı ve böylece bellek ek yüklerini azalttığı için yararlıdır.

19.6 Sorun: Değiştirme Politikası

Herhangi bir önbellekte ve dolayısıyla TLB'de olduğu gibi, dikkate almamız gereken bir sorun daha **önbellek değiştirmedir(cache replacement).** Spesifik olarak, TLB'ye yeni bir giriş kurarken, eskisini **değiştirmeliyiz(replace)** ve dolayısıyla şu soru: Hangisini değiştirmeli?

EN ÖNEMLİ NOKTA: TLB DEĞİŞTİRME POLİTİKASI NASIL TASARLANIR

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

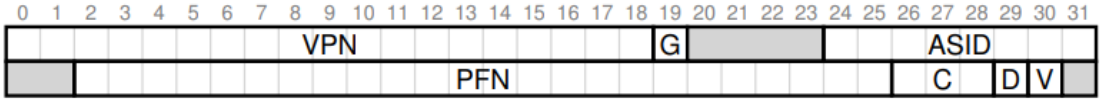
Bir sanal bellek sisteminde sayfaları diske aktarma sorununu ele alırken bu tür politikaları biraz ayrıntılı olarak inceleyeceğiz; burada sadece birkaç tipik politikayı vurgulayacağız. Yaygın bir yaklaşım, **en son kullanılan(least-recently-used)** veya **LRU** girişini çıkarmaktır. Buradaki fikir, bellek referans akışında yerellikten yararlanmaktır; bu nedenle, yakın zamanda kullanılmamış bir girişin tahliye için iyi bir aday olması muhtemeldir, çünkü (belki) yakında tekrar referans alınmayacaktır. Başka bir tipik yaklaşım, **rastgele(random)** bir politika kullanmaktır. Rastgelelik bazen kötü bir karar verir ama kötümser davranışa neden olabilecek herhangi bir tuhaf köşe vakası davranışının

olmaması, örneğin n+1 sayfaya erişen bir döngü, n boyutunda bir TLB ve bir LRU değiştirme ilkesi gibi güzel bir özellik.

19.7 Gerçek TLB Girişi

Sonunda, kısaca gerçek bir TLB'ye bakalım. Bu örnek, yazılım tarafından yönetilen TLB'leri kullanan modern bir sistem olan MIPS R4000'den [H93] alınmıştır. Bu TLB girişinin 64 bitinin tamamı Şekil 19.4'te görülebilir.

MIPS R4000, 4KB sayfalı 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 var; Görünüşe göre, kullanıcı adresleri adresin yalnızca yarısından gelecek alan (gerisi çekirdek için ayrılmıştır) ve bu nedenle yalnızca 19 bit VPN gereklidir. VPN, 24 bit fiziksel çerçeve numarasına (PFN) kadar çevirir ve dolayısıyla 64 GB'a kadar (fiziksel) ana belleğe (2 24 4 KB sayfa) sahip sistemleri destekleyebilir.
MIPS TLB'de birkaç ilginç parça daha var. Süreçler arasında global olarak paylaşılan sayfalar için kullanılan global bir bit (G) görüyoruz. Böylece, global bit ayarlanmışsa, ASID göz ardı edilir. Ayrıca, işletim sisteminin adres alanlarını ayırt etmek için kullanabileceği 8 bitlik ASID'yi de görüyoruz.



Şekil 19.4: **Bir MIPS TLB Girişi(A MIPS TLB Entry)**

Yukarıda tarif edilen). Size bir soru: Bir seferde 256'dan (2^8) fazla işlem çalışıyorsa işletim sistemi ne yapmalıdır? Son olarak, bir sayfanın donanım tarafından nasıl önbelleğe alınacağını belirleyen 3 Coherence (C) biti görüyoruz (bu notların kapsamının biraz ötesinde); sayfa yazıldığında işaretlenen kirli bir bit (bunun kullanımını daha sonra

İPUCU: RAM HER ZAMAN RAM DEĞİLDİR (CULLER YASASI)

Erken erişim belleği(random-access memory) veya **RAM**, bellekte herhangi bir yeri diğerine göre daha hızlı erişebileceğiniz anlamına gelir. Genellikle RAM'i bu şekilde düşünmek iyi bir fikirdir, ancak TLB gibi donanım/işletim sistemi özellikleri nedeniyle belirli bir sayfaya erişmek maliyetli olabilir, özellikle de TLB tarafından şu anda eşlenmemişse. Bu nedenle, sürekli olarak kullanım ipucunu hatırlamak iyi bir fikirdir: **RAM her zaman RAM değildir(RAM isn't always RAM.).** Zaman zaman adres alanınıza rastgele erişmek, özellikle erişilen sayfaların TLB kapsamını aşıyorsa, ciddi performans cezalarına neden olabilir. Öğretim elemanlarından biri olan David Culler, performans sorunlarının çoğunun kaynağının TLB olduğuna işaret ettiği için, bu yasayı onun adına verdik: **Culler Yasası(Culler's Law).**

göreceğiz); girişte geçerli bir çeviri olup olmadığını donanıma bildiren geçerli bir bit. Birden çok sayfa boyutunu destekleyen bir sayfa maskesi alanı da (gösterilmemiştir) vardır; daha büyük sayfalara sahip olmanın neden yararlı olabileceğini ileride göreceğiz. Son olarak, 64 bitin bir kısmı kullanılmamaktadır (şemada gri gölgeli).

MIPS TLB'ler genellikle bu girdilerden 32 veya 64 tanesine sahiptir ve bunların çoğu kullanıcı işlemleri tarafından çalışırken kullanılır. Ancak, birkaçı işletim sistemi için ayrılmıştır. Donanıma işletim sistemi için TLB'nin kaç yuvasının ayrılacağını söylemek için işletim sistemi tarafından kablolu bir kayıt ayarlanabilir; işletim sistemi bu ayrılmış eşlemeleri kritik zamanlarda erişmek istediği kod ve veriler için kullanır, burada bir TLB kaçırmanın sorunlu olacağı (örneğin, TLB kaçırma işleyicisinde).

MIPS TLB yazılım tarafından yönetildiğinden, TLB'yi güncellemek için talimatların olması gerekir. MIPS bu tür dört talimat sağlar: TLB'yi belirli bir çevirinin orada olup olmadığını görmek için araştıran TLBP; Kayıtlara bir TLB girişinin içeriğini okuyan TLBR; Belirli bir TLB girişinin yerini alan TLBWI; ve rastgele bir TLB girişinin yerini alan TLBWR. İşletim sistemi, TLB'nin içeriğini yönetmek için bu talimatları kullanır. Bu talimatların **ayrıcalıklı(privileged)** olması elbette önemlidir; TLB'nin içeriğini değiştirebilseydi bir kullanıcı işleminin neler yapabileceğini hayal edin (ipucu: hemen hemen her şey, makineyi ele geçirmek, kendi kötü niyetli "işletim sistemini" çalıştırmak ve hatta Sun'ı ortadan kaldırmak dahil).

19.8 Özet

Donanımın adres çevirisini daha hızlı yapmamıza nasıl yardımcı olabileceğini gördük. Bir adres çeviri önbelleği olarak küçük, özel bir çip üzerinde TLB sağlayarak, çoğu bellek referansı, ana bellekteki sayfa tablosuna erişmek zorunda kalmadan umarız işlenir. Bu nedenle, ortak durumda, programın performansı neredeyse hiç sanallaştırılmamış gibi olacaktır, bu bir işletim sistemi için mükemmel bir başarıdır ve kesinlikle modern sistemlerde sayfalama kullanımı için gereklidir.

Ancak, TLB'ler var olan her program için dünyayı pembeleştirmez. Özellikle, bir programın kısa sürede eriştiği sayfa sayısı, TLB'ye sığan sayfa sayısını aşarsa, program çok sayıda TLB hatası üretecek ve bu nedenle oldukça yavaş çalışacaktır. Bu olguyu **TLB kapsamının(TLB coverage**) aşılması olarak adlandırıyoruz ve belirli programlar için oldukça sorun olabilir. Bir sonraki bölümde tartışacağımız gibi bir çözüm, daha büyük

sayfa boyutları için destek eklemektir; anahtar veri yapılarını programın adres alanının daha büyük sayfalarla eşlenen bölgelerine eşleyerek, TLB'nin etkin kapsamı artırılabilir. Büyük sayfalara yönelik destek, hem büyük hem de rastgele erişilen belirli veri yapılarına sahip bir **veritabanı yönetim sistemi (database management system - DBMS)** gibi programlar tarafından sıklıkla kullanılır.

Bahsetmeye değer başka bir TLB sorunu: TLB erişimi, özellikle **fiziksel olarak indekslenmiş önbellek(physically-indexed cache)** olarak adlandırılan şeyle, CPU ardışık düzeninde kolayca 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 adreslerle önbelleklere erişmenin her türlü akıllı yolunu aradılar ve böylece bir önbellek isabeti durumunda pahalı çeviri adımından kaçındılar. Bu tür **sanal olarak dizine alınmış bir önbellek(virtuallyindexed cache),** bazı performans sorunlarını çözer, ancak donanım tasarımına da yeni sorunlar getirir. Daha fazla ayrıntı için Wiggins'in ince anketine bakın [W03].

Referanslar

[BC91] "Mimariden Performans: Bir RISC ve CISC'yi Benzer Donanım Organizasyonuyla Karşılaştırma"
D. Bhandarkar ve Douglas W. Clark
ACM'nin iletişimleri, Eylül 1991
RISC ve CISC arasında harika ve adil bir karşılaştırma. Sonuç olarak: benzer bir donanımda RISC, performansta yaklaşık üç kat daha iyi

[CM00] "IBM'de RISC teknolojisinin gelişimi"
John Cocke ve V. Markstein
IBM Journal of Research and Development, 44:1/2
Birçok kişinin ilk gerçek RISC mikro işlemcisi olarak kabul ettiği IBM 801'in arkasındaki fikirlerin ve çalışmaların bir özeti.

[C95] "Black Canyon Computer Corporation'ın Çekirdeği"
John Couleur
IEEE Annals of History of Computing, 17:4, 1995
Bu büyüleyici tarihsel notta Couleur, TLB'yi 1964'te çalışırken nasıl icat ettiğinden bahsediyor.
GE için ve MIT'deki Proje MAC çalışanları ile ortaya çıkan tesadüfi işbirliği.

[CG68] “Paylaşılan Erişimli Veri İşleme Sistemi”
John F. Couleur ve Edward L. Glaser
Patent 3412382, Kasım 1968
Adres çevirilerini depolamak için çağrışımsal bellek fikrini içeren patent. Fikir,
Couleur'a göre 1964'te geldi.

[CP78] "IBM System/370 mimarisi"
RP Case ve A. Padegs
ACM'nin iletişimleri. 21:1, 73-96, Ocak 1978
Belki de çeviri görünümlü arabellek terimini kullanan ilk makale. Adı, Manchester Üniversitesi'nde Atlas sistemini geliştirenler tarafından çağrılan bir arabellek olan bir önbelleğin tarihsel adından gelir; böylece bir adres çeviri önbelleği, çeviriye bakan bir arabellek haline geldi. Bakış tamponu terimi gözden düşmüş olsa da, TLB her ne sebeple olursa olsun takılıp kalmış gibi görünüyor.

[H93] “MIPS R4000 Mikroişlemci Kullanım Kılavuzu”.
Joe Heinrich, Prentice-Hall, Haziran 1993
Mevcut: http://cag.csail.mit.edu/raw/
belgeler/R4400 Uman kitabı Ed2.pdf

[HP06] "Bilgisayar Mimarisi: Nicel Bir Yaklaşım"
John Hennessy ve David Patterson
Morgan-Kaufmann, 2006
Bilgisayar mimarisi hakkında harika bir kitap. Klasik ilk baskıya özel bir bağlılığımız var.

[I09] “Intel 64 ve IA-32 Mimarileri Yazılım Geliştirici Kılavuzları”
Intel, 2009
Mevcut: http://www.intel.com/products/processor/manuals
Özellikle “Cilt 3A: Sistem Programlama Kılavuzu Bölüm 1” ve “Cilt 3B:
Sistem Programlama Kılavuzu Bölüm 2”

[PS81] “RISC-I: A Reduced Instruction Set VLSI Computer”
D.A. Patterson and C.H. Sequin
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 Performans Değerlendirmesi ve Yürütme Süresi Tahmini
Dar Spektrum Kıyaslamasının Kullanılması"
Rafael H. Saavedra-Barrera
EECS Departmanı, Kaliforniya Üniversitesi, Berkeley
Teknik Rapor No. UCB/CSD-92-684, Şubat 1992
www.eecs.berkeley.edu/Pubs/TechRpts/1992/CSD-92-684.pdf
Uygulamaların yürütülme süresinin, onları oluşturan parçalara ayırarak ve her bir parçanın maliyetini bilerek nasıl tahmin edileceğine dair harika bir tez. Muhtemelen bu çalışmadan çıkan en ilginç kısım, önbellek hiyerarşisinin ayrıntılarını ölçen araçtır (Bölüm 5'te açıklanmıştır). Buradaki harika diyagramları kontrol ettiğinizden emin olun.

[W03] "Önbelleğe Alma, Çeviri ve Koruma Arasındaki Etkileşim Üzerine Bir Araştırma"
Adam Wiggins
New South Wales Üniversitesi TR UNSW-CSE-TR-0321, Ağustos 2003
TLB'lerin CPU ardışık düzeninin diğer bölümleriyle, yani donanım önbellekleriyle nasıl etkileşime girdiğine dair mükemmel bir araştırma.

[WG00] "SPARC Mimarisi El Kitabı: Sürüm 9"
David L. Weaver ve Tom Germond, Eylül 2000
SPARC Uluslararası, San Jose, Kaliforniya
Mevcut: http://www.sparc.org/standards/SPARCV9.pdf

Ödev (Ölçme)

Bu ödevde, bir TLB'ye erişimin boyutunu ve maliyetini ölçeceksiniz. Fikir, tümü çok basit bir kullanıcı düzeyinde programla önbellek hiyerarşilerinin çeşitli yönlerini ölçmek için basit ama güzel bir yöntem geliştiren Saavedra-Barrera'nın [SB92] çalışmasına dayanmaktadır. Daha fazla ayrıntı için çalışmalarını okuyun.

Temel fikir, büyük veri yapısı (örneğin bir dizi) içindeki bazı sayfalara erişmek ve bu erişimleri zamanlamaktır. Örneğin, bir makinenin TLB boyutunun 4 olduğunu varsayalım (ki bu çok küçük ama bu tartışmanın amaçları açısından yararlı olacaktır). 4 veya daha az sayfaya dokunan bir program yazarsanız, her erişim bir TLB isabeti olmalı ve bu nedenle nispeten hızlı olmalıdır. Bununla birlikte, bir döngüde art arda 5 veya daha fazla sayfaya dokunduğunuzda, her erişimin maliyeti birdenbire bir TLB'nin kaçırılmasına neden olacaktır.

Bir dizide bir kez döngü yapmak için temel kod şöyle görünmelidir:

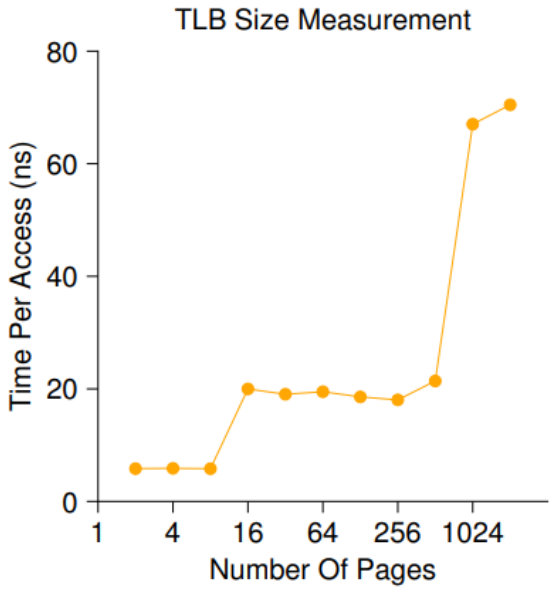
```
int jump = PAGESIZE / sizeof(int);
 for (i = 0; i < NUMPAGES * jump; i += jump) {
        a[i] += 1;
}
```

Bu döngüde, NUMPAGES tarafından belirtilen sayfa sayısına kadar a dizisinin her sayfası için bir tamsayı güncellenir. Böyle bir döngüyü art arda zamanlayarak (örneğin,

bunun etrafındaki başka bir döngüde birkaç yüz milyon kez veya birkaç saniye çalışması için birçok döngüye ihtiyaç duyulursa), her erişimin (ortalama olarak) ne kadar sürdüğünü ölçebilirsiniz. NUMPAGES arttıkça maliyetteki sıçramalara bakarak, birinci seviye TLB'nin ne kadar büyük olduğunu kabaca belirleyebilirsiniz, ikinci düzey bir TLB'nin var olup olmadığını (ve varsa ne kadar büyük olduğunu) belirleyin ve genel olarak TLB isabetlerinin ve kayıplarının performansı nasıl etkileyebileceği konusunda iyi bir fikir edinin.

İşte bir örnek grafik:

Grafikte görebileceğiniz gibi, 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 yaklaşık 1024 sayfada gerçekleşir ve bu noktada her erişim yaklaşık 70 nanosaniye sürer. Bu verilerden, iki seviyeli bir TLB hiyerarşisi olduğu sonucuna varabiliriz; ilki oldukça küçüktür (muhtemelen 8 ila 16 giriş tutar); ikincisi daha büyük ama daha yavaştır (yaklaşık 512 giriş tutar). Birinci seviye TLB'deki isabetler ile ıskalamalar arasındaki genel fark oldukça büyüktür, kabaca on dört kattır. TLB performansı önemlidir!



Şekil 19.5: TLB Boyutlarını ve Kaçırılan Maliyetleri Keşfetme

Sorular

•1-) Zamanlama için, gettimeofday() tarafından sunulana benzer bir zamanlayıcı kullanmanız gerekir. Böyle bir zamanlayıcı ne kadar hassastır? Tam olarak zamanlamanız için bir operasyonun ne kadar sürmesi gerekir? (bu, başarılı bir şekilde zamanlamak için bir sayfa erişimini bir döngüde kaç kez tekrarlamanız gerekeceğini belirlemenize yardımcı olacaktır)

- gettimeofday() fonksiyonu, bir işletim sisteminde geçen zamanı nanosaniye düzeyinde ölçebilir. Örneğin, bir işletim sisteminde bir işlemi yaparken, gettimeofday() fonksiyonu kullanarak bu işlemin ne kadar sürdüğünü nanosaniye düzeyinde ölçebilirsiniz. Ancak, bu işlemin ne kadar hassas bir şekilde ölçülebileceği, işletim sistemi ve makinenin özelliklerine bağlı olarak değişebilir.

• 2-) Her bir sayfaya erişim maliyetini kabaca ölçebilen tlb.c adlı programı yazın. Programın girdileri şunlar olmalıdır: dokunulacak sayfa sayısı ve deneme sayısı.

tlb.c

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

int main(int argc, char *argv[])
{
    if (argc < 3) {
        printf("Kullanım: %s <sayfa_sayisi> <deneme_sayisi>\n", argv[0]);
        return 1;
    }

    int page_count = atoi(argv[1]);
    int trial_count = atoi(argv[2]);

    int *pages = malloc(page_count * sizeof(int));
    if (pages == NULL) {
        printf("Hata: yeterli hafıza yok!\n");
        return 1;
    }

    srand(time(NULL));

    int i, j;
    double total_time = 0;

    for (i = 0; i < trial_count; i++) {
        for (j = 0; j < page_count; j++) {
            pages[j] = rand();
        }

        clock_t start = clock();

        for (j = 0; j < page_count; j++) {
            pages[j] *= 2;
        }

        clock_t end = clock();

        total_time += (double)(end - start) / CLOCKS_PER_SEC;
    }

    double avg_time = total_time / trial_count;

    printf("Ortalama zaman: %f saniye\n", avg_time);

    free(pages);

    return 0;
}
```

```
enes@enes-laptop:~$ cd Desktop/
enes@enes-laptop:~/Desktop$ cd tlb/
enes@enes-laptop:~/Desktop/tlb$ gcc tlb.c -o tlb
enes@enes-laptop:~/Desktop/tlb$ ./tlb 100 1000
Ortalama zaman: 0.000001 saniye
enes@enes-laptop:~/Desktop/tlb$
enes@enes-laptop:~/Desktop/tlb$
```

• 3-) Şimdi bu programı çalıştırmak için en sevdiğiniz betik dilinde (csh, python, vb.) bir betik yazın ve erişilen sayfa sayısını 1'den birkaç bine kadar değiştirin, belki yineleme başına iki kat artırın. Komut dosyasını farklı makinelerde çalıştırın ve bazı veriler toplayın. Güvenilir ölçümler elde etmek için kaç deneme gerekiyor?

-

```python
# Python kütüphanesini import et
import subprocess

# Sayfa sayısı, deneme sayısı ve artış faktörlerini ayarla
num_pages = 1
num_trials = 100
num_pages_increment = 2

# Sayfa ve deneme sayısını artıran şekilde tlb.c programını çalıştır
while num_pages <= 1000 and num_trials <= 10000:
  # Belirtilen deneme sayısı kadar tlb.c programını çalıştır
  for i in range(num_trials):
    command = "/home/enes/Desktop/tlbb/tlb.c {} {}".format(num_pages, num_trials)
    result = subprocess.run(command, shell=True)
    print("Number of pages: {}\tNumber of trials: {}\tResult: {}".format(num_pages, num_trials, result))

  num_pages *= num_pages_increment
```

```
Number of pages: 256	Number of trials: 100	Result: CompletedProcess(args='/home/enes/Desktop/tlbb/tlb.c 256 100', returncode=126)
Number of pages: 256	Number of trials: 100	Result: CompletedProcess(args='/home/enes/Desktop/tlbb/tlb.c 256 100', returncode=126)
Number of pages: 256	Number of trials: 100	Result: CompletedProcess(args='/home/enes/Desktop/tlbb/tlb.c 256 100', returncode=126)
Number of pages: 256	Number of trials: 100	Result: CompletedProcess(args='/home/enes/Desktop/tlbb/tlb.c 256 100', returncode=126)
Number of pages: 256	Number of trials: 100	Result: CompletedProcess(args='/home/enes/Desktop/tlbb/tlb.c 256 100', returncode=126)
Number of pages: 256	Number of trials: 100	Result: CompletedProcess(args='/home/enes/Desktop/tlbb/tlb.c 256 100', returncode=126)
Number of pages: 256	Number of trials: 100	Result: CompletedProcess(args='/home/enes/Desktop/tlbb/tlb.c 256 100', returncode=126)
Number of pages: 256	Number of trials: 100	Result: CompletedProcess(args='/home/enes/Desktop/tlbb/tlb.c 256 100', returncode=126)
Number of pages: 256	Number of trials: 100	Result: CompletedProcess(args='/home/enes/Desktop/tlbb/tlb.c 256 100', returncode=126)
Number of pages: 256	Number of trials: 100	Result: CompletedProcess(args='/home/enes/Desktop/tlbb/tlb.c 256 100', returncode=126)
Number of pages: 256	Number of trials: 100	Result: CompletedProcess(args='/home/enes/Desktop/tlbb/tlb.c 256 100', returncode=126)
Number of pages: 256	Number of trials: 100	Result: CompletedProcess(args='/home/enes/Desktop/tlbb/tlb.c 256 100', returncode=126)
Number of pages: 256	Number of trials: 100	Result: CompletedProcess(args='/home/enes/Desktop/tlbb/tlb.c 256 100', returncode=126)
Number of pages: 512	Number of trials: 100	Result: CompletedProcess(args='/home/enes/Desktop/tlbb/tlb.c 512 100', returncode=126)
Number of pages: 512	Number of trials: 100	Result: CompletedProcess(args='/home/enes/Desktop/tlbb/tlb.c 512 100', returncode=126)
Number of pages: 512	Number of trials: 100	Result: CompletedProcess(args='/home/enes/Desktop/tlbb/tlb.c 512 100', returncode=126)
Number of pages: 512	Number of trials: 100	Result: CompletedProcess(args='/home/enes/Desktop/tlbb/tlb.c 512 100', returncode=126)
Number of pages: 512	Number of trials: 100	Result: CompletedProcess(args='/home/enes/Desktop/tlbb/tlb.c 512 100', returncode=126)
Number of pages: 512	Number of trials: 100	Result: CompletedProcess(args='/home/enes/Desktop/tlbb/tlb.c 512 100', returncode=126)
Number of pages: 512	Number of trials: 100	Result: CompletedProcess(args='/home/enes/Desktop/tlbb/tlb.c 512 100', returncode=126)
Number of pages: 512	Number of trials: 100	Result: CompletedProcess(args='/home/enes/Desktop/tlbb/tlb.c 512 100', returncode=126)
```

100000 deneme yaparak oldukça istikrarlı bir sonuç elde edilebilir ancak daha fazla veri toplandığında daha kesin bir sonuca ulaşılabilir.

• 4-) Ardından, yukarıdakine benzer bir grafik oluşturarak sonuçları çizin. Ploticus gibi iyi bir araç kullanın. Görselleştirme genellikle verileri sindirmeyi çok daha kolaylaştırır; Neden bu olduğunu düşünüyorsun?

• 5-) Dikkat edilmesi gereken bir şey derleyici optimizasyonudur. Derleyiciler, programın başka hiçbir bölümünün kullanmadığı değerleri artıran döngüleri kaldırmak da dahil olmak

üzere her türlü zekice şeyi yapar. Derleyicinin yukarıdaki ana döngüyü TLB boyut tahmincinizden çıkarmamasını nasıl sağlayabilirsiniz?

- Bir derleyici optimizasyonunun yapılmamasını sağlamak için derleyiciye aşağıdaki gibi bir seçenek belirtebilirsiniz:

```
gcc -O0 tlb.c -o tlb
```

gcc -O0 tlb.c -o tlb

Bu seçenek, derleyiciye optimizasyon yapmamasını söyler ve böylece derleyici programdaki döngüleri kaldırmaz. Bu sayede, tlb boyutu tahmini yapılan döngü kaldırılmaz ve program doğru bir şekilde çalışır. Optimizasyon seçeneğini kullanmak programınızın daha yavaş çalışacağını unutmayın. Ancak bu seçenek, programınızın doğru bir şekilde çalıştığından emin olmak için kullanılabilir

```
enes@enes-laptop:~/Desktop/tlbb$ gcc -O0 tlb.c -o tlb
enes@enes-laptop:~/Desktop/tlbb$ ./tlb 100 1000
Ortalama zaman: 0.000001 saniye
enes@enes-laptop:~/Desktop/tlbb$
```

• 6-) Dikkat edilmesi gereken başka bir şey de, günümüzde çoğu sistemin birden fazla CPU ile gönderildiği ve elbette her CPU'nun kendi TLB hiyerarşisine sahip olduğu gerçeğidir. Gerçekten iyi ölçümler elde etmek için, programlayıcının kodu bir CPU'dan diğerine atlamasına izin vermek yerine, kodunuzu yalnızca bir CPU'da çalıştırmanız gerekir. Nasıl yaparsın? (ipucu: bazı ipuçları için Google'da "pinning a thread" konusuna bakın) Bunu yapmazsanız ve kod bir CPU'dan diğerine geçerse ne olur?

- Çoklu işlemciye sahip sistemlerde, her bir işlemci kendi TLB hiyerarşisine sahiptir. Bu nedenle, doğru ölçümler elde etmek için kodunuzu sadece bir işlemcide çalıştırmanız gerekir. Bu, işletim sistemi tarafından işlemciler arasında yönlendirilmesini engelleyerek yapılabilir. Bu işlemi yapmak için aşağıdaki adımları izleyebilirsiniz:

1-)İlk olarak, programınızı çalıştırmadan önce bir thread oluşturun.

```
import threading

def run_program():
    # Programı burada çalıştırın

thread = threading.Thread(target=run_program)
```

2-)Thread'i oluşturduktan sonra, thread'i belirli bir işlemciye bağlayın. Örneğin, aşağıdaki Python kodu ile thread'i CPU 0'a bağlayabilirsiniz:

```
import os

# Thread'i CPU 0'a bağlayın
os.sched_setaffinity(thread.ident, {0})
```

3-) Thread'i bağladıktan sonra, thread'i başlatabilirsiniz:

```
thread.start()
```

Eğer kodunuzu bir işlemciden diğerine hareket ettirirseniz, ölçümlerinizin doğruluğu ve güvenilirliği sorgulanabilir hale gelebilir. Bunun nedeni, her bir işlemcinin kendi TLB hiyerarşisi olduğu için, kodun bir işlemciden diğerine hareket ettiğinde, TLB hiyerarşisi de değişebilir. Bu da, elde edilen ölçümlerin doğruluğunu ve güvenilirliğini sorgulanabilir hale getirebilir. Bu nedenle, kodunuzu bir işlemciden diğerine hareket ettirmeyin ve bir işlemcide çalıştırın. Bu sayede, elde edeceğiniz ölçümlerin doğruluğu ve güvenilirliği artacaktır.

• 7-) Ortaya çıkabilecek başka bir sorun, başlatmayla ilgilidir. Diziye erişmeden önce yukarıdaki a'yı başlatmazsanız, talebin sıfırlanması 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?

-Eğer bir diziyi erişmeden önce başlatmazsanız, bu diziye ilk kez erişim çok pahalı olabilir. Bu durum, dizinin ilk kez erişim sırasında demand zeroing gibi ilk erişim maliyetlerinden kaynaklanabilir. Bu, kodunuzu ve zamanlama bilgilerini etkileyebilir. Örneğin, dizinin ilk kez erişimi sırasında diziye eklenen maliyetler, zamanlama bilgilerinizde yanlış sonuçlar doğurabilir. Bu potansiyel maliyetleri karşılamak için, dizinin başlatılmasını sağlamalısınız. Örneğin, diziyi başlatmak için calloc() gibi bir fonksiyon kullanabilirsiniz. Bu sayede diziye ilk kez erişim sırasında maliyetler oluşmayacak ve zamanlama bilgileriniz daha doğru olacaktır.