BİLGİSAYAR MİMARİSİ DERSİ

ARAŞTIRMA ÖDEVİ-III

Ayben GÜLNAR-191180041

KASIM 2022

İçindekiler Tablosu

1.GİRİŞ

Çoklu çekirdeklerin tanıtılmasıyla, bilimsel ve teknik uygulamalar daha iyi performans elde edebilir hale geldi.

N her işlemci için ayrı bir önbelleğe sahip paylaşımlı bellek çok işlemcili bir sistemde, işlenenin birçok kopyasına sahip olmak mümkündür. Örneğin bir kopya ana bellekte bulunurken diğer kopyalar önbellekte yer alabilir. Bir işlenenin bir kopyası değiştiğinde, diğer kopyalar da güncellenmelidir. Böylece, önbellek tutarlılığı, tüm işlenenlerdeki değişikliklerin sistem boyunca zamanında yayılmasını sağlar. Önbellek, bu tür sistemlerin tasarımında önemli bir rol oynar, ancak ölçeklenebilirlik, çoklu çekirdekler tarafından benimsenen önbellek tutarlılık mekanizmalarının performansını etkileyebilir. Günümüzün çoklu çekirdekleri, paylaşılan bellek mimarisi kullanılarak uygulanmaktadır ve herhangi bir yerel değişiklik, önbellek tutarsızlığı sorunuyla sonuçlanan tutarsız bir bellek görünümüne neden olur. Önbellek tutarsızlığı, iki farklı çekirdeğin aynı bellek konumu için iki farklı değere sahip olması olarak tanımlanabilir. Özel önbelleklere sahip paylaşımlı bellekte, birden çok işlemcide önbelleğe alınan bellek konumlarında bazı değişiklikler yapıldığında, tutarlılığı sağlamak için bazı mekanizmalar mevcut olmalıdır.[1,2]

Önbellek tutarlılığı, birden çok yerel önbellekte paylaşılan ve depolanan verilerin tutarlılığıdır. Bu nedenle, farklı çok işlemcili önbellek tarafından aynı bellek konumuna Update veya yazmadan doğabilecek tutarsız veri sorununu çözmek için önbellek tutarlılık protokolleri kullanılır. Ayrıca, bu protokoller istenen bellek modelinin uygulanmasına yardımcı olur ve herhangi bir bellek konumuna erişimin yine de o konumda yazılan en son değeri döndürmesini garanti eder. Sonuç olarak, önbellek tutarlılık protokolleri, paylaşılan önbellek blokları arasında veri tutarlılığını sağlamak için bir çözüm olarak düşünülebilir. Önbellek tutarlılığının korunması, yazılım veya donanım çözümü yoluyla sağlanabilir. Her iki yaklaşımda da bir işlemci tarafından yapılan Updateler diğerlerine iletilmeli ve tüm işlemcilerin belleğinin tutarlı bir görünümü olmalıdır. Yani ana fikir, işlemcinin tüm işlemlerini diğer tüm işlemcilere yayınlaması ve diğer tüm işlemcilerin buna göre paylaşılan bloğun durumunu değiştirmesi gerektiğidir. En yaygın kullanılan önbellek tutarlılık protokolleri şunlardır: 1. MI (Değiştirilmiş (M), Nvalid (I)) 2. MSI (Değiştirilmiş (M), Paylaşılan (S), Nvalid (I)) 3. MESI (Değiştirilmiş (M) , Özel (E), Paylaşılan (S) ve Nvalid (I)), 4. MOSI (Değiştirilmiş (M), Sahipli (O), Paylaşılmış (S) ve Nvalid (I)) 5. MOESI (Değiştirilmiş (M)), Sahip Olunan (O), Özel (E), Paylaşılan (S), Nvalid (I)) 6. MESIF (Değiştirilmiş (M), Özel (E), Paylaşılan (S), Nvalid (I),

İlet (F)) 7. MESRSI (Değiştirilmiş (M), Özel (E), Paylaşılan (S), En Son veya Salt Okunur (R), Nvalid (I)) 8. Bir kez yaz ve Synapse, Berkeley, Firefly ve Dragon Protokolleri.[2]

Bu araştırmamda öncelikli olarak multicore ve multiprocessorların ne olduğunu detaylıca incelendi. Bu iki yapı karşılaştırıldı. Tanımlamalar anlaşıldıktan sonra ise cache coherence kavramına ve sonrasında ise detaylıca protokoller ele alındı. Protokollerin tüm detay ve şekillerini aşağıdaki bölümlerde yer verilmiştir.

2.MULTIPROCESSORS

Paylaşılan bellek çok işlemcili, iki veya daha fazla CPU'nun ortak bir RAM'e tam erişimi paylaştığı bir bilgisayar sistemidir. CPU'lardan herhangi birinde çalışan bir program, normal bir sanal adres alanı görür. Bu sistemin sahip olduğu tek alışılmadık özellik, CPU'nun bir bellek sözcüğüne bir değer yazabilmesi ve ardından sözcüğü geri okuyabilmesi ve farklı bir değer alabilmesidir. Doğru düzenlendiğinde, bu özellik işlemciler arası iletişimin temelini oluşturur: bir CPU bazı verileri belleğe yazar ve diğeri verileri okur. Çoğunlukla, çok işlemcili işletim sistemleri sadece normal işletim sistemleridir. Sistem çağrılarını yönetirler, bellek yönetimi yaparlar, bir dosya sistemi sağlarlar ve G/Ç cihazlarını yönetirler. Yine de benzersiz özelliklere sahip oldukları bazı alanlar vardır. Bunlara süreç senkronizasyonu, kaynak yönetimi ve zamanlama dahildir.

2.1 Multiprocessor Hardware

Tüm çoklu işlemciler, her CPU'nun tüm belleğe hitap edebilme özelliğine sahip olmasına rağmen, bazı çoklu işlemciler, her bellek kelimesinin diğer her bellek sözcüğü kadar hızlı okunabilmesi gibi ek bir özelliğe sahiptir. Bu makinelere UMA (Tekdüzen Bellek Erişimi) çok işlemcili denir. Buna karşın, NUMA (Düzensiz Bellek Erişimi) çoklu işlemcilerinde bu özellik yoktur.

2.1.1UMA Bus-Based SMP Mimarisi

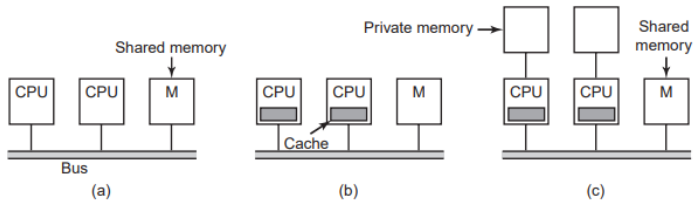


Figure 8-1. Three bus-based multiprocessors. (a) Without caching. (b) With caching. (c) With caching and private memories.

Şekil 2.1.1

En basit çoklu işlemciler, Şekil 8-1(a)'da gösterildiği gibi tek bir veri yoluna dayalıdır. İki veya daha fazla CPU ve bir veya daha fazla bellek modülü, iletişim için aynı veri yolunu kullanır. Bir CPU bir hafıza kelimesini okumak istediğinde, önce veri yolunun meşgul olup olmadığını kontrol eder. Eğer veri yolu boşta ise, CPU istediği kelimenin adresini veri yoluna koyar, birkaç kontrol sinyali verir ve bellek veri yoluna istenen kelimeyi koyana kadar bekler. Bir CPU belleği okumak veya yazmak istediğinde veri yolu meşgulse, CPU sadece veri yolu boşta kalana kadar bekler. İşte bu tasarımdaki sorun burada yatıyor. İki veya üç CPU ile, veri yolu çekişmesi

yönetilebilir olacaktır; 32 veya 64 ile dayanılmaz olacak. Sistem, veri yolunun bant genişliği ile tamamen sınırlandırılacak ve CPU'ların çoğu zaman boşta kalacaktır. Bu sorunun çözümü, Şekil 8-1(b)'de gösterildiği gibi her CPU'ya bir önbellek eklemektir. Önbellek, CPU çipinin içinde, CPU çipinin yanında, işlemci kartında veya üçünün bir kombinasyonu olabilir. Artık birçok okuma yerel önbellekten karşılanabileceğinden, çok daha az veri yolu trafiği olacak ve sistem daha fazla CPU'yu destekleyebilecektir. Genel olarak önbelleğe alma, tek tek sözcük bazında değil, 32 veya 64 baytlık bloklar temelinde yapılır. Bir kelimeye referans verildiğinde, bloğun tamamı ona dokunan CPU'nun önbelleğine alınır. Her bir önbellek bloğu, salt okunur (bu durumda aynı anda birden fazla önbellekte bulunabilir) veya okuma-yazma (bu durumda diğer önbelleklerde bulunmayabilir) olarak işaretlenir. Bir CPU, bir veya daha fazla uzak önbellekte bulunan bir kelimeyi yazmaya çalışırsa, veri yolu donanımı yazmayı algılar ve diğer tüm önbellekleri yazma hakkında bilgilendiren veri yoluna bir sinyal koyar. Diğer önbelleklerin "temiz" bir kopyası, yani bellekte olanın tam bir kopyası varsa, kopyalarını atabilir ve yazarın önbellek bloğunu değiştirmeden önce bellekten almasına izin verebilirler. Başka bir önbelleğin "kirli" (yani değiştirilmiş) bir kopyası varsa, yazma işlemi devam etmeden önce onu belleğe geri yazmalı veya veri yolu üzerinden doğrudan yazıcıya aktarmalıdır. Birçok önbellek aktarım protokolü mevcuttur. Yine başka bir olasılık, Şekil 8-1(c)'deki tasarımdır; burada her CPU yalnızca bir ön belleğe değil, aynı zamanda tahsis edilmiş (özel) bir veri yolu üzerinden eriştiği yerel, özel bir belleğe de sahiptir. Bu yapılandırmayı en iyi şekilde kullanmak için, derleyici tüm program metnini, dizeleri, sabitleri ve diğer salt okunur verileri, yığınları ve yerel değişkenleri özel belleklere yerleştirmelidir. Paylaşılan bellek daha sonra yalnızca yazılabilir paylaşılan değişkenler için kullanılır. Çoğu durumda, bu dikkatli yerleştirme, veri yolu trafiğini büyük ölçüde azaltacaktır, ancak derleyicinin aktif iş birliğini gerektirir.

2.1.2 UMA Multiprocessors Using Crossbar Switches

En iyi önbelleğe almada bile, tek bir veri yolunun kullanılması, bir UMA çok işlemcisinin boyutunu yaklaşık 16 veya 32 CPU ile sınırlar. Bunun ötesine geçmek için farklı türde bir ara bağlantı ağına ihtiyaç vardır. N CPU'yu k belleğe bağlamak için en basit devre, Şekil 8-2'de gösterilen çapraz çubuk anahtarıdır. Çapraz çubuk anahtarları, bir grup gelen hattı bir dizi giden hatta keyfi bir şekilde bağlamak için telefon anahtarlama santrallerinde onlarca yıldır kullanılmaktadır. Yatay (gelen) ve dikey (giden) çizginin her kesişme noktasında bir kesişme noktası vardır. Çapraz nokta, yatay ve dikey hatların bağlanıp bağlanmayacağına bağlı olarak elektriksel olarak açılıp kapatılabilen küçük bir anahtardır. Şekil 8-2(a)'da aynı anda kapalı olan ve (CPU, bellek) arasında bağlantılara izin veren üç çapraz nokta görüyoruz.

(001, 000), (101, 101) ve (110, 010) çiftleri aynı anda. Başka birçok kombinasyon da mümkündür. Aslında, kombinasyonların sayısı, sekiz kalenin bir satranç tahtasına güvenli bir şekilde yerleştirilebileceği farklı yolların sayısına eşittir.

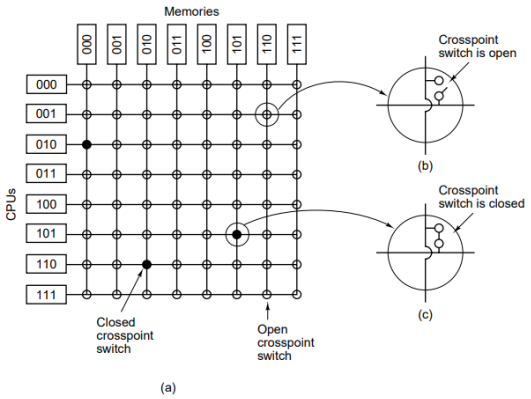


Figure 8-2. (a) An 8 × 8 crossbar switch. (b) An open crosspoint. (c) A closed crosspoint.

Şekil 2.1.2

Çapraz çubuk anahtarının en güzel özelliklerinden biri, bloke etmeyen bir ağ olmasıdır; bu, bazı çapraz noktalar veya hatlar zaten dolu olduğu için (bellek modülünün kendisinin mevcut olduğu varsayılarak) hiçbir CPU'nun ihtiyaç duyduğu bağlantıyı reddetmediği anlamına gelir. ayrıca önceden planlamaya gerek yoktur. Halihazırda yedi rasgele bağlantı kurulmuş olsa bile, kalan CPU'yu kalan belleğe bağlamak her zaman mümkündür.

Çapraz çubuk anahtarının en kötü özelliklerinden biri, çapraz nokta sayısının n2 olarak artmasıdır. 1000 CPU ve 1000 bellek modülü ile bir milyon çapraz noktaya ihtiyacımız var. Böyle büyük bir çapraz çubuk anahtarı mümkün değildir. Bununla birlikte, orta ölçekli sistemler için bir çapraz çubuk tasarımı uygulanabilir.

2.1.3 NUMA Multiprocessors

Tek veri yolu UMA çoklu işlemcileri genellikle birkaç düzineden fazla CPU ile sınırlı değildir ve çapraz çubuk veya anahtarlamalı çoklu işlemciler çok sayıda (pahalı) donanım gerektirir ve o kadar da büyük değildir. 100'den fazla CPU'ya ulaşmak için bir şeyler vermek gerekir. Genellikle, tüm bellek modüllerinin aynı erişim süresine sahip olduğu fikrini verir. Bu imtiyaz, yukarıda bahsedildiği gibi NUMA çok işlemcili fikrine yol açar. UMA kuzenleri gibi, tüm CPU'larda tek bir adres alanı sağlarlar, ancak UMA makinelerinden farklı olarak, yerel bellek modüllerine erişim uzak bellek modüllerine erişimden daha hızlıdır. Böylece tüm UMA programları NUMA makinelerinde değişmeden çalışacak, ancak performans aynı saat hızında bir UMA makinesinden daha kötü olacaktır.

NUMA makineleri, hepsinin sahip olduğu ve onları diğer çoklu işlemcilerden ayıran üç temel özelliğe sahiptir:

1. Tüm CPU'lar tarafından görülebilen tek bir adres alanı vardır.

2. Uzak belleğe erişim LOAD ve STORE komutları ile yapılır.

3. Uzak belleğe erişim, yerel belleğe erişimden daha yavaştır.

Uzak belleğe erişim zamanı gizlenmediğinde (çünkü önbelleğe alma yoktur), sisteme NC-NUMA adı verilir. Tutarlı önbellekler mevcut olduğunda, sistem CC-NUMA (Önbellekle Uyumlu NUMA) olarak adlandırılır.

Şu anda büyük CC-NUMA çoklu işlemcileri oluşturmak için en popüler yaklaşım, Directory tabanlı çoklu işlemcidir. Buradaki fikir, her bir önbellek satırının nerede olduğunu ve durumunun ne olduğunu söyleyen bir veri tabanı sürdürmektir. Bir önbellek satırına referans verildiğinde, veri tabanının nerede olduğu ve temiz mi kirli mi (değiştirilmiş) olup olmadığı sorgulanır. Bu veri tabanının belleğe başvuran her talimatta sorgulanması gerektiğinden, bir veri yolu döngüsünün çok kısa bir bölümünde yanıt verebilen son derece hızlı özel amaçlı donanımda tutulmalıdır.

Şekil 8-5(c)'de satırın önbelleğe alınmadığını görüyoruz, bu nedenle donanım 4. satırı yerel RAM'den alıyor. , onu düğüm 20'ye geri gönderir ve Directory girişi 4'ü satırın artık düğüm 20'de önbelleğe alındığını gösterecek şekilde günceller. Şimdi bu kez düğüm 36'nın satır 2'sini soran ikinci bir isteği ele alalım. Bu satırın 82. düğümde önbelleğe alındığını görüyoruz. Bu noktada donanım, hattın artık 20. düğümde olduğunu söylemek için Directory girişi 2'yi güncelleyebilir ve ardından 82. düğüme, hattı 20. düğüme geçirmesi ve nvalid kılması talimatını veren bir mesaj gönderebilir. [3]

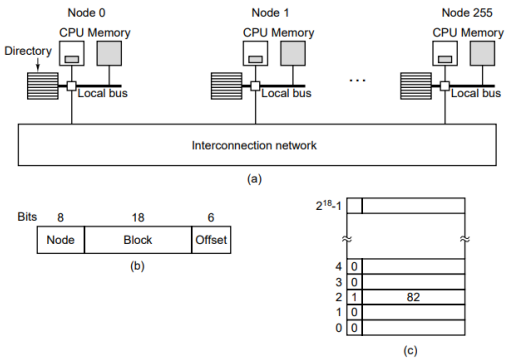


Figure 8-5. (a) A 256-node directory-based multiprocessor. (b) Division of a 32-bit memory address into fields. (c) The directory at node 36.

Şekil 2.1.3 [3]

3. MULTİCORE PROCESSORS

Bir işlemci çipindeki mimari organizasyon, sağlanan kaynakları verimli bir şekilde kullanmak için açıkça paralel programların kullanılmasını gerektirebilir. Gereken çoklu kontrol akışlarına genellikle iş parçacığı adı verildiğinden buna iş parçacığı düzeyinde paralellik denir. Karşılık gelen mimari organizasyon, çip çoklu işleme (CMP) olarak da adlandırılır. CMP'ye bir örnek, tüm yürütme kaynaklarıyla birlikte birden çok bağımsız yürütme çekirdeğinin tek bir işlemci çipine yerleştirilmesidir. Ortaya çıkan işlemcilere çok çekirdekli işlemciler denir.

Moore yasasına göre, bir işlemci çipindeki transistör sayısı her 18-24 ayda bir ikiye katlanıyor. Bu muazzam artış, donanım üreticilerinin uzun yıllardır uygulama programları için önemli bir performans artışı sağlamasına olanak sağlamıştır, ayrıca bkz. 2.1. Bu nedenle, tipik bir bilgisayar en fazla 5 yıl sonra eski moda ve çok yavaş kabul edilir ve müşteriler oldukça sık yeni bilgisayar satın alır. Donanım üreticileri bu nedenle bilgisayar satış rakamlarının düşmemesi için elde ettikleri performans artışını en azından mevcut seviyede tutmaya çalışıyorlar. Bölümde tartışıldığı gibi. 2.1'de, yıllık performans artışı için en önemli faktörler, saat hızındaki artış ve talimatların ardışık düzende yürütülmesi ve çoklu işlevsel birimlerin kullanımı gibi paralel işlemenin dahili kullanımı olmuştur. Ancak bu geleneksel teknikler esas olarak sınırlarına ulaştı:

• İşlemci çipine ek işlevsel birimler koymak mümkün olsa da, tek bir kontrol iş parçacığının yönergeleri arasındaki bağımlılıklar bunların paralel yürütülmesini engellediğinden, bu çoğu uygulama programı için performansı artırmaz. Tek bir kontrol akışı, çok sayıda işlevsel birimi meşgul etmek için yeterli komut düzeyinde paralellik sağlamaz.

• İşlemci saatlerinin hızının önemli ölçüde artırılamamasının iki ana nedeni vardır [106]. İlk olarak, bir çip üzerindeki transistör sayısındaki artış, esas olarak transistör yoğunluğunun arttırılmasıyla sağlanır. Ancak bu aynı zamanda kaçak akım ve güç tüketimi nedeniyle güç yoğunluğunu ve ısı üretimini artırır, dolayısıyla soğutma için daha fazla efor ve daha fazla enerji gerektirir.

İkincisi, bellek erişim süresi, işlemci saat periyodu ile aynı oranda azaltılamaz. Bu, bellek erişimi için artan sayıda makine döngüsüne yol açar. Örneğin, 1990'da tipik bir masaüstü bilgisayar sistemi için ana belleğe erişim 6 ila 8 döngü arasındayken, 2006'da ana belleği oluşturmak için kullanılan DRAM teknolojisine bağlı olarak bellek erişimi tipik olarak 100 ila 250 döngü arasında sürdü. Bu nedenle, bellek erişim süreleri daha fazla performans artışı için

sınırlayıcı bir faktör haline gelebilir ve bunu önlemek için önbellek bellekleri kullanılır. İşlemci tasarımcılarının yüzleşmesi gereken daha fazla sorun var: İşlemci mimarisinin karmaşıklığını artırmak için artan sayıda transistör kullanmak, işlemcinin fonksiyonel birimleri arasında kontrol ve veri aktarımı için işlemci-dahili kablo uzunluğunda bir artışa da yol açabilir. Burada, kablolar içindeki sinyal transferlerinin hızı sınırlayıcı bir faktör haline gelebilir. Kullanılabilecek pin sayısı, dolayısıyla CPU ve ana bellek arasındaki bant genişliğini sınırlar. Bu, bazen bellek duvarı olarak adlandırılan bir işlemciden belleğe performans boşluğuna yol açabilir. Bu, verimli bir önbellek hiyerarşisi ile yüksek bant genişliğine sahip bellek mimarilerinin kullanımını gerekli kılar. Tüm bu nedenler, geleneksel teknikler kullanılarak önceki hızda bir işlemci performansı artışını engeller. Bunun yerine, yeni işlemci mimarilerinin kullanılması gerekiyor ve tek bir işlemci kalıbında birden fazla çekirdeğin kullanılması en umut verici yaklaşım olarak görülüyor. Bu yaklaşım, bir işlemci çipinin iç organizasyonunun karmaşıklığını daha da artırmak yerine, nispeten basit bir mimariye sahip birden çok bağımsız işlem çekirdeğini tek bir işlemci çipinde birleştirir. Bu, bir işlemci çipinin enerji tüketiminin gerekirse boşta kalma sırasında kullanılmayan işlemci çekirdeklerini kapatarak azaltılabilmesi gibi ek bir avantaja sahiptir. Çok çekirdekli işlemciler, birden çok yürütme çekirdeğini tek bir işlemci çipinde birleştirir. İşletim sistemi için her yürütme çekirdeği, işlevsel birimler veya yürütme boru hatları gibi ayrı yürütme kaynaklarına sahip bağımsız bir mantıksal işlemciyi temsil eder. Her çekirdeğin ayrı ayrı kontrol edilmesi gerekir ve işletim sistemi, paralel bir yürütme elde etmek için farklı çekirdeklere farklı uygulama programları atayabilir. Virüs kontrolü, görüntü sıkıştırma ve kodlama gibi arka plan uygulamaları, kullanıcının uygulama programlarına paralel olarak çalışabilir. Paralel programlama tekniklerini kullanarak, hesaplama açısından yoğun bir uygulama programını (bilgisayar oyunları, bilgisayar görüşü veya bilimsel simülasyonlar gibi) bir dizi çekirdek üzerinde paralel olarak yürütmek de mümkündür, böylece yürütme süresini bir çekirdek üzerinde yürütmeye kıyasla azaltır. Tek çekirdekli veya sıralı durumda olduğu gibi daha fazla hesaplama yaparak daha doğru sonuçlara yol açar. Gelecekte, bilgisayar oyunları gibi standart uygulama programlarının kullanıcıları muhtemelen bir işlemci çipinin yürütme çekirdeklerinin verimli bir şekilde kullanılmasını bekleyeceklerdir. Bunu başarmak için, programcılar paralel programlama tekniklerini kullanmak zorundadır. Tek bir işlemci çipinde birden çok çekirdeğin kullanılması, metin işleme, ofis uygulamaları veya bilgisayar oyunları gibi standart programların arka planda ayrı bir çekirdekte hesaplanan ek özellikler sağlamasına da olanak tanır, böylece kullanıcı herhangi bir gecikme fark etmez. Ancak yine de uygulama için paralel programlama teknikleri kullanılmalıdır. [4]

Tablo 1 Karşılaştırma Tablosu

Table 4

	Multiprocessor system	Multicore system
Integration level	Each processor in a chip	All processors on the same chip
Processor performance	High	Low
System performance	Very high	High
Processor power consumption	High	Low
Total power consumption	Relatively high	Relatively low

4. CONSISTENCY & COHERENCE

Tutarlılık modelleri, önbelleklere veya tutarlılığa atıfta bulunmadan, yükler ve depolar. Paylaşılan bellek davranışını tanımlar. Tutarlılık modellerine neden ihtiyacımız olduğuna dair bazı gerçek dünya sezgileri elde etmek için ders programını çevrimiçi olarak yayınlayan bir üniversiteyi düşünün. Bilgisayar Mimarisi dersinin başlangıçta 152 numaralı odada olması planlandığını varsayalım. Derslerin başlamasından bir gün önce, üniversite kayıt memuru sınıfı 252 numaralı odaya taşımaya karar verir. Ve birkaç dakika sonra, kayıt memuru tüm kayıtlı öğrencilere yeni güncellenen programı kontrol etmeleri için bir metin mesajı gönderir. Çalışkan bir öğrencinin kısa mesajı aldığı, hemen çevrimiçi programı kontrol ettiği ve yine de (eski) sınıfın yerini gözlemlediği bir senaryo hayal etmek zor değil - örneğin web sitesi yöneticisi Updateyi hemen gönderemeyecek kadar meşgulse - Oda 152. Çevrimiçi program sonunda 252 numaralı odaya güncellense ve kayıt memuru "yazmaları" doğru sırayla gerçekleştirmiş olsa da bu çalışkan öğrenci onları farklı bir sırayla gözlemledi ve bu nedenle yanlış odaya gitti. Tutarlılık modeli, bu davranışın doğru (ve dolayısıyla bir kullanıcının istenen sonuca ulaşmak için başka bir işlem yapması gerekip gerekmediğini) veya yanlış (bu durumda sistemin bu yeniden sıralamaları engellemesi gerekir) olup olmadığını tanımlar. Bu yapmacık örnek birden çok ortam kullansa da benzer davranış sıra dışı işlemci çekirdekleri, yazma arabellekleri, önceden getirme ve çoklu önbellek bankaları olan paylaşılan bellek donanımlarında meydana gelebilir. Bu nedenle, programcıların ne bekleyeceklerini bilmeleri ve uygulayıcıların sağlayabileceklerinin sınırlarını bilmeleri için paylaşılan bellek doğruluğunu - yani hangi paylaşılan bellek davranışlarına izin verildiğini - tanımlamamız gerekir. Paylaşılan bellek doğruluğu, bir bellek tutarlılık modeli veya daha basit bir şekilde bir bellek modeli tarafından belirlenir. Bellek modeli, paylaşılan bellekle çalışan çok iş parçacıklı programların izin verilen davranışını belirtir. Belirli girdi verileriyle çalışan çok iş parçacıklı bir program için bellek modeli, dinamik yüklerin hangi değerleri döndürebileceğini ve isteğe bağlı olarak belleğin olası son durumlarının neler olduğunu belirtir. Tek iş parçacıklı yürütmeden farklı olarak, birden çok

doğru davranışa genellikle izin verilir, bu da bellek tutarlılık modellerinin anlaşılmasını incelikli hale getirir. Dikkat edilmezse, birden çok aktörün (örneğin, birden çok çekirdek) bir verinin birden çok kopyasına (örneğin, birden çok önbellekte) erişimi varsa ve en az bir erişim bir yazma ise, bir tutarlılık sorunu ortaya çıkabilir. Bellek tutarlılığı örneğine benzer bir örnek düşünün. Bir öğrenci çevrimiçi ders programını kontrol eder, Bilgisayar Mimarisi dersinin 152 numaralı odada yapıldığını gözlemler (veriyi okur) ve bu bilgiyi cep telefonundaki takvim uygulamasına kopyalar (veriyi önbelleğe alır). Ardından, üniversite kayıt memuru sınıfın 252 numaralı odaya taşınmasına karar verir, çevrimiçi programı günceller (veriye yazar) ve öğrencilere bir metin mesajı ile bilgi verir. Öğrencinin kopyası artık eski ve tutarsız bir durumla olur. 152 numaralı odaya giderse sınıfını bulamayacak. Bilgi işlem dünyasından tutarsızlık örnekleri, bilgisayar mimarisi hariç, eski web önbelleklerini ve güncellenmemiş kod havuzlarını kullanan programcıları içerir. Eski verilere erişim (tutarsızlık), bir sistem içindeki dağıtılmış aktörler grubu tarafından uygulanan bir dizi kural olan bir tutarlılık protokolü kullanılarak engellenir. Temel olarak, varyantların tümü, yazmayı tüm önbelleklere yayarak, yani takvimi çevrimiçi programla senkronize halde tutarak, bir işlemcinin yazmasını diğer işlemciler tarafından görünür kılar. Ancak protokoller, senkronizasyonun ne zaman ve nasıl gerçekleştiği konusunda farklılık gösterir. Tutarlılık protokollerinin iki ana sınıfı vardır. İlk yaklaşımda tutarlılık protokolü, yazma işlemlerinin önbelleklere eşzamanlı olarak yayılmasını sağlar. Çevrimiçi program güncellendiğinde, tutarlılık protokolü öğrencinin takviminin de güncellenmesini sağlar. İkinci yaklaşımda, tutarlılık protokolü, tutarlılık modelini onurlandırmaya devam ederken, yazma işlemlerini önbelleklere eş zamansız olarak yayar. Tutarlılık protokolü, çevrimiçi program güncellendiğinde yeni değerin öğrencinin takvimine de yayılacağını garanti etmez; ancak protokol, kısa mesaj cep telefonuna ulaşmadan önce yeni değerin yayılmasını sağlar. [5,7]

4.1 Heterojen Sistemlerde Consıstency And Coherence

Modern bilgisayar sistemleri ağırlıklı olarak heterojendir. Günümüzde bir cep telefonu işlemcisi yalnızca çok çekirdekli bir CPU içermekle kalmaz, aynı zamanda bir GPU ve diğer hızlandırıcılara (örneğin sinir ağı donanımı) sahiptir. Programlanabilirlik arayışında, bu tür heterojen sistemler paylaşılan belleği desteklemeye başlıyor. 10. Bölüm, bu tür heterojen işlemciler için tutarlılık ve uyumluluğu ele alır.

Bu bölüm, günümüzün tartışmasız en popüler hızlandırıcıları olan GPU'lara odaklanarak başlıyor. Bu bölüm, GPU'ların başlangıçta donanım önbellek tutarlılığını desteklememeyi seçtiğini gözlemliyor, çünkü GPU'lar verileri çok fazla senkronize etmeyen veya paylaşmayan

utanç verici derecede paralel grafik iş yükleri için tasarlandı. Bununla birlikte, donanım önbelleği tutarlılığının olmaması, GPU'lar ayrıntılı senkronizasyon ve veri paylaşımı ile genel amaçlı iş yükleri için kullanıldığında programlanabilirlik ve/veya performans sorunlarına yol açar. Bu bölüm, bu sınırlamaların üstesinden gelen umut vaat eden tutarlılık alternatiflerinden bazılarını ayrıntılı olarak tartışıyor - özellikle aday protokollerin tutarlılığı agnostik bir şekilde tutarlılık uygulamak yerine neden tutarlılık modelini doğrudan uyguladığını açıklayarak. Bölüm, CPU'lar ve hızlandırıcılar arasındaki tutarlılık ve tutarlılık hakkında kısa bir tartışma ile sona eriyor. [5]

4.2 Specıfyıng And Valıdatıng Memory Consıstency Models And Cache Coherence

Tutarlılık modelleri ve tutarlılık protokolleri karmaşık ve inceliklidir. Yine de bu karmaşıklık, çoklu çekirdeklerin programlanabilir olmasını ve tasarımlarının doğrulanabilmesini sağlamak için yönetilmelidir. Bu hedeflere ulaşmak için, tutarlılık modellerinin resmi olarak belirtilmesi çok önemlidir. Resmi bir belirtim, programcıların bellek modeli tarafından hangi davranışlara izin verildiğini ve hangi davranışlara izin verilmediğini (araç desteğiyle) açık ve kapsamlı bir şekilde anlamalarını sağlar. İkinci olarak, uygulamaları doğrulamak için kesin bir resmi belirtim zorunludur. [5]

4.3 Coherence Temelleri

Temel sistem modeli, Şekil 2.1'de gösterildiği gibi tek bir çok çekirdekli işlemci çipi ve çip dışı ana bellek içerir. Çok çekirdekli işlemci çipi, her biri kendi özel veri önbelleğine ve tüm çekirdekler tarafından paylaşılan son düzey önbelleğe (LLC) sahip birden çok tek iş parçacıklı çekirdekten oluşur. Bu kılavuz boyunca, "önbellek" terimini kullandığımızda, LLC'yi değil, bir çekirdeğin özel veri önbelleğini kastediyoruz. Her çekirdeğin veri önbelleğine fiziksel adreslerle erişilir ve geri yazılır. Çekirdekler ve LLC birbirleriyle bir ara bağlantı ağı üzerinden iletişim kurar. LLC, işlemci çipinde olmasına rağmen mantıksal olarak bir "bellek tarafı önbelleğidir" ve bu nedenle başka bir düzeyde tutarlılık sorunu getirmez. LLC mantıksal olarak belleğin hemen önündedir ve bellek erişimlerinin ortalama gecikmesini azaltmaya ve belleğin etkin bant genişliğini artırmaya hizmet eder. LLC ayrıca bir çip üzerinde bellek denetleyicisi olarak da hizmet vermektedir. Bu temel sistem modeli, yaygın olan ancak bu ilk metnin çoğu için gerekli olmayan birçok özelliği atlar. Bu özellikler, talimat önbelleklerini, çok düzeyli önbellekleri, birden çok çekirdek arasında paylaşılan önbellekleri, sanal olarak adreslenmiş önbellekleri, TLB'leri ve tutarlı doğrudan bellek erişimini (DMA) içerir. Temel sistem modeli ayrıca birden fazla çok çekirdekli yonga olasılığını da göz ardı eder. Tüm bu özellikleri daha sonra tartışacağız, ancak şimdilik gereksiz bir karmaşıklık katacaklardır. [5]

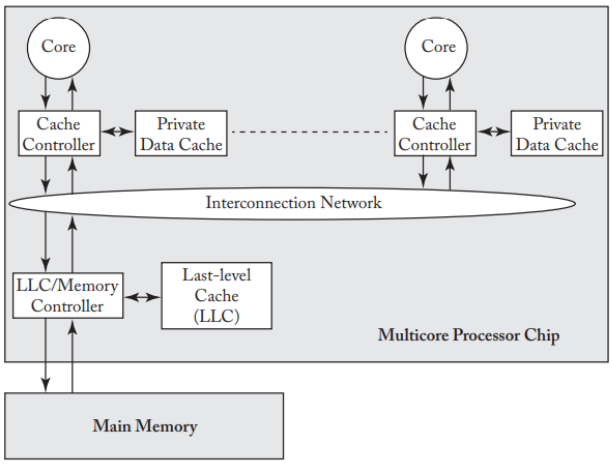


Figure 2.1: Baseline system model used throughout this primer.

Şekil 4.3.1 [5]

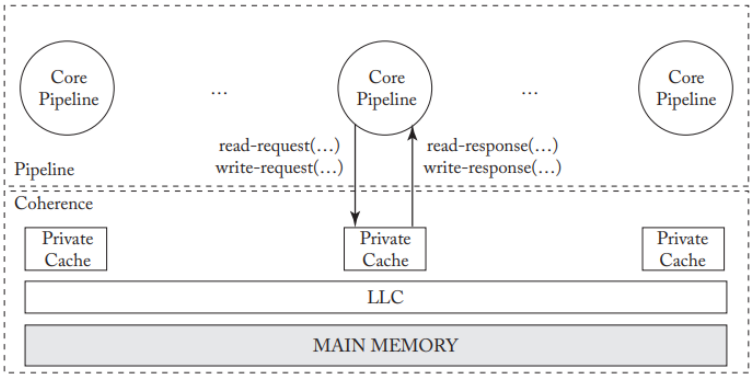


Figure 2.2: The pipeline-coherence interface.

Şekil 4.3.2 [5]

Tutarlılığı, tek yazarlı-çoklu okuyuculu (SWMR) değişmezi aracılığıyla tanımlarız. Belirli bir bellek konumu için, herhangi bir zamanda, onu yazabilen (ve okuyabilen) tek bir çekirdek veya onu okuyabilen birkaç çekirdek vardır. Bu nedenle, belirli bir bellek konumunun bir çekirdek tarafından yazılabileceği ve aynı anda diğer çekirdekler tarafından okunabileceği veya yazılabileceği bir zaman asla yoktur. Bu tanımı görmenin başka bir yolu, her bir bellek konumu için, bellek konumunun ömrünün çağlara bölündüğünü düşünmektir. Her çağda, ya tek bir çekirdeğin okuma-yazma erişimi vardır ya da birkaç çekirdeğin (muhtemelen sıfır) salt okunur erişimi vardır. Şekil 2.3, SWMR değişmezini koruyan dört döneme bölünmüş örnek bir bellek konumunun ömrünü göstermektedir. [5]

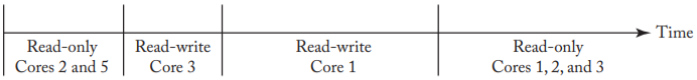


Figure 2.3: Dividing a given memory location's lifetime into epochs.

Şekil 4.3.3

Coherence invariants

1. **Single-Writer, Multiple-Read (SWMR) Invariant.** For any memory location A, at any given time, there exists only a single core that may write to A (and can also read it) or some number of cores that may only read A.
2. **Data-Value Invariant.** The value of the memory location at the start of an epoch is the same as the value of the memory location at the end of the its last read-write epoch.

Şekil 4.3.4

5. COHERENCE PROTOCOLS

Tutarlılık protokolünün amacı, değişmezleri uygulayarak tutarlılığı sürdürmektir.

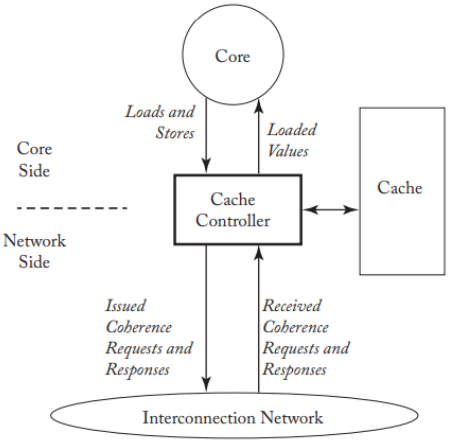


Figure 6.1: Cache controller.

Şekil 5.1

Bu değişmezleri uygulamak için, her depolama yapısıyla (her bir önbellek ve LLC/bellek) tutarlılık denetleyicisi adı verilen sonlu durum makinesini ilişkilendiririz. Bu tutarlılık denetleyicilerinin toplanması, her blok için SWMR'nin ve veri değeri değişmezlerinin her zaman korunmasını sağlamak için denetleyicilerin birbirleriyle mesaj alışverişinde bulunduğu dağıtılmış bir sistem oluşturur. Bu sonlu durum makineleri arasındaki etkileşimler tutarlılık protokolü tarafından belirlenir. Tutarlılık denetleyicilerinin çeşitli sorumlulukları vardır. Önbellek denetleyicisi olarak adlandırdığımız önbellekteki tutarlılık denetleyicisi Şekil 6.1'de gösterilmektedir. Önbellek denetleyicisi, iki kaynaktan gelen isteklere hizmet vermelidir. "Çekirdek tarafında", önbellek denetleyicisi işlemci çekirdeğine arayüz oluşturur. Kontrolör çekirdekten gelen yükleri ve depoları kabul eder ve yük değerlerini çekirdeğe döndürür. Bir önbellek hatası, denetleyicinin, çekirdek tarafından erişilen konumu içeren blok için bir tutarlılık talebi (örneğin, salt okunur izin talebi) yayınlayarak bir tutarlılık işlemi başlatmasına neden olur. Bu tutarlılık isteği, ara bağlantı ağı üzerinden bir veya daha fazla tutarlılık denetleyicisine gönderilir. Bir işlem, bir istek ve diğer mesaj(lar)dan oluşur.Talebi yerine getirmek için değiş tokuş edilir (örneğin, başka bir tutarlılık denetleyicisinden talep sahibine

gönderilen bir veri yanıt mesajı). İşlem türleri ve her işlemin bir parçası olarak gönderilen mesajlar, belirli tutarlılık protokolüne bağlıdır. Önbellek denetleyicisinin "ağ tarafında", önbellek denetleyicisi, ara bağlantı ağı aracılığıyla sistemin geri kalanıyla arabirim oluşturur. Denetleyici, işlemesi gereken tutarlılık isteklerini ve tutarlılık yanıtlarını alır. Çekirdek tarafta olduğu gibi, gelen tutarlılık mesajlarının işlenmesi, belirli tutarlılık protokolüne bağlıdır. Bellek denetleyicisi olarak adlandırdığımız LLC/bellekteki tutarlılık denetleyicisi Şekil 6.2'de gösterilmektedir. Bellek denetleyicisi, genellikle yalnızca bir ağ tarafına sahip olması dışında önbellek denetleyicisine benzer. Bu nedenle, tutarlılık talepleri (yükler veya depolar adına) yayınlamaz veya tutarlılık yanıtları almaz. G/Ç aygıtları gibi diğer aracılar, özel gereksinimlerine bağlı olarak önbellek denetleyicileri, bellek denetleyicileri veya her ikisi gibi davranabilir. Her tutarlılık denetleyicisi, bir dizi sonlu durum makinesi (blok başına mantıksal olarak bir bağımsız, ancak aynı sonlu durum makinesi) uygular ve bloğun durumuna bağlı olarak olayları (ör. gelen tutarlılık mesajları) alır ve işler. B bloğuna yönelik E tipi bir olay (örneğin, çekirdekten önbellek denetleyicisine bir depolama talebi) için, tutarlılık denetleyicisi, E'nin ve B'nin durumu (örn. salt okunur). Bu eylemleri gerçekleştirdikten sonra, denetleyici B'nin durumunu değiştirebilir. [5]

Tablo 2

94 6. COHERENCE PROTOCOLS

Table 6.1: Tabular specification methodology. This is an incomplete specification of a cache coherence controller. Each entry in the table specifies the actions taken and the next state of the block.

States	Events			
		Load request from core	Store request from core	Incoming coherence request to obtain block in read-write state
	Not readable or writeable (N)	Issue coherence request for read-only permission/RO	Issue coherence requests for read-write permission/RW	<No action>
	Read-only (RO)	Give data from cache to core	Issue coherence request for read-write permission/RW	<No action>/N
	Read-write (RW)	Give data from cache to core	Write data to cache	Send block to requestor/N

5.1. Sımple Coherence Protocol Örnekleri

Tutarlılık protokollerini anlamaya yardımcı olmak için şimdi basit bir protokol sunuyoruz. Her önbellek bloğu, iki kararlı tutarlılık durumundan birinde olabilir: I(nvalid) ve V(alid). LLC/bellekteki her blok aynı zamanda iki tutarlılık durumundan birinde olabilir: I ve V. LLC/bellekte, I durumu tüm önbelleklerin bloğu I durumunda tuttuğunu ve V durumu bir önbelleğin tuttuğunu gösterir. V durumundaki blok. Önbellek blokları için tek bir geçici durum

da vardır, IVD, aşağıda tartışılmaktadır. Sistem başlangıcında, tüm önbellek blokları ve LLC/bellek blokları durumdadır.

I. Her çekirdek, önbellek denetleyicisine yükleme ve depolama istekleri gönderebilir; önbellek denetleyicisi, başka bir bloğa yer açması gerektiğinde dolaylı olarak bir Evict Block olayı oluşturur. Önbellekte eksik olan yükler ve depolar, önbellek bloğunun geçerli bir kopyasını elde etmek için aşağıda açıklandığı gibi tutarlılık işlemlerini başlatır. Bu ilk metindeki tüm protokoller gibi biz de bir geri yazma önbelleği varsayıyoruz; yani, bir mağaza vurduğunda, mağaza değerini yalnızca (yerel) önbelleğe yazar ve yazmak için bekler.

Evict Block olayına yanıt olarak tüm blok LLC'ye/belleğe geri döner. Üç tür veri yolu mesajı kullanılarak uygulanan iki tür tutarlılık işlemi vardır: Get bir bloğa istekte bulunur, DataResp bloğun verilerini aktarır ve Put bloğu bellek denetleyicisine geri yazar. Bir yükleme veya depolama hatası durumunda, önbellek denetleyicisi bir Get mesajı göndererek ve karşılık gelen DataResp mesajını bekleyerek bir Get işlemi başlatır. Get işlemi atomiktir, çünkü başka hiçbir işlem (Get veya Koy), önbelleğin Get'i göndermesi ile bu Get için DataResp'in veri yolunda görünmesi arasında veri yolunu kullanamaz. Bir Evict Block olayında, önbellek denetleyicisi tüm önbellek bloğuyla birlikte bir Put mesajı gönderir. Bellek denetleyicisi. Kararlı tutarlılık durumları arasındaki geçişleri Şekil 6.3'te gösteriyoruz. Belirli önbellek denetleyicisi tarafından başlatılan işlemlere ilişkin mesajları diğer önbellek denetleyicileri tarafından başlatılan işlemlere karşı ayırmak için “Own” ve “Other” önsözlerini kullanırız. Belirli önbellek denetleyicisi V durumundaki bloğa sahipse ve başka bir önbellek bunu bir Get mesajıyla (OtherGet olarak gösterilir) talep ederse, sahip olan önbelleğin bir blokla (gösterilmemiş olan bir DataResp mesajı kullanarak) ve durum I'e geçişle yanıt vermesi gerekir. [5]

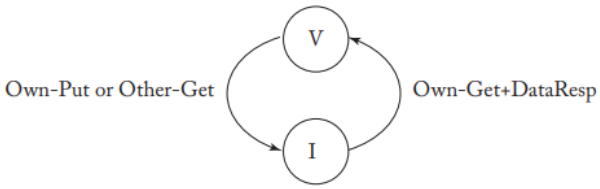


Figure 6.3: Transitions between stable states of blocks at cache controller.

Şekil 5.1.1

Bu tutarlılık protokolü basit ve birçok yönden verimsizdir, ancak bu protokolü sunmanın amacı, protokollerin nasıl belirlendiğini anlamaktır. [5]

Tablo 3 Coherence States

Table 7.22: Power5 L2 cache coherence states

State	Permissions	Description
I	None	Invalid
S	Read-only	Shared
SL	**Read-only**	**Shared local data source, but can respond with data to requests from processors in same node (sometimes referred to as F state, as in Intel QuickPath protocol (Section 8.8.4))**
S(S)	Read-only	Shared
Me (E)	Read-write	Exclusive
M (M)	Read-write	Modified
Mu	**Read-write**	**Modified unsolicited—received read-write data in response to read-only request**
T	**Read-only**	**Tagged—was M, received GetS. T is sometimes described as being a read-write state, which violates the SWMR invariant since there are also blocks in state S. A better way to think of T is that it is like E: it can immediately transition to M. However, unlike E, this transition is not silent: a store to a block in T state immediately transitions to M but (atomically) issues an invalidation message on the ring. Although other caches may race with this request, the T state has priority, and thus is guaranteed to be ordered first and thus does not need to wait for the invalidations to complete.**

6. MAJOR PROTOCOL DESIGN OPTIONS

Bir tutarlılık protokolü tasarlamanın birçok farklı yolu vardır. Aynı durumlar ve işlemler için bile birçok farklı olası protokol vardır. Protokolün tasarımı, her tutarlılık denetleyicisinde hangi olayların ve geçişlerin mümkün olduğunu belirler; durumların ve işlemlerin aksine, protokolden bağımsız olarak olası olayların veya geçişlerin bir listesini sunmanın bir yolu yoktur. Tutarlılık protokolleri için muazzam tasarım alanına rağmen, protokolün geri kalanı üzerinde büyük etkisi olan iki temel tasarım kararı vardır.

Tutarlılık protokollerinin iki ana sınıfı vardır: Snooping ve Directory.

• Snooping protokolü: Bir önbellek denetleyicisi, diğer tüm tutarlılık denetleyicilerine bir istek mesajı yayınlayarak bir blok isteği başlatır. Tutarlılık denetleyicileri toplu olarak "doğru olanı yaparlar", örneğin, eğer sahiplerse başka bir çekirdeğin isteğine yanıt olarak veri göndermek. Snooping protokolleri, yayın mesajlarını tüm çekirdeklere tutarlı bir sırayla iletmek için ara bağlantı ağına güvenir. Çoğu Snooping protokolü, isteklerin, örneğin paylaşılan bir kablolu veri yolu aracılığıyla toplam bir sırada geldiğini varsayar, ancak daha gelişmiş ara bağlantı ağları ve rahat siparişler mümkündür.

• Directory protokolü: Bir önbellek denetleyicisi, bloğu o bloğun evi olan bellek denetleyicisine tek noktaya yayınlayarak bir blok için istek başlatır. Bellek denetleyicisi, mevcut sahibin kimliği veya mevcut paylaşanların kimlikleri gibi LLC/bellekteki her blok hakkında durumu tutan bir Directory tutar. Bir blok talebi eve ulaştığında, bellek denetleyicisi bu bloğun Directory durumuna bakar. Örneğin, istek bir GetS ise, bellek denetleyicisi sahibini belirlemek

için Directory durumuna bakar. LLC/belleğin sahibi ise, bellek denetleyicisi, istek sahibine bir veri yanıtı göndererek işlemi tamamlar. Sahip bir önbellek denetleyicisiyse, bellek denetleyicisi isteği sahip önbelleğine iletir; sahip önbelleği iletilen isteği aldığında, istek sahibine bir veri yanıtı göndererek işlemi tamamlar. Snooping ve Directory seçimi, ödün vermeyi içerir. Snooping protokolleri mantıksal olarak basittir, ancak yayın ölçeklenmediğinden çok sayıda çekirdeğe ölçeklenemezler. Directory protokolleri, tek noktaya yayın yaptıkları için ölçeklenebilirdir, ancak ev sahibi olmadığında gönderilecek fazladan bir ileti gerektirdiğinden birçok işlem daha fazla zaman alır. Ek olarak, protokol seçimi ara bağlantı ağını etkiler (örneğin, klasik Snooping protokolleri, istek mesajları için tam bir sıra gerektirir).

Invalidate ve Update

Tutarlılık protokolündeki diğer önemli tasarım kararı, bir çekirdek bir bloğa yazdığında ne yapılacağına karar vermektir. Bu karar, protokolün Snooping mi yoksa Directory mi olduğundan bağımsızdır. İki seçenek vardır.

• Invalidate protokolü: Bir çekirdek bir bloğa yazmak istediğinde, diğer tüm önbelleklerdeki kopyaları Invalidatek için bir tutarlılık işlemi başlatır. Kopyalar geçersiz kılındıktan sonra, istekte bulunan kişi, başka bir çekirdeğin bloğu okuma olasılığı olmadan bloğa yazabilir. Eski değer. Başka bir çekirdek, kopyası geçersiz kılındıktan sonra bloğu okumak isterse, bloğu elde etmek için yeni bir tutarlılık işlemi başlatmak zorunda kalır ve onu yazan çekirdekten bir kopya alarak tutarlılığı korur.

• Update protokolü: Bir çekirdek bir blok yazmak istediğinde, bloğa yazdığı yeni değeri yansıtacak şekilde diğer tüm önbelleklerdeki kopyaları Updatek için bir tutarlılık işlemi başlatır. Çekirdeğin bir GetS işlemini başlatması ve tamamlanmasını beklemesi gerekmediğinden, bir çekirdeğin yeni yazılmış bir bloğu okuma gecikmesi. Ancak, Update protokolleri genellikle Invalidate protokollerinden çok daha fazla bant genişliği kullanır çünkü Update mesajları Invalidate mesajlarından daha büyüktür (yalnızca bir adres yerine bir adres ve yeni bir değer). Ayrıca Update protokolleri, birçok bellek tutarlılık modelinin uygulanmasını büyük ölçüde karmaşıklaştırır. Örneğin, yazma atomikliğini korumak birden çok önbelleğin bir bloğun birden çok kopyasına birden çok Update uygulaması gerektiğinde çok daha zor hale gelir. Update protokollerinin karmaşıklığı nedeniyle nadiren uygulanırlar. [5]

Hibrit Tasarımlar

Her iki önemli tasarım kararı için de seçeneklerden biri hibrit geliştirmektir. Snooping ve Directory protokollerinin özelliklerini birleştiren protokoller vardır ve Invalidate ve Update protokollerinin özelliklerini birleştiren protokoller vardır. Tasarım alanı zengindir ve mimarlar belirli bir tasarım stilini takip etmekle sınırlı değildir. [5]

5.2 Snooping Coherence Protocols - Hıgh-Level Protocol Specıfıcatıon

Temel protokolün yalnızca üç kararlı durumu vardır: M, S ve I. Böyle bir protokole tipik olarak MSI protokolü denir. Bölüm 6.3'teki protokol gibi, bu protokol de bir geri yazma önbelleği varsayar. Bir blok, M durumunda bir önbellekte olmadığı sürece LLC'ye/belleğe aittir. Ayrıntılı belirtimi sunmadan önce, temel davranışlarını anlamak için önce protokolün daha yüksek düzeyde bir soyutlamasını gösteriyoruz. Şekil 7.1 ve 7.2'de sırasıyla önbellek ve bellek denetleyicilerindeki kararlı durumlar arasındaki geçişleri gösteriyoruz. [5]

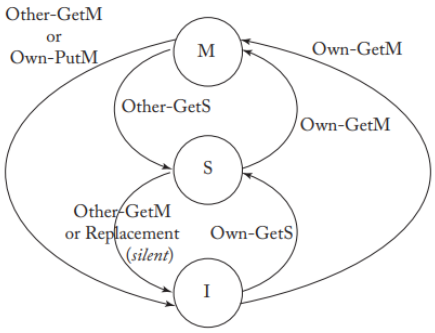


Figure 7.1: MSI: Transitions between stable states at cache controller.

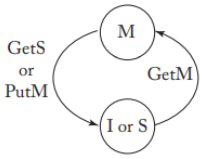


Figure 7.2: MSI: Transitions between stable states at memory controller.

Şekil 5.2.1

5.2.1 Sımple Snoopıng System Model: Atomıc Requests,Atomıc Transactıons

Spesifik olarak, bu sistem Atomik İstekler ve Atomik İşlemler olarak tanımladığımız iki atomik özellik uygular. Atomic Requests özelliği, bir tutarlılık talebinin verildiği döngüde sıralandığını belirtir. Bu özellik, bir talebin verilmesi ile sipariş edilmesi arasında başka bir çekirdeğin tutarlılık talebi nedeniyle bloğun durumunun değişme olasılığını ortadan kaldırır. [5]

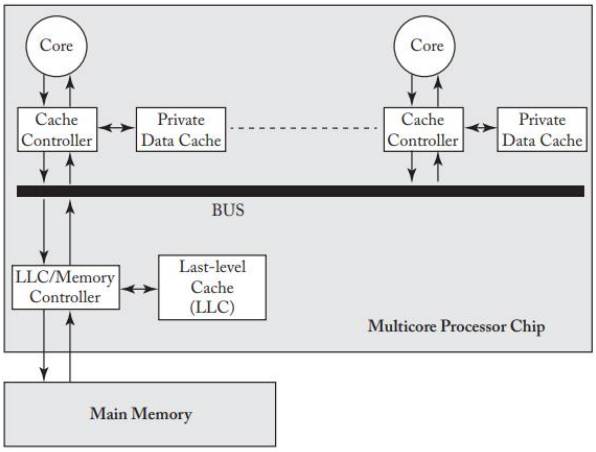


Figure 7.3: Simple snooping system mode.

Şekil 5.2.1.1

Şekil 7.4 ve 7.5'te MESI protokolündeki kararlı durumlar arasındaki geçişleri gösteriyoruz. MESI protokolü hem önbellekte hem de LLC/bellekte temel MSI protokolünden farklıdır. Önbellekte, bir GetS isteği, GetS sipariş edildiğinde LC/bellekteki duruma bağlı olarak S veya E'ye geçiş yapar. Ardından, E durumundan, blok sessizce M olarak değiştirilebilir. Bu protokolde, ayrı bir PutE kullanmak yerine, E'deki bir bloğu çıkarmak için bir PutM kullanırız; bu karar, protokol spesifikasyonunun kısa ve öz tutulmasına yardımcı olur ve protokol işlevselliği üzerinde hiçbir etkisi yoktur. LLC/belleğin, MSI protokolündekinden bir kararlı durumu daha vardır. LLC/bellek şimdi ayırt etmelidir

MSI protokolünde yapıldığı gibi bunları tek bir durumda birleştirmek yerine, sıfır veya daha fazla önbellek tarafından paylaşılan bloklar ve hiç paylaşılmayan bloklar (I) arasında. Bu ilk yazıda, E durumunu, protokol üzerinde önemli bir etkiye sahip olan bir sahiplik durumu olarak değerlendiriyoruz. Bununla birlikte, E durumunu bir sahiplik durumu olarak kabul etmeyen protokoller vardır.[5]

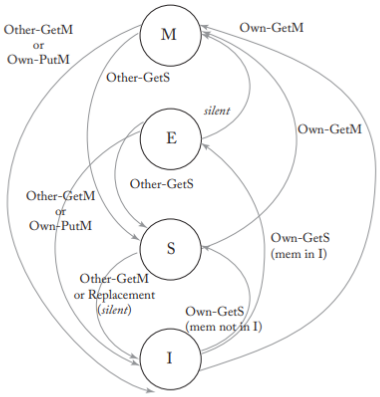


Figure 7.4: MESI: Transitions between stable states at cache controller.

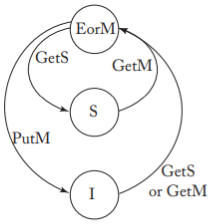


Figure 7.5: MESI: Transitions between stable states at memory controller.

Şekil 5.2.1.2

Şekil 7.6 ve 7.7'de kararlı durumlar arasındaki geçişlerin üst düzey bir görünümünü belirtiyoruz. Temel fark, M durumunda bir bloğa sahip bir önbellek başka bir çekirdekten bir GetS aldığında olan şeydir. Bir MOSI protokolünde, önbellek blok durumunu O olarak değiştirir (S yerine) ve bloğun sahipliğini korur (sahipliği LLC/belleğe aktarmak yerine). Böylece O durumu, önbelleğin LLC/belleği güncellemekten kaçınmasını sağlar. [5]

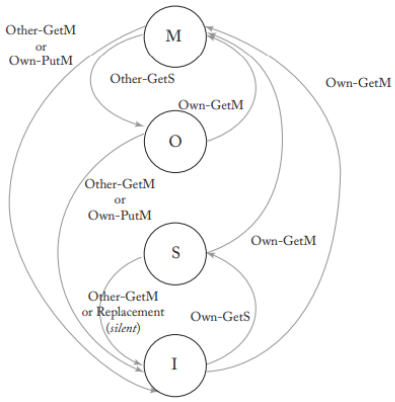


Figure 7.6: MOSI: Transitions between stable states at cache controller.

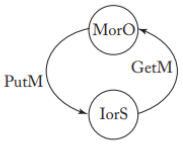


Figure 7.7: MOSI: Transitions between stable states at memory controller.

Şekil 5.2.1.3

MOESI protokolü: Bu protokol, ek donanım karmaşıklığı pahasına tutarlılık performansını daha da artırmak için Sweazey ve Smith tarafından önerilmiştir. Bu protokol, O durumunu tanıtarak, bazı işlemciler S durumunda aynı bloğa sahip olabileceğinde, ana bellekteki bazı bellek bloklarının en güncel olması gerekmemesine izin verir. Bu, M durum satırına sahip bir önbellek, belleği güncellemeden önbellekten önbelleğe aktarımı başlatan BusRd'yi gözlemlediğinde gerçekleşir. Bu işlem ayrıca gözetleme işlemcisinin önbelleğinde M→O geçişini ve istekte bulunan işlemcinin önbelleğinde I→S geçişini de içerir. SRAM tabanlı önbellek, DRAM ana belleğinden önemli ölçüde daha hızlı olduğundan, bu tür yeni durum dahil etme, önbellekten önbelleğe aktarımla eş zamanlı bellek güncellemesine kıyasla daha hızlı aktarım sağlar. MESI protokolünde, M→S geçişinin önbellekten önbelleğe aktarımla aynı anda ana belleği güncellediğine dikkat edin. AMD64 mimarisi ve SUN Microsystems'in UltraSPARC gibi MOESI protokolünün çeşitleri modern mikroişlemcilerde de kullanılmaktadır. [6]

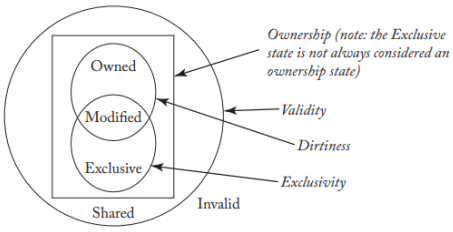


Figure 6.4: MOESI states.

Şekil 5.2.1.3

5.3 Directory Coherence Protocols

Snoopy protokollerinden farklı olarak, ara bağlantı ağının topolojisi kasıtlı olarak belirsizdir. Bir ağ, simit veya mimarın kullanmak istediği başka herhangi bir topoloji olabilir. Bu bölümde varsaydığımız ara bağlantı ağında bir kısıtlama, noktadan noktaya sıralamayı zorlamasıdır. Yani, kontrolör A, kontrolör B'ye iki mesaj gönderirse, mesajlar gönderildikleri sırada kontrolör B'ye ulaşır.1 Noktadan noktaya sıralamaya sahip olmak, protokolün karmaşıklığını azaltır ve bir tartışmayı erteleriz. Dizini boyutlandırmanın ve düzenlemenin birçok yolu vardır ve şimdilik en basit modeli varsayıyoruz: bellekteki her blok için karşılık gelen bir dizin girişi vardır. Ayrıca, tek bir dizin denetleyicisi olan yekpare bir LLC varsayıyoruz. [5]

5.3.1 Hıgh-Level Protocol Specıfıcatıon

Temel dizin protokolünün yalnızca üç kararlı durumu vardır: MSI. Bir blok, M durumunda bir önbellekte olmadığı sürece, dizin denetleyicisine aittir. Her bloğun dizin durumu, kararlı tutarlılık durumunu, sahibinin kimliğini (blok M durumundaysa) ve kimliklerini içerir. Paylaştırıcılar bir sıcak bit vektörü olarak kodlanmıştır (blok S durumundaysa). Şekil 8.2'de bir dizin girişini gösteriyoruz. [5]

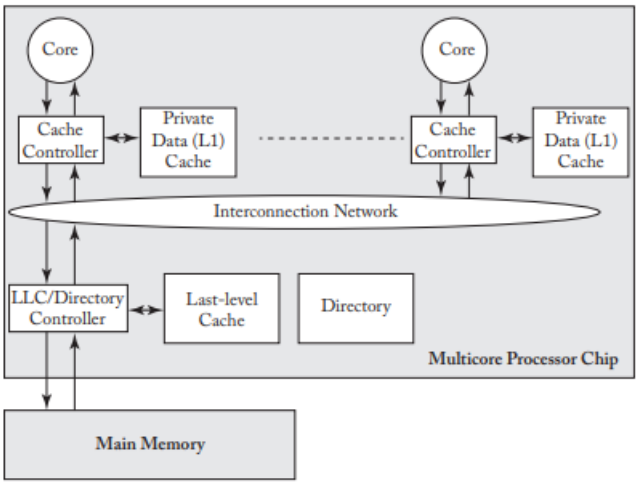


Figure 8.1: Directory system model.

Şekil 5.3.1.1

5.3.2 Deadlocks Sorunu

Bu protokolde, bir mesajın alınması tutarlılık denetleyicisinin başka bir mesaj göndermesine neden olabilir. Genel olarak, A olayı (örneğin, mesaj alımı) B olayına (örneğin, mesaj gönderme) neden olabiliyorsa ve bu olayların her ikisi de kaynak tahsisi gerektiriyorsa (örneğin, ağ bağlantıları ve arabellekler), o zaman şu durumlarda meydana gelebilecek kilitlenmeleri önlemek için dikkatli olmalıyız. Döngüsel kaynak bağımlılıkları ortaya çıkar. Örneğin, bir GetS isteği, dizin denetleyicisinin bir Fwd-GetS mesajı yayınlamasına neden olabilir; bu mesajlar aynı kaynakları (ör. ağ bağlantıları ve arabellekler) kullanıyorsa, sistem potansiyel olarak kilitlenebilir. Şekil 8.4'te, iki tutarlılık denetleyicisi C1 ve C2'nin birbirlerinin isteklerine yanıt verdiği, ancak gelen kuyrukların zaten diğer tutarlılık istekleriyle dolu olduğu bir kilitlenmeyi gösteriyoruz. Kuyruklar FIFO ise, yanıtlar istekleri geçemez. Kuyruklar dolu olduğu için, her denetleyici bir yanıt göndermeye çalışırken durur. Kuyruklar FIFO olduğundan, denetleyici sonraki bir istekte çalışmaya geçemez (veya yanıta ulaşamaz). Böylece sistem kilitleniyor. Tutarlılık protokollerinde kilitlenmeyi önlemek için iyi bilinen bir çözüm, her mesaj sınıfı için ayrı ağlar kullanmaktır. Ağlar fiziksel olarak ayrı veya mantıksal olarak ayrı olabilir (sanal ağlar olarak adlandırılır), ancak anahtar, mesaj sınıfları arasındaki bağımlılıklardan kaçınmaktır. [5]

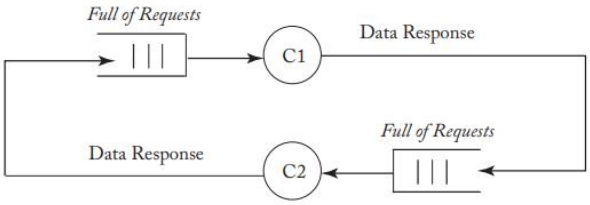


Figure 8.4: Deadlock example.

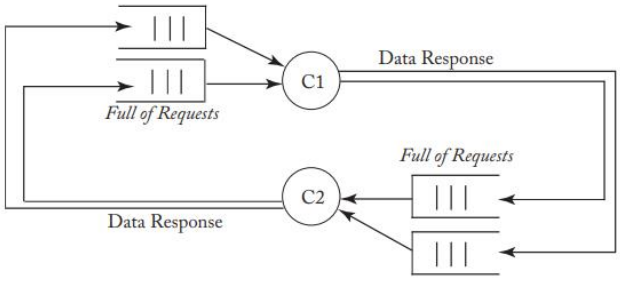


Figure 8.5: Avoiding deadlock with separate networks.

Şekil 5.3.2.1

5.3.3 Protocol Operations

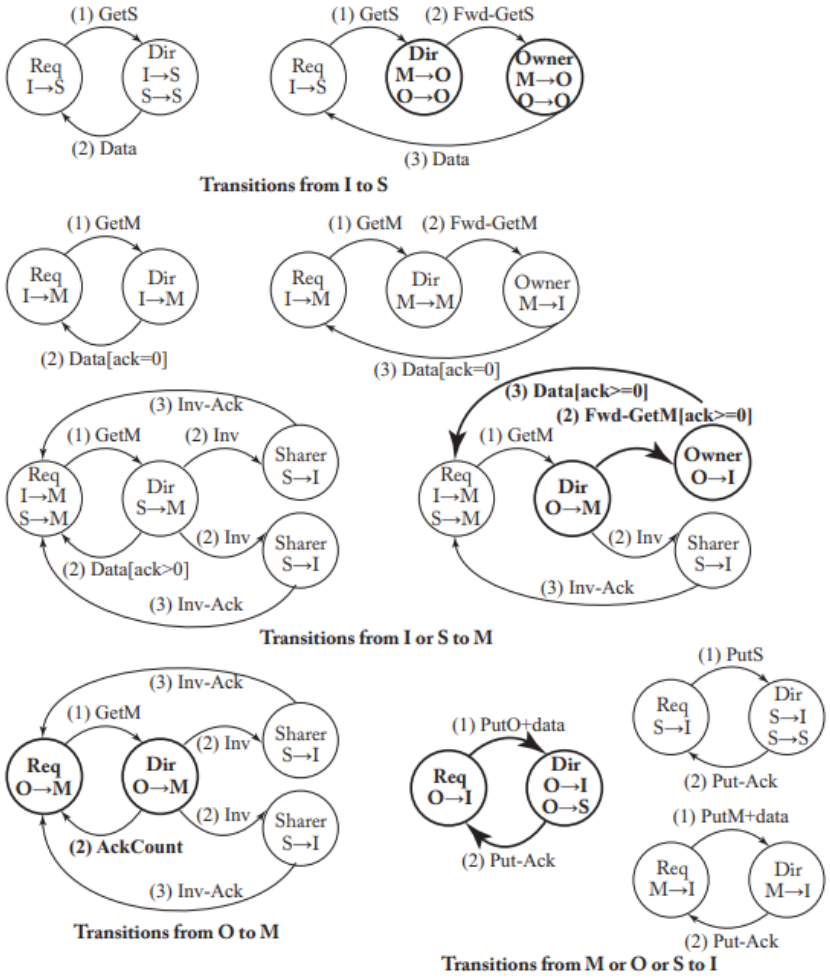


Figure 8.7: High-level description of MOSI directory protocol. In each transition, the cache controller that requests the transaction is denoted "Req."

Şekil 5.3.3.1

5.4. Interconnectıon Networks Wıthout Poınt-To-Poınt Orderıng

• Noktadan noktaya sıralama ile (Şekil 8.12'de gösterilmektedir): C1 Fwd-GetS'yi alır, Data ile yanıt verir ve blok durumunu O olarak değiştirir. C1 daha sonra Fwd-GetM'yi alır, Data ile yanıt verir ve bloğu değiştirir durum I. Bu beklenen sonuçtur.

• Noktadan noktaya sıralama olmadan (Şekil 8.13'te gösterilmektedir): C3'ten Fwd-GetM önce C1'e ulaşabilir. C1, C3'e Veri ile yanıt verir ve blok durumunu I olarak değiştirir. C2'den Fwd-GetS daha sonra C1'e ulaşır. C1, I'de ve yanıt veremiyor. C2'den gelen GetS isteği hiçbir zaman karşılanmayacak ve sistem sonunda kilitlenecektir. [5]

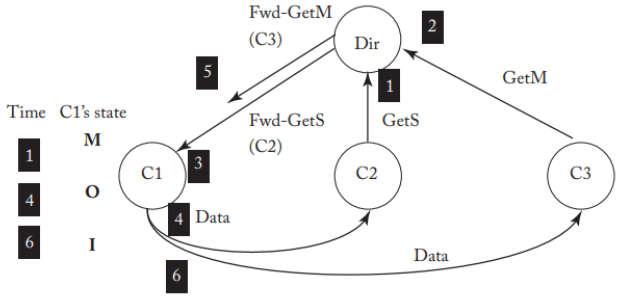


Figure 8.12: Example with point-to-point ordering.

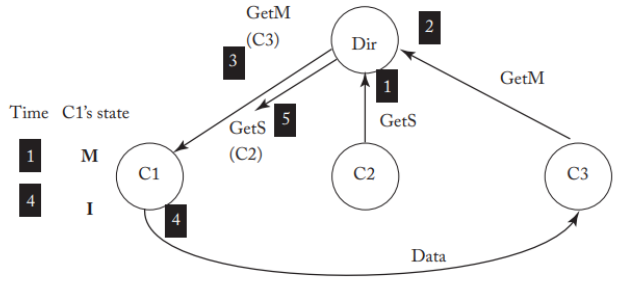


Figure 8.13: Example without point-to-point ordering. Note that C2's Fwd-GetS arrives at C1 in state I and thus C1 does not respond.

Şekil 5.4.1

5.5 Diğer Protokoller

Dragon protokolü, geri yazma ilkesi tabanlı bir protokoldür. Bu protokolde, bir işlemcideki her bloğa aşağıdaki durumlar belirtilir (Geçersiz durum, değiştirilmiş durum, özel durum, paylaşılan temiz ve paylaşılan değiştirilmiş). Dragon protokolünün amacı, ana belleğin yapması gereken depolama işlemi miktarını en aza indirmektir. . İki paylaşımlı duruma ayrılmıştır, biri paylaşımlı temizlik ve diğeri paylaşımlı değiştirilmiş durum. Paylaşılan temiz durum, paylaşanların olduğunu ve bu durumda işlemcinin önbellek satırının sahibi olmadığını gösterir. Paylaşılan değiştirilmiş durum, ana belleğin henüz güncellenmediğini ve bu durumda işlemcinin önbellek satırının sahibi olduğunu gösterir. [8]

Wikipedia'ya göre bazı protokoller:[9]

"Firefly (DEC) protocol States D-VE-S (MES)

- No "Invalid" state

- "Write-broadcasting"+"Write Through"

- Use of "shared line"

- "Write-broadcasting" avoid the necessity of "Invalid" state

- Simultaneous intervention from all caches (shared and dirty intervention – on not modified that modified data)

- This protocol requires a synchronous bus

Berkeley protocol States D-SD-V-I (MOSI)

- As with MOESI without E state

- No use of "shared line"

Synapse protocol States D-V-I (MSI)

- The characteristic of this protocol is ti have a single-bit tag with each cache line in MM, indicating that a cache have the line in D state.

- This bit prevents a possible race condition if the D cache does not respond quickly enough to inhibit the MM from responding before being updating.

- The data comes always from MM

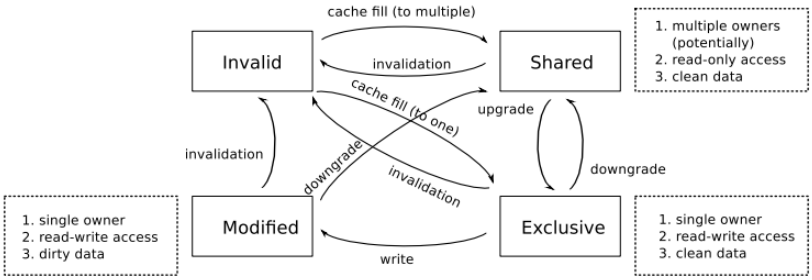
- No use of "shared line" "

6. MULTICORE PROCESSORS İÇİN CACHE COHERENCE TEKNİKLERİ

Çok çekirdeğe geçiş, sürekli üstel performans kazanımları elde etmek için paralel yazılıma dayanacaktır. Ticari pazardaki çoğu paralel yazılım, tüm işlemcilerin aynı fiziksel adres alanına eriştiği paylaşılan bellek programlama modeline dayanır. İşlemciler mantıksal olarak aynı belleğe erişse de, işlemciler tarafından yapılan bellek referanslarının çoğu için hızlı performans elde etmek için çip üzerindeki önbellek hiyerarşileri çok önemlidir. Bu nedenle, paylaşılan bellek çoklu işlemcilerinin temel sorunu, çeşitli önbellek hiyerarşileriyle tutarlı bir bellek görünümü sağlamaktır. Bu önbellek tutarlılık sorunu, kritik doğruluk ve performansa duyarlı bir tasarımdır. Önbellek tutarlılık mekanizmaları, yalnızca bir paylaşılan bellek çok işlemcisindeki iletişimi yönetmekle kalmaz, aynı zamanda tipik olarak 3 bellek sisteminin işlemciler, önbellekler ve bellek arasında verileri nasıl aktardığını da belirler. Paylaşılan bellek programlama modelinin öne çıkmaya devam ettiğini varsayarsak, gelecekteki iş yükleri önbellek tutarlı bellek sisteminin performansına bağlı olacaktır ve bu alanda sürekli yenilik, bilgisayar tasarımındaki ilerleme için çok önemlidir. Önbellek tutarlılığı, araştırma camiasında büyük ilgi gördü, ancak önceki çalışma, birden çok tek çekirdekli işlemciden oluşan çok

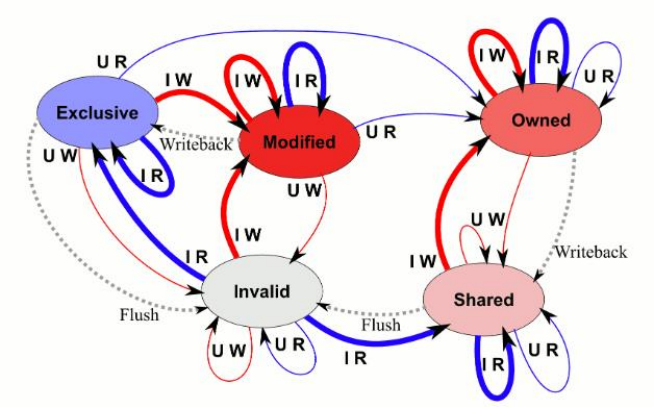
işlemcili makineleri (MP'ler) hedefliyordu. CMP'lerin tasarımındaki belki de önceki MP'lere kıyasla en önemli fark, tasarıma bütüncül bir yaklaşım benimseme fırsatıdır. Önceki makineler genellikle, tasarımın tek çekirdekli performansa odaklandığı ticari tek işlemcilerden yapılmıştır. Önbellek tutarlı bellek sistemi artık çip düzeyinde birinci dereceden bir tasarım sorunudur. [10]

Birden çok işlemci aynı bellek hiyerarşisini paylaştığında, ancak kendi L1 verilerine ve yönerge önbelleklerine sahip olduğunda, iki işlemcinin önbelleğinde belirli bir önbellek bloğunun iki veya daha fazla kopyası varsa ve bu bloklardan biri değiştirilmişse hatalı yürütme meydana gelebilir. İşlemci 0 ve İşlemci 1'in her ikisinin de L1 veri önbelleklerinde Blok X'in bir kopyasına sahip olduğu durumu düşünün. Başlangıçta blok, her iki işlemcinin paylaştığı bir L2 önbelleğinden okundu, bu nedenle bloğun her iki kopyası da aynı değerleri içeriyor. İşlemci 0 veya İşlemci 1, Blok X'teki bir değeri okumak için bir yükleme talimatı gerçekleştirirse, doğru yürütme gerçekleşir. Ancak İşlemci 0, Blok X'teki bir değeri değiştiren bir saklama talimatı gerçekleştirirse ve İşlemci 1 daha sonra Blok X'ten bir yükleme talimatı gerçekleştirirse, yükleme talimatı yeni değeri görmelidir. Bu nedenle, yeni değer bir şekilde İşlemci 1'deki Blok X kopyasına yayılmalıdır. Buna önbellek tutarlılık sorunu denir. Bu sorunu çözmek için birden fazla önbelleğin nasıl etkileşime girdiğini yöneten bir dizi kurala önbellek tutarlılık protokolü denir. Tutarlılığı destekleyen bir önbellek sistemi, çeşitli şekillerde oluşturulabilir. Bu laboratuvar için, günümüzde kullanılan en yaygın şemalardan birine, geçersiz kılmaya dayalı bir önbellek tutarlılık protokolüne odaklanacağız. Geçersiz kılmaya dayalı bir protokol, bir işlemci bir önbellek bloğuna yazmak istediğinde, o işlemcinin bloğu içeren başka bir işlemcinin önbelleğindeki bloğun kopyasını geçersiz kılmasını (kaldırmasını) sağlayarak önbellek tutarlılık sorununu çözer. İstekte bulunan işlemci daha sonra önbellek bloğunun tek kopyasına sahiptir ve içeriğinde değişiklikler yapabilir. Daha sonra, herhangi bir işlemci bloğu okumaya çalıştığında, bir önbellek kaybı yaşar ve yeni verileri, verileri değiştiren işlemciden alması gerekir. (Bu etkileşimin tam olarak nasıl gerçekleştiği aşağıda açıklığa kavuşacaktır.) Geçersiz kılmaya dayalı bir önbellek tutarlılık protokolü, aşağıdaki değişmezi zorunlu kılarak (gerektiğinde önbellek bloklarını geçersiz kılarak) tüm işlemcilerin önbelleklerinde doğru değerleri görmesini sağlar: Geçersiz kılma tabanlı Protokol Değişmezi. Herhangi bir önbellek bloğu X, yazılabilir bir durumda en fazla bir önbellekte bulunur veya salt okunur bir durumda birden fazla önbellekte bulunur. Bu laboratuvarda, MESI önbellek tutarlılık protokolünün basit bir sürümünü uygulayacağız. MESI protokolü, L1 önbelleğindeki bir önbellek bloğunun sahip olabileceği dört durumdan sonra adlandırılan geçersiz kılma tabanlı bir protokoldür: Değiştirilmiş, Özel, Paylaşılan veya Geçersiz. Bu durumlar şu şekilde tanımlanır: [11]



Şekil 6.1.1 [11]

Çok çekirdekli bir makinede, farklı konumlara yazma işlemlerinin düzgün bir şekilde birleştirilmesini sağlamak için CPU'ların kullandığı bir önbellek tutarlılık protokolüne ihtiyacınız vardır. Bu, diğer çekirdeklerden okuma ve yazma işlemleriyle başa çıkmak için durumların eklenmesi anlamına gelir. Bu, MOESI'nin bir çeşididir: [12]



Şekil 6.2 [12]

7. MULTİPROCESSOR SİSTEMLERDE CACHE COHERENCE YAKLAŞIMLARI

Şekil 4'te gösterildiği gibi birbirine bağlı işlemci kümelerinden oluşur. Her bir işlemci kümesi ortak bir veri yolunu paylaşır ve kümeler arası iletişimi gerçekleştirmek için bir küme denetleyicisi gerektirir. Burada açıklanan protokolün sürümü için, küresel ara bağlantının da tek bir paylaşılan veri yolu olduğu varsayılır.

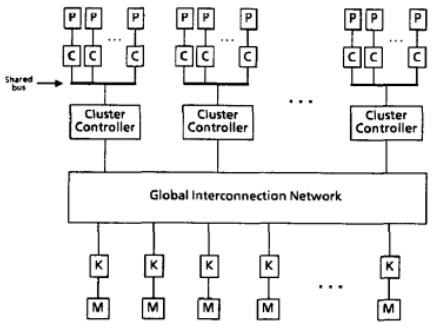


Figure 4: Two-Level Hierarchical Multiprocessor Organization

Şekil 7.1

Bu protokol, etkin paylaşım desteği sağlamak için her küme içinde etkili bir güncelleme protokolü kullanır. Aşağıdaki protokol açıklamasında, sunumu basitleştirmek için temiz sahiplik ve REMOTEWRITE sayaç durumları dahil edilmemiştir, ancak bunlar muhtemelen bir uygulamada istenecektir ve zorlanmadan eklenebilirler. Aslında, temiz sahiplik çok önemli hale gelir çünkü kümedeki diğer önbellekler tarafından hizmet verilen referanslar, ana bellek tarafından hizmet verilenlerden çok daha az zaman gerektirir. İdeal olarak, tüm küme içi isabetler, önbellekten önbelleğe aktarımlarla sonuçlanacaktır.

Kümeler arasında yazılabilir verilerin paylaşımının nadir olduğu varsayılır ve genel veri yolu üzerindeki protokol bir geçersiz kılma protokolü şeklini alır. (Salt okunur veriler veya talimatlar kümeler arasında verimli bir şekilde paylaşılabilir.)

Dahili Küme Protokolleri; Aşağıdaki altı yerel önbellek durumu kullanılır:

GEÇERSİZ. (TERS)

DEĞİŞTİRİLMİŞ-ÖZEL. (UNMOD-EXC) Bu, tüm sistemdeki önbelleğe alınmış tek kopyadır ve değiştirilmemiştir.

DEĞİŞTİRİLMİŞ-ÖZEL. (MOD-EXC) Bu, tüm sistemdeki önbelleğe alınmış tek kopyadır ve değiştirildiğinde geri yazılmalıdır.

DEĞİŞTİRİLMEMİŞ-PAYLAŞILAN-SADECE OKUNABİLİR. (UNMOD-SHDR) Bloğun diğer kopyaları bu ve diğer kümelerde bulunabilir.

DEĞİŞTİRİLMEMİŞ-PAYLAŞILMIŞ-YAZILABİLİR. (UNMOD-SHDW) Bloğun diğer kopyaları bulunabilir, ancak yalnızca kümede bulunur.

DEĞİŞTİRİLMİŞ-PAYLAŞILMIŞ. (MOD-SHD) Bloğun diğer kopyaları yalnızca kümede bulunabilir ve blok değiştirildiğinde geri yazılmalıdır.

UNMOD-SHD-R dışında bir durumda geçerli bir kopyası varsa, bir kümenin bir bloğa sahip olduğu söylenir. Bu, başka hiçbir kümenin bloğun bir kopyasına sahip olmadığı anlamına gelir. Bir blok kümeler arasında paylaşılıyorsa, herhangi bir kümeye ait değildir ve tüm önbelleğe alınmış kopyalar UNMOD-SHD-R olmalıdır. Bir kümeye ait bir blok, dağıtılmış yazma protokollerinde olduğu gibi kümedeki herhangi bir önbellek tarafından çok verimli bir şekilde değiştirilebilir. Sahip olunmayan blokları değiştirmenin ek yükü oldukça yüksektir. Küme protokolü iki özel veri yolu hattı gerektirir:

SHARED ve CLUSTER-OWNERSHIP, aşağıda C-O olarak kısaltılmıştır. EDWP protokolünde olduğu gibi, özel veri yolu hatları çeşitli veri yolu işlemleri sırasında bilgi sağlar. Protokolün önbellek bölümünün kesin bir özelliği, Şekil 5'teki durum geçiş diyagramında verilmiştir.

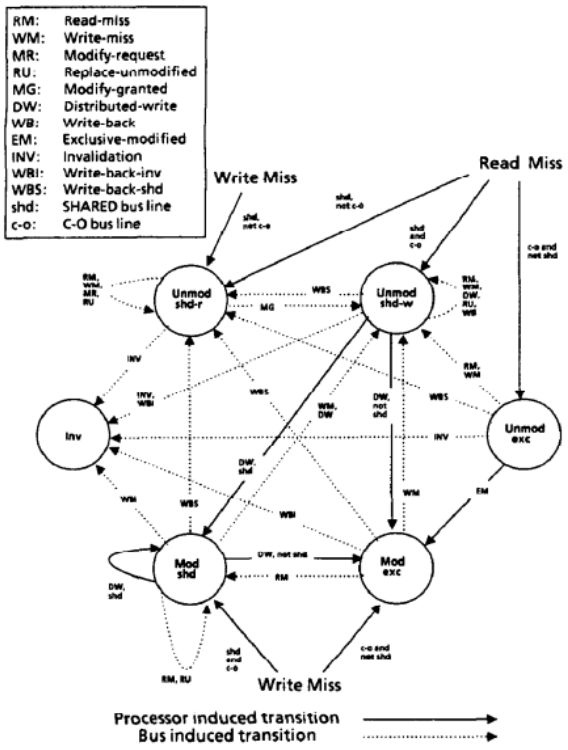


Figure 5: Cache State Transition Diagram

Şekil 7.2

Küresel Protokoller: Genel ara bağlantı paylaşılan bir veri yolu olduğundan, tıpkı gözetleme protokollerinin tamamen önbellek denetleyicisi düzeyinde uygulandığı gibi, protokolün genel bölümünün tamamen küme denetleyicisi düzeyinde uygulanması mümkündür. Küme denetleyicisi, küme ile genel ara bağlantılar arasında bir ağ geçidi görevi görür. Akıllı bir ağ geçidi olmadan, tüm küme veri yolu sinyallerinin küresel veri yolunda ve tüm küresel sinyallerin küme veri yolunda gönderilmesi gerekir, bu da anahtarı etkin bir şekilde tek bir paylaşılan veri yoluna indirger. İki düzey arasındaki gereksiz trafiği önlemek için, küme denetleyicisi tüm düzeyler için bir durum bilgisi tablosu sağlamalıdır. Küme içinde önbelleğe alınan bloklar. Bu bilgiler için olası bir organizasyon, IBM System 138'deki TLB'ye benzer bir karma tablo veya bazı veri akışı sistemlerinde eşleşen birimlerdir. [13]

8.SONUÇ

Bu çalışmada, multicore ve multiprocessor kullanan büyük çok işlemcili sistemler için özel olarak bir önbellek tutarlılık protokollerinin kullanımları detaylo bir şekilde ele alındı. Multiprocessor sistemler, ortak bir veri yolunu paylaşan işlemci kümeleri halinde düzenlenmiştir. Küme içindeki tutarlılık protokolü, paylaşılan yazmalarda önbelleğe alınan tüm kopyaları güncelleyen bir gözetleme protokolleri sınıfına benzer. Hiyerarşik protokolün küme kısmının organizasyonel detaylarının çoğunu motive etmek için yüksek verimliliğe sahip bir gözetleme protokolü sunuldu. Protokolün genel kısmı, veri yolu protokollerinin dağıtılmış ve gözetleme doğasını da kullanır. Tutarlılık mekanizması, global veri yolunda birbirinin aktivitelerini izleyen küme denetleyicileri üzerinden dağıtılır. Sunulan protokol, büyük sistemler için bir tutarlılık yaklaşımı örneğidir. Bu ve diğer çözümlerin maliyet ve performans dengelerini belirlemek için daha çok çalışma yapılması gerekebilir.

Snooping sistemleri, bilinen basitlikleri ve ölçeklenebilirlik eksikliklerinin piyasaya hâkim olan nispeten küçük sistemler için bir önemi olmaması nedeniyle erken dönem çok işlemcili sistemlerde yaygındır. Snooping ayrıca ölçeklenemeyen sistemler için performans avantajları sunar. Avantajlarına rağmen, gözetleme artık yaygın olarak kullanılmamaktadır. Dizin protokolleri piyasaya hakim olmaya başladı. Bunun yanı sıra küçük ölçekli sistemlerde bile, dizin protokolleri, büyük ölçüde ara bağlantı ağında noktadan noktaya bağlantıların kullanımını kolaylaştırdıkları için gözetleme protokollerinden daha yaygın olduğu görülmüştür. Ayrıca dizin protokolleri, ölçeklenebilir önbellek tutarlılığı gerektiren sistemler için tek seçenektir. Snooping darboğazlarını hafifletebilecek çok sayıda optimizasyon ve uygulama hilesi olmasına rağmen, temelde hiçbiri bu darboğazları ortadan kaldıramaz. Yüzlerce hatta binlerce düğüme ölçeklenmesi gereken sistemler için tutarlılık için tek uygun seçenek bir dizin protokolü olduğu görülmüştür.

Özetle bu çalışmamda bütün sistemler ve protokoller tek tek incelendiğinde hepsinin kullanım alanları, avantaj-dezavantajları ve tüm teknik detayları araştırılıp ele alınmıştır.

Not: Ödevimin birçoğunda her yerde kullanılan bir kitabı referans alarak araştırmalarımı tamamladım. Bu yüzden birçok kısımda aynı kaynakça verildi ayrıca şekillerimi kendi isimlendirmemin yanında makale ve kitaplardakiyle aynı şekilde aldım çünkü açıklamaları önemliydi.

KAYNAKÇA

[1] N. Parvathy, B. R. Upadhyay and T. S. B. Sudarshan, "Cache coherence: A walkthrough of mechanisms and challenges," 2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT), 2016, pp. 2251-2256, doi: 10.1109/ICEEOT.2016.7755093.

[2] Alwaisi, Zainab & Opoku Agyeman, Michael. (2017). An Overview of On-Chip Cache Coherence Protocols. 10.1109/IntelliSys.2017.8324309.

[3] Tanenbaum, A. S., Bos, H. (2014). *Modern Operating Systems*. Boston, MA: Pearson. ISBN: 978-0-13-359162-0

[4] Thomas Rauber and Gudula Rnger. 2013. Parallel Programming: for Multicore and Cluster Systems (2nd. ed.). Springer Publishing Company, Incorporated.

[5] Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2011. A Primer on Memory Consistency and Cache Coherence (1st. ed.). Morgan & Claypool Publishers.

[6]Internet:https://smartech.gatech.edu/bitstream/handle/1853/14065/suh_taeweon_200612_phd.pdf;sequence=1

[7]Internet: Cache Coherence, https://www.cs.umd.edu/~meesh/411/CA-online/chapter/cache-coherence-i/index.html

[8] S, Muthukumar & Kumar, Dhinakaran. (2013). Hybrid Cache Coherence Protocol for Multi-Core Processor Architecture. International Journal of Computer Applications. 70. 24-29. 10.5120/12031-8060.

[9]Internet:Wikipedia:https://en.wikipedia.org/wiki/Cache_coherency_protocols_(examples)

[10] Internet: https://research.cs.wisc.edu/multifacet/theses/michael_marty_phd.pdf

[11] Internet: https://course.ece.cmu.edu/~ece447/s15/lib/exe/fetch.php?media=lab8.pdf

[12] Internet: https://www.cs.uaf.edu/courses/cs441/notes/cache-coherence/

[13] J. K. Archibald. 1988. A cache coherence approach for large multiprocessor systems. In Proceedings of the 2nd international conference on Supercomputing (ICS '88). Association for Computing Machinery, New York, NY, USA, 337–345.