19

Sayfalama : Daha Hızlı Çeviriler (Etkin Sayfalar Ön Belleği)

Disk belleğini sanal belleği desteklemek için çekirdek mekanizma olarak kullanmak, yüksek performans harcamalarına yol açabilir. Adres alanını küçük, sabit boyutlu birimlere (yani sayfalara) bölmek,büyük miktarda eşleme bilgisi gerekir. Çünkü haritalama bilgisi genellikle fiziksel hafızada saklanır, haritalama mantıksal olarak fazladan hafıza için program tarafından oluşturulan adrese bakar. Her talimat getirmeden veya açık yüklemeden veya depolamadan önce çeviri bilgileri için belleğe gitmek sistemi çok yavaşlatır. Sorunumuz şu ki:

Asıl Nokta:
Adres Çevirisini Nasıl Hızladırırız

Adres çevirisini nasıl hızlandırabiliriz ve sayfalamanın gerektirdiği ek bellek referansından nasıl kaçabiliriz? Hangi Donanım Desteği Gereklidir ? Hangi işletim sistemi kalıtımı gerekir ?

İşleri hızlandırmak istediğimizde, işletim sisteminin genellikle biraz yardıma ihtiyacı vardır. Ve yardım genellikle işletim sisteminin eski dostundan gelir: Donanım. Adres çevirisini hızlandırmak için (tarihsel nedenlerden dolayı [CP78]) bir **etkin sayfalar önbelleği(Translations lookaside buffer)** eklemeliyiz ya da **TLB** [CG68, C95]. TLB ise **Bellek Yönetim Birimi(Memory Management Unit(MMU))'nin bir parçasıdır**, ve sanaldan fiziksele adres çevirilerinin popüler bir donanım **önbelleğidir**; bu nedenle,**adres çeviri önbelleği daha** iyi bir isimlendirme olacaktır. Her bir sanal bellek referansında, donanım istenen çevirinin orada tutulup tutulmadığını görmek için önce TLB'yi kontrol eder; Eğer orada ise, çeviri (hızlı bir şekilde) sayfa tablosuna (tüm çevirileri içerir) başvurmak zorunda kalmadan gerçekleşir. Muazzam performansı nedeniyle, TLB'ler gerçek anlamda sanal bellek kadar performans mümkün kılar [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'nin Kontrol Akış Şeması

19.1 TLB'nin Temel Algoritması

Şekil 19.1, donanımın bir sanal adres çevirisini nasıl işleyebileceğinin kabaca bir taslağını gösterir, basit bir **doğrusal sayfa tablosu** (örneğin, sayfa tablosu bir dizidir) ve **donanım tarafından yönetilen bir TLB** (yani donanım, sayfa tablosu erişimlerinin sorumluluğunun çoğunu üstlenir; aşağıda bununla ilgili daha fazla açıklama yapacağız).

Donanımın izlediği algoritma şu şekilde çalışır: önce sanal adresten sanal sayfa numarasını (VPN) çıkarır (Şekil 19.1'deki Satır 1), ve TLB'nin bu VPN için çeviriyi elinde tutup tutmadığını kontrol eder (Satır 2). Eğer varsa, bir **TLB bilgisi** vardır , bu da TLB'nin çeviriyi elinde tuttuğu anlamına gelir.. Başarılı! Artık sayfa çerçeve numarasını (Page Frame Number) ilgili TLB girişinden çıkarabiliriz, offset üzerinden orijinal sanal adrese bağlanır,ve istenen fiziksel adresi oluşturur(physical address) ve belleğe erişim (Satır 5–7), korumanın başarısız olmadığını varsayarak kontrol eder(Satır 4).

CPU çeviriyi TLB'de bulamazsa (**TLB ıskalaması**), daha yapacak çok işimiz var. Bu örnekte, donanım çeviriyi bulmak için sayfa tablosuna erişir (Satır 11–12), ve, süreç tarafından oluşturulan sanal bellek referansının geçerli ve erişilebilir olduğu varsayılarak (Satır 13, 15), TLB'yi çeviriyle günceller (Satır 18). Bu eylemler dizisi maliyetlidir. Öncelikle sayfa tablosuna (Satır 12) erişmek için ekstra bellek referansına ihtiyacı vardır. Son olarak,bir kere TLB güncellendiğinde donanım komutu yeniden dener. bu kez çeviri TLB'de bulunur ve bellek referansı hızlı bir şekilde işlenir.

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

19.2 Örnek: Bir Diziye Erişim

Bir TLB'nin çalışmasını garantilemek için, basit bir sanal adres izlemeyi inceleyelim ve bir TLB'nin performansını nasıl iyileştirebileceğini görelim. Bu örnekte, bellekte 10 adet 4 baytlık bir tamsayı dizimiz olduğunu varsayalım, sanal adres 100'den başlayarak Küçük bir 8 bitlik sanal adres alanımız olduğunu varsayalım, 16 baytlık sayfalarla; bir sanal adres 4-bit VPN (16 sanal sayfa vardır) ve 4-bit ofset (bu sayfaların her birinde 16 bayt vardır) olarak bölünür.

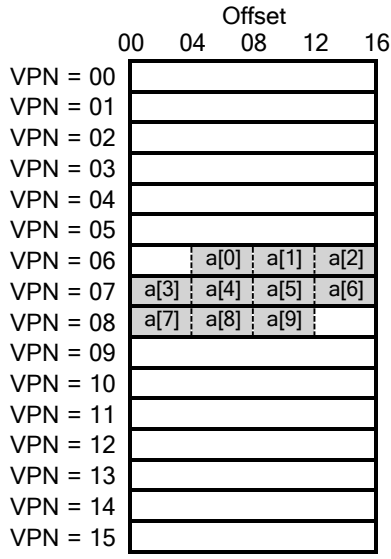
Şekil 19.2 (sayfa 4), sistemin 16 adet 16 baytlık sayfasında düzenlenen diziyi göstermektedir. Gördüğünüz gibi dizinin ilk girişi (a[0]) (VPN=06, offset=04) ile başlıyor; o sayfaya yalnızca üç adet 4 baytlık tamsayı sığar. Dizi bir sonraki sayfada devam eder (VPN=07), sonraki dört girişin (a[3] ... a[6])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

Şimdi her bir dizi öğesine erişen basit bir döngü düşünelim, C'de bu şekilde görünür:

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

Basitlik adına, yalnızca belleğin eriştiği gibi davranacağız. Bu döngü şöyle bir dizi oluşturur (i ve sum değişkenlerinin yanı sıra talimatların kendileri göz ardı edilerek). İlk dizi elemanına (a[0])a erişildiğinde, CPU, sanal adres 100'de bir yük görecektir. Donanım, VPN'yi bundan çıkarır (VPN=06), ve geçerli bir çeviri için TLB'yi kontrol etmek için bunu kullanır. Programın diziye ilk kez eriştiği varsayılırsa, sonuç bir TLB ıskalaması olacaktır.

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



Şekil 19.2: Örnek: Küçücük Bir Adres Alanındaki Bir Dizi

TLB'nin içinden dolayı başarımızın nedeni böyledir. a[2] erişimi benzer bir başarı ile karşılaşır (başka bir isabet), çünkü o da a[0] ve a[1] ile aynı sayfayı paylaşıyor.

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

Son olarak, a[7]'ye erişim, son bir TLB'nin ıskalamasına 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; donanım, çevirileri için TLB'ye baktığında, iki vurgun daha elde edilir.

Diziye on erişimimiz sırasındaki TLB etkinliğini özetleyelim: **ıskala**, isabet, isabet, **ıskala**, isabet, isabet, isabet, **ıskala**, isabet, isabet. Böylece, isabet sayısının toplam erişim sayısına bölünmesiyle elde edilen **TLB isabet** oranımız %70. Bu çok yüksek olmamakla birlikte (aslında %100'e yaklaşan isabet oranlarını arzu ediyoruz),en azından 0 değil, böyle bir durum sürpriz olabilirdi. Programın diziye ilk erişmesine rağmen, TLB, **mekansal konum** nedeniyle genel performansı artırır. Dizinin öğeleri, sayfalar halinde sıkıca paketlenir (yani, uzayda birbirlerine yakındırlar), ve bu nedenle, bir sayfadaki bir öğeye yalnızca ilk erişimin, TLB'nin ıskalamasından nasibini alır. Ayrıca bu örnekte sayfa boyutunun oynadığı role dikkat edin. Eğer sayfa boyutu daha büyük olsaydı bu isabet oranı daha yüksek olurdu ya da daha Küçük olsaydı isabet oranı düşerdi.

İPUCU: MÜMKÜN OLDUĞUNDA ÖNBELLEKLEMEYİ KULLANIN
Önbelleğe alma, bilgisayar sistemlerindeki en temel performans tekniklerinden biridir, "sıradan durumu hızlı" hale getirmek için tekrar tekrar kullanılan tekniklerden bir tanesi [HP06]. Donanım önbelleklerinin arkasındaki fikir, talimat ve veri referanslarında **konumdan** yararlanmaktır. Genellikle iki tür konum vardır: **zamansal konum** ve **mekansal konum**. Zamansal konum 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. Mekansal konum ile ilgili, fikir şu ki, eğer bir program x adresindeki belleğe erişirse, muhtemelen x belleğine yakında birdaha erişebilir. Burada bir tür diziden akış yaptığınızı, bir öğeye ve ardından diğerine eriştiğinizi hayal edin. Tabii ki, bu özellikler programın doğasına bağlıdır, ve bu nedenle katı ve hızlı yasalar değil, daha çok pratik kurallara benzer. Talimatlar, veriler veya adres çevirileri için (TLB'mizde olduğu gibi) donanım önbellekleri, belleğin kopyalarını küçük, hızlı çip üzerinde bellekte tutarak konumdan yararlanır.Talebi karşılamak için yavaş bir hafızaya gitmek yerine işlemci önce bir önbellekte yakındaki bir kopyanın olup olmadığını kontrol eder; eğer oradaysa, işlemci ona hızlı bir şekilde erişebilir (yani, birkaç CPU döngüsünde) 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 başka 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 hızlı olma amacını bozar. Bu nedenle, küçük, hızlı önbelleklerle sıkışıp kaldık; Geriye kalan soru, performansı artırmak için bunları en iyi nasıl kullanacağımızdır.

basitçe 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, her sayfa erişimde yalnızca tek bir kayıpla karşılaşarak mükemmel TLB performansı sağlar. TLB Performansı Hakkında Son Birşey: program, bu döngü tamamlandıktan hemen sonra diziye tekrar erişirse, gerekli çevirileri önbelleğe alacak kadar büyük bir TLB'ye sahip olduğumuzu varsayarsak, muhtemelen daha da iyi bir sonuç görürüz: isabet, isabet, isabet, isabet, isabet, isabet, isabet, isabet, isabet, isabet. Bu durumda TLB isabet oranı, **zamansal konum** nedeniyle, yani bellek öğelerinin **zaman** içinde hızlı bir şekilde yeniden referanslandırılması nedeniyle yüksek olacaktır. 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.

```
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)
```

19.3 Şekil 19.3: TLB Kontrol Akış Algoritması (OS Tarafından Yönetilen)

19.4 TLB'nin Iskalamasını Kim Halleder ?

Cevaplamamız gereken bir soru: TLB'nin ıskalamasını kim halleder ? İki cevap ile mümkündür : Donanım ya da Yazılım (OS). Önceki zamanlarda donanım karmaşık komut setlerine sahipti(buna **CISC** denildi , **karmaşık komut setli bilgisayalar**) ve donanımı oluşturan insanlar, o sinsi işletim sistemi çalışanlarına pek güvenmiyordu. Donanım, TLB eksikliğini tamamen giderebilir. Bunu yapmak için, donanımın sayfa tablolarının bellekte tam olarak nerede bulunduğunu bilmesi gerekir. (Şekil 19.1'de Satır 11'de kullanılan bir **sayfa tablosu temel kaydı** aracılığıyla), kesin biçimlerinin yanı sıra; ıskalama durumunda, donanım sayfa tablosunda "yürür", doğru sayfa tablosu girişini bulur ve istenen çeviriyi çıkarır, TLB'yi çeviriyle günceller ve talimatı yeniden dener. **Donanım tarafından yönetilen TLB'lere** sahip "eski" bir mimari örneği, Intel x86 mimarisidir. Sabit bir **çok düzeyli sayfa tablosu** kullananlar (ayrıntılar için bir sonraki bölüme bakın); geçerli sayfa tablosu CR3 kaydı [I09] tarafından işaret edilir. Daha modern mimariler (ör. MIPS R10k [H93] veya Sun's SPARC v9 [WG00], hem **RISC** hem de azaltılmış komut seti bilgisayarları), **yazılım tarafından yönetilen TLB** olarak bilinen şeye sahiptir. Bir TLB ıskalamasında, donanım basitçe bir istisna oluşturur (Şekil 19.3'teki 11. satır), mevcut talimat akışını duraklatan, ayrıcalık seviyesini çekirdek moduna yükselten ve bir **tuzak işleyicisine** atar. Tahmin edebileceğiniz gibi, bu tuzak işleyici, TLB ıskalamalarını işlemek amacıyla yazılmış işletim sistemi içindeki koddur. Çalıştırıldığında, kod 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 prosedür çağrısından hemen sonra talimata geri dönmesi gibi, tuzaktan dönüş OS'de tuzaktan sonra talimatta yürütmeye devam etmelidir.Önceki durumda, bir TLB ıskalama tuzağından dönerken, donanım, tuzağa neden olan talimatta yürütmeye devam etmelidir; bu yeniden deneme böylece talimatın tekrar çalışmasına izin verir, bu sefer bir TLB isabetiyle sonuçlanır.

RISC, CISC Karşı Karşıya

1980'lerde bilgisayar mimarisi camiasında büyük bir savaş yaşandı. Bir tarafta, **Karmaşık Komut Seti Hesaplamanın** kısaltması olan **CISC** kampı vardı; diğer tarafta, **Azaltılmış Komut Seti Hesaplama** [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 erkenden çalışması nedeniyle bir Turing ödülü ile tanındı [CM00]. CISC komut setlerinde çok sayıda talimat bulunur ve neredeyse her talimat 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 kompakt hale getirmek için talimatların ilkel olması gerektiğiydi. RISC komut setleri tam tersidir. RISC'nin arkasındaki önemli bir gözlem, komut setlerinin gerçekten derleyici hedefleri olduğu ve derleyicilerin gerçekten istediği tek şeyin, yüksek performanslı kod oluşturmak için kullanabilecekleri birkaç basit ilkel kod 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 istiyorlardı.

Başlarda, RISC çipleri fark edilir derecede daha hızlı oldukları için büyük bir etki yarattı [BC91]; birçok makale yazıldı; birkaç şirket kuruldu (örneğin, MIPS ve Sun). Bununla birlikte, zaman geçtikçe, Intel gibi CISC üreticileri, örneğin karmaşık talimatları daha sonra RISC benzeri bir şekilde işlenebilen mikro talimatlara dönüştüren öncelikli veriyolu hattı aşamaları ekleyerek, birçok RISC tekniğini işlemcilerinin çekirdeğine dahil ettiler. 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 yapılıyor.

Bu nedenle, bir tuzağın veya istisnanın nasıl oluşturulduğuna bağlı olarak, işletim sistemine yönlendirme sırasında donanımın zamanı geldiğinde düzgün bir şekilde 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 hata işleyicilerini fiziksel bellekte tutabilirsiniz (burada eşlenmemişler ve adres çevirisine tabi değiller) veya TLB'deki bazı girişleri kalıcı olarak geçerli çeviriler için ayırabilir ve bu kalıcı çevirilerden bazılarını kullanabilirsiniz. işleyici kodunun kendisi için yuvalar; bu şifreli çeviriler her zaman TLB'de görülür.

Yazılımla yönetilen yaklaşımın birincil avantajı esnekliktir: İşletim sistemi, donanım değişikliği gerektirmeden sayfa tablosunu uygulamak için istediği herhangi bir veri yapısını kullanabilir.

TLB GEÇERLİ BİTİ/= SAYFA TABLOSUNUN GEÇERLİ BİTİ
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 doğru 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 yüklenir 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 işlemden sanaldan fiziksele çeviriyi kullanmamasını sağlayabilir.

Diğer bir avantaj, TLB kontrol akışında görüldüğü gibi basitliktir (Şekil 19.1'deki 11–19 satırlarının aksine Şekil 19.3'teki 11. satır). Donanım ıskalama durumunda fazla bir şey yapmaz: sadece bir istisna oluşturur ve OS TLB'nin ıskalama işleyicisinin gerisini halletmesine izin verir.

19.5 TLB Hakkında : İçinde Neler Vardır ?

Donanım TLB'sinin detaylarına bakalım. Tipik bir TLB'nin 32, 64 veya 128 girişi olabilir ve **tamamen çağrışımsal** 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. Bir TLB girişi şöyle görünebilir:

VPN | PFN | diğer bitler

Bir çeviri bu konumlardan herhangi birinde sona erebileceğinden (donanım açısından TLB, tam olarak **çağrışımsal** 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 ilginçleri "Diğer Bitler". Örneğin, TLB genellikle girişin geçerli bir çevirisi olup olmadığını söyleyen **geçerli** bir bit içerir. Ayrıca bir sayfaya nasıl erişilebileceğini belirleyen (sayfa tablosundaki gibi) **koruma bitleri** de vardır. Örneğin, kod sayfaları okundu ve yürütüldü olarak işaretlenebilirken yığın sayfaları okundu ve yazıldı olarak işaretlenebilir. Bir **adres alanı tanımlayıcısı,** bir **kirli bit** vb. dahil olmak üzere birkaç başka alan da olabilir; daha fazla bilgi için aşağıya bakın.

19.6 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 alınan çevirileri yanlışlıkla kullanmadığından emin olmak için dikkatli olmalıdır. Bu durumu daha iyi anlamak için örneğe bakalım: Birinci Süreç (P1) çalıştığında, TLB'nin kendisi için geçerli olan, yani P1'in sayfa tablosundan gelen çevirileri önbelleğe alıyor olabileceğini varsayar. Bu örnek için, 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 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 tercüme edilir, ancak donanım hangi girişin hangi işlem için olduğunu ayırt edemez. Bu nedenle, TLB'nin sanallaştırmayı birden çok süreçte doğru ve verimli bir şekilde desteklemesi için biraz daha çalışmamız gerekiyor. Ve böylece, bir önemli nokta:

ÖNEMLİ NOKTA:
BİR İÇERİK ANAHTARINDA TLB İÇERİĞİ NASIL YÖNETİLİR ?
İşlemler arasında dizi 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. Bir yaklaşım, TLB'yi bağlam anahtarlarında basitçe temizlemek ve böylece bir sonraki işlemi çalıştırmadan önce onu **boşaltmaktır.** Yazılım tabanlı bir sistemde bu, açık (ve ayrıcalıklı) bir donanım talimatı ile 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 (ASID)** alanı sağlar. ASID'i bir **süreç tanımlayıcısı (PID)** olarak düşünebilirsiniz, ancak genellikle daha az bit içerir (örneğin, ASID için 8 bit, PID için 32 bit). Yukarıdan örnek TLB'mizi alıp ASID'leri eklersek, süreçlerin TLB'yi kolayca paylaşabileceği açıktır: aksi halde aynı olan çevirileri ayırt etmek için yalnızca ASID alanı gereklidir. 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, farklı işlemlerden gelen çevirileri aynı anda herhangi bir karışıklık olmadan tutabilir. Elbette donanımın çevirileri gerçekleştirmek için o anda hangi işlemin çalıştığını bilmesi gerekir ve bu nedenle işletim sisteminin bir bağlam anahtarında geçerli işlemin ASID'ine bazı ayrıcalıklı kayıtlar ayarlaması gerekir. 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.

19.7 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 yaparken, eskisini **değiştirmeliyiz** ve dolayısıyla şu soru: Hangisini değiştirmeliyiz?

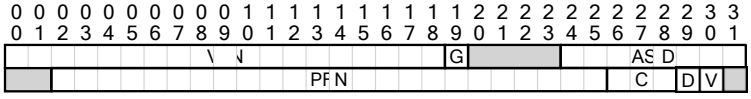
ÖNEMLİ NOKTA: TLB DEĞİŞTİRME POLİTİKASINI NASIL YAPARIZ ?

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

Sayfaları diske aktarma sorununu ele alırken bu tür politikaları biraz ayrıntılı olarak üstünde çalışacağız; burada sadece birkaç tipik politikayı vurgulayacağız. **Yaygın bir yaklaşım**, en son kullanılan veya **LRU** girişini çıkarmaktır. LRU, yakın zamanda kullanılmamış bir girişin tahliye için iyi bir aday olabileceğini varsayarak, bellek referans akışındaki yerin avantajlarından yararlanmaya çalışır. Başka bir yaygın yaklaşım, **rastgele** bir TLB eşlemesini çıkaran rastgele bir politika kullanmaktır. Böyle bir politika, basitliği ve köşe durum 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 olaylarda çok daha iyi sonuç verir.

19.8 Gerçek Bir TLB Girişi

Son olarak, gerçek bir TLB'ye bakalım. Bu örnek, yazılımla yönetilen TLB'leri kullanan modern bir sistem olan MIPS R4000'den [H93] alınmıştır; biraz basitleştirilmiş bir MIPS TLB girişi 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 beklerdik. Ancak TLB'de görebileceğiniz gibi VPN için sadece 19 bit var; Görünen o ki, kullanıcı adresleri adres alanının yalnızca yarısından gelecek (geri kalanı çekirdek için ayrılmış) ve bu nedenle yalnızca 19 bit VPN olması lazım. VPN, 24 bit fiziksel çerçeve numarasına (PFN) kadar çevirir ve dolayısıyla 64 GB'a kadar (fiziksel) ana belleğe izin verir 24(2 4KB sayfa). 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 (yukarıda açıklandığı gibi) 8 bitlik ASID'i de görüyoruz.



Şekil 19.4: **Bir MIPS TLB Girişi**

İPUCU: RAM HER ZAMAN RAM DEĞİLDİR (CULLER'S YASASI)
Rastgele erişim belleği veya **RAM** terimi, RAM'in herhangi bir bölümüne bir diğeri kadar hızlı erişebileceğiniz anlamına gelir. RAM'i bu şekilde düşünmek genellikle iyi olsa da, TLB gibi donanım/işletim sistemi özellikleri nedeniyle, belirli bir bellek sayfasına erişmek, özellikle de o sayfa şu anda TLB'niz tarafından eşlenmemişse, maliyetli olabilir. Bu nedenle, şu ipucunu hatırlamak her zaman iyidir: **RAM her zaman RAM değildir.** Bazen adres alanınıza rastgele erişerek, özellikle erişilen sayfa sayısı TLB kapsamını aşarsa, ciddi performans sorunlarına yol açabilir. Danışmanlarımızdan biri olan David Culler, birçok performans sorununun kaynağı olarak her zaman TLB'yi gösterdiği için, bu yasayı onun onuruyla adlandırıyoruz: **Culler Yasası**.

Size bir soru: Aynı anda çalışan 256'dan (2) fazla işlem varsa işletim sistemi ne yapmalıdır? Şöyle; 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 göreceğiz); girişte geçerli bir çeviri olup olmadığını donanıma bildiren geçerli bir bit var. Birden çok sayfa boyutunu destekleyen bir sayfa maskesi alanı da (gösterilmeyen) 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 ıskalamasının sorunlu olacağı durumlarda kullanmak için ayrıldı(örneğin, TLB kaybı işleyicisinde). MIPS TLB yazılım tarafından yönetildiğinden, TLB'yi güncellemek için talimatlar 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** 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).

Özet

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

Ancak, TLB'ler var olan her program için dünyayı güzelleştirmez. Özellikle, bir programın kısa sürede eriştiği sayfa sayısı, TLB'ye sığan sayfa sayısını aşarsa, program çok sayıda TLB ıskalaması üretecek ve bu nedenle oldukça yavaş çalışacaktır. Bu olguyu **TLB kapsamının 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 (DBMS)** gibi programlar tarafından sıklıkla istismar edilir.

Bahsetmeye değer başka bir TLB sorunu: TLB erişimi, özellikle **fiziksel olarak indekslenmiş önbellek** olarak adlandırılan şeyle, CPU ardışık düzeninde kolayca bir darboğaza dönüşebilir. Böyle bir önbellekte, önbelleğe erişilmeden önce adres çevirisinin yapılması gerekir, bu da işleri biraz yavaşlatabilir. Bu olası sorun nedeniyle, insanlar sanal adreslerle önbelleklere erişmenin her türlü akıllı yolunu aradılar ve böylece bir önbellek isabeti durumunda pahalı çeviri adımından kaçındılar. Bir tür bir **sanal dizinlenmiş önbellek**, bazı performans sorunlarını çözer, ancak donanım tasarımına da yeni sorunlar getirir. Daha fazla ayrıntı için Wiggins'in güzel anketine bakın [W03].

References

[BC91] "Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware 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 Patterson. 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. Available: http://www.intel.com/products/processor/manuals. *In particular, pay attention to "Volume 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. 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 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 (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 bir 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 deneyin amaçları açısından yararlı olacaktır). 4 veya daha az sayfaya etki eden bir program yazarsanız, her erişim bir TLB isabeti olmalı ve bu nedenle nispeten hızlı olmalıdır. Bununla birlikte, bir döngüde tekrarlanan 5 veya daha fazla sayfaya etki ettiğinizde, her erişimin maliyeti birdenbire bir TLB'nin ıskalamasına neden olacaktır.

Bir dizide bir döngü yapmak için temel kod budur:

```
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 sayı 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 sıçramaları arayarak, kabaca birinci düzey TLB'nin ne kadar büyük olduğunu belirleyebilir, ikinci düzey TLB'nin var olup olmadığını (ve varsa ne kadar büyük olduğunu) belirleyebilirsiniz ve genel olarak bu konuda iyi bir fikir edinebilirsiniz. TLB isabetlerinin ve ıskalamalarının performansı nasıl etkileyebileceği gibi.

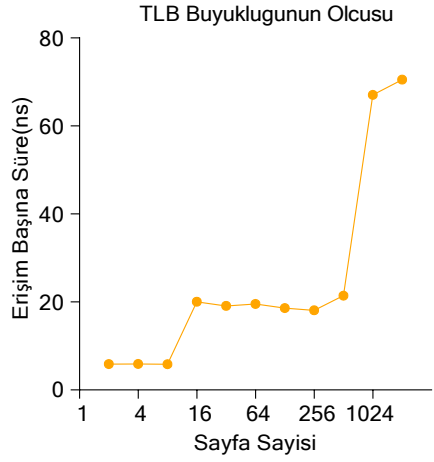


Figure 19.5: **Discovering TLB Sizes and Miss Costs**

Şekil 19.5 (sayfa 15), döngüde erişilen sayfa sayısı arttıkça erişim başına ortalama süreyi göstermektedir. Grafikte görebileceğiniz gibi, yalnızca birkaç sayfaya erişildiğinde (8 veya daha az), ortalama erişim süresi kabaca 5 nanosaniyedir.16 sayfadan fazlasına erişirse program 20 nanosaniyeye kadar çıkıyor. Son 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 büyük kabaca on dört kattır. TLB performansı önemlidir!

Sorular

1. Zamanlama için bir zamanlayıcı kullanmanız gerekir (ör. gettimeofday()). 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 gerektiğini belirlemenize yardımcı olacaktır.)

20 Çoğu sistemde, gettimeofday() birkaç mikrosaniye çözünürlüğe sahiptir. Bu, makul bir doğrulukla tamamlanması en az birkaç mikrosaniye süren işlemleri zamanlamak için gettimeofday() işlevini kullanabileceğimiz anlamına gelir. Ancak gettimeofday() gibi bir zamanlayıcının kesinliği, işletim sisteminin ve donanımın iş yükü ve sistemde çalışan diğer işlemlerin varlığı gibi çeşitli faktörlerden etkilenir. Sonuç olarak, çok kısa işlemleri zamanlamak için kullanmaya çalışmak yerine, tamamlanması en az birkaç milisaniye süren işlemleri zamanlamak için gettimeofday() gibi bir zamanlayıcı kullanmak genellikle en iyisidir. bir sayfa erişimini kaç kez tekrarlamanız gerektiğini belirlemek için, işlemin beklenen süresini ve istenen kesinlik düzeyini göz önünde bulundurmak gerekiyor. Mesela, bir sayfaya erişim süresini 1 milisaniye hassasiyetle ölçmek istiyorsanız ve sayfa erişiminin tamamlanması tahmini olarak 100 milisaniye sürüyorsa, sayfa erişimini 10 kez tekrarlayabilir ve ortalama süreyi sizin için alabilirsiniz. son ölçüm. Bu, yaklaşık %10'luk bir kesinlik ile sayfaya erişmeniz için geçen süre hakkında iyi bir tahmin verebilir.

1. Her sayfaya erişim maliyetini kabaca ölçebilen tlb.c adlı programı yazın. Programın girdileri şunlar olmalıdır: dokunulacak sayfa sayısı ve deneme sayısı.
2. Şimdi, erişilen sayfa sayısını 1'den birkaç bine kadar değiştirerek, belki yineleme başına iki kat artırarak, bu programı çalıştırmak için en sevdiğiniz dilde (eğlence?) bir dil yazın. Komut dosyasını farklı makinelerde çalıştırın ve bazı veriler toplayın. Güvenilir ölçümler elde etmek için kaç deneme gerekiyor?
3. Ardından, yukarıdakine benzer bir grafik oluşturarak sonuçları çizin. Ploticus ve hatta zplot gibi iyi bir araç kullanın. Görselleştirme genellikle verileri işlemeyi çok daha kolaylaştırır; Neden böyle olduğunu düşünüyorsun açıkla?
4. 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?

5. 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?
6. 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?
   Bu potansiyel maliyetleri dengelemek için, diziye erişmeden önce "a" dizisini başlatabilirsiniz. Bu, dizi boyunca döngü yaparak ve her öğeye bir değer atayarak veya diziyi sizin için başlatan dil veya çalışma zamanı tarafından sağlanan bir işlev kullanılarak yapılabilir. Bu, diziye erişmeye başlamadan önce dizinin düzgün bir şekilde başlatılmasını sağlar ve bu da kodunuzun performansını iyileştirmeye yardımcı olur.