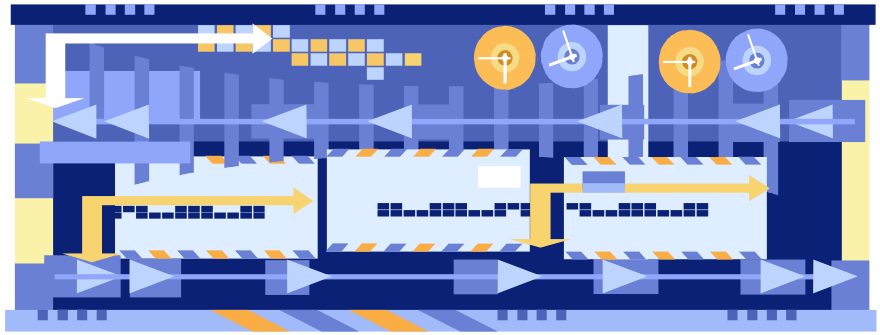
Önbellek-3

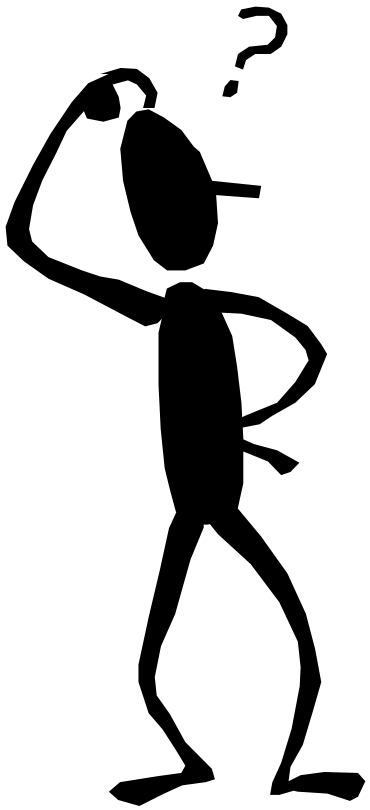
- Önbelleğe Yazma ve Performans

Cache'e Yazma ve Performans



- Bugün cache konusunda kalan şu son konuları konuşacağız :
  - Cache'e yazma: Belleği tutarlı tutmak ve yazma yöntemi.
  - Değişik cache yazma dizaynlarının getirilerini ve genel performansa etkisini tartışacağız.
  - Ayrıca sistem bellek performansının artmasına yardımcı olan ana bellek organizasyonlarını tekrar detaylıca ele alacağız.
- Sonraki derste, diskin cache'i gibi davranan sanal belleklerden bahsedeceğiz.

Dört Önemli Soru



1. Ana bellekten cache içine bir veriyi alırken tam olarak nereye yazacağız?
2. Bir veri cache içinde var mı, yok mu nasıl söyleriz? Orada yoksa ana bellekten çekmek lazım da...
3. Eninde sonunda bu küçücük bellek dolacak ve ana bellekten gelen bir veri diğeri üzerine yazılacaktır. Bu alan hangisi olacak?
4. Cache ve ana belleğe yazma stratejimiz ne olacak? Veriyi geri yazma gerekirse nereye yazacağız?

- İlk 3 soruya önceki slaytlarda cevap verdik, bu derste son soruya bakalım.

Bir Cache'e Yazmak

- Bir cache'e yazmak bir çok yeni konu açıyor.
- İlki, değerini değiştirmek istediğimiz adresin şu an zaten bir direkt atamalı cache'de bir yerlerde yazılı olduğunu varsayalım.

İndeks	V	Tag	Veri
...			
110	1	11010	42803
...			

Adres	Veri
...	
1101 0110	42803
...	

- Şimdi bu adrese yeni bir değer yazmak istersek, yeni veriyi direkt cache'e yazabiliriz ve pahalı olan ana belleğe yazmaktan kurtuluruz.

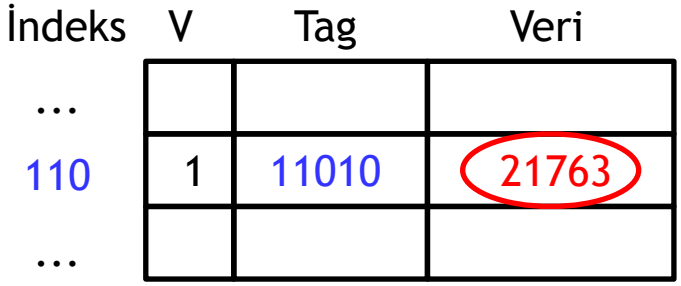
Mem[214] = 21763

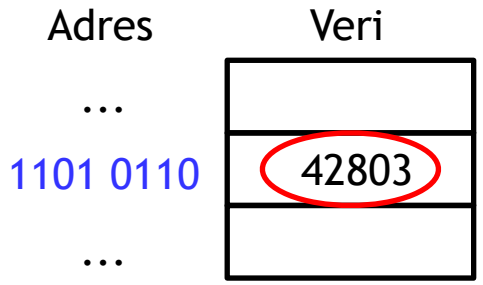
İndeks	V	Tag	Veri
...			
110	1	11010	21763
...			

Adres	Veri
...	
1101 0110	42803
...	

Tutarsız Bellekler

- Fakat artık cache ve bellek değişik içerikliler, bunun adı tutarsız veri!
- Bu noktadan sonra o adrese ihtiyaç duyulduğunda doğru verinin geldiğine nasıl emin olacağız?
- Eğer bir de bu bellekler çok işlemcili makinelerde olduğu gibi, başka cihazlar tarafından da kullanılan ortak belleklerse daha da problemli bir durumdur.



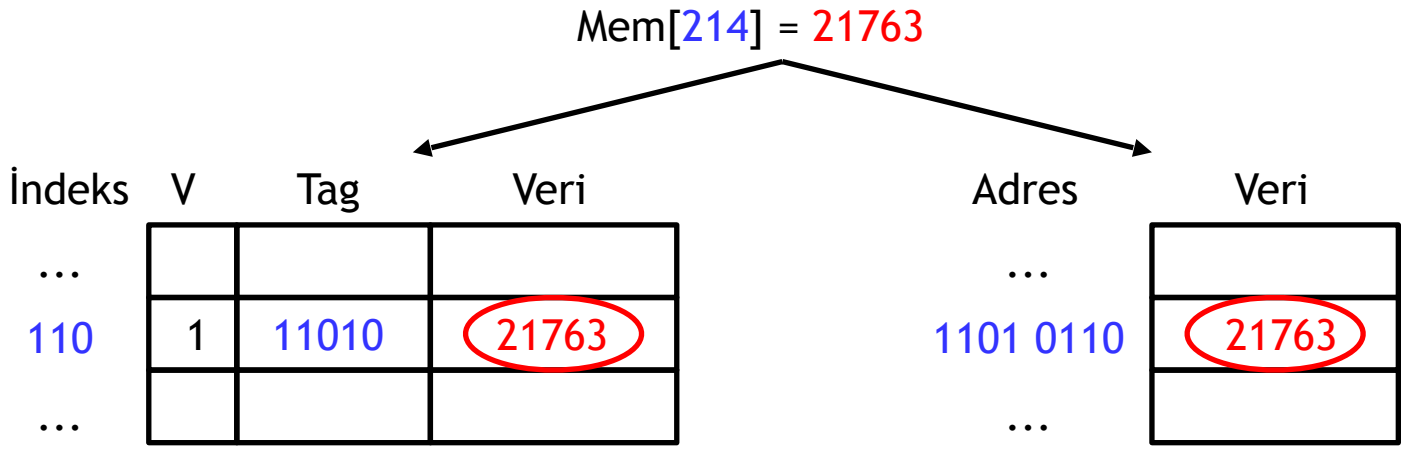


Daha fazla yatırım ($$$)

Cache bellek tutarlılığı (ing: Cache coherence)

İki Tarafa Yazan 'Write-Through' Cache'ler

- Bir 'write-through cache' tutarsızlık (ing: inconsistency) problemini yazma işlemini hem cache'e hem de ana belleğe yapmaya zorlayarak çözer.



- Güzel, basit bir çözüm, cache ve ana belleği tutarlı tutmanın en basit ve uygulanabilir çözümüdür.
- Ama aslında hiç te güzel değil... **Yavaş çünkü!**
- Bir kötülüğü de her yazmayı ana belleğe yollamak, cache ve bellek arasındaki hattın sürekli meşgul olması demektir.

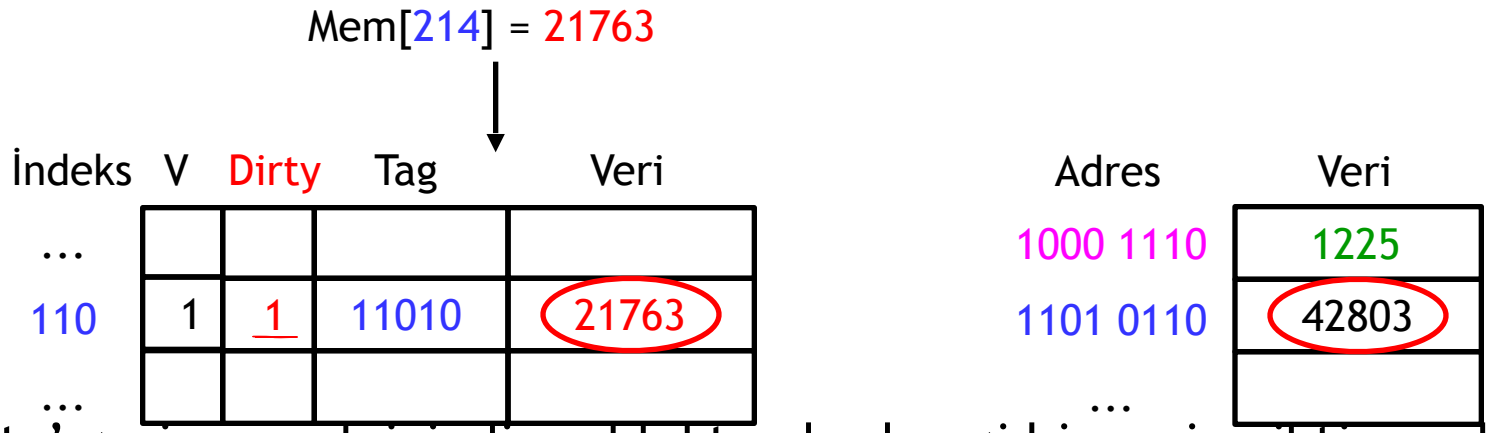
Yazma Tampon Bellekleri

- ‘Write-through’ cache’ler yavaş yazma işlemlerine yol açar, bu yüzden işlemciler bir de yazma tampon alanları (ing: write buffer) içerirler ki ana belleğe yazılacaklar burada kuyruğa girsin, işlemci işine devam edebilsin.

- Tampon alanlar iki farklı cihazın farklı hızlarda çalıştığı tüm durumlarda kullanılırlar.
  - Eğer bir üreteç/üretici (ing: producer) bir tüketicinin (ing: consumer) altından kalkamayacağı hızda veri üretirse, bu ekstra veriler tampon alan yazılır ve üreteç, tüketeni beklemeden başka işler üzerinde çalışmasına devam eder.
  - Tersi durumda, üreteç yavaşlayınca, tüketici son hızla tampon bellekten veri çekip, çalışmasını sürdürecektir.
- Bizim için üreteç CPU, tüketici ana bellektir.

Geri Yazmalı 'Write-Back' Cache'ler

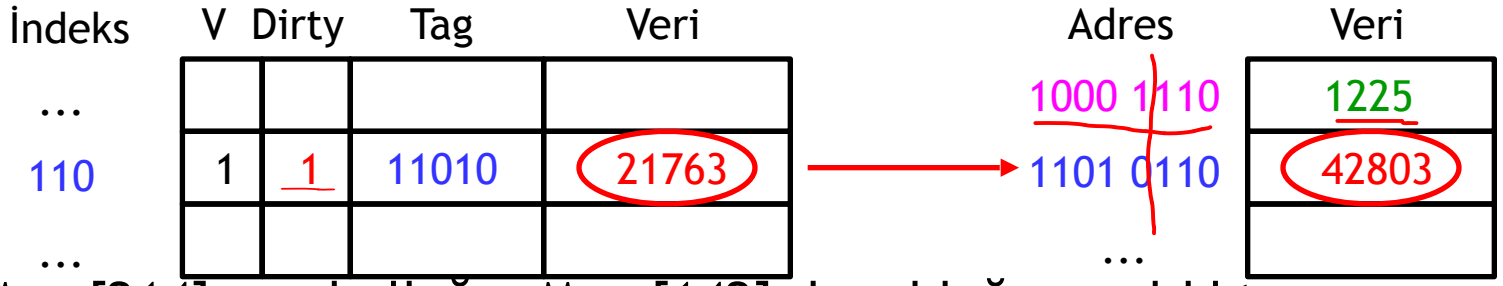
- Bir 'write-back cache' de ana belleğe yazma işlemi ancak o cache bloğu üzerine yazma gerekirse gerekli olur (Set assoc. cache için tüm set dolduğunda ancak gerekir).
- Ama acaba yazma gerekli mi? Bir cache bloğu değiştiğinde ana bellekle tutarsızlık olur, aksi durumda gerek yok.
  - Cache bloğuna yeni bir bit daha ekler ve bu blok değiştirildi/tutarsız (ing: dirty) diye işaretlemek için kullanırız.



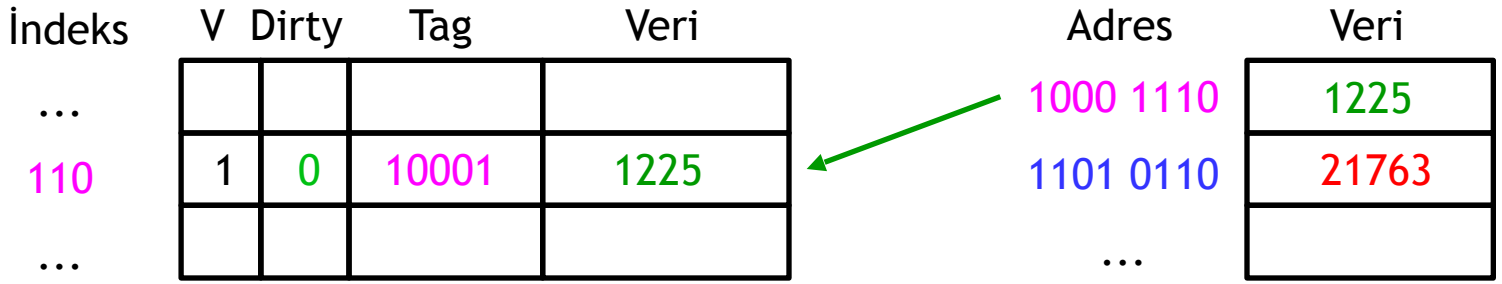
- 'Dirty' geri yazmak içindir, o bloktan herhangi bir veriye ihtiyaç oldukça, cache o an, en geçerli değeri tuttuğundan, değeri CPU'ya iletecektir.

Geri Yazmayı Tamamlamak

- Cache bloğu o yerde durdukça, bizim de yeni değerleri ana belleğe yazmamıza gerek yoktur.
- Örneğin, Mem[142]'den değer okunacaksa ve cache'de o blokta Mem[214] kayıtlı görünüyorsa, iki durum var.
  - 'Dirty' = 0 ise direkt üzerine yazılır, ana bellekten geldiği gibi duruyor değişmemiş demektir.
  - 'Dirty' = 1 ise önce cache'deki bloğu ana belleğe yazmak lazım.



- Mem[214] ana belleğe, Mem[142] de o bloğa yazıldıktan sonra aşağıdaki şekil oluşur.

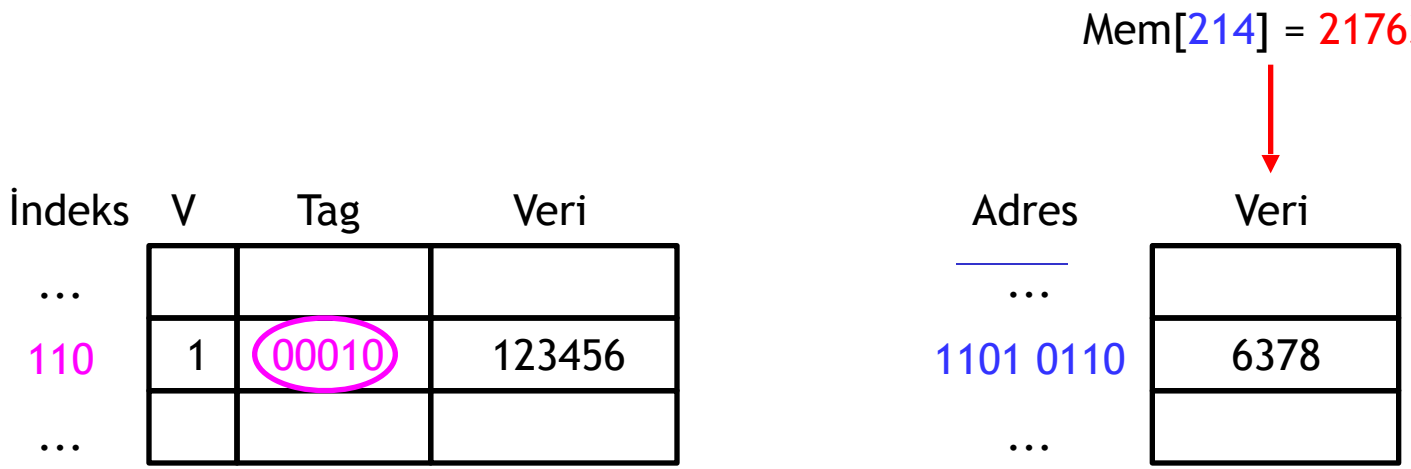


‘Write-Back’ Cache Tartışması

- ‘Write-back’ cache’lerin ‘write-through’ cache’lere göre avantajı tüm yazma işlemlerini ana belleğe göndermemesidir.
  - Eğer tek bir adres sürekli yazılıyorsa (toplam = toplam + A[i] gibi), ana belleğe hiç yazmadan cache bu işlemi yürütür.
  - Aynı cache bloğunda pek çok byte tutuyoruz. Eğer aynı blokta birden fazlası değiştiyse, tek bir geri-yazma işlemi ile ana belleğe yollanırlar.
    - Değişmemiş olanlar da mecburen yazılır :-)
- ‘Write-back’ cache’lerde her blok başında bir ‘dirty’ biti tutulmak zorundadır, yoksa istenmeyen geri-yazmalar olacaktı.
- Ana belleğe başvurma sorunu (ing: penalty) her zaman yaşanmıyor.
  - Örneklerde, Mem[214] adresine yazma işlemlerini cache hallediyor.
  - Fakat Mem[142]’den veri yüklemek gerekince *iki* bellek erişimi gerekiyor:
    - Biri adres 214’ye değişen veriyi yazma, diğeri adres 142’den okuma.
    - Yazma işlemleri yine tampon bellek aracılığıyla yapılırsa iyi olur.

Yazma Adresinin Cache'de Olmaması (Write Misses)

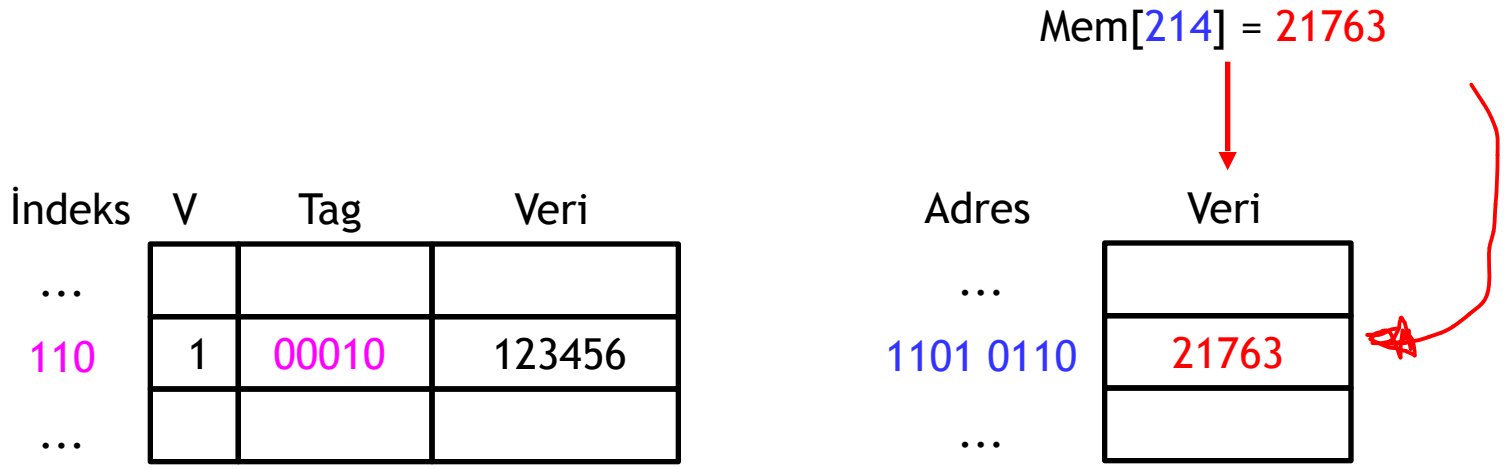
- İşte size ikinci ilginç senaryo: int x = 0; işleminde olduğu gibi, yazılacak adres cache'de halihazırda yoksa, buna 'write miss' diyorlar.
- Hadi 21763 değerini Mem[214]'e yazmaya çalışalım ama bu adres cache içinde olmasın.



- Acaba Mem[214]'e yapılan değişikliği aynı zamanda ilgili cache bloğuna da yazsak mı?

Sadece Belleğe Yazan 'Write Around' Cache'ler

- Bu yeni 'write around' (veya write-no-allocate) yöntemi, yazma işlemini cache'i etkiletmeden direkt olarak ana belleğe yönlendirir.

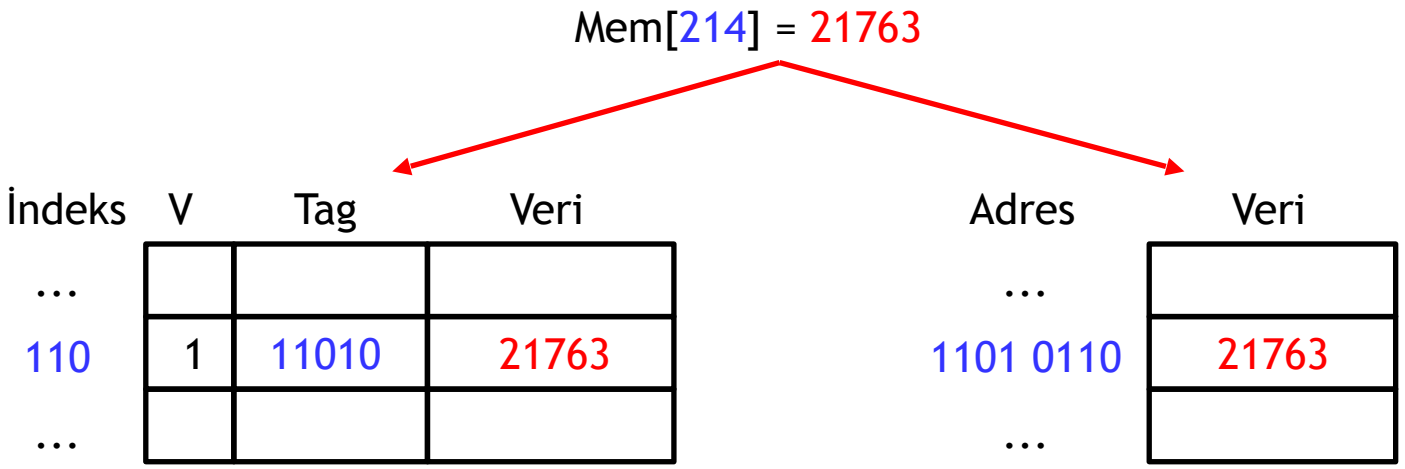


- Bu teknik verinin ana belleğe yazılıp ta, kod içinde hemen devam satırlarda ihtiyaç olmadığı durumlarda çok iyidir.

```
for (int i = 0; i < SIZE; i++)
        a[i] = i;
```

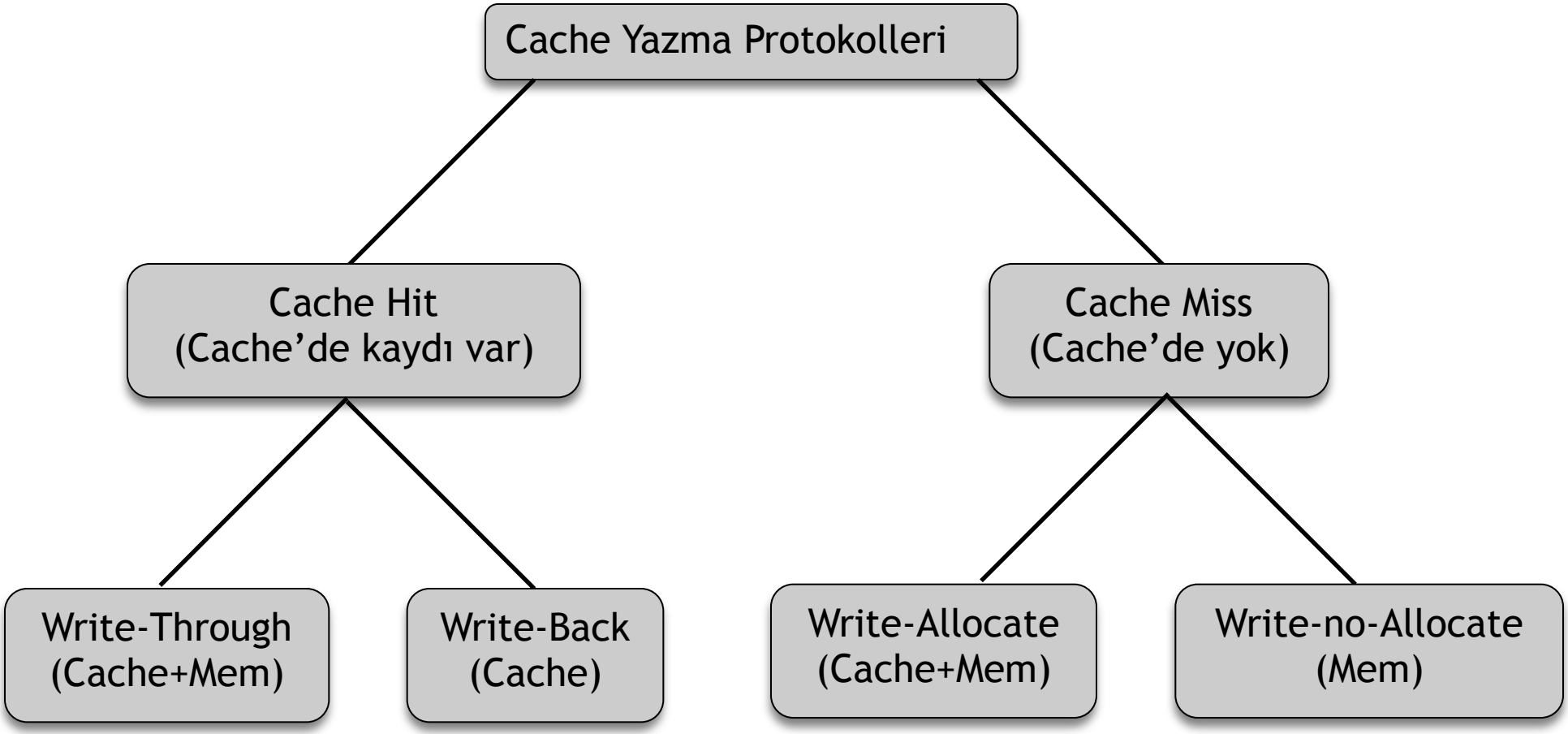
İki Taraf da Yazan ‘Allocate on Write’ Cache’ler

- Alternatifi hem ana belleğe, hem de cache’e yazan ‘allocate on write’ tekniğidir.



- Veri yazıldığı gibi kullanılacaksa cache içinde hazır olacaktır.

Cache Belleğe Yazma Protokolleri Özeti



Örnek: Hangi Teknik Kullanılmış?

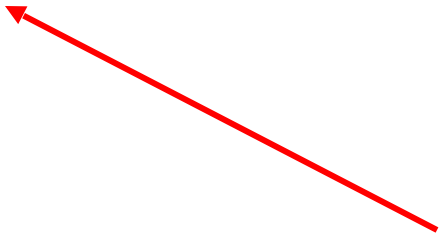
- Aşağıda bir bellek erişim akışı verilmiştir, cache'de 'write-allocate' mi yoksa 'write-no-allocate' mi kullanıldığını tespit edebilir misiniz?
  - A ve B adresleri farklı ve cache içinde aynı anda durabiliyorlar.

Miss	Load A
Miss	Store B
Hit	Store A
Hit	Load A
Miss	Load B
Hit	Load B
Hit	Load A

Örnek: Hangi Teknik Kullanılmış?

- Aşağıda bir bellek erişim akışı verilmiştir, cache'de 'write-allocate' mi yoksa 'write-no-allocate' mi kullanıldığını tespit edebilir misiniz?
  - A ve B adresleri farklı ve cache içinde aynı anda durabiliyorlar.

Miss	Load A
Miss	Store B
Hit	Store A
Hit	Load A
Miss	Load B
Hit	Load B
Hit	Load A

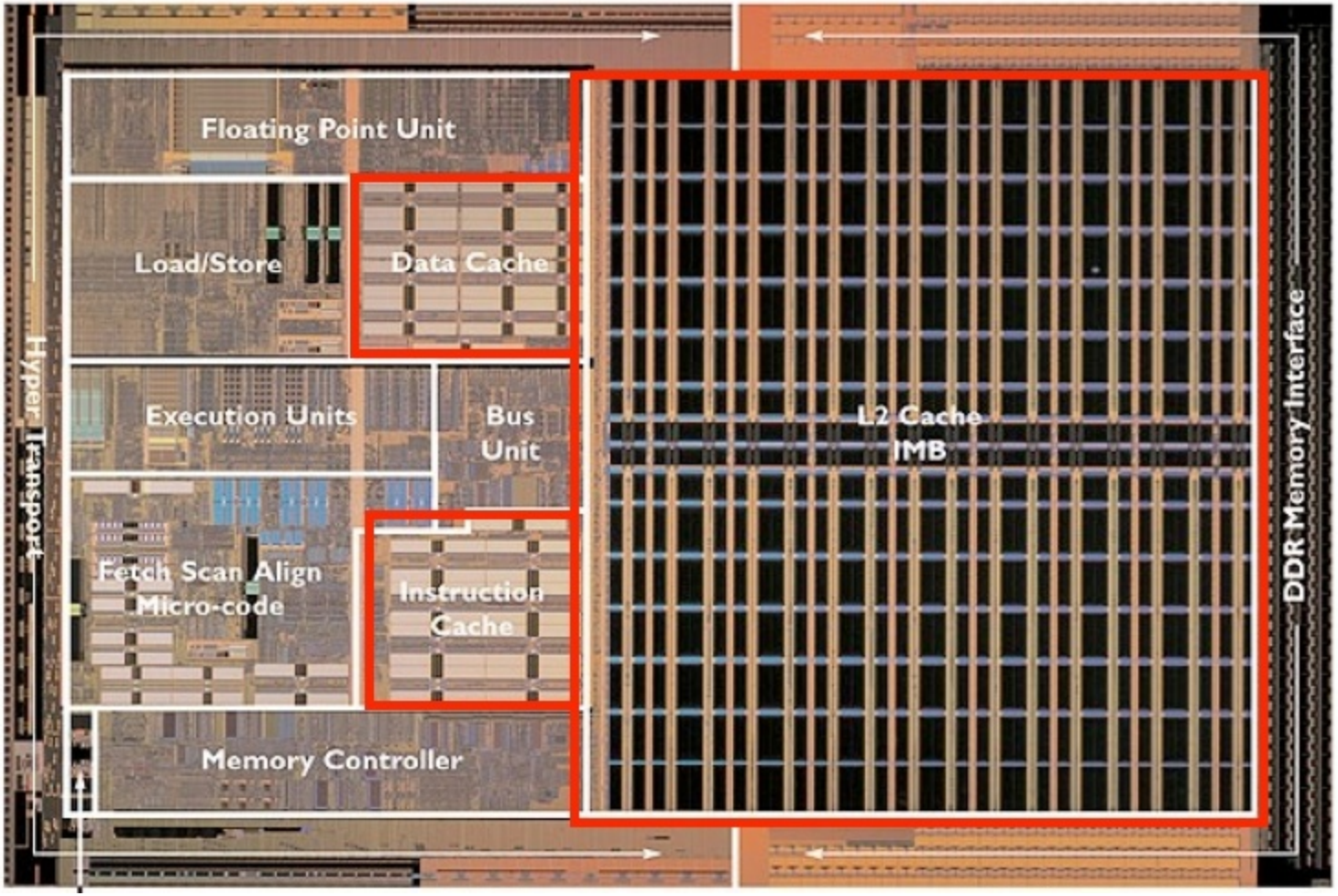


Eğer 'write-allocate cache' olsa burada hit alırdık

Cevap: Write-no-allocate

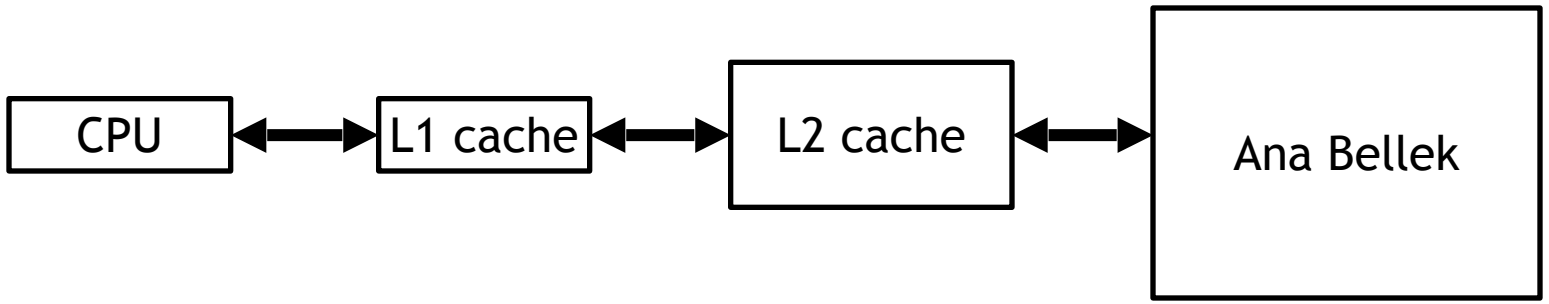
Günümüzden Örnekler

Gerçek Dizaynlar

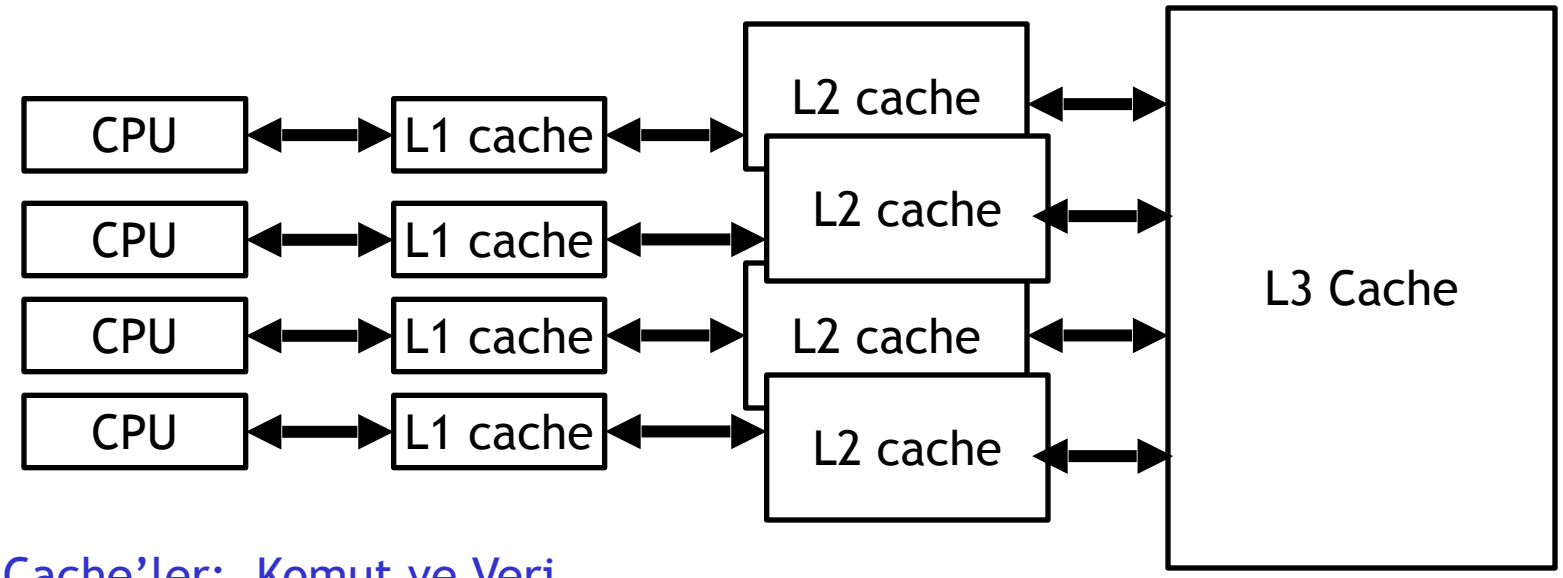


İlk Gözlemler

- Komut/Veri cache'leri ayrılır:
  - İyi tarafları: Komut ve veriler birbirlerini beklemezler.
    - Cache yavaşsa Load/Store komutlarında CPU biraz bekler.
  - Kötü tarafları: Komut ve veri alanları kalıcı olarak ayrıdır.
    - Setlerin içi farklı dolulukta olabilir.
    - Örneğin: az_kod/ÇOK_VERİ veya **ÇOK_CODE**/az_veri
- Cache hiyerarşileri:
  - Erişim zamanı ile hit oranı arasında tercih gerekir.
    - L1 cache hızlı erişime odaklanır (hit oranı ikinci planda)
    - L2 cache iyi hit oranına odaklanır (erişim zamanı ikinci planda)
  - Bu tür hiyerarşik yapılar da büyük bir fikir olmuştu.
  - Aşağıdaki yapıları bugün göreceğiz.



İntel-i5 6. Nesil Cache Özellikleri



- L1 Cache'ler: Komut ve Veri
  - 256 KB (komut:128 + veri:128)
  - 8-way set associative
- L2 Cache:
  - 1 MB
  - 4-way set associative
- L3 Cache:
  - 6 MB
- Ortak özellikleri
  - 64 Byte bloklar,
  - Write-back tekniği

Intel i7-7820X (Skylake X)

- 8 cores, 4.3 GHz (Turbo Boost), Mesh 2.4 GHz, 14 nm.
  - RAM: 4x 8 GB DDR4-3400 16-18-18-36.
- L1 Data cache = 32 KB, 64 B/line, 8-WAY
- L1 Instruction cache = 32 KB, 64 B/line, 8-WAY.
- L2 cache = 1024 KB, 64 B/line, 16-WAY
- L3 cache = 11 MB, 64 B/line, 11-WAY
- L1 Data Cache Latency = 4 cycles for simple access via pointer
- L1 Data Cache Latency = 5 cycles for access with complex address calculation (size_t n, *p; n = p[n]).
- L2 Cache Latency = 14 cycles
- L3 Cache Latency = 68 cycles (3.6 GHz)
- L3 Cache Latency = 79 cycles (4.3 GHz) (77-81 cycles for different cores)
- RAM Latency = 79 cycles + 50 ns
- Not: L3 Cache testlerde LRU veya Pseudo-LRU özellikleri göstermiyor. L3 Cache bazı eski satırları uzun süre korumuş. Rastgele yerleştirme var gibi.

Kaynak: https://www.7-cpu.com

Intel® Core™ i9-10850K İşlemci

- **İş Parçacığı Sayısı : 20 ! (10 Cpu)**
  - İşlemci Temel Frekansı : 3.60 Ghz
  - Maks Turbo Frekansı : 5.20 Ghz
  - Önbellek L1 : 640 KB 2 X (10 X 32KB) 8-way Set Assoc
  - Önbellek L2 : 2.5 MB (10 X 256 KB) 4-way Set Assoc
  - Önbellek L3 : 20 MB 16- Way Set Associative Paylaşımlı
- **Bellek Özellikleri :**
  - Maks Bellek Boyutu : 128 Gb
  - Bellek Türü : Ddr4-2933
  - Bellek Kanal Sayısı : 2
  - BUS RATE : 8 GT/S (Ddr5'e Uygun)
  - Maks Bellek Bant Genişliği : 45,8 Gb/S

Intel® Core™ i9-10980XE Extreme Edition İşlemci

- CPU Specifications
  - Fiziksel çekirdek sayısı 18
  - İş Parçacığı Sayısı 36
  - İşlemci Temel Frekansı 3,00 GHz
  - Maks Turbo Frekansı 4,60 GHz
- Önbellek
  - Önbellek L1 : 1125 KB 2 X (18 X 32KB) 8-way Set Assoc
  - Önbellek L2 : 18 MB (18 X 1 MB) 16-way Set Assoc
  - Önbellek L3 : 24.75 MB 11- Way Set Associative Paylaşımlı
- Veri Yolu Hızı 8 GT/s DMI3
- Bellek Teknik Özellikleri
  - Maksimum Bellek Boyutu (bellek türüne bağlıdır) 256 GB
  - Bellek Türleri
    DDR4-2933
  - Maks. Bellek Kanalı Sayısı 4
  - Maksimum Bellek Bant Genişliği94 GB/s

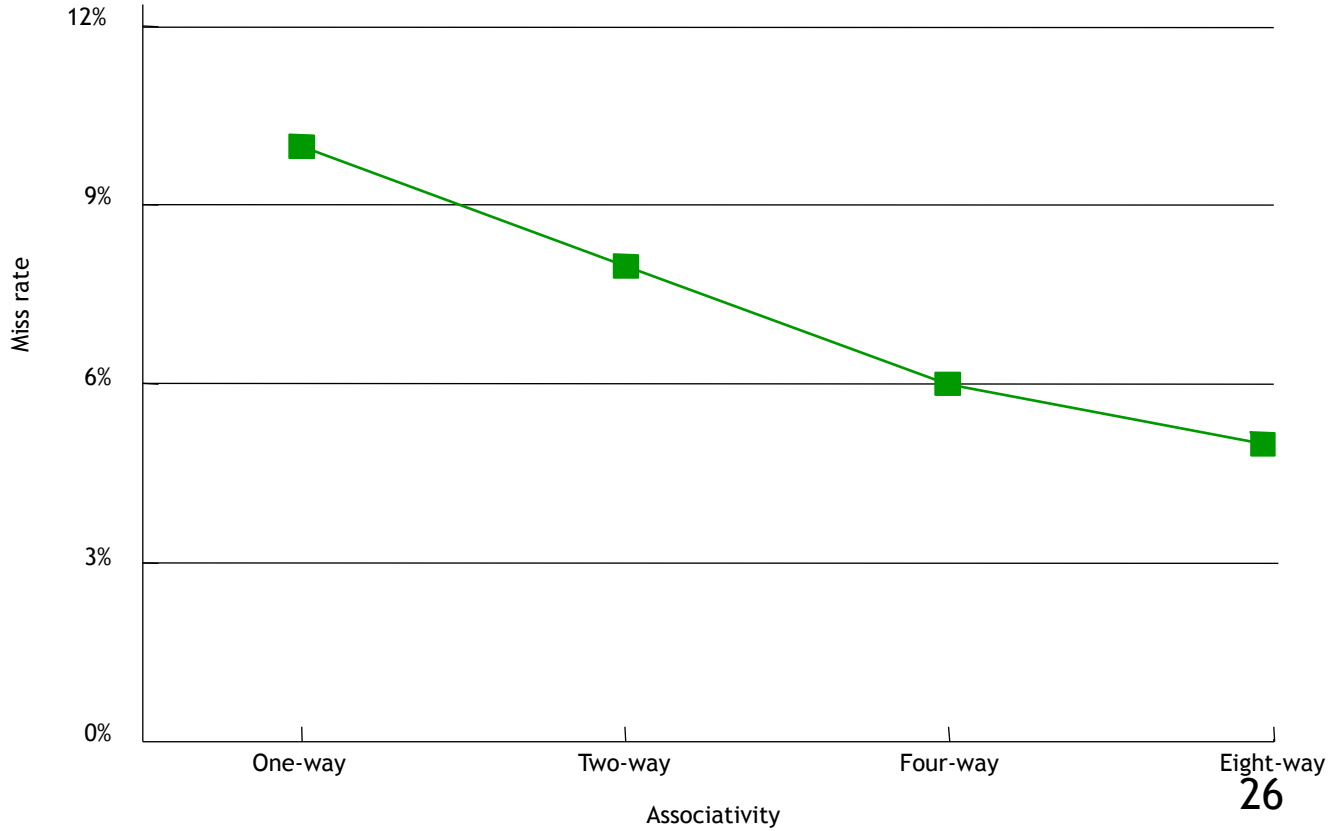
CACHE ORGANİZASYONLARI

Cache Organizasyonlarını Karşılaştırma

- Pek çok mimari özellikleri gibi, cache'ler de deneysel olarak ayarlanıyor.
  - Her zaman olduğu gibi, farklı programlar farklı bellek erişim desenlerine sahipken, performans ta gerçek hayatta karşımıza çıkacak komut karışımlarına bağlı oluyor.
  - Simülasyon yapmak veya gerçek uygulamaları çalıştırmak performans karakteristiklerini ölçmenin en doğru yoludur.
- Burada birkaç slayttaki grafikler birkaç cache değişik dizaynında simüle edilmiş miss oranlarını göstermektedir.
  - Düşük miss oranları genelde iyidir, ama unutmayın bu miss oranı ortalama bellek erişim zamanı ve çalışma zamanını sadece bir bileşenidir. Bunlar üzerinde başka kriterler de etkilidir.

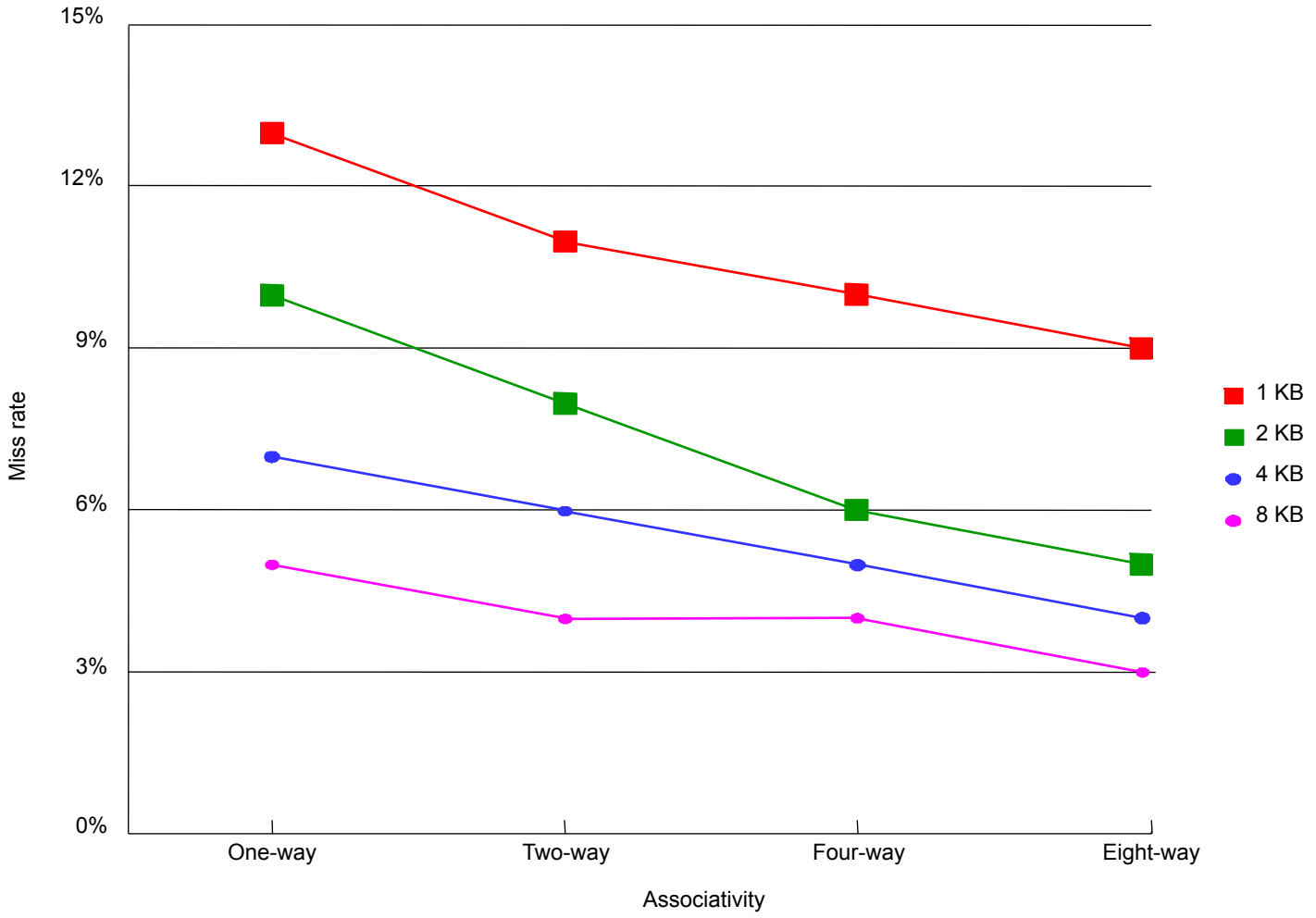
Birliksellik (Associativity) - Miss Oranları

- En son gördüğümüz konuda, yüksek birliksellik (Fully Associative'e doğru gidenler) daha kompleks donanım anlamına geliyordu.
- Fakat bu yüksek *k*-way cache'lerde de düşük miss oranları kaçınılmaz oluyor.
  - Her set daha çok satır içeriyor, dolayısıyla aynı setteki iki adresin birbirine engel teşkil etmesi olasılığı iyice düşüyor.
  - Toplamda, bu AMAT'ı düşürüyor ve belleği beklemeyi azaltıyor.
- Miss oranlarının düşüş grafiği yandaki gibi bulunmuş.



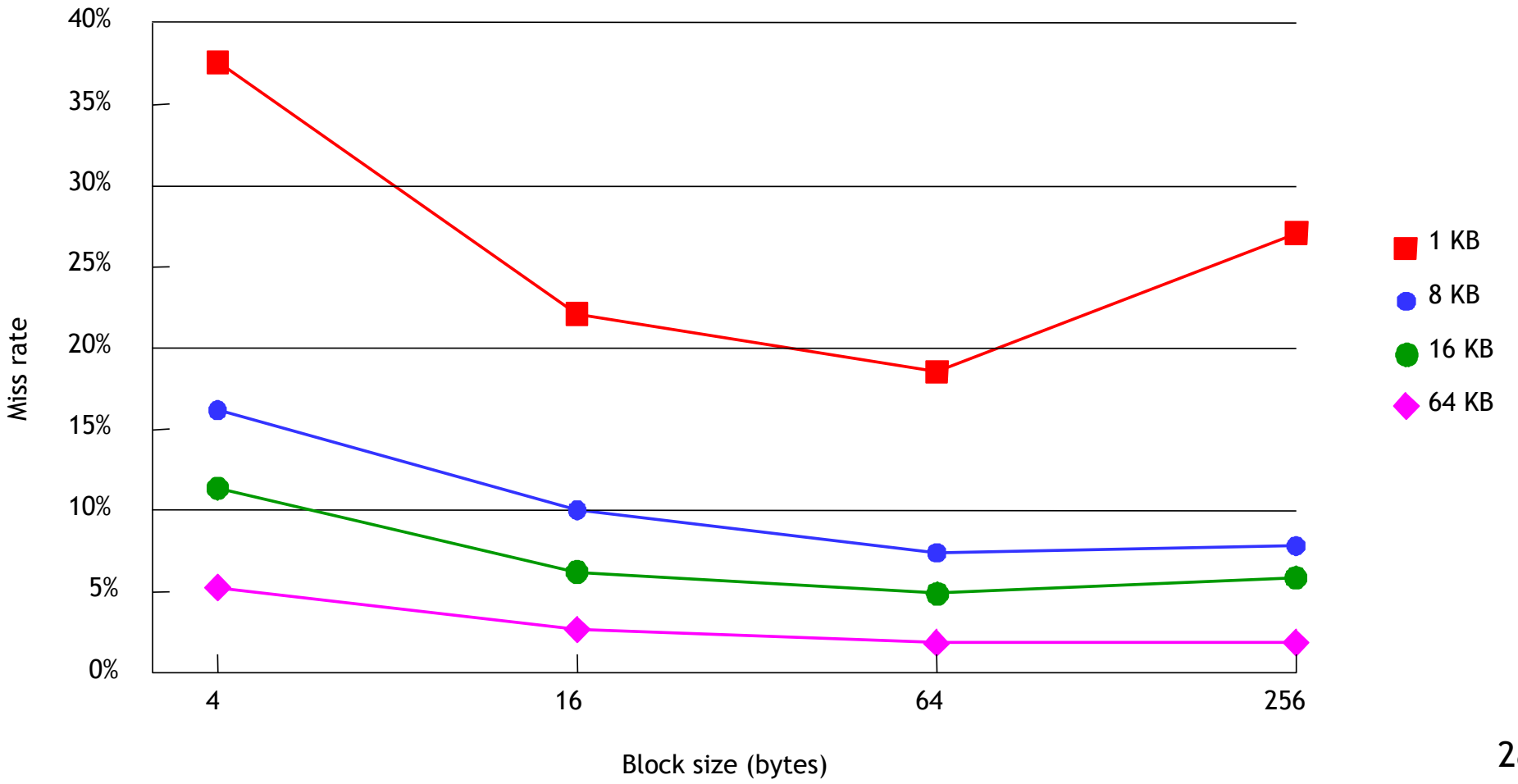
Cache Boyutu - Miss Oranları

- Cache boyutunun da performans üzerinde önemli etkisi vardır.
  - Büyük cache demek düşük ihtimalli adres çakışması demektir.
  - Yine bu miss oranı düşüşü demektir, yani bu AMAT'ı düşürüyor ve belleği beklemeyi azaltıyor.
- Cache boyutu ve *k*-way değerine bağlı miss oranları grafiği.



Satırda Tutulan Byte - Miss Oranları

- Miss oranlarının blok boyutu (byte) ve cache boyutuna bağlı değişimi grafiği.
  - Küçük bloklar uzaysal yerellik avantajından tam yararlanamıyorlar.
  - Fakat bloklar çok büyükse, az sayıda satır oluyor ve çakışmalar yaşanıyor.



PERFORMANS

Bellek ve Genel Performans

- Cache hit'leri ve miss'ler genel sistem performansını etkiler mi?
  - Cache hit alınca geçen süreyi bir CPU saat sinyali kabul edersek (gerçekte 3-5 arası saat sinyali oluyormuş), sürekli cache hit alınınca programın işleyişi olması gereken hızla sürüyor demektir.
  - Cache miss alınca, CPU'yu veri ana bellekten gelene dek oturup bekliyor (ing: stall) kabul ediyoruz.
- Toplam beklenen döngü sayısı (ing: total stall cycles) ne kadar cache miss yaşandığı ve ana belleğe gitmenin cezasıyla formüle ediliyor.

Memory stall cycles = Memory accesses x miss rate x miss penalty

- Sonuçta CPU'nun normal çalıştığı ve belleği beklediği döngüler toplanır, saat sinyalinin süresi ile çarpılarak programın çalışma süresi bulunur.

CPU time = (CPU execution cycles + Memory stall cycles) x Cycle time

Performans Örneği

- Diyelim ki programınızdaki komutları %33'ü belleğe erişiyor. Cache hit oranı %97 ve hit alınırsa veri bir saat sinyalinde, alınmazsa 20 saat sinyalinde geliyor.
  - Toplam komut sayısı `I` ise.

$$\text{Memory stall cycles} = \text{Memory accesses} \times \text{Miss rate} \times \text{Miss penalty}$$
$$= 0.33\ I \times 0.03 \times 20 \text{ saat sinyali}$$
$$= 0.2\ I \text{ saat sinyali}$$

- Bu kod mükemmel işleyen bir programdan 1.2 kez yavaştır.
  - Mükemmellik Cyle-Per-Instruction (CPI) = 1 ise olur.
  - Yani komut başına CPU saat sinyali = 1 olmalı.

Bellek Sistemleri Başa Beladır

CPU time = (CPU execution cycles + Memory stall cycles) x Cycle time

- İşlemci performansı ile belleğin performansı arasında uçurum olduğundan, bellek sistemleri tüm sistemin karadeliğidir (ing: bottleneck).
- Önceki örnekte, ideal olan CPI = 1 temelinde CPU zamanı:

CPU time = (I + 0.2 I) x Cycle time

- Şimdi biz bunda iyileştirme olsun diye CPU performansını iki kat arttırdık, CPI =0.5 oldu, ama bellek performansı aynı kaldı, ne olur?

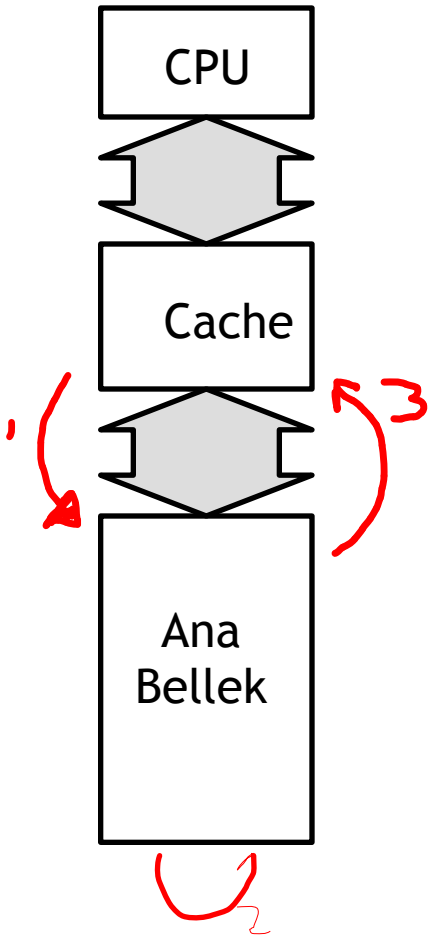
CPU time = (0.5 I + 0.2 I) x Cycle time

- Toplamda CPU zamanı sadece 1.2/0.7 = 1.7 kat iyileşti, 2 kat değil!
- Amdahl'ın meşhur yasaları der ki:
  - Sistemin tek bir parçasını iyileştirmek beklenen faydanın azalmasına neden olur.

BELLEK – CACHE BAĞLANTI DİZAYNLARI

En Temel Ana Bellek Dizaynı

- Cache'in de yardımıyla, ana belleğe başvurma cezasını (ing: miss penalties) düşürmek için ana bellek birkaç yolla organize edilebilir.
- Burada örnekleri verirken cache belleğin, ana bellekten veri alması için şu üç varsayımı kullanacağız:
  1. 1 saat sinyalinde bir adres RAM'e iletilir.
  2. Her RAM erişiminde 15 saat sinyali gecikme olur.
  3. 1 saat sinyalinde veri RAM'den gelir.
- Buradaki kurulumda, CPU'dan cache belleğe ve cache bellekten RAM'e bus'lar bir kelime genişliğindedir. Kelime bit sayısına 8/16/32 bit denilebilir, önemli değil.
- Eğer cache'in bir kelimelik blokları varsa, RAM'den gelen bilgiyle bloğu doldurmak (*yani* 'miss penalty') 17 saat sinyali süre alır.



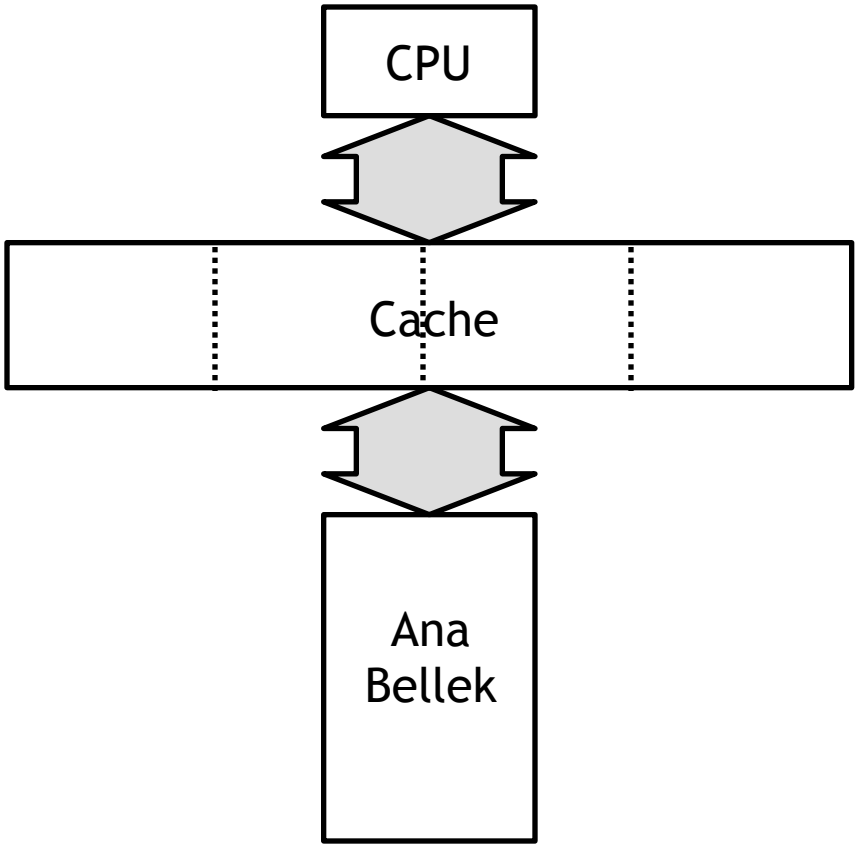
1 + 15 + 1 = 17 saat sinyali

- Cache kontrolcüsü istenen adresi RAM'e yollar, bu süre boyunca bekler ve veriyi alır.

Daha Geniş Cache Blokları Kullanılırsa

- Eğer cache 4-kelimelik bloklar içerirse, Bir blok veri almak demek 4 bağımsız bellek erişimi demektir ve ‘miss penalty’ 68 saat sinyali olur!

4 x (1 + 15 + 1) = 68 saat sinyali

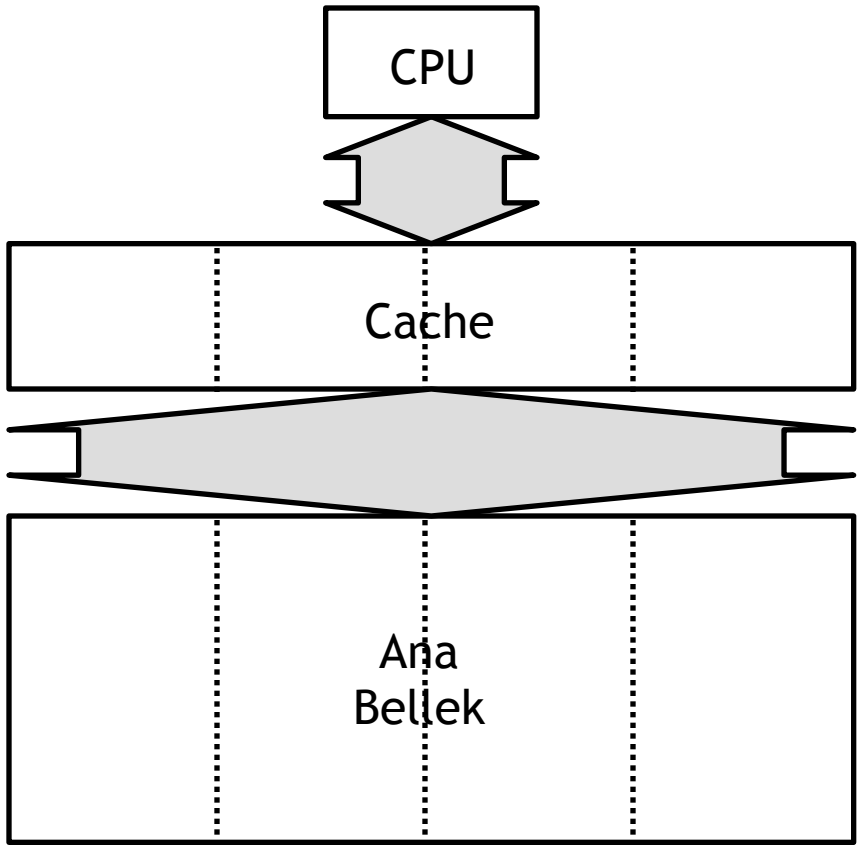


Bellek Cache'e Göre Genişletilebilir

- Geniş cache'e veri aktarırken 'miss penalty' değerini düşürmek için bir basit yol geniş bellek kullanmaktır. Böylece bir kerede daha çok kelime RAM'den okunabilir.
- Dört kelimeyi bellekten okumak için bus'ın genişlemesi lazım ki bu işlem sadece 17 saat sinyali sürsün.

  1 + 15 + 1 = 17 saat sinyali

- Geniş bus'ın maliyeti de büyük — her ek bellek biti cache'e ek bağlantı demektir.

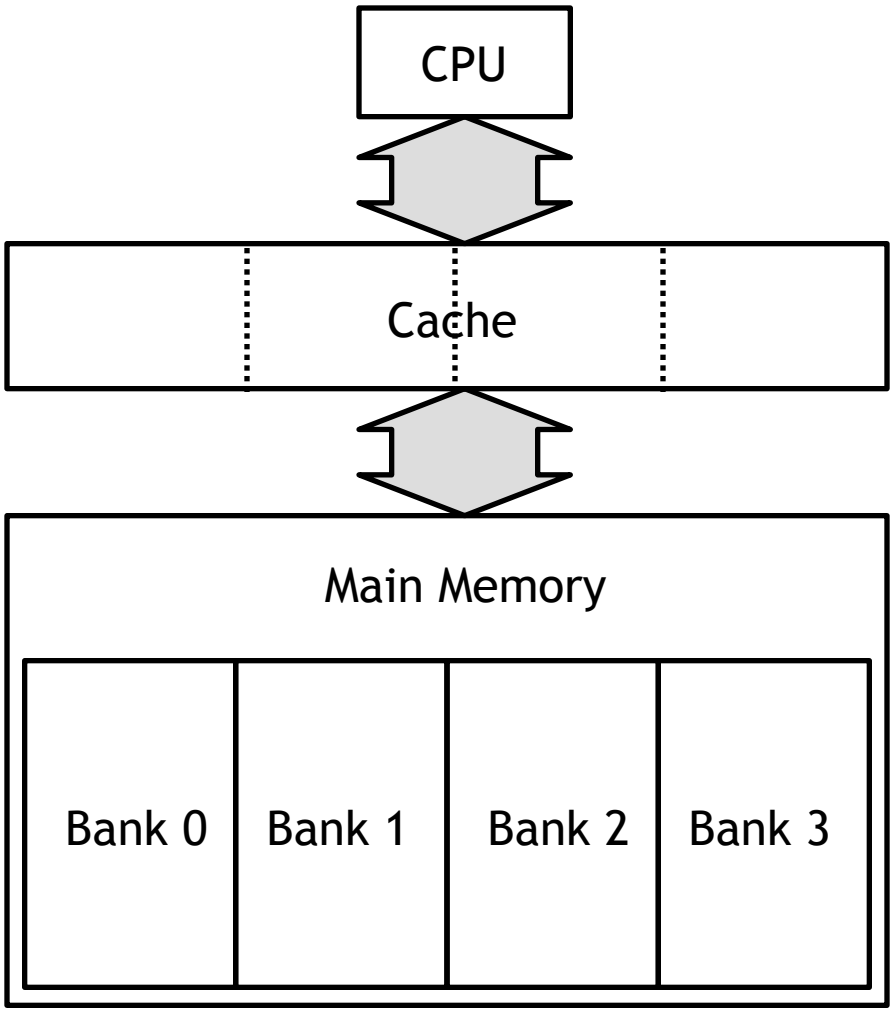


Bir İç İçe Geçmiş (Interleaved) Bellek

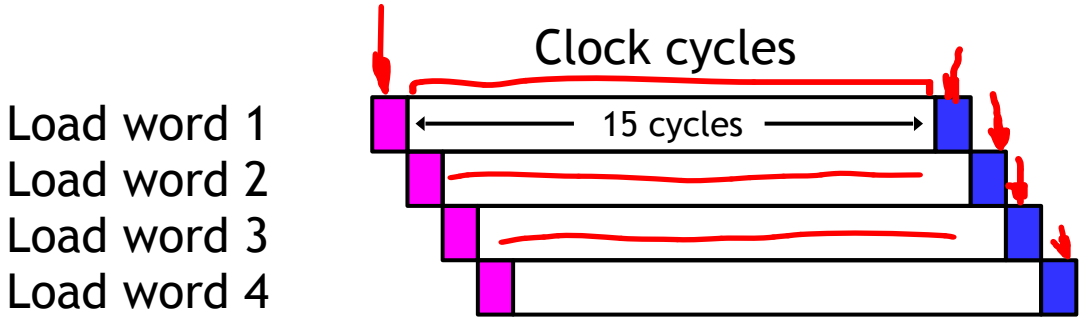
- Yeni bir yaklaşım ve en mantıklısı ‘interleaved’ bellektir. Bellek “banklar” halinde kullanılarak bağımsız erişim sağlanır.
- Bunun ana getirisi her kelimede yaşanan bellek gecikmesini üst üste koymaktır.
- Örneğin, ana belleğimiz 4 bank olsa ve her biri bir byte genişlikte olsa, 4 byte bilgi cache bloğuna sadece 20 saat sinyalinde ulaşır.

  1 + 15 + (4 x 1) = 20 saat sinyali

- Bus’ımızı bir byte genişliğe çektik, o yüzden cache’e her transfer 4 saat sinyali sürecektir.
- Bunu yapmak 4 byte bus’tan ucuz ama çok ta yavaş değil.



İç İçe Geçmiş Bellek Erişimleri



- Yukarıda bellek erişimlerinin nasıl iç içe geçtikleri gösteriliyor.
  - Mor saat sinyallerinde bir adres bellek bankına ulaştırılıyor.
  - Her bellek bankı 15-saat sinyali bekliyor ve mavi saat sinyali boyunca ana bellekten veri cache'e gidiyor.
- Bu iş hattı (ing: pipelining) ile aynı mantık!
  - Biz bir banktan veri istediğimizde peşinden diğer banktan veri istiyoruz.
  - Her bağımsız yükleme 17 saat sinyali ama iç içe geçmiş yüklemede 20 saat sinyaline gerek vardır.

Örnek: Hangisi Daha İyidir?

- Blok boyutunu (byte) arttırmak hit oranını iyileştiriyor (çünkü uzaysal yerellik artıyor), fakat transfer zamanı da artıyor. O halde hangi cache konfigürasyonu iyidir?

	Cache #1	Cache #2
Blok boyutu	32-byte	64-byte
Miss oranı	5%	4%

- Her iki cache'de de hit almak bir saat sinyalinde veriye ulaşmak demek varsayalım. Bellek erişimleri ise 15 saat sinyali ve bellek bus'ının genişliği 8-byte olsun:
  - Örneğin bir 16-byte bellek erişimi 18 saat sinyali sürer:

  1 (adres yollamak) + 15 (bellek erişimi) + 2 (iki adet 8-byte transfer)

... Sonraki slayda bakınız

Hatırlatma: AMAT = Hit time + (Miss rate x Miss penalty)

Örnek: Hangisi Daha İyidir?

- Blok boyutunu (byte) arttırmak hit oranını iyileştiriyor (çünkü uzaysal yerellik artıyor), fakat transfer zamanı da artıyor. O halde hangi cache konfigürasyonu iyidir?

	Cache #1	Cache #2
Blok boyutu	32-byte	64-byte
Miss oranı	5%	4%

- Her iki cache'de de hit almak bir saat sinyalinde veriye ulaşmak demek varsayalım. Bellek erişimleri ise 15 saat sinyali ve bellek bus'ının genişliği 8-byte olsun:

Cache #1:
Miss Penalty = 1 + 15 + 32B/8B = 20 saat sinyali
AMAT = 1 + (.05 * 20) = 2

Cache #2:
Miss Penalty = 1 + 15 + 64B/8B = 24 saat sinyali
AMAT = 1 + (.04 * 24) = ~1.96

CACHE DOSTU PROGRAM YAZMAK

Matris Çarpma Örneği

- Öncelikle belirtelim küçük matrislerde sorun yok, büyük matrislerde sıkıntı oluyor.
- Burada toplam cache boyutu da etkili şöyleki:
  - Geçici yerellik faydası var küçük bloklar cache'de kalıyor.
  - Uzaysal yerellik blok boyutuyla artıyor.

- Açıklama:
  - N x N matrisler çarpılıyor.
  - $O(N^3)$ toplam maliyet
  - Bellek erişimleri
    - Her iç döngüde N okuma
    - Her hedefe N değer toplama
      - Ama sum belki de işlemci register'ında tutulur.

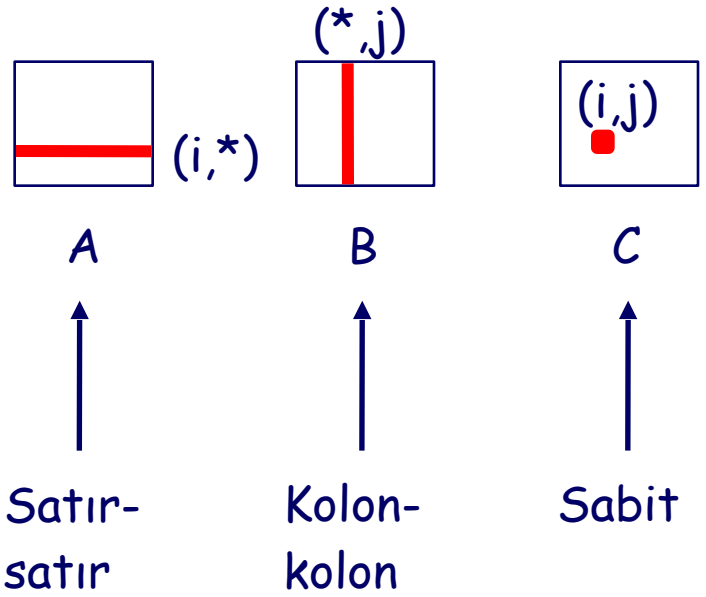
sum değişkeni
geçici yerellik

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Matris Çarpma Cache Miss Değerleri (ijk)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

İç döngü:

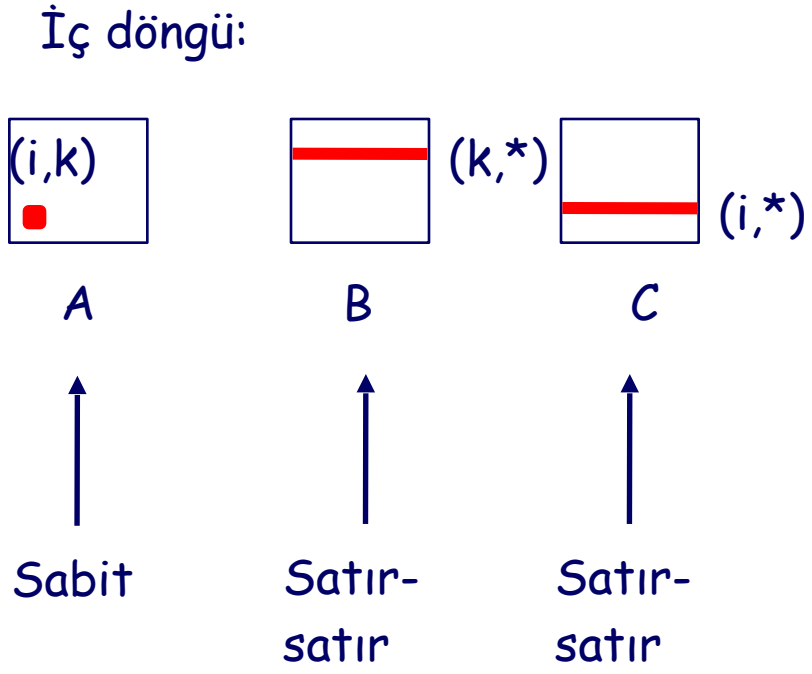


Her iç döngüdeki cache miss'ler:

A	B	C
0.25	1.0	0.0

Matris Çarpma Cache Miss Değerleri (kij)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```



Her iç döngüdeki cache miss'ler:

A	B	C
0.0	0.25	0.25

Özet

- Bir Cache'e yazmak için değişik senaryolar vardır.
  - Write-through ve write-back teknikleri cache belleği ana bellekle tutarlı tutuyor (write hit).
  - Write-around ve allocate-on-write write miss durumlarında etkili, ana belleğe direkt yazılan veriyi cache'e yazıp yazmamaya karar veriyorlar.
- Bellek sistem performansı gerçek ortamda çalışan programın hit time, miss rate ve miss penalty değerlerine bağlıdır.
  - Bu üç değerle ortalama bellek erişim zamanını bulabiliriz.
  - Ayrıca CPU zamanını da bellek bekleme süreleri (ing: stall cycles) ile daha doğru hesaplarız.

AMAT = Hit time + (Miss rate x Miss penalty)

Memory stall cycles = Memory accesses x miss rate x miss penalty

CPU time = (CPU execution cycles + Memory stall cycles) x Cycle time

- Bellek sisteminde seçilen organizasyon performansını direkt etkiler.
  - Cache boyutu, blok boyutu ve *k*-way (associativity) boyutu miss oranını etkiler.
  - Biz ana belleği 'miss penalties' değerini düşürecek şekilde, örneğin iç içe geçmiş veri transferleri kullanarak dizayn edebiliyoruz.