Önbellek (Cache) Sistemleri

Yazan: Kaan Aslan
14 Ocak 2015

1. Giriş

Bilgisayar sistemlerinde pek çok durumda erişim hızı bakımından iki tür belleğin söz konusu olduğunu söyleyebiliriz: Yavaş bellek ve hızlı bellek. Yavaş bellek ucuz ve boldur, hızlı bellek ise pahalı ve kıttır. Tabi buradaki yavaş ve hızlı kavramları görelidir. Sistemden sisteme yavaş ve hızlı bellekler farklılaşabilir. Örneğin bir sistemde *RAM* hızlı belleği temsil ederken disk yavaş belleği temsil ediyor olabilir. Başka bir sistemde ise *RAM* yavaş belleği temsil ederken işlemci içerisinde bulunan bellek hızlı belleği temsil ediyor olabilir. Ya da bir *web* sayfasının sunucuda saklandığı disk yavaş belleği temsil ederken onun istemcide saklandığı disk hızlı belleği temsil ediyor olabilir. Önbellek (*cache*) yavaş belleğin belli bölümlerinin hızlı bellekte tutularak yavaş belleğe erişim oranını azaltmak için kullanılan bir bellek yönetim tekniğidir. Önbellek sistemleri donanımsal ya da yazılımsal bir biçimde ya da karma bir biçimde gerçekleştirilebilmektedir.

2. Önbellek Terminolojisi

Önbellek sistemlerinde bol olan yavaş belleğin belli bölümleri az olan hızlı bellekte tutulur. Bilgiye erişilmek istendiğinde bilgi önce hızlı bellekte aranır. Bulunursa hemen oradan alınır, bulunamazsa yavaş belleğe başvurulur. Aranan bilginin hızlı bellekte bulunmasına önbelleğe isabet ettirme (*cache hit*), bulunamamasına önbelleği kaçırma (*cache miss*) denilmektedir. Önbelleğe isabet ettirme oranı (*cache hit ratio*) toplam erişimlerin yüzde kaçının önbellekten karşılandığını belirtmektedir. Örneğin önbelleğe isabet ettirme oranının %70 olduğu bir sistemde ortalama 100 erişimin 70'i hiç yavaş belleğe başvurulmadan hızlı bellekten karşılanabilmektedir. Önbelleğin isabet ettirildiği durumlarda bilginin önbellekten alınma süresine önbellek gecikme zamanı (*cache latency*) denilmektedir. Önbellek gecikme zamanı önbellek için kullanılan algoritmalara ve önbellek büyüklüğüne bağlı olarak değişebilmektedir.

Önbellek sistemleri yalnızca okunabilir (*read only*) ya da hem okunabilir hem de yazılabilir (*read/write*) biçimde tasarlanabilir. Yalnızca okunabilir önbellek sistemlerinde önbelleğe bir şey yazılmaz. Yazma işlemi doğrudan yavaş belleğe yapılır. Fakat hem okunabilir hem de yazılabilir önbellek sistemlerinde önbelleğe okuma amaçlı başvurulacağı gibi yazma amaçlı da başvurulabilmektedir. Şüphesiz hem okunabilen hem de yazılabilen önbellek sistemleri toplamda daha yüksek bir performans sunarlar. Ancak güç kaynağının kesilmesi gibi bir durumda önbellekteki bilgiler hedefe yazılamamış olacağından bilginin bütünlüğü bozulabilir. Yazma işlemi sırasında önbelleğin kullanılmasına ilişkin belirlemelere *önbellek sisteminin yazma politikası* (*writing policy*) denilmektedir.

Hem okunabilir hem de yazılabilir önbellek sistemlerinde kullanılan iki temel yazma politikası vardır. Birinci yazma politikasında yazma işlemi hem önbelleğe hem de yavaş belleğe yapılır. (Bu yazma politikasına *İngilizce write-through* deniyor.) Böylece güç kaynağının kesilmesi gibi anormal durumlarda önbellekte bulunan bilginin kaybolmasının önüne geçilmiş olur. İkinci yazma politikasında ise bilgi yalnızca önbelleğe yazılır, daha sonra yavaş belleğe aktarılır. (Bu yönteme de *İngilizce write-back* ya da *write behind* denilmektedir). Pekiyi hem okunabilir hem de yazılabilir

önbellek sistemlerinde yavaş belleğin yazma yapılacak kısmı o anda önbellekte bulunmuyorsa (yani yazma işleminde önbellek ıskalanmışsa) ne olacaktır? İşte bu durumda da tipik olarak iki strateji uygulanabilmektedir: Birinci strateji yavaş belleğin yazma yapılacak kısmını önce önbelleğe çekip yazmayı yine önbelleğe yapmak, (buna İngilizce *write-allocate* ya da *fetch on write* deniyor) ikinci strateji ise yazmayı önbelleğe değil doğrudan yavaş belleğe yapmaktır. (Buna da *İngilizce write-no-allocate* ya da *write around* deniyor.)

Sırasal yerelliği (*sequential locality*) yüksek olan bazı önbellek sistemlerinde yavaş belleğin ardışıl tek bir bloğunun önbellekte tutulması daha uygun olabilmektedir. Dosya işlemleri için oluşturulan tamponlar bu tür önbellek sistemlerine örnek verilebilir. Fakat pek çok sistemde yavaş belleğin birden fazla bölümü önbellek içerisinde tutulur. Önbelleğin yavaş bellekteki farklı bölümlerini tutan kısımlarına önbellek satırları (*cache lines*) ya da önbellek blokları (*cache blocks*) denilmektedir.

Önbelleğe çekilmiş olan bilgi burada ne kadar süre kalacaktır? Bilginin önbellekten çıkarılmasını tetikleyen en önemli olaylardan biri önbellekte yer açma isteğidir. Önbelleğin tıka basa dolu olduğunu ve yavaş belleğin başka bir bölümünün önbelleğe çekilmek istendiğini düşünelim. Bunun için önbellekte yer açılmalıdır değil mi? İşte bu noktada sistemin, yavaş belleğin önbellekte saklanan hangi bloğunun önbellekten çıkarılacağına karar vermesi gerekir. Önbelleğin hangi bloğunun yavaş bellekten çıkartılacağına karar veren algoritmaya *önbellek yer değiştirme algoritması* (*cache replacement algorithm*) denilmektedir. Sistem önbellekten hangi bloğun çıkartılacağına karar verdikten sonra çıkartılacak blok üzerinde bir güncelleme yapılıp yapılmadığını belirlemek zorundadır. Çünkü eğer çıkartılacak blok üzerinde güncelleme yapılmışsa onun önbellekten atılmadan önce yavaş belleğe geri yazılması gerekir. Sistemler genellikle önbellek bloklarındaki bilgilerin güncellenip güncellenmediğini birer bayrakla tutarlar. Bu bayrağa kirlenme bayrağı (*dirty flag)* denilmektedir.

Bazı sistemler güncellenmiş olan önbellek bloklarını belli periyotlarla tarayıp yavaş belleğe aktarırlar. Önbellekteki bloklarının çok bekletilmeden bu biçimde yavaş belleğe aktarılmasının bir nedeni güvenilirliği artırmak, diğer nedeni ise bilgilerin paketlenerek gruplar halinde daha uygun bir zamanda yazılmasını sağlamaktır. Örneğin işletim sistemlerinin disk işlemleri için oluşturduğu aşağı seviyeli önbellek sistemlerinde önbellekteki bilgiler çok da bekletilmeden diske yazılmaktadır. (Böyle gecikmeli yazımlara *İngilizce delayed-write* deniyor.)

Önbellek kavramı ile tampon (*buffer*) kavramı da bazen birbirlerine karıştırılabilmektedir. Önbellek de tampon da bir bellek bölgesi belirtiyor olsa da bu bölgenin organize edilme nedenleri farklıdır. Önbellek istemlerinin amacı yavaş belleğe erişimde etkinliği artırmaktır. Halbuki tamponlar bilgilerin geçici olarak sırası bozulmadan bekletilmesi amacıyla kullanılır. Örneğin dış dünyadan gelen bilgileri işleyemiyorsak onları bir tamponda bekletebiliriz ve uygun bir zamanda işleyebiliriz.

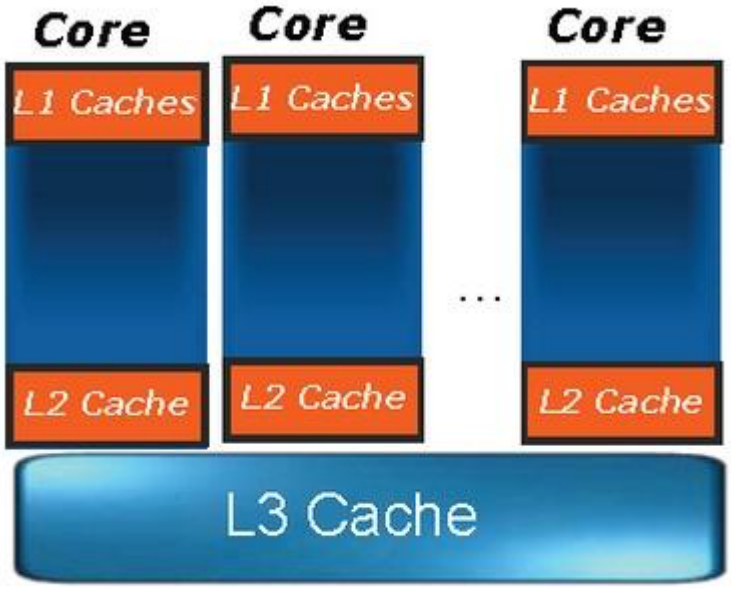
3. Çok Karşılaşılan Bazı Önbellek Sistemleri

Bu bölümde bilgisayar sistemlerinde çok kullanılan bazı önbellek sistemlerinden bahsedeceğiz ve bu sistemlerin işlevlerini açıklayacağız.

3.1. İşlemcilerin Önbellek Sistemleri

Mikroişlemciler elektriksel olarak fiziksel belleğe bağlıdır ve çalışma sırasında sürekli olarak okuma yazma amacıyla fiziksel belleğe erişmektedir. Bilindiği gibi entegre devre biçiminde üretilen fiziksel *RAM*'lerde iki temel tasarım kullanılmaktadır. *DRAM* (*Dynamic RAM*) tarzı tasarımlarında

tipik olarak blleğin her bit bir tansistör ve bir kapasitif elemanla oluşturulur. *SRAM* (*static RAM*) tarzı tasarımılarda ise bitler *flip flop* devresi ile (birkaç transistör kullanılarak) gerçekleştirilmektedir. *SRAM*'ler göreli olarak hızlı fakat pahalı, *DRAM*'lar ise yavaş fakat ucuzdur. Bunların yanı sıra *SRAM*'lerin güç harcamaları da daha azdır ve tazelenmeleri gerekmez. Bugün kişisel bilgisayarlarda kullandığımız kullandığımız *RAM*'ler *DRAM* tarzı belleklerdir [1]. İşte işlemcilerin *RAM*'e erişme miktarını azaltarak hız kazancı sağlamak için *DRAM*'ların belli bölümleri *CPU* içerisindeki *SRAM*'lerden oluşan önbelleklerde tutulmaktadır. Böylece *CPU RAM*'e erişeceği zaman önce bilgiyi kendi önbelleğinde arar, burada bulamazsa *DRAM*'a erişir.
Pek çok modern *CPU*'da önbellekler birden fazla kademeden oluşmaktadır. Daha hızlı erişilen birinci kademe önbelleğe *L1* (*L* harfi *level* sözcüğünden geliyor) önbelleği, sonrakilere *L2* ve *L3* önbellekleri deniyor. Eskiden iki kademeli önbellek sistemlerinde *L2* önbelleği işlemcinin dışında bulunuyordu. Sonraları pek çok tasarımda *L2* önbelleği de *CPU*'nun içerisine alınmıştır. Yeni *CPU*'larda artık tüm kademe önbellekler *CPU* ile aynı *chip* içerisine yerleştirilmektedirAşağıda *I7* işlemcilerinin önbellek mimarisini görüyorsunuz [2]:



3.2. TLB (Translation Lookaside Buffer)

Sayfalama (*paging*) mekanizmasına sahip mikroişlemcilerde program içerisindeki adreslere sanal adresler (*virtual addresses*) denilmektedir. Sanal adresler işlemci tarafından sayfa tablosu (*page table*) denilen bir tabloya bakılarak gerçek fiziksel adreslere dönüştürülmektedir. İşlemciler sayfa tablolarını önceden belirlenmiş bir yazmacın gösterdiği yerde ararlar. (Örneğin *Intel* işlemcilerinde *CR3* yazmacı.) Sayfa tabloları prosese özgüdür ve prosesler arası geçiş sırasında işlemci tarafından değiştirilmektedir [3]. İşte işlemcilerin her adres dönüştürmesinde sayfa tablosuna bakmak için *DRAM*'a erişmesini engellemek amacıyla ismine *TLB* (*Translation Lookaside Buffer)* denilen ayrı bir önbellek sistemi oluşturulmaktadır. *TLB* de tıpkı *L1* önbellek sistemleri gibi işlemcinin içerisine yerleştirilmiştir.

3.3. İşletim Sisteminin Organize Ettiği Disk Önbellek Sistemi

Bugünkü teknolojik düzeyde (makalenin yazım tarihini dikkate alınız) yarı iletkenlerle yapılan *flash SSD'ler* (*Solid State Disks*) elektromekanik sistemler olan *hard diskler*'den yaklaşık 5 ile 10 kat, *DRAM*'lar ise *SSD*'lerden yaklaşık 1000 kat daha hızlıdır. Bu nedenle işletim sistemleri disk

erişimini azaltmak için diskten okunan sektörleri *RAM*'de oluşturulan bir önbellekte saklamak isterler. Böylece aynı disk sektörlerine erişilmek istendiğinde hiç disk okuması yapılmadan içerik hızlı bir biçimde *RAM*'deki önbellekten karşılanır. Disk önbellek sistemleri işletim sistemlerinin performansları üzerinde en önemli etkiye sahip mekanizmalardan biridir. Disk önbellek sistemlerine İngilizce *Disk Cache*, *Buffer Cache* ya da *Page Cache* de denilmektedir. İşletim sisteminin organize ettiği disk önbellekleri genellikle hem okunabilen hem de yazılabilen biçimdedir. Pek çok sistem önbelleğe yazılan bilgiyi orada uzun süre tutmaz; uygun bir zaman diliminde kirlenmiş olan önbellek alanlarını diske geri yazar. Örneğin işletim sisteminin sunduğu sistem fonksiyonlarıyla (yani *Windows*'ta *ReadFile*, *Linux*'ta *sys_read* ile) bir dosyadan okuma yapmak isteyelim. Bu durumda okuma fonksiyonu önce dosyanın neresinden okuma yapılacağını belirler. Sonra disk önbelleğine başvurarak dosyanın okunacak kısmının disk önbelleğinde olup olmadığına bakar. Orada varsa doğrudan oradan alır; yoksa bu kez disk denetleyicisini programlayarak disk işlemini başlatır.

3.4. İşletim Sistemi Tarafından Oluşturulan Dizin Girişleri ve Dosya Bilgilerine İlişkin Önbellek Sistemleri

Modern işletim sistemleri yol ifadelerini çözümlerken (*path name resolution*) yol bileşenlerini ve onlara ilişkin dosya bilgilerini diskte bulduklarında sonraki erişimler için onları önbellek sistemlerinde saklayabilmektedir. Örneğin *UNIX/Linux* sistemlerinde *"/home/kaan/a.dat"* biçiminde bir yol ifadesini bir sistem fonksiyonuna vermiş olalım. Fonksiyon önce kök dizinin diskte yerini bulur; sonra kök dizinin içerisinde *home* girişini, *home* dizininin içerisinde *kaan* girişini, *kaan* dizininin içerisinde de *a.dat* girişini arar. Sistem her bulduğu yol bileşenini ve ona ilişkin dosya bilgilerini sonraki aramalarda daha hızlı bulabilmek için önbellek sistemlerinde saklamaktadır. *UNIX/Linux* sistemlerinde yol ifadelerinin hızlı çözümlenmesi amacıyla dizin girişleri için oluşturulan önbellek sistemlerine *directory entry cache (dentry cache), d*izin girişlerine ilişkin dosyaların diskteki bilgilerinin (örneğin uzunluklarının, erişim haklarının, disk üzerindeki yerlerinin vs.) saklandığı önbellek sistemlerine ise *i-node cache* denilmektedir [4]. Aynı önbellek mekanizmaları *Windows* sistemlerinde de benzer biçimde kullanılmaktadır.

3.5. Disk Birimi Tarafından Oluşturulan Disk Tamponları

Eskiden disk birimlerinde önbellek kullanılmazdı. Sonraları disk birimlerinin içerisinde de önbellekler oluşturuldu. Böylece işletim sistemi disk denetleme birimine başvurup diskten okuma yapacağı zaman bilgi önce disk birimi tarafından bu önbellekte aranır. Eğer burada bulunamazsa disk kafaları hareket ettirilerek gerçek okuma yapılmaktadır. Disk içerisinde kullanılan bu biçimdeki önbellek sistemlerine disk tamponları da denilmektedir. Disk tamponları genellikle -diskin büyüklüğüne oranlandığında- oldukça küçük tutulmaktadır (8MB, 16MB, 32MB, 64MB, 128MB). Disk tamponları yalnızca okuma amacıyla değil yazma amacıyla da kullanılabilmektedir. Böylece işletim sistemi yazılacak sektörleri disk birimine ilettiğinde yazmayı beklemeden işine devam edebilir. Ancak disk tamponlarının bu biçimde kullanılmasının bazı potansiyel sorunlara yol açabileceğini de belirtelim.

3.6. Web Dünyasında Kullanılan Önbellek Sistemleri

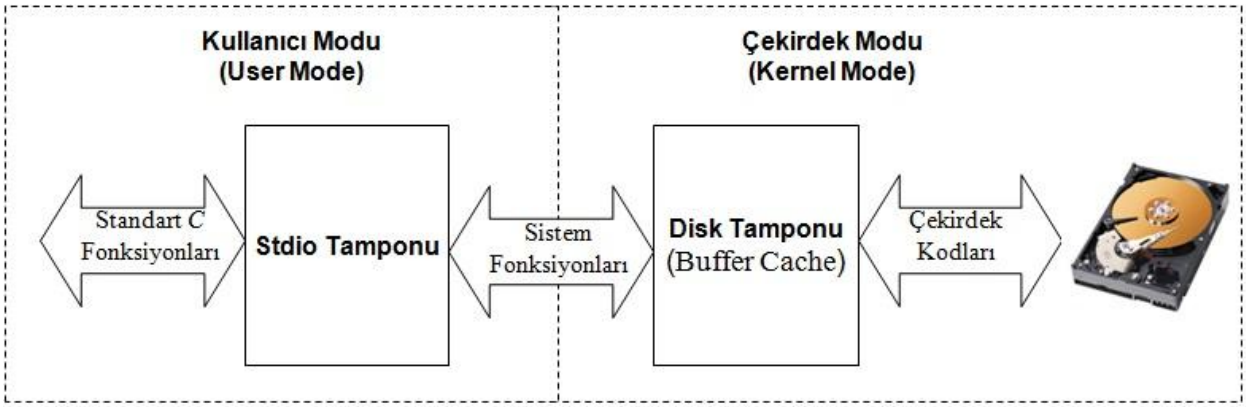
Daha önce ziyaret edilmiş sitelerin içeriklerini daha hızlı bir biçimde getirebilmek için tarayıcılar son ziyaret edilen *web* sitelerini yerel diskte bir önbellek alanında tutabilmektedir. Tarayıcıların oluşturdukları önbelleklerin büyüklükleri ve izledikleri önbellek politikaları tarayıcıdan tarayıcıya değişebilmektedir. Tarayıcıların kullandıkları önbellekler için aklınıza gelen ilk soru muhtemelen şu olacaktır: Tarayıcı *web* sayfasının güncellenip güncellenmediğini nasıl biliyor? Bunun yanıtı

bilmiyor biçimindedir. Tarayıcılar sayfa içeriğinin değişip değişmediğini bizim belirlemelerimize göre kontrol etmektedir.[1]

Web dünyasında *proxy* sunucular da benzer bir *web* önbelleği kullanabilmektedir. Örneğin bir *proxy* sunucusundan bir sayfayı talep ettiğinizde, o size daha önce kendi önbelleğinde saklamış olduğu sayda içeriğini hızlı bir biçimde getirebilir. Arama motorları da en yoğun önbellek kullanan *web* uygulamalarındandır. Arama motorları çok talep edilen arama sonuçlarını bir önbellekte hazır olarak bulundurabilmektedir. .

3.7. Dosya Fonksiyonlarının ve Sınıflarının Kullandıkları Önbellek Sistemleri

Bilindiği gibi her türlü dosya işlemi aslında işletim sistemlerinin sunduğu sistem fonksiyonlarıyla yapılmaktadır. Koruma mekanizmasına sahip işlemcileri kullanan işletim sistemlerinde sistem fonksiyonları çekirdek modunda (*kernel mode*) çalışırlar ve prosesin kullanıcı modundan (*user mode*) çekirdek moduna geçmesi ve geri dönmesi göreli bir zaman kaybına yol açar. Bir dosyadan *byte byte* okuma yapıp onları işlemek istediğinizi düşünelim. Her defasında bir *byte* okumak için işletim sisteminin çekirdek moduna geçen sistem fonksiyonunu kullanmak (örneğin *Windows* sistemlerinde *ReadFile*, *UNIX/Linux* sistemlerinde *read* fonksiyonunu) çok verimsiz bir yöntem olacaktır. Çünkü bir *byte* okumak için çekirdek moduna geçmek ve oradan geri dönmek yüzlerce makine komutunun çalışmasını gerektirmektedir. İşte programlama dillerinin bulundurdukları dosya fonksiyonları ve sınıfları çekirdek moduna geçişi azaltmak için önbellekler oluşturmaktadır. Böylece örneğin programcı dosyadan bir *byte* bile okumak istediğinde aslında bu fonksiyonlar bir blok bilgiyi okuyup önbelleğe yerleştirirler ve sonraki okumaları doğrudan bu önbellekten karşılarlar. Dosya fonksiyonlarının ve sınıflarının oluşturdukları bu önbellekler genellikle hem okunabilir hem de yazılabilir biçimdedir. Bu durumda dosyaya yazma yapan fonksiyonlar yazmayı doğrudan dosyaya değil önbelleğe yaparlar. Önbellekteki bilgiler süreç içerisinde çeşitli durumlarda işletim sisteminin sistem fonksiyonlarıyla dosyaya aktarılmaktadır. Programlama dillerinde dosya fonksiyonlarının ve sınıflarının kullandıkları bu mekanizmaya -aslında mekanizma bir önbellek sistemi olmasına karşın- genel olarak tamponlama (*buffering*) denilmektedir. Örneğin *C*'de *fgetc* gibi bir fonksiyonla dosyadan bir *byte* okuyacak olalım. Fonksiyon önce *stdio* kütüphanesinin kullanıcı modundaki (*user mode*) önbelleğinden okuma yapmaya çalışır. Eğer bu önbellekte (başka bir deyişle tamponda) okunacak bilgi kalmamışsa işletim sisteminin okuma yapan sistem fonksiyonunu kullanarak önbelleği doldurur. Tabi işletim sisteminin okuma fonksiyonun da önce işletim sisteminin disk önbelleğine baktığını, eğer okunacak bilgiyi burada bulamazsa gerçek disk okuması yaptığını biliyorsunuz.

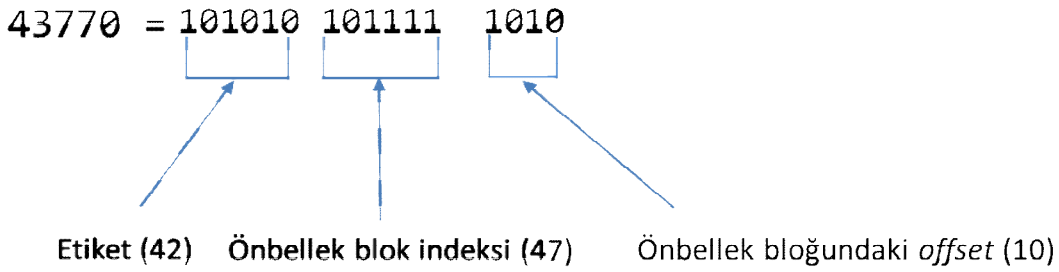


4. Önbellek İlişkilendirme Yöntemleri

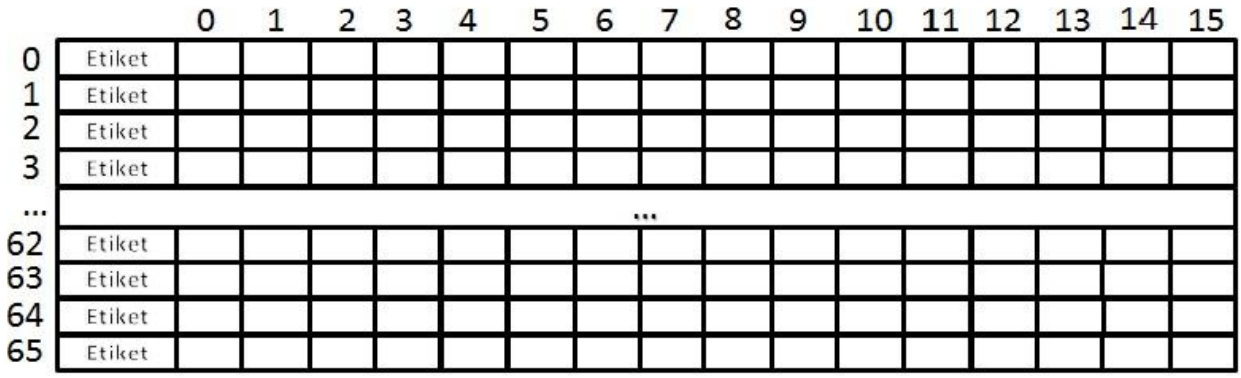
Yavaş bellekten alınan bilginin önbelleğin hangi bloklarına yerleştirilebileceğine ilişkin belirlemelere önbellek ilişkilendirmesi (*cache associativity*) denilmektedir. Önbellek ilişkilendirilmesi tipik olarak birkaç biçimde yapılabilmektedir.

4.1. Doğrudan Haritalanmış Önbellek (*Direct Mapped Cache*) ilişkilendirmesi

Bu ilişkilendirmede yavaş bellekteki bir blok önbelleğin yalnızca tek bir bloğuna yerleştirilebilir. Yavaş bellekteki hangi blokların önbelleğin hangi bloklarıyla ilişkilendirileceğini belirlemek için bir *hash* fonksiyonu kullanılmaktadır. Örneğin 64K uzunluğunda olan yavaş bellek için 1K'lık bir önbellek kullanacak olalım ve önbelleğin her bir bloğu 16 byte olsun. Bu durumda yavaş bellekte toplam 65536 / 16 = 4096, önbellekte ise toplam 1024 / 16 = 64 blok bulunacaktır. Yavaş belleğin bu 4096 bloğunu önbelleğin 64 bloğuyla ilişkilendiren basit *hash* fonksiyonu bölümden elde edilen kalan fonksiyonu (*mod* fonksiyonu) olabilir. Bu durumda bu bellek için doğrudan haritalama yönteminde erişilecek adresin 16'ya bölümünün 64'e bölümünden elde edilen kalan o yavaş bellek bloğunun ilişkilendirildiği önbellek bloğunu belirtiyor olacaktır. (Sayının 64'e bölümünden elde edilen kalan düşük anlamlı 6 bitidir). Örneğin yavaş belleğin 43770'inci adresine erişilmek istensin. Bu adres yavaş belleğin 2735'inci bloğu içerisindedir (43770 = 2735 * 16 + 10). Yavaş belleğin 2735'inci bloğu ise önbelleğin 47'nci bloğu ile ilişkilendirilir (2735 % 64 = 47). O halde yavaş belleğin 43770 adresine erişilmek istendiğinde eğer o blok önbelleğin 47'nci bloğunda yoksa yavaş belleğin 43760 adresinden itibaren 16 byte alınarak önbelleğin 47'nci bloğuna yerleştirilecektir. Şimdi aynı hesabı 2'lik sistemde yapalım:



Etiket (*label*) erişilen önbellek bloğunun gerçekten aranan adrese ilişkin olup olmadığını belirlemek için gerekmektedir. Bu durumda önbellekte her blok için bir etiket de tutulmak zorundadır. Buradaki önbellek aşağıdaki gibi düşünülebilir:



Böylece örneğin yavaş belleğin [43760, 43775] aralığındaki herhangi bir *byte*'a erişilmek istendiğinde bunların hepsinin etiketleri aynı olacaktır. Tabi böyle bir tasarımda çakışma (*collision*) kaçınılmazdır. Çakışma oluştuğunda söz konusu blok doluysa ya o yavaş bellek bloğu önbelleğe alınmaz ya da çakışmaya yol açan bloğun içeriği önbellekten atılır. Örneğin, sonraki zamanlarda

36598 adresine erişilmek istendiğini düşünelim. 36598 adresi yavaş belleğin 2287'inci bloğundadır (36598 = 2287 * 16 + 6). Yavaş belleğin 2287'inci bloğu önbelleğin yine 47'nci bloğu ile ilişkilendirilir (2287 % 64 = 47). Bu da bir çakışmaya yol açar. İşte bu durumda ya yavaş belleğin bu bloğu önbelleğe alınmayacaktır ya da önbelleğin 47'nci bloğunun içeriği önbellekten çıkarılacaktır. Doğrudan ilişkilendirilen önbellek sistemlerinde yavaş belleğin önbellekte olup olmadığının belirlenmesi çok hızlı yapılabilmektedir. Ancak çakışma durumunda önbellekte daha uygun bloklar varken çakışmanın oluştuğu bloğun önbellekten çıkartılması, önbellekten çıkartma konusunda esnekliği kısıtlamaktadır.

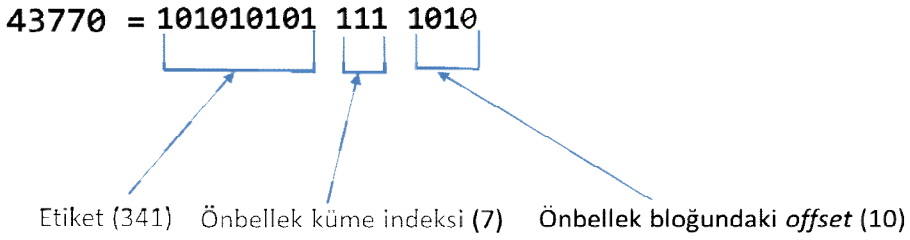
Doğrudan haritalamış önbellek sistemlerini daha iyi anlayabilmek için yavaş bellekteki adresleri üç bileşene ayırabiliriz:

Adresin *offset* kısmı önebellek bloğundaki *offset*'i, önbellek blok indeksi önbellek bloğunun numarasını belirtiyor. Etiket ilgili önbellek bloğundaki bilginin aranan adrese ilişkin olup olmadığını karş laştırmak amacıyla kullanılmaktadır.

4.2. Kümesel Önbellek İlişkilendirmesi (*Set Associated Cache*)

Kümesel önbellek ilişkilendirmesinde önbellek kümelere ayrılır. İşlemci bir bloğa erişmek istediğinde önce onun hangi kümede olduğunu belirler. Sonra küme içerisinde arama yapar. (Tabi donanımsal sistemlerde bu yapılan aramalar sıralı değil aynı anda, yani paralel biçimde gerçekleştirilmektedir.) Aranan blok ilgili kümede varsa önbellek isabet ettirilmiştir, yoksa önbellek ıskalanmıştır. Iskalama durumunda eğer önbellek kümesi tamamen dolu ise sonraki bölümde açıklanan algoritmalarla o önbellek kümesinden blok çıkarılır. Kümesel ilişkilendirme işlemcilerin önbellek sistemlerinde sıkça kullanılan bir ilişkilendirme yöntemidir. Kümesel ilişkilendirme yönteminde erişilmeye çalış lan yavaş bellek adresleri üç bileşene ayrılmaktadır: Etiket (*tag*), küme indeksi (*set*) ve *offset*:

Etiket ilgili bloğun önbellekte bulunup bulunmadığını belirlemekte kullanılır. Küme indeksi adresin hangi önbellek kümesine ilişkin olduğunu belirtmektedir. *Offset* de önbellek bloğundaki ilgili *byte*'ın *offset'*idir. Kümesel önbellek sistemleri bir kümedeki blok sayısına göre 2 yollu (2-*way*), 4 yollu (4-*way*), 8 yollu (8-way) ve genel olarak *n* yollu (*n-way*) biçiminde isimlendirilir. Örneğin 64KB yavaş bellek için 8 yollu kümesel ilişkilendirmeyle 1K'lık bir önbellek oluşturulacak olsun. Önbellek bloğunun da yine 16 *byte* olduğunu varsayalım. Bu durumda her kümede 8 blok bulunacağından toplam 1024 / 16 / 8 = 8 küme söz konusudur. Böylece küme indeksi 3'er bitten, *offset* alanları 4'er bitten, etiket de 9 bitten oluşacaktır. Böyle bir ilişkilendirmede biz yine yavaş belleğin 43770'inci adresine erişmek isteyelim. Bu adres aşağıdaki gibi parçalarına ayrılır:



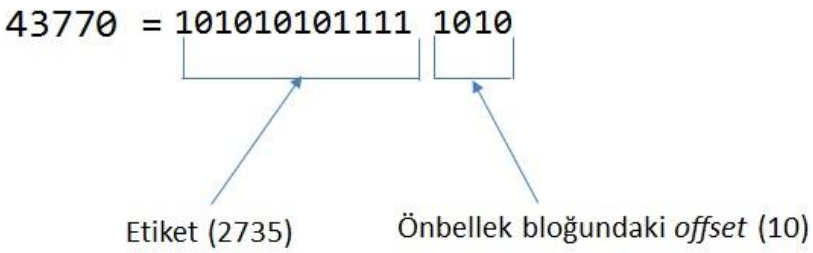
Adres önbellekte aranırken önce 7 numaralı kümeye gidilir. Bu kümedeki 8 blokta etikete göre arama yapılır.

Şimdi kümelere ayırmanın ne faydası var diye düşünebilirsiniz. Doğrudan haritalanmış önbellek ilişkilendirmesinde bir çakışma söz konusu olduğunda önbellekten çıkartılacak blok hakkında bir esneklik yoktur. Halbuki örneğin 8 yollu kümesel bir ilişkide çakışma olduğunda o kümedeki 8 bloktan herhangi biri önbellekten çıkartılabilmektedir. Bu da önbellekten çıkarma konusunda esneklik sağlamaktadır.

4.3. Tam Önbellek İlişkilendirilmesi (*Fully Associative Cache*)

Bu ilişkilendirme biçiminde yavaş bellekteki bir adres önbellekte aranırken önbelleğin tüm bloklarına bakılır. Tabi bu yöntem genellikle donanımsal olarak uygulandığı için bu bakma işlemi sıralı arama biçiminde değil paralel bir biçimde çok hızlı yapılmaktadır. Tam önbellek ilişkilendirmesinde yavaş bellekteki adres iki bölüme ayrılır: Etiket ve *offset*:

Şimdi de 64KB yavaş bellek için tam ilişkilendirmeyle 1K'lık bir önbellek oluşturulmak istensin. Yine bir önbellek bloğunun 16 byte uzunlukta olduğunu varsayalım. Böylece önbellek de yine 64 bloktan oluşacaktır. Bu durumda adresin etiket kısmı 12 bit, *offset* kısmı 4 bit olacaktır. Yavaş belleğin 43770 adresine erişilmek istendiğinde bu adres aşağıdaki gibi iki parçaya ayrılır:



Sistem 2735 etiketini paralel bir biçimde hızlı belleğin 64 bloğunda arar. Bulursa erişimi o bloktan karşılar. Bulmazsa kullanılan algoritmaya göre herhangi bir bloğu önbellekten çıkarabilir.

İlişkilendirme yöntemlerini dikkatle incelediyseniz bir şeyi fark etmişsinizdir: Aslında yukarıda ele aldığımız üç önbellek ilişkilendirmesi de bir çeşit kümesel ilişkilendirme olarak ele alınabilir. Doğrudan haritalanmış önbellek ilişkilendirmesini önbellekteki blok sayısı kadar kümeden oluşan kümesel ilişkilendirme, tam önbellek ilişkilendirmesini de 1 tane kümeden oluşan kümesel ilişkilendirme gibi düşünebiliriz.

5. Önbellekten Blok Çıkartma Algoritmaları

Bu bölümde önbellekten blokların nasıl ve hangi kurala göre atılacağını belirlemek için kullanılan önbellek algoritmaları (*cache replacement algorithms*) üzerinde duracağız. Çeşitli sistemler için onların özel durumlarına uygun düşen pek çok önbellek algoritması önerilmiş olsa da bunların birkaçı genel gereksinimlerin büyük çoğunluğunu karşılayabilecek niteliktedir.

Öncelikle ideal bir önbellek yer değiştirme algoritmasının nasıl olabileceği üzerinde durmak istiyoruz. Evet, eğer çalıştırılacak program hakkında her şeyi biliyor olsaydık ideal önbellek yer değiştirme algoritması nasıl olurdu? Sezgisel olarak bu soruya hemen şöyle yanıt verilebilir: Önbellekten bir blok çıkarılacağı zaman en sonra gereksinim duyulacak blok önbellekten atılabilir. Örneğin önbellekte 5 blok olsun ve bunların o andan itibaren sonraki gereksinim duyulma süreleri 0.1, 0.2, 0.3, 0.4 ve 0.5 saniye olsun. Bizim 0.5 saniye sonra gereksinim duyulacak bloğu önbellekten atmamız gerekir. Sonraki atımlarda da aynı yöntemi izlersek en iyi çözümü elde etmiş oluruz. Tabi bu ideal algoritmanın uygulanabilmesi için bizim her önbellek bloğuna gelecekte ne zaman gereksinim duyulacağını bilmemiz gerekir. Ancak böyle bir bilgi kestirilebilir değildir.[2] Bunun için çeşitli önbellek yer değiştirme algoritmaları önerilmiştir. Aşağıda dört temel önbellek yer değiştirme algoritmasını açıklayacağız. Diğer algoritmalar için başka kaynaklara başvurabilirsiniz.

5.1. Son Zamanlarda En Az Kullanılan (Least Recently Used) Bloğun Önbellekten Çıkarılması

LRU (*Least Recently Used*) olarak da ifade edilen bu algoritmada önbellekten blok çıkarılacağı zaman son zamanlarda en az kullanılan bloğun çıkartılması yoluna gidilir. Bu algoritmanın dayandığı fikir, son zamanlarda en az kullanılanın yakın bir gelecekte de az kullanılmaya devam edeceği beklentisidir. Gerçekten de pek çok sistemde bu algoritma iyi bir performans ortaya koymaktadır. *LRU* algoritması yazılımsal olarak çeşitli biçimlerde gerçekleştirilebilir. En tipik yöntemlerden biri bir bağlı liste oluşturup, önbellek bloklarını bağlı liste biçiminde ifade etmek ve bir blok kullanıldığında onu oradan koparıp bağlı listenin önüne almaktır. (Örneğin *Linux* çekirdeğindeki bazı önbellek sistemlerinde bu yöntem kullanıyor). Böylece son zamanlarda kullanılanlar bağlı listenin önlerinde kullanılmayanlar da sonlarında kalır. Çıkarma yapılacağı zaman bağlı listenin sonunda kalan önbellek bloğu çıkartılır. Diğer bir yöntem ise her önbellek bloğu için bir yaşlanma sayacı (*age counter*) tutmaktır. Bir önbellek bloğuna erişildiğinde diğer tüm önbellek bloklarının yaşlanma sayaçları bir artırılır. Böylece yaşlanma sayacı en yüksek olan bloklar son zamanlarda en az kullanılmış olan blokları belirtiyor durumda olur. Tabi aynı şey tersten de yapılabilir. Yani erişilen bloğun sayacı artırılır. Böylece sayacı en düşük olan blok en yaşlı blok durumunda olacaktır. Böylece önbellekten blok çıkartılacağı zaman yaşlanma sayacı en yüksek olan blok çıkartılır. Diğer bir yöntemde ise önbellek bloğuna her erişildiğinde erişimin mutlak ya da göreli olarak ne zaman yapıldığı o bloğa ilişkin bir alanda tutulur. Önbellekten blok çıkartılmak istendiğinde blokların tutulan bu son erişim zamanlarına bakılır. Zamanı en geride olan blok önbellekten çıkartılır. *LRU* algoritması pek çok yaygın sistem için en uygun algoritma olarak kabul edilmektedir.

5.2. Son Zamanlarda En Fazla Kullanılan (Most Recently Used) Bloğun Önbellekten Çıkarılması

MRU algoritması denilen bu yöntemde *LRU*'nun tersine son zamanlarda en fazla kullanılmış olan blok önbellekten atılmaktadır. İlk bakışta sizlere anlamsız gibi gözüken bu algoritma bazı sistemler için uygun olabilmektedir.

5.3. En Az Kullanılan (Least Frequently Used) Bloğun Önbellekten Çıkarılması

LFU algoritması denilen bu algoritmada son duruma bakılmaz toplamda en az kullanılmış olan blok önbellekten atılmaya çalışılır. Bazı sistemlerde bellek blokları sürekli kullanılır durumdadır. Bu tür sistemlerde toplamda en az kullanılan önbellek bloklarının atılması uygun olabilmektedir. *LFU* algoritmasını gerçekleştirmek için tipik olarak her önbellek bloğu için bir sayaç tutulur. Bloğa erişildiğinde o bloğun sayacı bir artırılır. Önbellekten blok çıkartılacağı zaman bloklar incelenir ve sayacı en düşük olan blok çıkartılır. *LRU* için O(1) karmaşıklıkta algoritmalar da önerilmiştir.

5.4. Rastgele Bir Bloğun Önbellekten Çıkarılması

Bazı sistemler için önbellekte hangi blokların tutulması gerektiği hakkında tutarlı bilgiler olmayabilir ya da bazı sistemlerde yavaş belleğe erişim tamamen rastgele düzeyde gerçekleşiyor olabilir. Bu tür sistemlerde belli blokların önbellekte tutulmaya çalışılmasının bir anlamı kalmaz. Böylece rastgele bir blok önbellekten çıkarılabilir.

5.5. Uyumlanabilir (Adaptive) Önbellek Algoritmaları

Bu yöntemlerde dinamik bir yol izlenmektedir. Başka bir deyişle uyumlanabilir yöntemler ”en az kullanılan boğun önbellekten çıkartılması (*LFU*)” yöntemiyle “son zamanlarda en az kullanılan bloğun önbellekten çıkartılması (*LRU*)” yönteminin dinamik biçimde birleştirilmesiyle oluşturulmuş yöntemlerdir.

---

[1]: Örneğin *Internet Explorer*'da bunun için kullanıcıya birkaç seçenek sunulmuştur. Seçeneklerden biri sayfa her ziyaret edildiğinde kontrolün yapılması biçimindedir. Diğer bir seçenek, tarayıcı her çalıştırıldığında bunu bir yapması biçimindedir. Diğeri de her zaman önbellektekini kullanması biçimindedir.

[2]: Eğer aynı program ikinci kez çalıştırılacak olsa ve birinci çalıştırmadaki aynı durumlar ikinci çalıştırmada oluşacak olsa belki sistem programın ilk çalışması sırasında bu bilgileri toplayıp diğer çalıştırmalarında bunlardan faydalabilir. Fakat programlardaki olası akışlar o anda başka koşullara bağlı olabileceğinden bu da pek mümkün gözükmemektedir.

Kaynaklar

[1] B. Jacob ve ark., *Memory Systems*, Burlington: Morgan Kaufmann Publishers, 2008, bl. 7, ss. 315-351.

[2] F. Abi-Chahla (2008, Kasım. 14). *Intel Core i7 (Nehalem): Architecture By AMD?* [Online]. URL: http://www.tomshardware.com/reviews/Intel-i7-nehalem-cpu,2041-10.html

[3] K. Aslan, *Intel İşlemcileri Korumalı Mod Yazılım Mimarisi*, İstanbul: Pusula Yayıncılık, 1997, bl. 3, ss. 63-90.

[4] D.P. Bovet ve M. Cesati, *Understanding The Linux Kernel,* 3th Edition , Sebastopol: O'Reilly, 2005, bl. 12, ss. 412-478.