TOBB EKONOMİ VE TEKNOLOJİ ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

ÇOK ÇEKİRDEKLİ GÖREV-KRİTİK İŞLEMCİLER İÇİN ÖNBELLEK TASARIMI VE GERÇEKLENMESİ

YÜKSEK LİSANS TEZİ

MERT ATAMANER

Bilgisayar Mühendisliği Anabilim Dalı

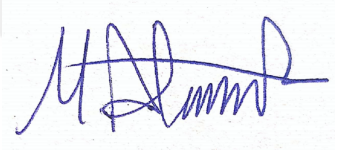
Tez Danışmanı: Prof. Dr. Oğuz Ergin

HAZİRAN 2020

TEZ BİLDİRİMİ

Tez içindeki bütün bilgilerin etik davranış ve akademik kurallar çerçevesinde elde edilerek sunulduğunu, alıntı yapılan kaynaklara eksiksiz atıf yapıldığını, referansların tam olarak belirtildiğini ve ayrıca bu tezin TOBB ETÜ Fen Bilimleri Enstitüsü tez yazım kurallarına uygun olarak hazırlandığını bildiririm.

Mert Atamaner



ÖZET

Yüksek Lisans Tezi

ÇOK ÇEKİRDEKLİ GÖREV-KRİTİK İŞLEMCİLER İÇİN ÖNBELLEK TASARIMI VE GERÇEKLENMESİ

Mert Atamaner

TOBB Ekonomi ve Teknoloji Üniversitesi
Fen Bilimleri Enstitüsü
Bilgisayar Mühendisliği Anabilim Dalı

Tez Danışmanı: Prof. Dr. Oğuz Ergin

Tarih: Haziran 2020

Çok çekirdekli sistemlerde paralel çalışmaya bağlı olarak performansı optimize etmek için çekirdeklere özgü ilk düzey önbellekler ve paylaşımlı üst düzey önbellekler bulunmaktadır. Çekirdeklere özgü önbelleklerin varlığı kullanılan verilerin aynı adresteki veriler olmasına bağlı olarak tutarlılık sorununu oluşturmaktadır. Modern sistemler bu sorunu tutarlılık protokolleri kullanarak çözmektedir. Tutarlılık protokollerinden MESI ve MOESI, günümüz sistemlerinde en çok kullanılan protokollerdendir. Bu protokoller tutarlılık sorununu başarılı bir şekilde çözmelerine karşın farklı amaçlarla da sistemde kullanılabilmektedir. Tez kapsamında tutarlılık etiketleri kullanılarak hata düzeltimi yapılabileceği gösterilmiştir. Çekirdeğe özgü önbelleklerde hata olduğu zaman hata düzeltme kodu kullanmak boru hattı ile iç içe olduklarından hız gereksinimini karşılamamaktadır. Güncel işlemcilerde bu önbellekler eşlik bitleriyle ve hata oluşunca sistemi geri alma yöntemiyle korunmaktadır. Önerilen mekanizma ise var olan tutarlılık altyapısının kullanılarak hataları hızlı bir şekilde düzeltmeyi amaçlamaktadır. Paylaşımlı tutarlılık etiketine sahip önbellek satırlarını takip eden dizin(ler) yardımıyla, hata oluşan önbelleklerdeki verileri düzeltmek mümkündür. Yapılan çalışmalar ışığında bu mekanizma kullanılarak programların dörtte birine kadar kısmını koruma altına almak mümkündür ve bu mekanizmanın kullanılamayacağı bir an neredeyse hiç oluşmamaktadır.

Hata düzeltiminin yanında, Yao vd., tutarlılık etiketlerinin güvenlik açığı yarattığını ortaya koymuştur [1]. Buna göre "S" ve "E" etiketlerine yapılan erişimlerin deterministik olarak ayrıştırılabilir olması, sistemde çalışan ajan ve truva atı uygulamalarının bir zamanla yan kanalı oluşturarak aralarında seri haberleşmesini mümkün kılmaktadır. İletişim, KSM(kernel same page merging)'nin çalışmasına bağlı olarak kullanılabilmektedir ve bu uygulamanın kullanılmaması performansı ciddi ölçüde etkilemektedir. Bu çalışma, tutarlılık etiketi kullanılarak oluşturulacak yan kanalları engellemek için KSM'nin kapatılmasına gerek olmayan bir çözüm öne sürmektedir. Önbellek satırlarına arka arkaya gelen yükle buyruklarını takip ederek iletişime gürültü ekleyen bu yöntem, gönderilen bitlerin %90'ını bozabilecek kapasitede iken, en kötü koşullar altında yaklaşık %15, en iyi durumda %2 yürütme zamanı ekleyerek programın performansını düşürmektedir.

Anahtar Kelimeler: Çok çekirdekli işlemciler, Önbellek, Tutarlılık protokolleri, Hata toleransı, Hata düzeltici kodlar, Güvenilirlik, Güvenlik, Önbellek yan kanalları, Gem5

ABSTRACT

Master of Science

CACHE DESIGN AND IMPLEMENTATION FOR MISSION-CRITICAL MULTICORE PROCESSORS

Mert Atamaner

TOBB University of Economics and Technology
Institute of Natural and Applied Sciences
Department of Computer Engineering

Supervisor: Prof. Dr. Oğuz Ergin

Date: June 2020

Mutlicore architecutres optimizes the performance and energy overhead by using private and shared caches in the memory hierarchy. Different cores having private caches introduces coherency problem in between different cores. Modern systems employ a coherence protocol scheme into the system to tackle this problem. Two of the most used protocols today are MOESI and MESI protocols. These protocols solve the coherency problem efficiently, however recently it was shown that these protocols can be exploited or used for different purposes. Thesis includes a such a scheme that uses MOESI labels for error correction. Private caches are closely connected to the pipeline and thus requires to be fast. Because of this fact, modern CPUs do not use ECC in private caches(L1). Instead, parity bits are checked for errors and if an error occurs systems crash or are reload to a safe state. Proposed mechanism suggests an alternative for using ECC while being fast enough for pipeline utilizing coherency labels. Shared labeled cache blocks imply that there is at least one other copy of the same data in another L1 cache, and it can be used to reload when a parity bit of such cache blocks imply error. In this thesis, it is shown that it is possible to protect a program during one fourth of its lifetime.

Yao et al., reports that coherency protocols can also be exploited to create timing side channels. Since the accesses to shared and exclusive labeled cache blocks takes

deterministically distinguishable time, it enables a spy-trojan pair to serially communicate by only measuring the access times to such cache blocks [1]. This communication is made possible by KSM(kernel same page merging) and disabling KSM incurs a significant performance loss. This work proposes a new scheme to prevent communicating through this side channel without the need of disabling KSM. Since the communication depends on back-to-back load operations, it is possible to track and monitor loads to the same cacheblock and introduce noise to the side channel. It is shown that it is possible to disrupt up to %90 of the communication while increasing the runtime %2 to %15.

Keywords: Multi core processors, Cache, Coherency protocols, Error tolerance, Error correcting codes, Reliability, Security, Cache Side channels, Gem5

TEŞEKKÜR

Lisans ve yüksek lisansım boyunca desteğini esirgemeyen, akademik hayatımda kendimi geliştirmemi sağlayan ve her zaman yol gösteren değerli hocam ve tez danışmanım Prof. Dr. Oğuz Ergin'e, tezimi geliştirme aşamasıda yardımcı olan ve savunma sanayii için araştırmacı yetiştirme programı dahilinde danışmanlığımı yapan ASELSAN mensuplarından sayın Dr. Fatih Say'a ve Çağla Irmak Rumelili Köksal'a, tezin geliştirme sürecinde finansal desteğinden dolayı ASELSAN A.Ş. ve Savunma Sanayi Başkanlığı'na, lisans eğitimimi tam burslu ve yüksek lisans eğitimimi özel başarı burslu tamamlamama olanak sağlayan TOBB Ekonomi ve Teknoloji Üniversitesi'ne ve eğitim hayatıma katkıları için üniversitenin öğretim üyelerine, destekleriyle her zaman yanımda olan aileme ve arkadaşlarıma teşekkürlerimi sunarım.

İÇİNDEKİLER

ŞEKİL LİSTESİ

Sayfa

KISALTMALAR

HDK : Hata Düzeltme Kodları (Error Correcting Codes)
TMR : Üç Kopyalı Tekrar (Triple Modular Redundancy)
KSM : Kernel Same Page Merging
CPU : Mekezi İşlemci (Central Processing Unit)

1. GİRİŞ

Çok çekirdekli işlemci mimarileri çağımızda hem endüstride hem de akademik çalışmalarda tercih edilen standart haline gelmiştir. Çok çekirdekli işlemciler, programların paralel olarak çalıştırılmasına dayalı olarak aynı anda birden fazla iş yapabildiği için performans- tan; çekirdeklerin daha düşük frekansta (ya da dinamik değişken) çalışmasına olanak sağladığı için enerjiden yüksek kazanç sağlamaktadır [2]. Bu sebeple, bu mimarilerin optimizasyonu ve geliştirilmesi için çalışmalar gündemden düşmemektedir.

Günümüz mimarilerinde özellikle performans optimizasyonları göz önünde bulundurularak önbellekler paylaşımlı ve çekirdeklere özgü olarak ayrılmıştır. Bu önbelleklerden çekirdeğe özgü olanları, boru hattı ile iletişim içinde bulunduğundan, bellek buyruklarına hızlı cevap vermesi gerekmektedir. Bu durum, özellikle bu önbelleklerin geçici hatalara karşı optimal şekilde korunmasını zorlaştırmaktadır. Bunun ana sebebi, standart olarak hata düzeltmek için kullanılan HDK metodunun kodlama ve kod çözme için her bellek erişiminde ekstra zaman harcamaya yol açmasıdır. Ayrıca HDK kullanmak fazladan bellek alanı istediğinden çekirdek çipi üzerinde önemli ölçüde yer kaplamaktadır [3, 4].

Oluşan sistem hatalarına ek olarak, önbellek hiyerarşisi içerisinde donanım güvenliği de gün geçtikçe önem kazanmaya devam eden bir konudur.Donanım güvenliği, son dönemlerde özellikle Spectre ve Meltdown saldırılarının keşfedilmesi sonrasında donanım tasarımında dikkat edilen en kritik kıstaslardan birisi haline gelmiştir. Temelinde güvenlik açıkları, çekirdekler ve iş parçacıkları arasında paylaşılan kaynaklar sebebiyle oluşmaktadır. Paylaşılan kaynaklar var olan veriyolları dışında yollarla bilgi aktarımı sağlayabilmektedir. Bu şekilde oluşturulan yollara yan kanal denir. Ana kanallar sistemlerde monitör edilebilirken bu kanallar, kontrol edilmediğinden, ciddi güvenlik açıklarına işaret eder. Bu yollar donanımdaki çeşitli optimizasyonların kötüye kullanılması ile ortaya çıkmaktadır [5, 6]. Önbellek hiyerarşisi ve tasarımı içerisinde de bahsedilen yan kanalların varlığı araştırılmış ve literatürde yayınlanmıştır [7, 8].

Bahsedilen iki ana problemden ilki ele alındığında, hata düzeltimi için önbellek tutarlılık protokollerinin kullanılabileceğini ve HDK gibi bir çözüme oranla gecikmesinin çok daha az olabileceği görüldü. Sistemin hata toleransını farklı birimlerde farklı yöntemlerle artıracak araştırmalar literatürde bulunmaktadır [9, 10, 11, 12]. Fakat bu çalışma, önbellek tutarlılık protokol etiketleri kullanılarak birinci düzey paylaşımlı önbellekler için hata toleransını artırmaya yönelik ilk çalışmadır.

Önbellek yan kanallarının özellikle bulut sistemleri için büyük bir tehlike oluşturdu-

ğu bilinmektedir. Bu konu ile ilgili literatürde çeşitli açıklar bulunmuş, saldırılar oluşturulmuş ve bunlara karşı alınabilecek önlemler ve engelleme metodları geliştirilmiştir. Bunlardan bir tanesi olan [1], tutarlılık protokolü etiketlerinin erişim zamanına etkisini kullanarak bir saldırı tanımlamaktadır. Protokol etiketlerini kulanılarak oluşturulan bir yan kanaldan seri haberleşme sağlayan bir şema önerilen bu çalışma kapsamında alınabilecek önlemlerden bahsedilmemiştir. Aynı zamanda literatürde başka bir araştırma tarafından da bu probleme karşı bir savunma mekanizması önerilmemiştir. Bu çalışma, önbellek tutarlılık protokollerinin kötüye kullanımı sonucunda oluşan yan kanallar üzerindeki haberleşme trafiğini azaltma ve engelleme konusunu ele alan ilk araştırmadır.

1.1 Tezin Katkıları

Tez kapsamında, önbellek tutarlılık protokollerinin performansı ve haberleşme trafiği profillenmiş ve hiyerarşide var olan problemler üzerine yeni mekanizmalar önerilerek karşılaştırmalı analizler sunulmuştur. Bu kapsamda literatüre yapılan katkılar aşağıda sıralanmıştır:

- Arama ve Dizin tabanlı protokollerin çekirdek sayılarına göre oluşturdukları trafik ve performansları SPLASH2 program bütünü uygulamaları kullanılarak profillendi.
- HDK yerine birinci düzey paylaşımlı önbelleklerin hata toleransını artıracak yeni bir mekanizma öne sürüldü ve SPLASH2 program bütünü uykulamaları kullanılarak analizi yapıldı
- Önbellek tutarlılık protokol etiketlerinin kötüye kullanılmasıyla oluşan yan kanalların etkisini azaltacak yeni bir mekanizma öne sürüldü ve sağlanılan güvenlik SPLASH2 program bütünü uygulamaları kullanılarak analiz edildi.

1.2 Literatür Araştırması

Önbelleklerde hata düzeltimi ve hata toleransı sistem performansına ve çalışabilme süresine doğrudan etkilerinin önemi açısından literatürde geniş bir yer tutmaktadır. Sadler vd. , birinci düzey önbellekler için var olan hata düzeltme modellerini incelemiş ve HDK'nın performans üzerindeki etkilerini göstermiştir [3]. Yoon vd. ve Farbeh vd., HDK'nin getirdiği performans ve alan kaybını azaltmak için farklı HDK modelleri öne sürmüş ve hata toleransını düşüren ancak efektif çalışan mekanizmalar önermişlerdir [10, 13]. Wilkerson vd. farklı bir yaklaşımla önbelleklerde HDK ugulaması kolay yeni teknolojilerin entegre edilmesi üzerine çalışmıştır [14]. Mofrad vd., önbellek yapısını değiştirmeden hata eşlemeleri çıkarıp düzeni HDK uygulamayı kolaylaştıracak ve enerji

kullanımını düşürecek bir mekanizma öne sürmüştür [15]. Mohr vd. hata düzeltimi için HDK yerine daha küçük saklama alanları için daha efektif çalışan bir algoritma geliştirmiş ve devre düzeyinde gerçeklemesini ortaya koymuştur [16].

Donanım güvenliği sorunu özellikle Specte ve Meltdown ortaya çıktıktan sonra araştırılan bir konu haline gelmiştir [5]. Bununla beraber farklı yan kanallar ortaya konulmuş ve sistemlerde var olan güvenlik açıkları tespit edilerek kapatılmaya çalışılmıştır. Yarom vd. önbelleklerin nasıl profillenerek yan kanal oluşturmada kullanıldığını uygula- malı olarak anlatmış ve yeni oluşturdukları atomik işlem tabanlı profilleme algoritmasını göstermiştir [7, 8]. Bununla beraber önbellek erişimlerinin zamanlarının ölçülmesiyle oluşturulan yan kanallar literatürde geniş bir yere sahiptir [1, 6, 17]. Yan vd., önbellek tutarlılık şemalarının da yan kanal oluşturmak için kullanılabileceğini ortaya koymuştur. Yaptıkları çalışmada dizin girdilerini ele alarak yan kanal oluşturmayı başarmışlardır[6]. Benzer şekilde Yao vd., tutarlılık protokolleri etiketlerini kötüye kullanarak MESI durum makinesinin S ve E durumlarındaki adreslere erişimin deterministik olarak farklılık gösterdiğini bulmuş ve bu gözlem üzerine bir seri haberleşme yöntemi ortaya koymuştur [1].

Yapılan araştırmalar sonucunda literatürde tutarlılık protokollerinin birinci düzey paylaşımlı önbelleklerde hata düzeltimi için kullanımının önerilmediği görülmüştür. Ayrıca güvenlik konusunda [1]'ın oluşturduğu kritik açık ve önlem olarak bir çözüm gösterilmemesi tez kapsamında motivasyon niteliğinde olmuştur.

1.3 Tez Taslağı

Tez kapsamında yapılan çalışmalar şu şekilde sıralanmıştır:

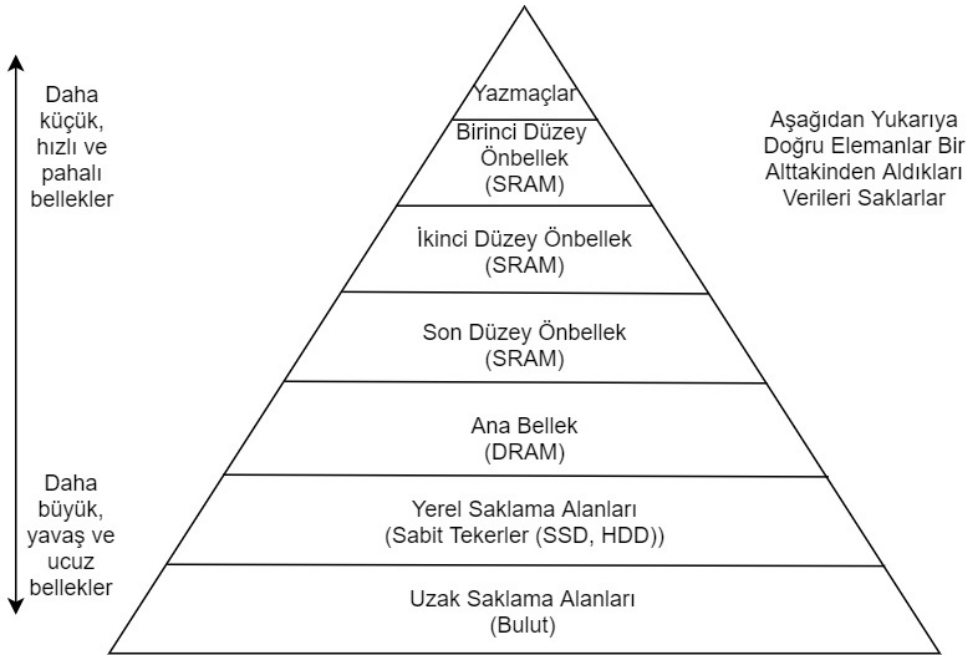
Bölüm 2 içerisinde kısım 2.1, 2.2, 2.3 bellek hiyerarşini, önbellek yapısını ve tutarlılık probleminin açıklamaları yer almaktadır. Kısım 2.4'da tutarlılık protokolleri ayrıntılı incelenirken, kısım 2.5'te güvenilirlik konusu ele alınıp geçici hatalar anlatılmış ve kısım 2.6'de donanım güvenliği ve tutarlılık şemalarının güvenlik açısından önemi değerlendirilmiştir. Bölüm 3, önerilen yöntemlerin detaylarını içerirken Bölüm 4, deney düzenini, uygulama profillerini ve analizleri ortaya koymaktadır. Bölüm 5, deney sonuçları ışığında yapılan değerlendirmeleri bulundurmaktadır. Son olarak Bölüm **??**, olası gelecek çalışmalara yol göstermektedir.

2. ÖN BİLGİ

2.1 Bellek Hiyerarşisi

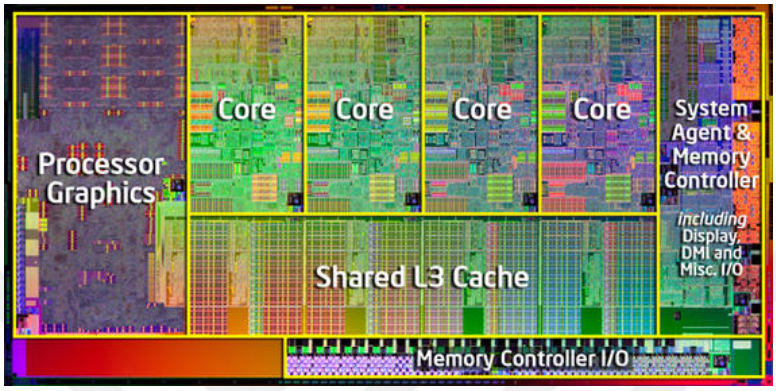
İyi bir bellek tasarımı, olabildiğince veri saklayabilecek kadar büyük, çok sayıda ve hızlıca üretilebilecek kadar ucuz ve erişim gecikmesi sistemin hızını kısıtlamayacak kadar hızlı olmalıdır. Ancak büyük ve hızlı erişilebilen bir bellek hem maliyetli olmakta hem de kapladığı alan aynı boyutta ancak erişim gecikmesi daha yüksek bir belleğe göre çok daha fazla olmaktadır. Büyük ve ucuz bir bellek tasarımı ise yüksek erişim gecikmesi nedeniyle performansı kısıtlar.

Bellekte tutulan verilerin fiziksel konumu ve bu verilere erişim sıklığı arasında bir ilişki vardır. Örneğin bir kod parçası bir bellek adresine erişmek isterse, aynı bellek adresine, ya da erişilen adresin civarındaki bellek adreslerine kısa zamanda tekrar erişim isteği gönderme ihtimali yüksektir. Belleğin belli alanlarına belli zamanlarda diğer adreslerden daha sık erişilmesine yerellik adı verilir. Yerellik kavramı, büyük bellek tasarımlarının yalnızca bir bölümünün daha küçük ve daha hızlı bellek birimlerine taşınmalarına izin verir. Modern sistemler, fiziksel bellek ve işlemci arasında, erişim hızı gittikçe artan, ancak kapasitesi azalan çok sayıda ara bellek birimi kullanır. Fiziksel bellek ve bellek ile işlemci arasındaki bu yapıların tamamı bellek hiyerarşisini oluşturur [18].



Şekil 2.1: Bellek hiyerarşisi

Anlatılan bellek hiyerarşisi Şekil 2.1'de gösterilmiştir. Şekildeki piramitte aşağıya doğru bellek elemanlarının boyutları artmata, hızları düşmekte ve saklanan bit başına harcanan enerjileri azalmaktadır. Bellek elemanlarından yazmaçlar, birinci ve ikinci düzey önbel- lekler çekirdek içinde bulunmaktadır. Paylaşımlı son düzey önbellekler, diğer elemanlara sağlanacak bağlantı ve bunların kontrolcüleri işlemci çipi üzerinde yer almaktadır. Şekil 2.2 Intel Sandy Bridge'in yapısını ve çip üzerinde yer alan bölümleri göstermektedir.



Şekil 2.2: Intel Sandy Bridge tasarımı

2.2 Önbellek

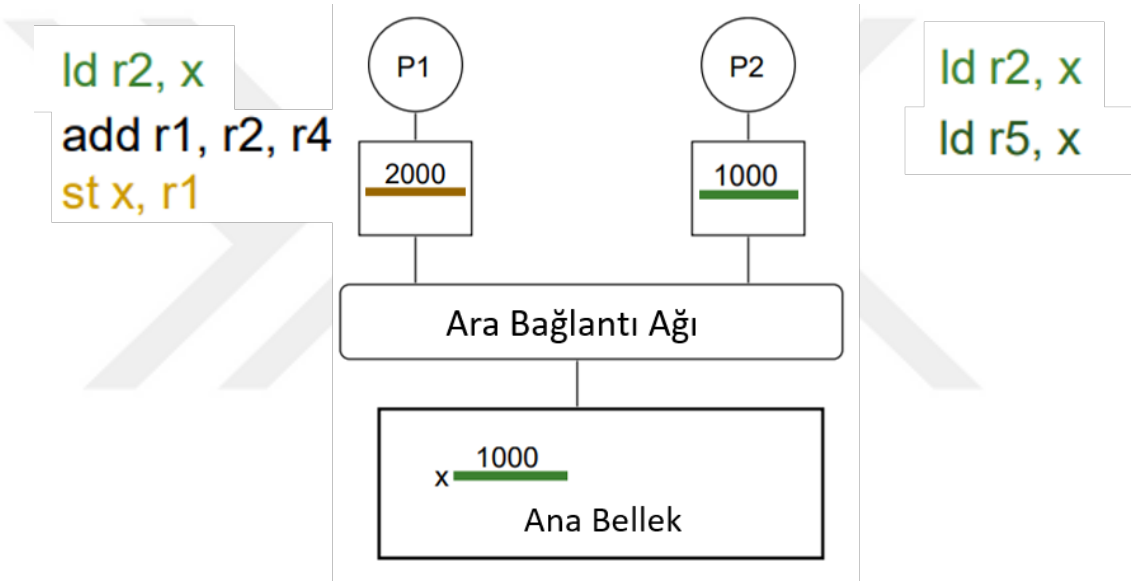
Bellek hiyerarşisinde, fiziksel bellek ve işlemci arasında kalan daha küçük bellek birimlerine önbellek adı verilir. Önbellek, fiziksel belleğin bir alt kümesi gibi davranır ve genelde fiziksel bellek adresinin tamamı ya da bir kısmıyla adreslenir. Fiziksel belleğe erişim, işlemciye uzaklığı ve belleğin büyüklüğü gibi nedenlerle yüzlerce saat çevrimi sürebilirken, önbelleğe erişim gecikmesi bir çevrime kadar düşebilmektedir.

Sistemde bir bellek erişimi yapıldığında, işlenecek veri fiziksel bellekten önbelleğe taşınır. Veri, daha sonraki erişimlerde önbellekten temin edilebilir. Önbellek, fiziksel bellekten çok daha küçük olduğundan, (Örneğin RAM boyutu birkaç gigabayt olabilirken, birinci düzey önbelleğin boyutu yalnızca birkaç kilobaytla sınırlıdır) fiziksel bellekte farklı adreslere sahip veriler önbellekte aynı adrese eşlenebilir. Bazı sistemler, önbellek- lerine aldıkları veriyi, bu veri başka bir veri tarafından çıkarılıncaya kadar fiziksel belleğe yazmadan güncelleyebilirler. Bu da fiziksel bellekte tutulan veri ile önbellekteki veri arasında bir uyumsuzluğa yol açar [18].

2.3 Önbellek Tutarlılığı

Modern çok çekirdekli sistemlerde, her çekirdeğin kendine ait bir ya da iki düzey önbelleği bulunmaktadır. Sistemde çok sayıda çekirdek bulunması durumunda, aynı

anda birden fazla çekirdek, aynı veri üzerinde işlem yapmak isteyebilir. Bu durum, aynı verinin aynı anda birden fazla çekirdeğin önbelleklerinde bulunabileceği anlamına gelmektedir. Her çekirdeğin, kendi önbelleğinde tuttuğu veri üzerinde, sistemin geri kalanından bağımsız olarak işlem yapabiliyor olması durumunda, bellekte ve tüm çekirdekler tarafından paylaşılmayan her önbellekte tutulan veriler arasında bir uyumsuzluk ortaya çıkması muhtemeldir [19]. Bunun bir örneği şekil 2.3'te gösterilmiştir. Bu şekle göre P1 işlemcisi ve P2 işlemcisi x adresinde duran 1000 verisini bellekten kendi önbelleklerine yüklemişlerdeir. P1 işlemcisi bu sayıyı güncelleyerek 2000 yapmıştır. Bu noktada P2'nin çalıştıracağı ikinci yükleme işleminin kullanacağı verinin doğruluğu problemi oluşmaktadır. P2 işlemcisi x adresinde yazan değeri kendi önbelleğinde 1000 olarak görmektedir, ancak P1 işlemcisinin önbelleğinde duran en güncel veriyi görememektedir. Bu oluşan sorun tutarlılık problemini göstermektedir.



Şekil 2.3: Tutarlılık problemi

Çok çekirdekli sistemlerde önbellek tutarlılık problemi, sistemde fiziksel bellekteki veriler üzerinde işlem yapabilen her birimin, her zaman verinin doğru kopyasıyla işlem yapması anlamına gelir.Bu problem, yazılım seviyesinde yapılacak eniyilemelerle sağlanabileceği gibi, sisteme eklenecek donanım birimleriyle donanım düzeyinde de sağlanabilir. Yazılım düzeyinde tutarlılık problemini çözmek programcılar için fazlasıyla yük oluşturduğundan [20], donanım çözümleri günümüzde kabul edilen çözüm haline gelmiştir.

Birden fazla çekirdek tarafından paylaşılan değişkenler gibi, önbelleğe alınması tutarlılığı tehlikeye atacak verilerin, derleyiciler tarafından belirlenerek önbelleğe alınmaması, tutarlılık adına yazılım düzeyinde uygulanmaktadır. Bu çözüm, paylaşılan veri her zaman birden fazla çekirdek tarafından kullanılmak isteniyorsa iyi sonuç

verebilir. Ancak, paylaşılan verinin yalnızca tek bir çekirdek tarafından kullanıldığı, ya da güncellenmeyip yalnızca okunduğu durumlar da mevcut olduğundan, bahsedilen çözümün neredeyse her zaman, daha sofistike önlemlerle önlenebilecek bir performans kaybına sebep olacağı açıktır [19].

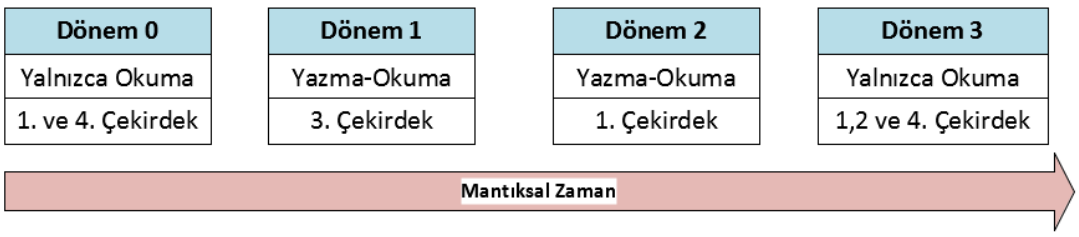
Donanım düzeyinde önbellek tutarlılığını sağlamaya yönelik çözümler, önbellek tutarlılık protokolleri olarak adlandırılır. Önbellek tutarlılık protokolleri, yalnızca sistemde bir tutarlılık sorunu mevcut olduğunda devreye girdikleri için, yazılımdan ve programcıdan bağımsız olmakla kalmayıp, tutarlılık protokolleriyle, derleyici düzeyinde uygulanan çözümlere göre daha verimli sonuçlar elde edilebilmektedir [19]. Kısım 2.4'de donanım çözümleri detaylı bir şekilde incelenmektedir.

2.4 Önbellek Tutarlılık Protokolleri

Önbellek tutarlılığını sağlamak için donanım düzeyinde uygulanan yöntemlerin genel adı, literatürde önbellek tutarlılık protokolleri (cache coherence protocols) olarak geçmektedir. Bir tutarlılık protokolünde dikkat edilmesi gereken iki önemli durum vardır:

- Tek yazma, Çoklu Okuma (Single -Write, Multiple Read): Herhangi bir mantıksal zamanda, bir bellek adresine yalnızca bir çekirdeğin yazma-okuma yetkisi, birden fazla çekirdeğin yalnızca okuma yetkisi bulunabilir. Bir bellek adresi aynı anda bahsedilen iki durumda da bulunamaz(Şekil 2.4)

- Veri Tutarlılığı: Yukarıda, bir bellek adresinin iki farklı durumda bulunabileceğini söylemiştik. Bellek adreslerinin bu iki durumdan birinde bulunduğu periyotların her birini bir dönem (epoch) olarak tanımlarsak, veri tutarlılığı durumunu, bir bellek adresinin değerinin, her yeni dönemin başında, o bellek adresine en son yazma yetkisi verilen dönemle aynı olması olarak açıklayabiliriz [21].



Şekil 2.4: Bellek adreslerine, çekirdeklerin kullanımına açılma durumuna göre mantıksal zaman içerisinde erişim dönemleri

Çoğu önbellek tutarlılık protokolü, yukarıda bahsedilen iki durumu sağlamak için tasarlanmıştır. Bu protokollerde kısaca, bir çekirdek bir adres üzerinde işlem

yapmak istediğinde, o adresin en güncel halini edinmek ve işlem yapılacak verinin aynı anda başka çekirdekler tarafından kullanımda olmadığından emin olmak adına sisteme çeşitli sinyaller gönderir. Bu sinyaller, bellek adresinin o an içinde olduğu dönemi sonlandırarak, gönderilen isteğin türüne göre yeni bir okuma-yazma ya da yalnızca yazma dönemi başlatır. Tutarlılık için kontrol edilen verinin boyutu, genelde önbellekteki bir veri öbeğinin büyüklüğü kadardır [21].

Tutarlılık protokollerinin gerektiği şekilde uygulandığından emin olmak adına, her bir bellek birimi için (farklı düzey önbellekler ve fiziksel bellek) tutarlılık denetleyicisi adı verilen bir donanım birimine ihtiyaç duyulur. Tutarlılık denetleyicisi, çekirdek ve birimler arası ağ arasında tutarlılık sinyallerinin iletilmesini sağlayan bir arayüz görevi görür.

Tutarlılık denetleyicisi tarafından denetlenen sinyaller, uygulanan tutarlılık protokolüne göre değişiklik gösterebilir. Son seviye önbellek ve fiziksel bellek tutarlılık denetleyicilerine bellek denetleyicisi adı da verilir. Her bir tutarlılık denetleyicisi, sorumlu olduğu bellek biriminin her bir bloğu için, uygulanan tutarlılık protokolüne göre değişebilen bir sonlu durum makinesi oluşturur. Bir tutarlılık protokolü, aslında bu sonlu durum makinelerinin gerçeklenme biçimidir [21].

Önbellek tutarlılık sinyallerinin haberleşme modelleri dizin tabanlı ve arama tabanlı olmak üzere iki ana başlıkta incelenir. Bu protokoller çekirdekler arasında tutarlılık sinyallerinin nasıl iletileceğini tanımlar. Tez kapsamında bu iki protokolün performans ve trafik analizi yapılmıştır. Ancak, ana odak tutarlılığı sağlayan sonlu durum makineleri ve bu durum sinyallerinin etkileridir. Bu durum makinelerini içeren tutarlılık protokolleri de yaz ve güncelle ile yaz ve geçersiz kıl olmak üzere iki ana başlık altında toplanmaktadır.

Dizin temelli tutarlılık modellerinde, genelde bellek denetleyicisinin bir parçası olan ve hangi bellek adreslerinin birer kopyasının hangi çekirdeklerde bulunduğu bilgisini tutan merkezi bir denetleyici, ve fiziksel bellekte bulunup, farklı çekirdeklerde bulunan verilerin sistem gözünde (global) durumunu tutan bir dizin yer almaktadır.Bir çekirdek, önbelleğinde bulunan bir veriyi güncellemek istediğinde, merkezi denetleyiciye, bu verinin yalnızca kendi kullanımına ayrılmasını sağlayacak bir istek gönderir. Kontrolcü, istek gönderen çekirdeğe izin vermeden önce, verinin bir kopyasını bulunduran diğer tüm önbelleklere istek göndererek, verinin diğer tüm kopyalarının geçersiz kılınmasını sağlar. Bu işlemden sonra, başka bir çekirdek söz konusu veriye erişmek istediğinde bu durum, veri önbellekte bulunamamış gibi işlem görür. Durum bilgisinin güncel tutulabilmesi adına, çekirdeklerin lokalde yaptığı, global durumu değiştirebilecek her işlemin, denetleyiciye bildirilmesi gerekmektedir [22].

Arama tabanlı protokoller, tutarlılığı sağlama görevini, çok çekirdekli sistemdeki

tüm çekirdeklerin önbellek kontrolcüleri arasında bölüştürmeyi amaçlar. Bu yöntemde, her önbellekte, paylaşılan bir verinin durum bilgisi tutulmaktadır. Paylaşılan bir veri bloğu güncellenmek istendiğinde, bu istek tüm önbelleklere bir arama ağı üzerinden duyurulur. Çekirdeklerin önbellek kontrolcüleri, bu ağı yoklayarak, ağ üzerinden duyurulan bildirimlere erişebilir [22].

2.4.1 Yaz ve güncelle protokolleri

Yaz ve güncelle protokolleri, aynı anda aynı veriye birden fazla çekirdekten gelen yazma isteğine izin verir. Bu yöntemde, bir veri bloğu güncellenmek istendiğinde, güncellenecek veri, o verinin bir kopyasını bulunduran tüm önbelleklere gönderilir. Böylece tüm önbelleklerde verinin her zaman en güncel hali bulunmuş olur. Bu başlık altında, modern sistemlerde kullanılmış olan yaz ve güncelle protokolleri kısaca açıklanmıştır.

2.4.1.1 Firefly protokolü

Bu protokol, her bir veri öbeği için Valid-Exclusive, Shared ve Dirty olmak üzere üç farklı durum tanımlar. Bu durumların açıklamaları aşağıda verilmiştir:

- **Valid-Exclusive:** Verinin fiziksel bellektekiyle uyumlu bir kopyası bir ve yalnız bir önbellekte bulunmaktadır.
- **Shared:** Verinin lokal önbellekteki kopyası fiziksel bellektekiyle uyumludur ve verinin kopyaları birden fazla önbellekte bulunuyor olabilir.
- **Dirty:** Verinin, fiziksel bellektekinden farklı bir kopyası bir ve yalnız bir önbellekte bulunmaktadır [23].

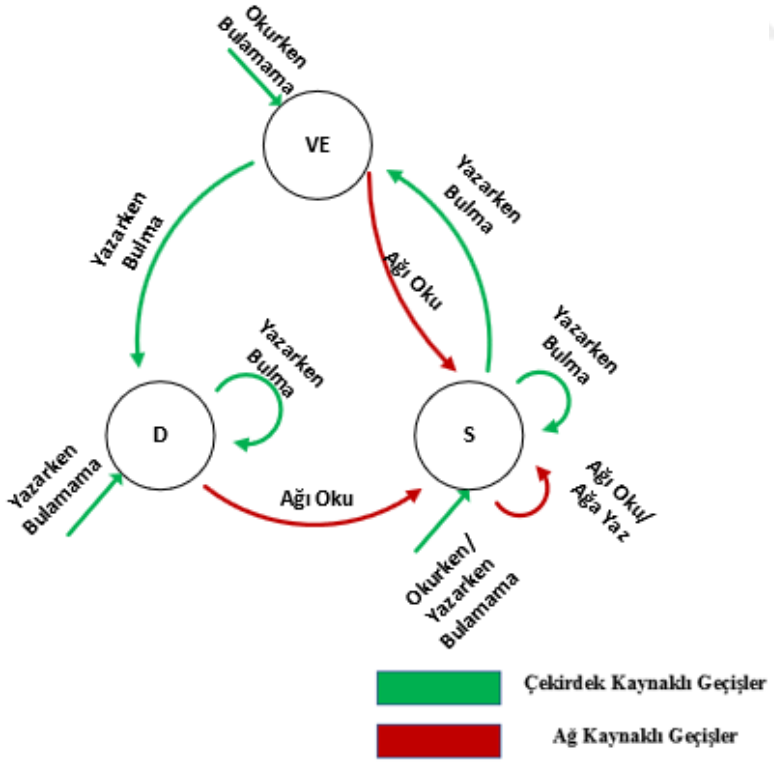
Archibald ve Baer, olası yazma-okuma işlemlerinde bu protokolün işleyişini aşağıdaki şekilde açıklamıştır:

- **Okurken Bulamama:** Veri başka bir önbellekte bulunmuyorsa, Valid-Exclusive durumunda fiziksel bellekten alınır. Verinin bir kopyasının başka bir önbellekte bulunması durumunda veri bu önbellekten edinilir. Bu önbellekte bulunan veri Dirty durumundaysa veri aynı zamanda fiziksel belleğe de yazılır. Verinin durumu, veriyi bulunduran tüm önbelleklerde Shared olarak değiştirilir.
- **Yazarken Bulma:** Veri Shared durumunda bulunuyorsa, Veri fiziksel belleğe yazılmanın yanı sıra, verinin bir kopyasını bulunduran tüm önbellekler kendi lokal kopyalarını günceller. Diğer durumlarda yazma işlemi gecikme olmaksızın gerçekleşebilir, verinin son durumu Dirty olarak güncellenir.

- **Yazarken Bulamama:** Veri fiziksel bellekten alınıyorsa Dirty durumunda alınır. Verinin başka bir önbellekten alınması durumu, yazarken bulma bölümünde anlatılan shared olma durumuyla benzerdir [23].

Durumların açıklamalarına göre oluşan sonlu durum makinesi Şekil 2.5'te gösterilmiştir. Şekildeki "Ağı Oku" komutu, bir önbelleğin ağ üzerinden yayınlanan veriyi, kendi kopyasının üzerine yazması anlamına gelmektedir [23]. Şekil 2.6 verilerin herhangi iki önbellekte aynı anda bulunabileceği durumları içermektedir.

Firefly protokolünde, paylaşılan veri bloklarına yapılan tüm yazma işlemlerinde verinin tüm önbelleklerdeki kopyaları güncellendiğinden, Invalid durumuna ihtiyaç yoktur [23].



Şekil 2.5: Firefly protokolüne ilişkin sonlu durum makinesi şeması.

	VE	S	D
VE	×	×	×
S	×	✓	×
D	×	×	×

Şekil 2.6: Verinin, herhangi iki önbellekte bulunabileceği durumlar

2.4.1.2 Dragon protokolü

Bu protokolü uygulayan sistemlerde, veri öbeği Valid-Exclusive, Shared-Dirty, Shared-Clean ve Dirty olmak üzere dört farklı durumda bulunabilir. Bu durumların açıklamaları aşağıdaki gibidir:

- **Valid-Exclusive:** Veri öbeğinin fiziksel bellekle uyumlu bir kopyası yalnızca bir önbellekte bulunmaktadır.
- **Shared Dirty:** Veri birkaç önbellek tarafından paylaşılıyor olabilir, ancak fiziksel belektekiyle uyumlu değildir.
- **Shared-Clean:** Veri birkaç önbellek tarafından paylaşılıyor olabilir ve verinin önbelleklerdeki kopyası fiziksel bellektekiyle uyumludur.
- **Dirty:** Verinin fiziksel bellektekiyle uyuşmayan bir kopyası yalnızca bir önbellekte bulunmaktadır[24].

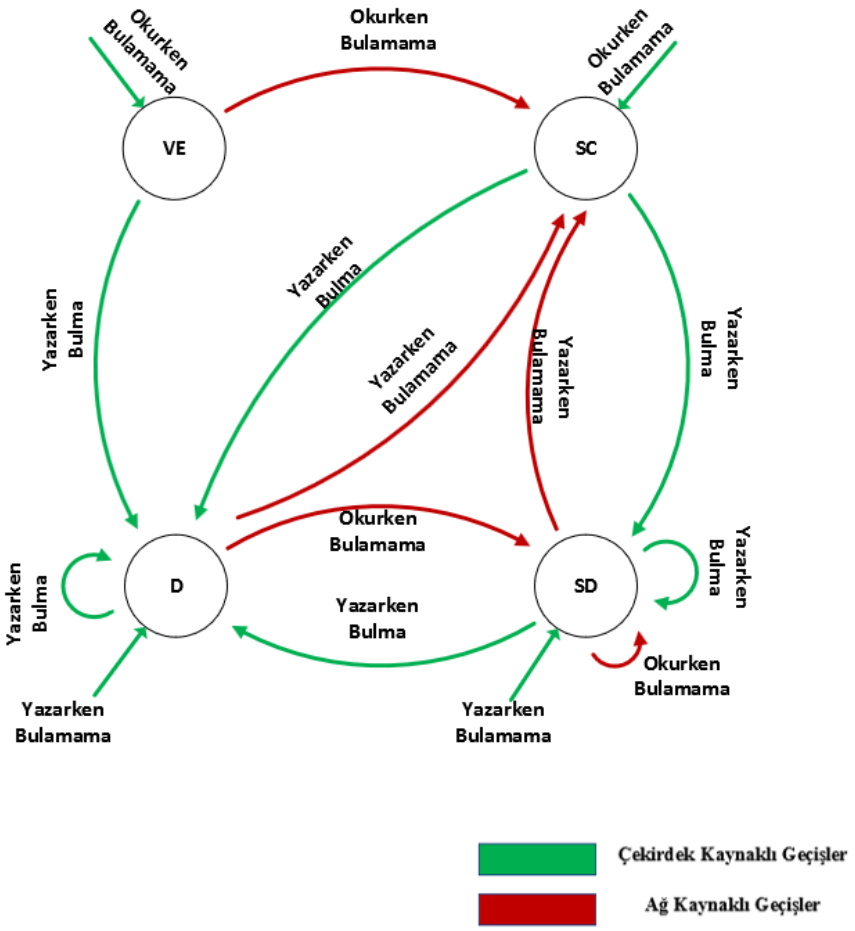
Dragon protokolü uygulayan sistemlerde, yapılan bir işlem sonucu oluşabilecek durumları Archibald ve Baer şu şekilde incelemiştir:

- **Okurken Bulamama:** Veriyi Dirty ya da Shared-Dirty durumlarından birinde bulunduran başka bir önbellek varsa veri öbeği bu önbellekten alınır. Veriyi sağlayan önbellek verinin durumunu Shated-Dirty olarak değiştirir. Diğer durumlarda veri fiziksel bellekten alınır. Verinin bir kopyası birden fazla önbellekte bulunuyorsa veri öbeğinin durumu Shared-Clean olur. Veri son durumda paylaşılmıyorsa önbellekte Valid-Exclusive olarak tutulur.
- **Yazarken Bulma:** Veri öbeği Shared durumlarından birindeyse veri öbeğinin yeni değeri ağ üzerinden sistemdeki diğer önbelleklerle paylaşılır. Veriyi bulunduran tüm önbellekler kendi kopyalarını güncelleyerek Shared-Clean durumuna

çekerler. Bu durumda yazma isteğini gönderen önbellek, veri başka önbellekler tarafından paylaşılıyorsa kendi kopyasının durumunu Shared-Dirty'ye, paylaşılmıyorsa Dirty'ye çeker. Başlangıçta veri öbeği Shared durumlarından birinde değilse yazma işlemi gecikme olmaksızın tamamlanır, verinin son durumu Dirty olur.

- **Yazarken Bulamama:** Verinin önbelleğe getirilmesi, okurken bulamama durumuyla benzerdir.İşlem tamamlandıktan sonra verinin sistemdeki durumuna karar verilmesi, yazarken bulamama bölümünde verinin Shared durumlarından birinde olması durumuyla benzerdir [24].

Durumların açıklamalarına göre oluşan sonlu durum makinesi Şekil 2.7'de gösterilmiştir [24]. Şekil 2.8 verilerin herhangi iki önbellekte aynı anda bulunabileceği durumları içermektedir.



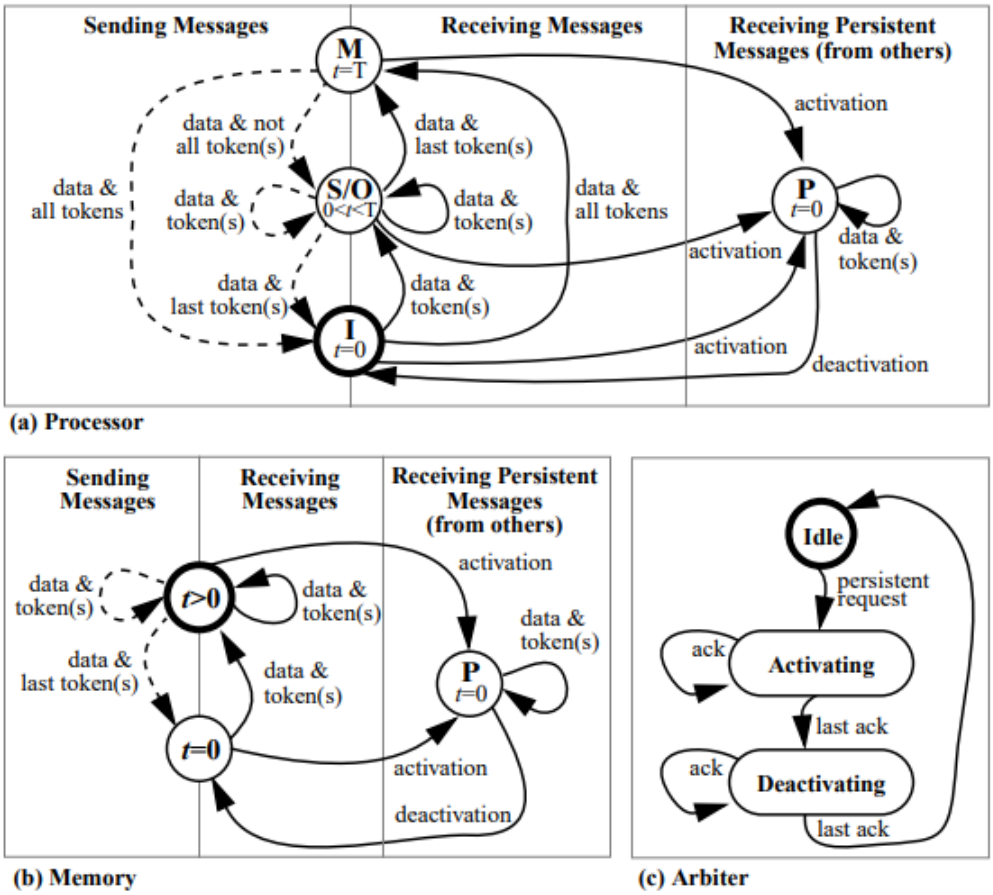
Şekil 2.7: Dragon protokolüne ait sonlu durum makinesi şeması

	VE	SD	SC	D
VE	×	×	×	×
SD	×	✓	✓	×
SC	×	✓	✓	×
D	×	×	×	×

Şekil 2.8: Verinin, herhangi iki önbellekte bulunabileceği durumlar

2.4.1.3 Ayraç tabanlı tutarlılık protokolü

Bu protokolün oluşturulma amacı performansın ve doğruluğun birbirinden ayrı değerlendirilmesidir. Bunu sağlayabilmek için geliştirilen bu sistem hem arama temelli hem de dizin temelli tutarlılık protokollerinin iyi taraflarını kompleks bir yapı karşılığında elde edebilmektir. Temel olarak sık karşılaşılan durumları hızlandırmak üzerine kurulan bir yapı ve doğruluğu sağlamak için geliştirilen ayraç sisteminden oluşur. Ayraç sistemi aynı anda birden çok okuma işlemine ve bir yazma işlemine izin vermektedir bunun kontrolü isteklerin dolaylandırıldığı bir global dizin sisteminde tutulur. Bu dizin sistemde ayrılan tüm ayraçları her bellek bloğu için tutar. Dizin istekler geldiği zaman bu isteklerle beraber ayraçların istek yapan işlemcilere verilmesini de sağlar. Okuma için ayraçlar birer birer dağıtılırken, yazma işlemleri için tüm ayraçlar bir işlemciye dağıtılır. Ayraçların durumunun kontrolü için diğer çekirdekler bu ayraçların varlığını aralarındaki ağ üzerinden dinleyerek görür. Böylece iki temel protokolün avantajları kullanılarak ayraçların daha çok kullanılan bellek bloklarına dağıtılması ve kontrolün bu sık kullanılan bloklarda hızlı gerçekleşmesini sağlar. Doğruluk içinse isteklere zaman aşımı veya bir zamanlama uygulanmadığından açlık durumu köşe durumlar için performans kaybına sebep olsa da sağlanmış olur. Ancak [25]'de incelendiğine göre bu değişim genel durumlar karşısındaki kazançlara oranla önemsiz kalmıştır. Basitçe çalışma prensibi Şekil 2.9'da özetlenmiştir.

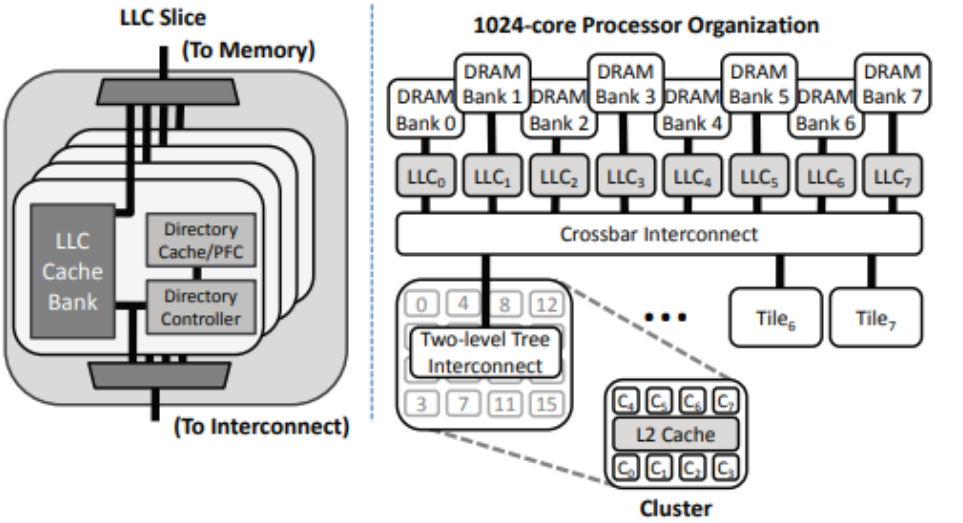


Şekil 2.9: Dizinden ayrılan ayraçların ve verinin trafiğini gösteren durum makineleri şemaları (a) İşlemci için (b) bellek blokları için (c) köşe durumlar için

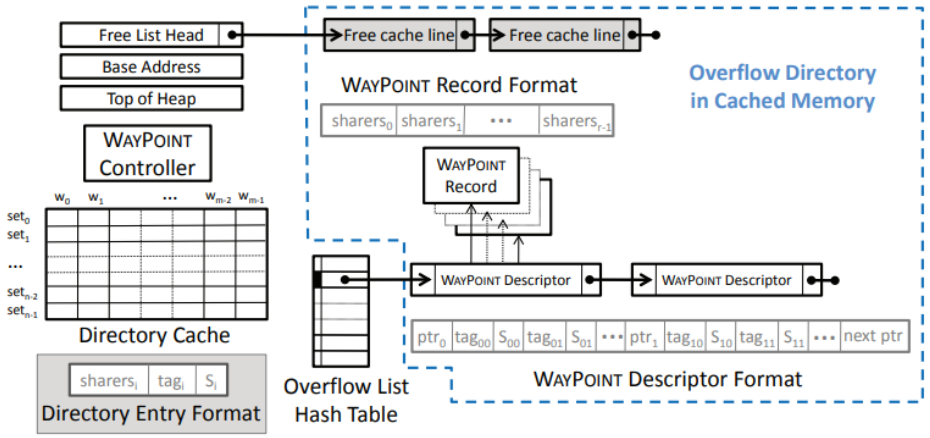
Ayrıntılı olarak bakıldığında işlemciler arasında tutarlılık kontrol edilirken MSI protokolüne benzer bir durum makinesi dinleme tabanlı tutarlılığı sağlamak için kullanılır. Burada aynı zamanda global olarak ayrılan ayraçlar da aktarılarak izinlerin takibi sağlanmış olur. Bu şekilde erişilen adreslerin bellekteki durumlarına bakmaksızın durum makinesindeki değişiklikler istek tipine (okuma/yazma) ortak ağ üzerinde bu ayraçların değiş tokuşu ve varlığı üzerinden işler. Bellek sisteminde ise var olan verinin ve ayraçların işlemciye gönderilen ve işlemciden gelen mesajlara göre trafiğini görebilirsiniz. Aktivasyon ve deaktivasyon işlemleri, gelen isteklerin denk geldiği bellek adreslerindeki ayraç durumunu kontrol etmek için kullanılır. Aktif duruma gelemeyen yani global olarak ayraç atanmamış bloklara olan istekler kalıcı istek olarak boş durumunda bekleyerek ayraç ayrılma işinin global dizin tarafından yapılmasını bekler [24].

2.4.1.4 WAYPOINT tutarlılık protokolü

WAYPOINT protokolü özellikle 1000 işlemciden fazla sistemlerde kullanılması için düzenlenmiş bir dizin tabanlı tutarlılık protokolüdür. Bağlama modeli olarak crossbar'a sahip birçok işlemcinin gruplanarak paylaştığı çeşitli derecelerde paylaşılan önbellek modeli üzerinde çalışmaktadır. Bu gruplama sistemi 2.10'da görülmektedir.



Şekil 2.10: Temel alınan işlemci bağlantıları



Şekil 2.11: WAYPOINT mikromimarisi

Kabul edilen modele göre optimize edilmiş bir dizin mimarisine dayanan bu sistemin

çalışma mekanizmasını Şekil 2.11'de detaylı olarak bulabilirsiniz. Bu sistemde sıralı liste şeklinde tutulan bir önbellekleme yapısı kullanılır. Öncelikle dizin önbelleğine erişen istekler bulunamazsa taşma dizini olarak adlandırılan bir yapıya yönlendirilir. Her bir dizin önbelleğine bağlı bu taşma dizinleri paylaşılan son seviye önbelleklerinin boş satırlarında saklanır. Bir sonraki satırı gösteren bu taşma satırları, dizin önbelleğindeki bulamama oranını azaltmak için oluşturulan listelerdir. Bu şekilde Dizinlerin efektifliği önbelleklerin boş satırlarını da kullanarak artırılmış olur. Bu listelerde taşma satırlarının bulunduğu ve bellekte duran hash tablolarının göstergeleri tutulur. Bu sayede belleğe erişim sırasında büyük dizinlere hızlı erişim sağlanmış olunur [26].

2.4.2 Yaz ve geçersiz kıl protokolleri

Bir veriye, herhangi bir zamanda birden fazla çekirdeğin okuma isteği göndermesine izin veren, ancak aynı anda birden fazla çekirdeğin aynı veriyi güncellemesine izin vermeyen protokollerdir. Bir çekirdek, paylaşılan bir veriyi güncellemeden önce, verinin diğer önbelleklerdeki kopyalarının geçersiz kılınmasını sağlayacak bir istek gönderir. Böylece, söz konusu veri bloğu, yalnızca isteği gönderen çekirdeğe ayrılmış olur. Write-Invalidate protokollerini gerçeklemek adına çoğu sistemin önbelleğinde, her bir veri satırının durum bilgisini tutmak adına, her satır için etiket kısmında fazladan iki bit tutulmalıdır. Önbellekte tutulan veri, bu iki bitin durumlarına göre M(odified), E(xclusive), S(hared) ya da I(nvalid) adı verilen dört durumdan birinde olabilir [24]. Önbellek tutarlılığı kavramının ortaya çıkışından itibaren çok sayıda write-invalide protokolü ortaya atılmıştır.Bu bölümde öne sürülen farklı protokollerin detayları bulunmaktadır .

2.4.2.1 Write-Once tutarlılık protokolü

Kronolojik olarak, ilk write-invalidate tutarlılık protokolü, James R. Goodman tarafından 1983'te ortaya atılmış olan write-once protokolüdür [27]. Bu protokolde, bir veri bloğu Invalid, Valid, Reserved ve Dirty olmak üzere dört farklı durumda bulunabilir.

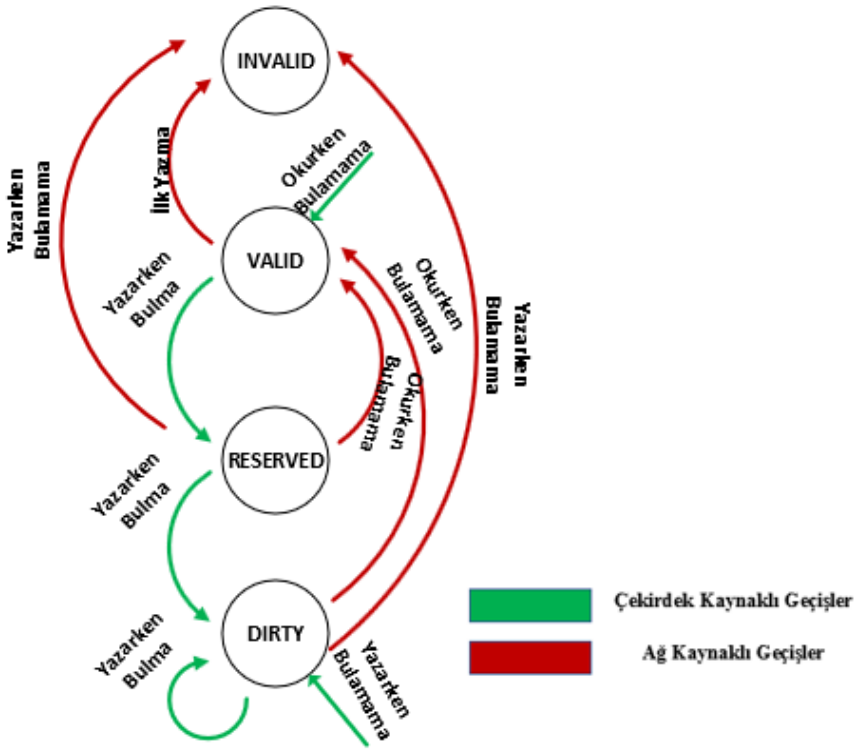
- **Invalid:** Lokal önbellekteki veri bloğundaki verinin geçersiz olduğunu ifade eder.
- **Valid:** Verinin kullanılabilir bir kopyasının önbellekte bulunduğunu gösterir.
- **Reserved:** Verinin, ana bellektekiyle uyumlu bir kopyasının bir ve yalnız bir önbellekte bulunduğunu gösterir.
- **Dirty:** Verinin, ana bellektekinden farklı bir kopyasının bir ve yalnız bir önbellekte bulunduğunu gösterir [28].

Archibald ve Baer, write-once protokolünün olası üç durum için analizini şu şekilde yapmıştır:

- **Okurken Bulamama:**Verinin bir kopyası başka bir önbellekte Dirty durumundaysa, veri, ana bellek yerine verinin güncel kopyasını bulunduran önbellek tarafından temin edilir. Bu işlem yapılırken veri bloğu aynı zamanda ana belleğe de yazılır. Verinin hiçbir önbellekte bulunmaması durumunda fiziksel belleğe erişilir. İşlem sonunda verinin güncel bir kopyasını bulunduran tüm önbellekler veri bloğunun durumunu Valid'e çevirir.

- **Yazarken Bulma:** Dirty ya da Reserved durumundaki bloklara yazma işlemi gecikme olmaksızın tamamlanır. Bu işlem sonunda blok Dirty durumunda bulunur. Blok Valid durumunda bulunuyorsa, veri önbelleğe yazılırken aynı zamanda fiziksel belleğe de yazılır. Bu işlemin sonunda veri bloğu, yazma işlemini gerçekleştiren önbellekte Reserved durumunda, verinin yazılmadan önceki kopyalarını bulunduran önbelleklerde Invalid durumunda bulunur.

- **Yazarken Bulamama:** Verinin önbelleğe getirilmesi işlemi, okurken bulamama durumuyla bezerdir. Ancak isteği gönderen önbellek dışındaki önbellekler, yazarken bulamama durumunda kendi kopyalarını Invalid durumuna çekerler. İşlem sonunda veri bloğu, isteği gönderen önbellekte Dirty durumunda bulunur [23].



Şekil 2.12: Write-Once protokolü için sonlu durum makinesi şeması

Detaylandırılan sonlu durum makinesinin şeması Şekil 2.12'de verilmiştir. Şekilde yeşil renkle gösterilen oklar, verinin, isteği gönderen çekirdeğin önbelleğindeki durum değişikliğini, kırmızı renkle gösterilen oklar, bir işlem yapıldığında önbellekler arası ağda iletilen emirleri göstermektedir. Örneğin, "valid" durumundaki bir veriye yapılan ilk yazma isteği, isteği gönderen önbellekte "yazarken bulma" durumuna, diğer önbellekte "ilk yazma" durumuna denk gelmektedir [23]. Şekil 2.13 verilerin iki önbellekte aynı anda bulunabileceği durumları göstermektedir.

	V	R	D	I
V	✓	×	×	✓
R	×	×	×	✓
D	×	×	×	✓
I	✓	✓	✓	✓

Şekil 2.13: Verinin herhangi iki önbellekte bulunabileceği durumlar

2.4.2.2 SYNAPSE protokolü

Bu tutarlılık protokolünde, bir veri bloğu Invalid, Valid ve Dirty olmak üzere üç farklı durumda bulunabilir. SYNAPSE protokolünde, her bir veri bloğunun etiket bölümünde, verinin, önbellekte bulunamama durumunda, fiziksel bellekten çekilip çekilmeyeceğini gösteren fazladan 1-bitlik bir alan bulunur. Bu sayede, aranan veri bloğunu temin edebilecek bir önbelleğin yeterince hızlı cevap vermemesi durumunda fazladan bellek erişimine gerek kalmaz [23].

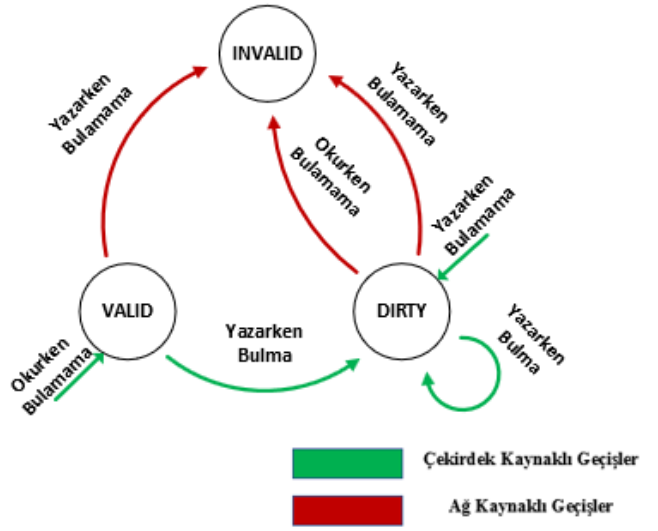
Archibald ve Baer, bu protokolü kullanan sistemlerde olası üç durumu aşağıdaki şekilde açıklamıştır:

- **Okurken Bulamama:** Verinin bir kopyası başka bir önbellekte Dirty durumunda bulunuyorsa, veri bloğu- nun sahibi olan önbellek veriyi belleğe yazar, veri bloğunun bu önbellekteki durumu Invalid olarak güncellenir. Daha sonra okuma isteği yollayan önbellek veriyi bellekten okur. Veri hiçbir önbellekte bulunmuyorsa doğrudan fiziksel bellekten edinilir. Verinin, okuma isteği gönderen önbellekteki son durumu Valid olur.

- **Yazarken Bulamama:** Bellek erişimleri bir önceki durumla benzerdir. Verinin bir kopyasına sahip tüm önbellekler veri öbeğinin durumunu Invalid olarak değiştirir. Veri öbeği, isteği gönderen önbelleğe Dirty durumunda getirilir.

- Yazarken Bulma: Veri öbeği Dirty durumundaysa işlem gecikme olmaksızın tamamlanır. Diğer durumlar yazarken bulamama durumuyla benzerdir [23].

SYNAPSE protokolünün sonlu durum makinesinin şeması Şekil 2.14'te verilmiştir [23]. Şekil 2.15 verinin iki önbellekte aynı anda bulunabileceği durumları göstermektedir.



Şekil 2.14: SYNAPSE protokolüne ilişkin sonlu durum makinesi şeması

	V	D	I
V	✓	×	✓
D	×	×	✓
I	✓	✓	✓

Şekil 2.15: Herhangi iki önbellekte bir verinin bulunabileceği durumlar

2.4.2.3 Berkeley ownership protokolü

Bu tutarlılık protokolünde, bir veri bloğu Invalid(INV), UnOwned(UNO), Owned Exclusively(EXC) ve Owned NonExclusively(NON).olmak üzere dört farklı durumda bulunabilir [29]. Bu dört durumun açıklaması aşağıdaki gibidir:

- **INV:** Verinin önbellekteki kopyası geçersizdir.
- **UNO:** Veri birkaç önbellek tarafından paylaşılıyor olabilir, ancak izin alınmadan veri üzerinde işlem yapılamaz.

- **EXC:** Veri bir ve yalnız bir önbellekte geçerli olarak bulunmaktadır. Veri üzerinde istenildiği gibi işlem yapılabilir.

- **NON:** İstek yollayan önbellek veri bloğunun sahipliğini(ownership) elinde tutar, ancak veri üzerinde diğer önbelleklere bildirilmeden işlem yapılamaz [29]

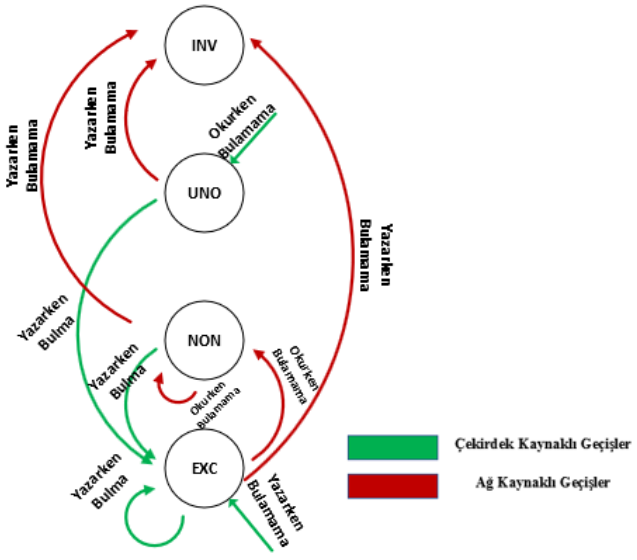
Berkeley protokolünün işleyişi, Archibald ve Baer tarafından aşağıdaki gibi analiz edilmiştir:

- **Okurken Bulamama:** Veriyi EXC ya da NON durumunda bulunduran bir önbellek varsa bu önbellek veriyi isteği gönderen önbelleğe iletir, verinin bu önbellekteki kopyası NON durumuna çevrilir. Diğer durumlarda veri fiziksel bellekten getirilir. İsteği gönderen önbellekte verinin son durumu UNO olarak belirlenir.

- **Yazarken Bulma:** Veri bloğu EXC durumundaysa işlem gecikme olmaksızın tamamlanır. Diğer durumlarda, verinin diğer önbelleklerdeki kopyaları INV durumuna getirilir. Daha sonra isteği gönderen önbellek yazma işlemini yaparak veri bloğunun durumunu EXC olarak değiştirir.

- **Yazarken Bulamama:** Verinin isteğin sahibi önbelleğe getirilmesi okurken bulamama durumuyla benzerdir. Verinin bu önbellekteki son durumu EXC olurken, verinin kopyalarını bulunduran diğer önbellekler kendi kopyalarını INV olarak günceller [29]

Berkeley ownership protokolüne ait sonlu durum makinesi Şekil 2.16'da verilmiştir [23]. Şekil 2.17 verinin iki önbellekte aynı anda bulunabileceği durumları göstermektedir.



Şekil 2.16: Berkeley protokolüne ilişkin sonlu durum makinesi şeması

	UNO	EXC	NON	INV
UNO	✓	×	✓	✓
EXC	×	×	×	✓
NON	✓	×	×	✓
INV	✓	✓	✓	✓

Şekil 2.17: Bir verinin herhangi iki önbellekte bulunabileceği durumlar.

2.4.2.4 M.E.S.I. (Illinois) protokolü

MESI, donanım düzeyinde önbellek tutarlılığı sağlamak için sıkça kullanılan bir protokoldür. Bu protokolde önbellekte bulunan her veri öbeği Modified, Exclusive, Shared ve Iinvalid olmak üzere dört farklı durumda bulunabilir. Bu dört durumu ifade edebilmek için önbellekte her bir veri öbeğinin etiket bölümünde fazladan iki bit yer alır [19].Protokolü ilk ortaya atan Papamarcos ve Patel, M, E, S ve I durumlarını aşağıdaki biçimde açıklamıştır:

- **Modified:** Verinin önbellekte bulunan kopyası, fiziksel bellektekinden farklı ve daha günceldir. Bir veri aynı anda en fazla bir önbellekte M durumunda bulunabilir.
- **Exclusive:** Verinin önbellekte bulunan kopyası fiziksel bellektekiyle aynıdır ve başka bir önbellekte bulunmamaktadır.
- **Shared:** Verinin önbellekte bulunan kopyası fiziksel bellektekiyle aynıdır ve aynı veri başka önbelleklerde bulunuyor olabilir.
- **Invalid:** Verinin önbellekte bulunan kopyası geçersizdir. I durumunda bulundurduğu bir veriye yapılan erişim isteklerine CACHE MISS gibi davranılır [19, 30]

MESI protokolünü uygulamak için, her bir çekirdekteki her önbellek satırı için, sayılan dört farklı durumu gerçekleyen bir sonlu durum makinesi kullanmak gerekir 2.18

MESI protokolünün işleyişi, 2.18'de gösterilen durum makinesine göre [19]'de aşağıdaki gibi açıklanmıştır:

- **Okurken Bulamama:** Okuma isteği gönderilen bir adresin, o çekirdeğin önbelleğinde bulunmaması durumudur. Bu durumda veriye erişmek isteyen çekirdek, bulunamayan adresi bellekten çekmek için fiziksel belleğe okuma isteği gönderir.

Aynı zamanda arama(snoop) ağına da yapılacak okuma isteğini diğer çekirdeklerin önbelleklerine duyurulması adına sinyal gönderilir.

  - Eğer verinin geçerli bir kopyası başka bir önbellekte bulunmuyorsa, veriye erişmek isteyen çekirdek, veriyi fiziksel bellekten okur. Verinin durumu I'dan E'ye geçirilir. Bu durumda isteği gönderen çekirdeğe başka çekirdeklerden bir sinyal dönmez.
  - Erişilmek istenen veri başka bir çekirdeğin önbelleğinde S durumunda bulunuyorsa, veriyi S durumunda bulunduran çekirdekler, erişim isteği gönderen çekirdeğe, verinin paylaşımda olduğu bilgisini ileten bir sinyal gönderir. Daha sonra veri, isteği gönderen çekirdek tarafından fiziksel bellekten okunur. Verinin durumu I'dan S'ye geçirilir.
  - Erişilmek istenen veri başka bir çekirdeğin önbelleğinde E durumunda bulunuyorsa, veriyi E durumunda bulunduran çekirdekler, erişim isteği gönderen çekirdeğe, verinin paylaşımda olduğu bilgisini ileten bir sinyal gönderir. Daha sonra, veriyi E konumunda bulunduran önbellek, verinin durumunu E'den S'ye çevirir. Veri, isteği gönderen önbellek tarafından fiziksel bellekten okunur, verinin bu önbellekteki durumu I'dan S'ye geçirilir.
  - Erişilmek istenen veri, başka bir çekirdeğin önbelleğinde M durumunda bulunuyorsa, veriyi M durumunda bulunduran çekirdek, veriye erişmek isteyen çekirdek tarafından gönderilen okuma isteğini durdurur. Bu durumda bazı sistemlerde veri, veriyi M durumunda bulunduran önbellekten temin edilirken, verinin önce M durumunda bulunduğu önbellek tarafından fiziksel belleğe yazılması, ardından isteği gönderen çekirdek tarafından fiziksel bellekten okunması da kullanılan bir yöntemdir. İşlem sonunda veriyi bulunduran tüm önbellekler, verinin durumunu S'ye çevirir.

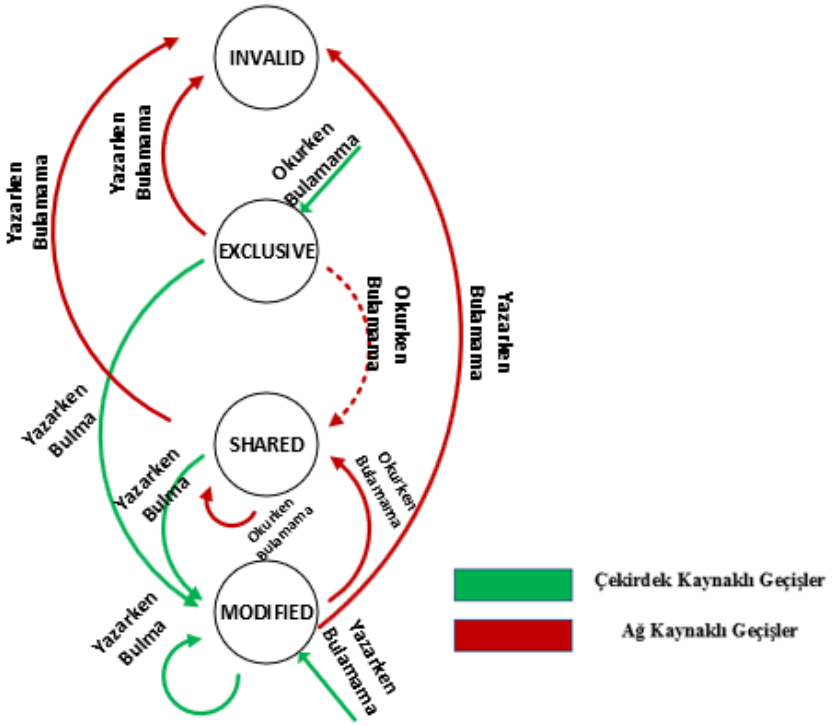
- **Okurken Bulma:** Okunmak istenen verinin, o çekirdeğin önbelleğinde bulunması durumudur. Bu durumda, okuma isteğini gönderen çekirdek veriyi kendi önbelleğinden okur. Verinin durumunda bir değişiklik yapılmasına gerek yoktur.

- **Yazarken Bulamama:** Üzerine yazılmak istenen verinin, yazma isteği gönderen çekirdeğin önbelleğinde bulunmaması durumudur. Bu durumda önbellek tutarlılığı açısından dikkat edilmesi gereken önemli durumlardan biri, erişilmek istenen verinin başka bir çekirdeğin önbelleğinde M durumunda olmasıdır. Bu durumda yazma isteği, veriyi M durumunda bulunduran önbellek veriyi fiziksel belleğe yazana kadar bekletilir. Veri daha sonra başka bir önbellek tarafından değiştirileceğinden bu önbellek, M durumunda bulunan veri öbeğini I durumuna geçirir.

Yazma isteği gönderen önbellek veriyi fiziksel bellekten okur ve M durumuna geçirir.Erişilmek istenen veri başka bir önbellekte M konumunda değilse, isteği gönderen önbellek veriyi fiziksel bellekten okur. Veri, daha önce başka bir önbellekte S ya da E durumunda bulunuyorsa bu önbellekler o veriye karşılık gelen öbeğin durumunu I'ya geçirir.

- **Yazarken Bulma(WRITE HIT):** Bir çekirdeğin, üzerine yazmak istediği veriyi kendi önbelleğinde bulması durumudur. Bu durumda veri öbeği M ya da E durumlarından birindeyse yazma işlemi tamamlanır, veri öbeği M durumuna çekilir. Veri öbeği S durumundaysa, yazma isteği ağ üzerinden diğer çekirdeklere bildirilir. Verinin bir kopyasını bulunduran önbellekler kendi kopyalarını I durumuna çekerler [19].

MESI protokolüne ilişkin durum makinesi Şekil 2.18'de verilmiştir. Bu protokolün Berkeley protokolünden farkı noktalı kırmızı okla gösterilmiştir [23]. Şekil 2.19 verinin iki önbellekte aynı anda bulunabileceği durumları belirtmektedir.



Şekil 2.18: MESI protokolüne ilişkin sonlu durum makinesi şeması

	M	E	S	I
M	×	×	×	✓
E	×	×	×	✓
S	×	×	✓	✓
I	✓	✓	✓	✓

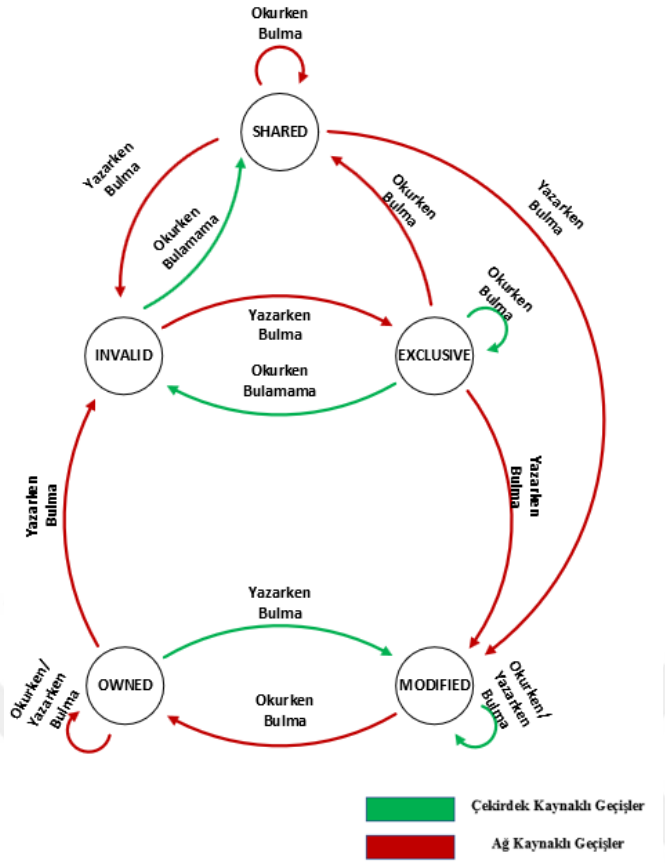
Şekil 2.19: Herhangi iki önbellekte aynı verinin bulunabileceği durumlar

2.4.2.5 M.O.E.S.I. Protokolü

1986'da Smith ve Sweazy tarafından ortaya atılan bu protokol, M.E.S.I. protokolüne Owned adı verilen beşinci bir durum eklenmesiyle oluşturulmuştur. M, E, S ve I durumları, M.E.S.I. protokolüyle benzer olup, O durumunun açıklaması aşağıda verilmiştir:

- **Owned:** Önbellekte bulunan veri geçerlidir, ancak ana bellektekiyle uyumlu olması gerekmez. Veriyi O durumunda bulunduran önbellek, veri bloğunu sistemin geri kalanına sağlamakla sorumludur. Her verinin aidiyeti ya sistemde yalnızca bir önbellekte ya da ana bellektedir [19, 31].

Bu protokolün sonlu durum makinesi Şekil 2.20'de verilmiştir [23]. Şekil 2.21 verinin iki önbellekte aynı anda bulunabileceği durumları göstermektedir.



Şekil 2.20: MOESI protokolüne ilişkin sonlu durum makinesi şeması

	M	O	E	S	I
M	×	×	×	×	✓
O	×	×	×	✓	✓
E	×	×	×	×	✓
S	×	✓	×	✓	✓
I	✓	✓	✓	✓	✓

Şekil 2.21: Herhangi iki önbellekte aynı verinin bulunabileceği durumlar.

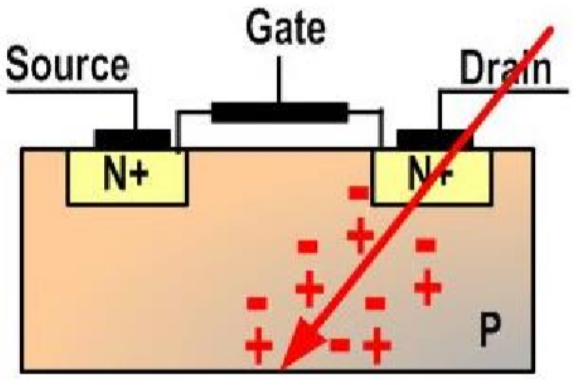
2.5 Güvenilirlik

Güvenilirlik, bir sistemin çalışmaya devam ettiği sürece beklenen sonuçları beklenen şekilde oluşturup oluşturamayacağının ölçüsüdür. Bir sistemin güvenilirliği çalışma sürecinde oluşacak hata sayısına, türüne ve bu hataların düzeltilip düzeltilemeyeceğine

bağlıdır. Donanım güvenilirliği için oluşabilecek hatalar kalıcı ve geçici hatalar olarak iki ana başlık altında incelenebilir. Kalıcı hatalar donanımın doğru şekilde çalışmasını etkileyecek fiziksel bir sorunun varlığını göstermektedir. Bu hataların çözümü, literatürde farklı yollar gösterilmiş olsa da, temelinde yanlış çalışan donanımı devre dışı bırakarak yedek bir donanım kullanmak olarak özetlenebilir [32]. Geçici hatalar ise donanımın fiziksel olarak bozulduğunu değil çalışma sırasında farklı etkiler sebebiyle bellekte saklanan verinin değişmesi gibi hataları işaret eder. Bu hatalar sistemin çalışmasına her zaman yansımadığı için fark edilmesi ve düzeltilmesi literatürde bir araştırma konusudur. Tez kapsamında geçici hatalar ve çözüm yolları ele alınacakır.

2.5.1 Geçici hatalar

Geçici hatalar, kozmik ışımalar ya da elektromanyetik çeşitli etkilerle oluşan ve sistem durumlarını çeşitli derecelerde etkileyen ve rastgele gelişen hatalardır (Şekil 2.22). Şekil içerisinde kırmızı ok, yüklü parçacıkları teslim etmektedir. Kalıcı olmadıklarından bu hatalar sistem yeniden başlatıldığında düzeltilebilir. Ancak; donanım çalışırken yapılan hatalar işleyişi ve doğruluğu bozabilmektedir. Oluşan hatalar nerede ve ilgili verinin hangi bölgesinde olduğuna üç farklı durum bulunmaktadır. Maskelenen hatalar, etkilediği hafıza bölgesinde işleyişi ve doğruluğu etkilemeyen hatalardır. Örneğin bir "ve" kapısına giren iki girişten birisi hatalı veri ile mantık-0 ise bu hata önemli olmadan maskelenmiş olabilir. Hafıza kaynaklarında oluşan hatalar sistem çöküşüne sebep olabilmektedir ve bu duruma kalıcı hata denir. Kalıcılık donanım hatası ile değil sistemde yarattığı etki ilgilidir. Sessiz veri bozulması ise oluşan hatalar sebebiyle etkilenen verilerin çalışma sürecini bozmaması, ancak veriler hatalı olduğu için sonuçların olması gerektiği gibi oluşmamasıdır. Bu çeşit hatalar çalışmayı etkilemediği için fark edilmesi zor hatalardır [33, 34, 35, 36].



Şekil 2.22: Transistörde geçici hata oluşması

Geçici hataların etkileri düşünüldüğünde hafıza birimlerinde bu hatalardan korunmak

gerekliliği açıktır. Bunun için en çok kullanılan yöntem eşlik biti ekleme, üç kopyalı çalışma (TMR) ve hata düzeltme kodlarıdır (HDK). Eşlik bitleri tek bit içerisinde oluşan hataları fark edebilirken düzeltmeye olanak sağlamaz. Bu sebeple hata fark edildiğinde sistem güvenli bir duruma geri alınır. TMR hata fark etme ve düzeltme için aynı işlemi aynı veriler üzerinde üç kere uygular (paralel ya da sıralı) ve sonuçları oylama sistemi ile karşılaştırır. Bu metot etkili olarak çalışsa da kapladığı ekstra alan ve harcadığı işlem gücü sebebiyle sıkça tercih edilen bir yöntem değildir. HDK metodu var olan verilere ekstra bilgi içeren bitlerin eklenmesiyle hafıza birimlerini koruma altına alır. Veriler belirli bir kodlama yöntemi kullanılarak hata tespiti ve düzeltimi için etkili ve az yer kaplayacak şekilde kodlanır. Veriler kullanılacağı zaman (okuma yapılacağında) verinin yanında bulunan HDK çözümlenerek veri ile karşılaştırılarak hata için sorgulama yapılır. HDK günümüzde hafıza birimlerinde sıkça kullanılan bir yöntem olsa da kapladığı ek alan ve kodlama-çözme aşamasının işlemlere eklediği ek zamanlar sebebiyle performansı düşürmektedir [33, 34, 35, 36].

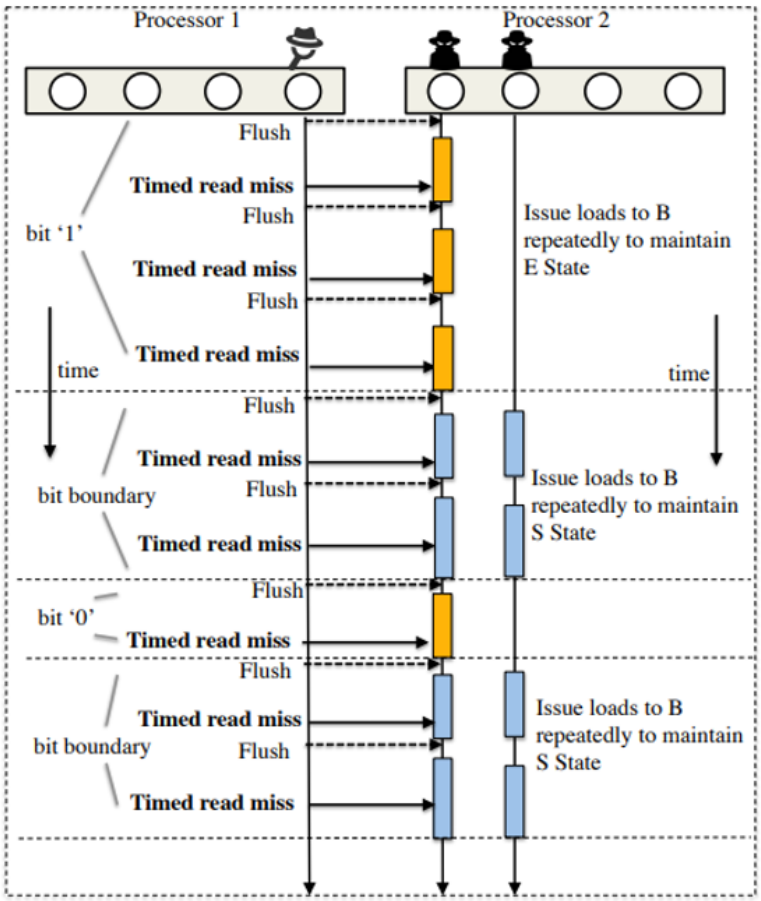
Günümüzde önbellek hiyerarşisi içerisinde, özellikle Intel mimarilerinde, veriler HDK kullanılarak korunmaktadır. Ancak; birinci seviye önbellekler boru hattı ile bütünleşik olduğundan bu aşamadaki verilerin HDK ile korunması performansı oldukça kötü etkilemektedir. Bu sebeple birinci seviye önbelleklerde HDK kullanmak bir opsiyon olarak sunulmaktadır ve varsayılan koruma yöntemi eşlik bitleridir [3].

2.6 Donanım Güvenliği

Donanım güvenliği son dönemlerde özellikle Spectre ve Meltdown ataklarının keşfedilmesi sonrasında üzerinde sıkça durulan bir konudur. Temelinde güvenlik açıkları, çekirdekler ve iş parçacıkları arasında paylaşılan kaynaklar sebebiyle oluşmaktadır. Paylaşılan kaynaklar var olan veriyolları dışında yollarla bilgi aktarımı sağlayabilmektedir. Bu şekilde oluşturulan yollara yan kanal denir. Yan kanalların iki türü bulunmaktadır. İlki kanalın iki tarafının da iletişimden haberdar olduğu ve iletişimde aktif bir rol aldığını (ing: covert channel), ikincisi bir verinin yalnızca tek taraflı aktarıldığını ve iletişimi yalnızca bir tarafıın aktif olarak oluşturduğunu (ing: side channel) ifade eder. Tezde yan kanal terimi ilk tanımı karşılayacak şekilde kullanılmaktadır.

Ana kanallar sistemlerde monitör edilebilirken bu kanallar kontrol edilmediğinden ciddi bir güvenlik açığına işaret eder. Bu yollar donanımdaki çeşitli optimizasyonların kötüye kullanılması ile ortaya çıkmaktadır [6, 5]. Çalışmamızda, önbellek açıklarına ve özellikle zamanlama yan kanallarına odaklanacağız. Bu yan kanal yapılan işlerin donanıma göre farklı sürelerde tamamlanmasıyla mümkün olmaktadır. Örneğin; bir bulut sisteminde erişilen donanım- daki önbellek boyutu doğrudan öğrenilemezken, belleğe sıralı erişim yapan bir kod çalıştırılıp erişimlerin aldığı süre ölçüldüğünde, süre

önbellekte bulma durumlarında kısa olacakken bulamama durumunda fark edilecek boyutta uzun olacaktır. Bu şekilde yapılan erişimler sonunda erişimlerin sayılmasıyla verilerin boyutları göz önünde bulundurularak doğrudan ulaşılamayan önbellek boyutunu keşfetmek mümkündür.



Şekil 2.23: Tutarlılık etiketlerine erişim zamanı farkından yararlanarak seri haberleşme

Donanım güvenliği kapsamında [1]'da aktarılan saldırı modeli tez kapsamında ele alınmış ve zamanlama yan kanalları ile ilgili çalışmlalar bu örnek üzerinden değerlendirilmiştir. Makalede, MESI tutarlılık protokolü etiketlerinin erişim zamanına etkisini kullanarak bir atak oluşturmuştur. Bu atak için "kernel same page merging" (KSM) adı verilen bir optimizasyon kötüye kullanılmaktadır. KSM, farklı iş parçacıklarının sayfa tablosu satırlarında aynı fiziksel alana dönüşüm yapılıyorsa bu sayfaların birleştirilmesini sağlayarak ciddi bir performans artışı sağlamaktadır. Fakat bu kernelin çalışma mekanizmasını kötüye kullanacak bir kod yazarak kötü amaçlı iki işlemin (trojan-spy) aynı fiziksel adresleri paylaşması sağlanabilmektedir. Bu şekilde önbellek erişimlerinin aynı satırlara eşleşeceği garantilenmiş olur. Sonrasında, MESI protokolünde S ve M etiketlerine erişimin farklı sürelerde tamamlandığı göz önünde bulundurularak bahsedilen iki iş parçacığı arasında monitörlenemeyen bir bilgi akışı sağlanmıştır. Bu

erişimlerin belli bir şablonla olması ve bu erişimlerin hangi erişimler olduğunun sadece süre ölçülerek görülebilmesi bu iki iş parçacığının seri haberleşmesine olanak verir. Bu durum bulut sistemlerinde KSM kernelinin kapatılarak performans kaybına rağmen güvenlik tehdidinin engellenmesini gerektirecek kadar önemli bir güvenlik açığıdır. Atak için gerçekleşen seri haberleşme yöntemi Şekil 2.23'te gösterilmektedir.

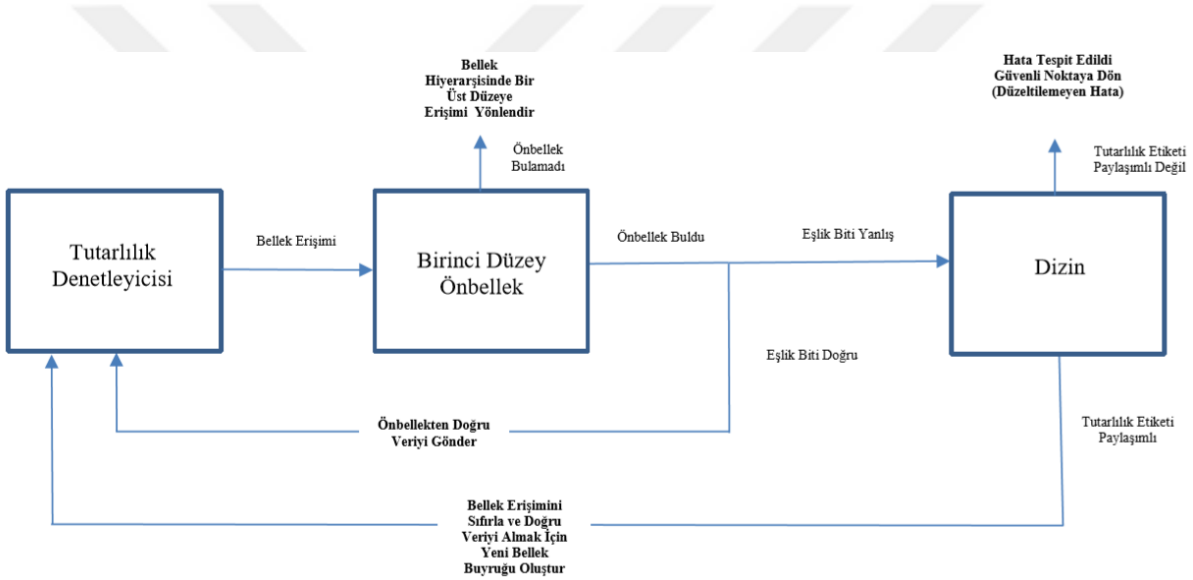
3. METODOLOJİ

Tez kapsamında, bu önbelleklerde oluşan geçici hataların düzeltimi için, önbellek tutarlılık protokollerinin sağladığı altyapı aracılığı ile daha ucuz ve etkili yeni bir mekanizma önerilmektedir. Diğer araştırma konusu ise donanım güvenliğidir ve bu alanda tutarlılık protokollerinin çalışma mekanizmalarının kötüye kullanıldığı durumlar incelenmiş ve bunları engelleyecek yeni mekanizmalar önerilmiştir.

3.1 Tutarlılık Etiketleri Kullanılarak Hata Düzeltme

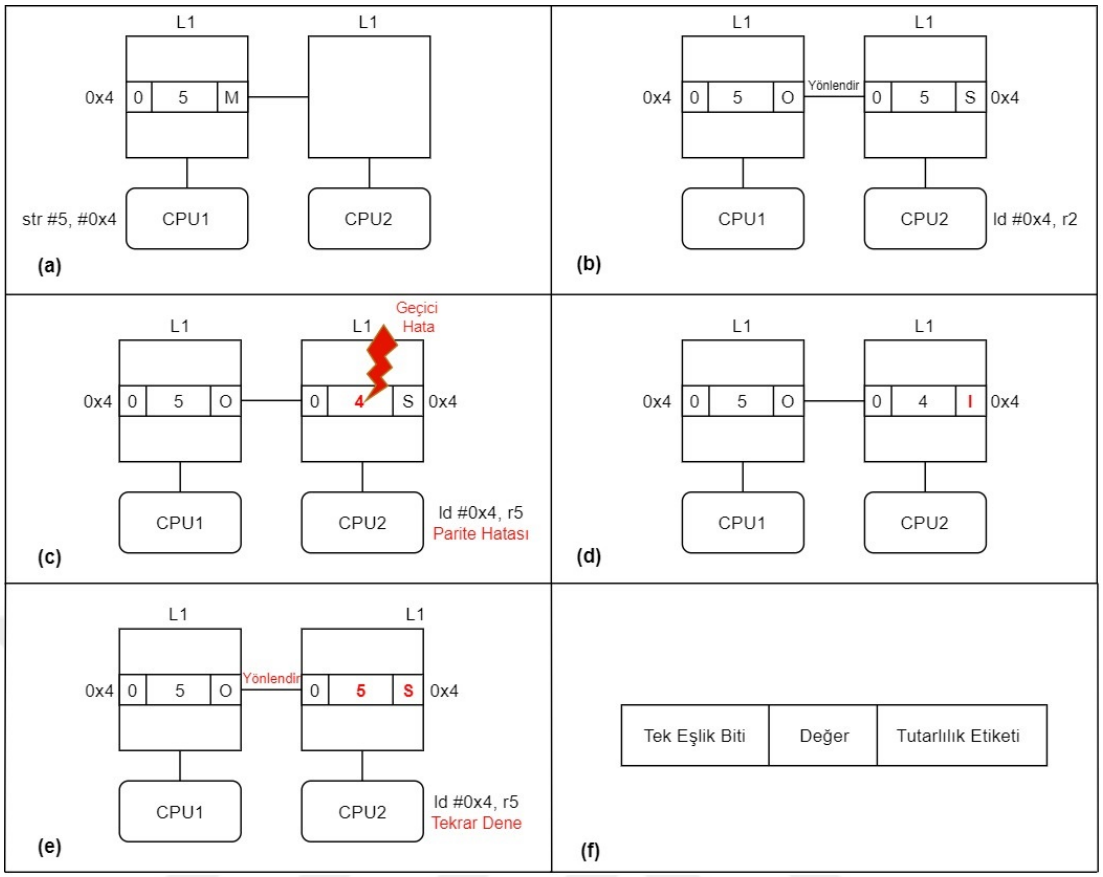
Donanımda hata düzeltme bilgi tekrarına dayanmaktadır. Oluşan bir hatayı düzeltebilmek için bu veri ile ilgili ek bilgilerin donanımda saklanıyor olması gerekmektedir. Tutarlılık protokolleri incelendiğinde, çok çekirdekli sistemlerde her çekirdeğin özel önbelleklerinde diğer çekirdeklerin özel önbellekler ile aynı verilerin bulunduğu gözlemlenmektedir. MESI protokolünü ele alırsak, “S” harfi ile gösterilen paylaşımlı olarak etiketlenmiş veriler diğer çekirdeklerde aynı verinin aynı şekilde üzerinde bir değişiklik olmadan bulunduğunu göstermektedir. MOESI protokolü için "S" ile etiketli bir verinin başka bir kopyasının ya "O" ya da "S" durumunda başka bir önbellekte daha bulunduğu bilinmektedir. MESI protokolüne ek olarak "O" durumunun bulunması, gelecek bir yükleme buyruğunun, eğer "O" etiketli bir adresten yükleme yapılıyorsa, hangi çekirdeğin önbelleğinden yönlendireceğini göstermektedir. MESI protokolünde de benzer bir yönlendirme kullanılabilmesine karşın tüm çekirdeklere bu yükleme buyruğu ile ilgili bilgi ulaştırılacağından yönlendirme daha maliyetlidir. Bu yönlendirmenin yapılabilmesi için, arama tabanlı protokollerde değiştirilmiş verilerin ve protokol yayınlarının çekirdekler arasında aktarılması için bir veri yolu bulunmaktadır. Dizin tabanlı protokollerde ise bu veri aktarımını bir üst seviye önbelleklerde ayarlayan bir dizin mevcuttur. Bu sebeple, var olan bu veri kopyaları hata düzeltmekte kullanılmaya elverişlidir. Önbelleklerde, özellikle birinci seviye önbelleklerde, ECC için harcanılan ekstra zaman ve yer performans açısında oldukça kritiktir. ECC yerine daha etkili bir mekanizma kullanılması performansı önemli ölçüde etkileyecektir. Bunun için “S” ile etiketli veriler için eşlik biti mekanizması kullanmak daha faydalı olabilir. Önerilen metot, “S” ile etiketli verilerde ECC kodlama ve kod çöz mesi yerine sadece eşlik biti kontrolünün yer almasıyla zamandan tasarruf sağlayacaktır. Bir hata fark edildiğinde eşlik kontrolü bu hatayı fark edecek ve çekirdeği uyaracaktır. Eklenecek bir mekanizma eğer bu eşlik kontrolü hatayı anlarsa bu veriye sahip diğer çekirdeklerden verinin doğru halini talep edecek ve bahsedilen veri yolları kullanılarak verinin doğru hali önbelleklere

yüklenerek işlem kaldığı yerden devam edecektir. 3.1 bu algoritmanın akış diyagramını göstermektedir. Arama tabanlı protokoller için dizin yerine çekirdekler kendi önbelleklerindeki verileri kontrol eder ve “S” etiketi bulunuyorsa diğer çekirdeklere veri isteklerini yayınlarlar. Buradaki performans kazancı var olan veri yolunun kullanılmasının ECC kodlama-kod çözme zamanından daha düşük olmasına bağlıdır. Kapsayan önbellek yapısında (alt seviyeler üst seviyelerin alt kümesi olduğu durum) bir üst düzey bellek hiyerarşisinde aynı verinin bir kopyasının olduğu bilinmektedir, ancak bu seviye önbelleklerden tekrar yükleme yapmak bu yollardan daha uzun zaman alacaktır. Benzer şekilde [37], her hata için tekrar yükleme yapılmasını önermiştir, fakat bu durumda performans açısından kayıplar yaşanacaktır. Bununla beraber, eğer hata olan önbellek bloğu eğer "S" etiketinde değilse bu şema uygulanarak üst önbelleklerden yükleme yapılması düşünülebilir.



Şekil 3.1: Önerilen hata düzeltme algoritması ve veri akışı

Önerilen bu yapının çalışma mekanizması örnek üzerinden şekil 3.2’te açıklanmıştır. Bu örnek incelendiğinde MOESI tutarlılık protokolü kullanan bir sistemde S ve O etkiketlerinin nasıl hata düzeltimi için kullanılabileceği görülmektedir. (a) ve (b) S ve O durumları için gerekli önkoşulları, (c) geçici hatanın etkisini, (d) ve (e) önerilen mekanizma ile hata düzeltmeyi göstermektedir. (f) önbellek bloğunun değişmiş yapısını içermektedir. (a), (b), (c), (d) ve (e) kronolojik olarak sıralıdır. Gösterilen şemada hata düzeltimi için O satırı kullanılmış olsa da aynı şekilde hata düzeltimi kullanılacak biçimde başka bir S etiketli satır da kullanılabilir. Bu da MOESI ile beraber MESI ya da MSI gibi protokollerin de aynı şekilde işleyeceğini ortaya koymaktadır.

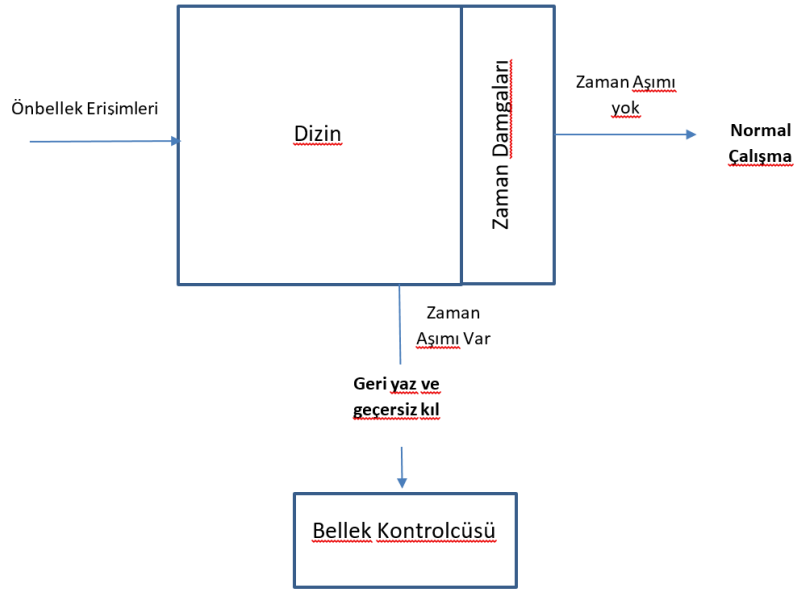


Şekil 3.2: (a) M etiketine geçiş, (b) S durumu önkoşulunu oluşturma (MOESI kullanıldığından O durumuna geçiş), (c) geçici hatanın oluşması ve parite kontrolü, (d) hata sonucunda geçersiz kılma, (e) yükle buyruğunun tekrar çalıştırıması ve veri yönlendirmesi, (f) önbellek satırlarının açıklaması

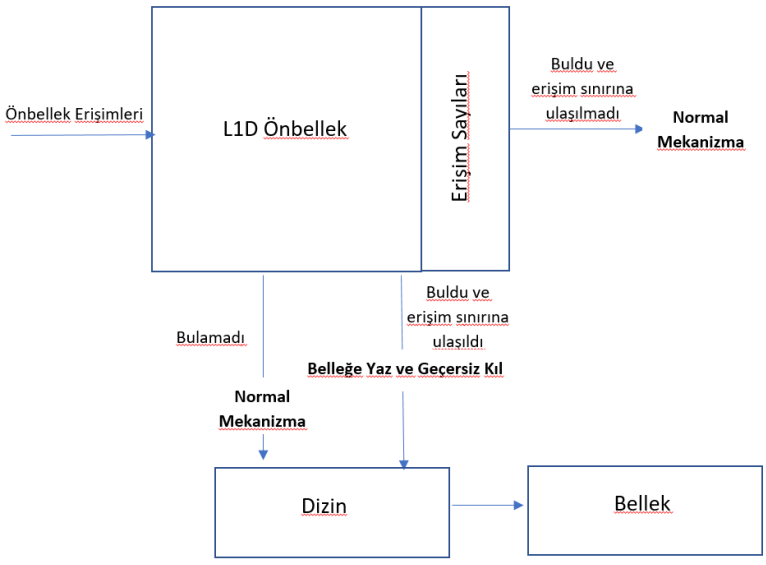
3.2 Önbelleklerde Zamanlama Yan Kanallarını Engelleme

Zamanlama tabanlı yan kanalları engelleyebilmek için erişim zamanlarının aynı kalması gerekmektedir. Fakat her erişim için bunu sağlamak performansı gereksiz bir şekilde düşürebilir. Bunun yerine donanıma eklenecek bir mekanizma ile tüm önbellek erişimleri monitörlenebilir ve atak tehdidi olarak görülen örüntülü erişimler tespit edilebilir ve buna göre önlem alınarak yan kanal oluşması engellenebilir. Tutarlılık tabanlı seri haberleşme sağlayan atak üzerinde çalışılacak olursa KSM ile aynı fiziksel adresler kullanılıp sürekli aynı adreslere erişim yapıldığı göz önünde bulundurulursa bu erişimleri takip etmek mümkündür. Özellikle dizin tabanlı tutarlılık protokolleri için her dizin satırına zaman aşımı için ekstra bilgi konulması bu problemi çözebilir. Bu şekilde normal çalışma süreci dışında kötü amaçlı önbellek erişimleri tespit edilebilir. Belirtilen atak "S" ve "E" etiketleri için çalıştığından, zaman aşımı sağlandığında ilgili adreslerde geri yazıp etiketi "I" olarak güncellemek doğruluk açısından bir götürüsü olmadan erişim zamanlarının ölçülebilirliğini ortadan kaldıracaktır. Bu ölçülebilirliği

ortadan kaldırmak üzere iki mekanizma önerilmiştir. Dizin üzerinden veya birinci düzey önbellekler üzerinden bu algoritmayı gerçeklemek mümkündür. Mekanizmanın dizin üzerinden akışı Şekil 3.3 'te, birinci düzey önbellek üzerinden akışı Şekil 3.4'te gösterilmektedir.



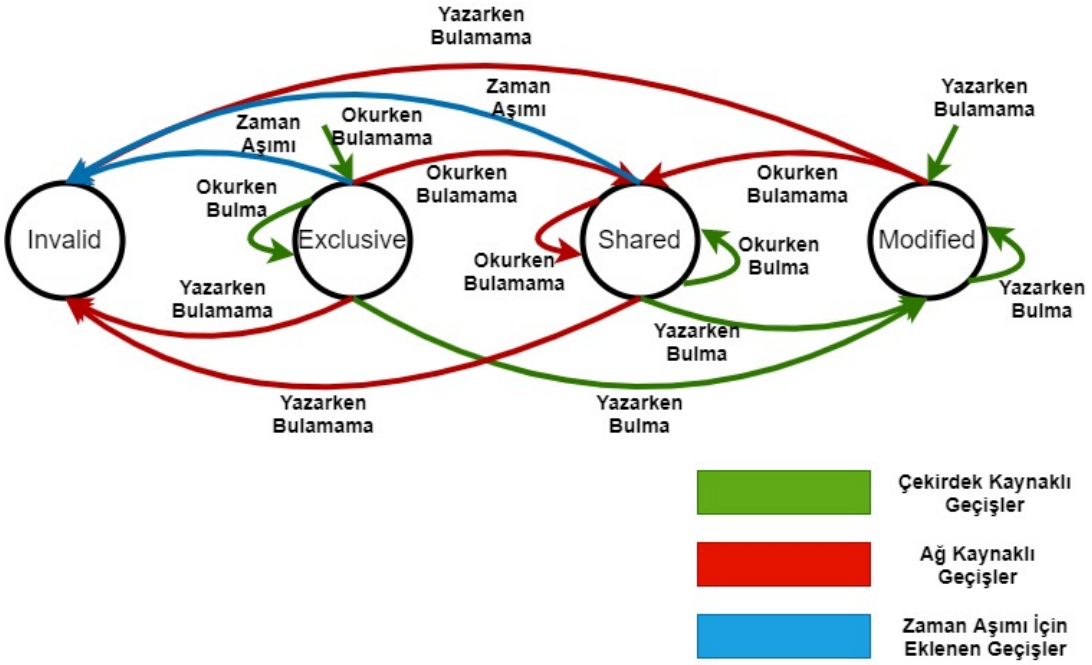
Şekil 3.3: Zamanlama yan kanalını engellme şeması (dizin üzerinden)



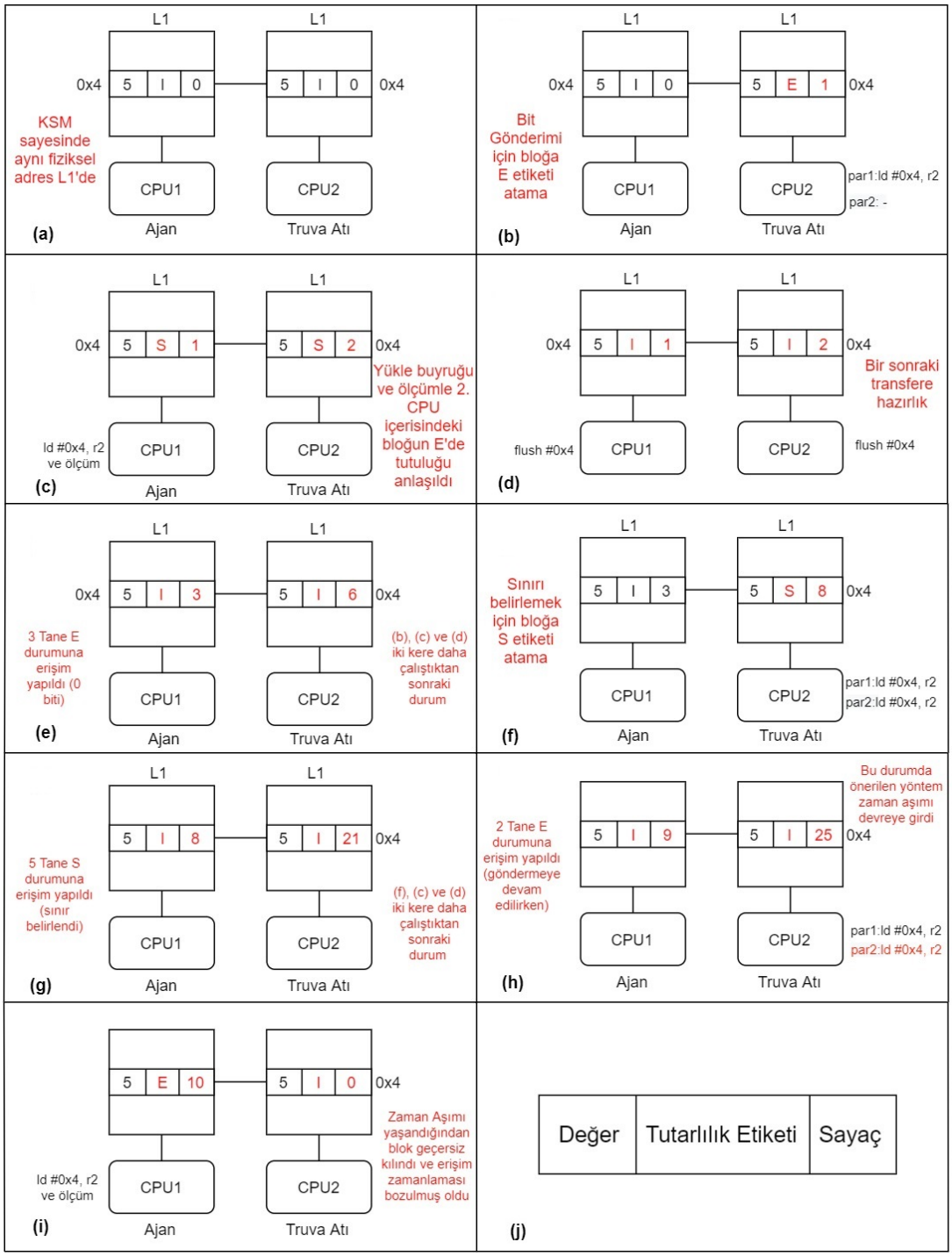
Şekil 3.4: Zamanlama yan kanalını engellme şeması(birinci düzey önbellek üzerinden)

Önerilen mekanizma kullanıldığında Şekil 3.5'te gösterildiği gibi MESI protokolünün durum makinesi değişmektedir. Şekilde mavi oklarla gösterilen geçişler eklenerek zaman aşımı durumları değerlendirilmektedir. Zaman aşımı için gerekli sayaçların

değişimi ise şekil 3.6'te örneklenerek anlatılmıştır. Örnekte 0 biti 3 E erişimi ile, sınır 5 S erişimi ile kontrol edilmektedir. (a) [1]'un önerdiği şekilde KSM kullanılarak yapılan kalibrasyonu göstermektedir. Görsel olarak anlaşılmayı kolaylaştırmak adına zaman aşımı için sınır 25 erişim olarak belirlenmiştir. Bu aşamadan sonra bir bit (3 adet E erişim ölçümü) göndermek için truva atı ve ajan uygulamalarının çalışması (b), (c) ve (d)'de gösterilmiştir. (b)'de iki iş parçacıklı truva atı uygulamasının E durumuna geçiş için yaptığı erişim ve sayaç güncellemelerini, (c) ajanın ölçüm için yaptığı erişimi ve durum değişikliklerini, (d) ise bir ölçüm sonunda yeni ölçümden önce yapılan değişiklikleri göstermektedir. Burada dikkat edilmesi gereken önemli nokta sayaç değişimlerinin arka arkaya gelen E veya S etiketlerinin takibini yapması ve flush durumlarında sayacın sıfırlanmamasıdır. (e) bir bit gönderimi sonunda oluşacak durumu göstermektedir. (f) truva atının S etiketini oluşturmak için yaptığı erişimleri ve değişiklikleri, (g) sınır sonundaki durumu ortaya koymaktadır. (h)'de zaman aşımına ulaşılan ilk nokta görülmektedir. (i) zaman aşımından sonra oluşacak tutarlılık etiketi değişimlerini ve erişim için ek sürelerin kaynağını içermektedir. Burada önerilen yapının ajan için bir değişiklik yaratmayacağı görülmektedir. Ajanın önbellek satırı 25 olduktan sonra sabit kalarak çalışmaya devam edecektir. Bu durumun sebebi durum değişikliği için E veya S durumuna bir yükleme gelmesinin kontrolüdür. Ajanda ise yükle buyrukları her zaman I etiketli adreslere yapılmaktadır. Son olarak (j) belirtilen yapıdaki önbellek satırlarının düzenini göstermektedir.



Şekil 3.5: Güncellenmiş MESI durum makinesi



Şekil 3.6: (a) KSM çalıştıktan sonraki durum, (b) truva atı uygulamasının E etiketi için erişimleri, (c) ajanın erişim ve ölçümü, (d) bir ölçüm sonu ve diğer ölçüm için hazırlık, (e) 3 E ile 0 biti gönderildikten sonraki durum, (f) truva atı uygulamasının S etiketi için erişimleri, (g) sınır için yapılan erişimlerin sonu, (h) zaman aşımı durumunun fark edilmesi, (i) zaman aşımı çalıştıktan sonraki durum ve ekstra gecikmenin sebebi, (j) önbellek satırlarının yapısı

4. DENEYLER VE SONUÇLAR

4.1 Deney Düzeni

Tez çalışmaları kapsamında arama temelli protokoller ile dizin tabanlı protokollerin performans ve iletişim trafiğinin analizi yapılmıştır. Bu analiz sonucunda görülmek istenen, yapılacak deneylerde kullanılacak sistemlerde hangi protokolün optimal sonucu vereceğini görmek ve bu protokollerin davranışlarını çekirdek sayılarına göre karakterize etmektir. Analiz için kullanılacak test yazılımları ve çalıştırma biçimleri için çalışmalar yürütülmüştür. Test yazılımları için yapılan makale taramaları sonucunda SPLASH2 ve PARSEC yazılım bütünleri, çok çekirdekli mimarilerin efektifliğini ölçebilmek için hem bir arada çalıştırılabilen (SPLASH2) sunucu aplikasyonları hem de çok iş parçacıklı çalıştırılabilecek aplikasyon ihtiyaçlarını karşılaması sebebiyle uygun görüldü. Bu iki yazılım bütünü hakkında ayrıntılı bilgi Şekil 4.1'de gösterilmektedir [38]. Bununla beraber Splash2 ve PARSEC uygulamalarının bellek karakterizasyonları da hangi uygulamaların kullanılacağını ve beklenen sonuçların nasıl olması gerektiğini göstermektedir. Bu sebeple Şekil 4.2'de [38]'a ait yapılan çalışmalar gösterilmektedir. Bu sonuçlar göz önünde bulundurularak kullanılan test ortamı oluşturulmaya çalışılmıştır.

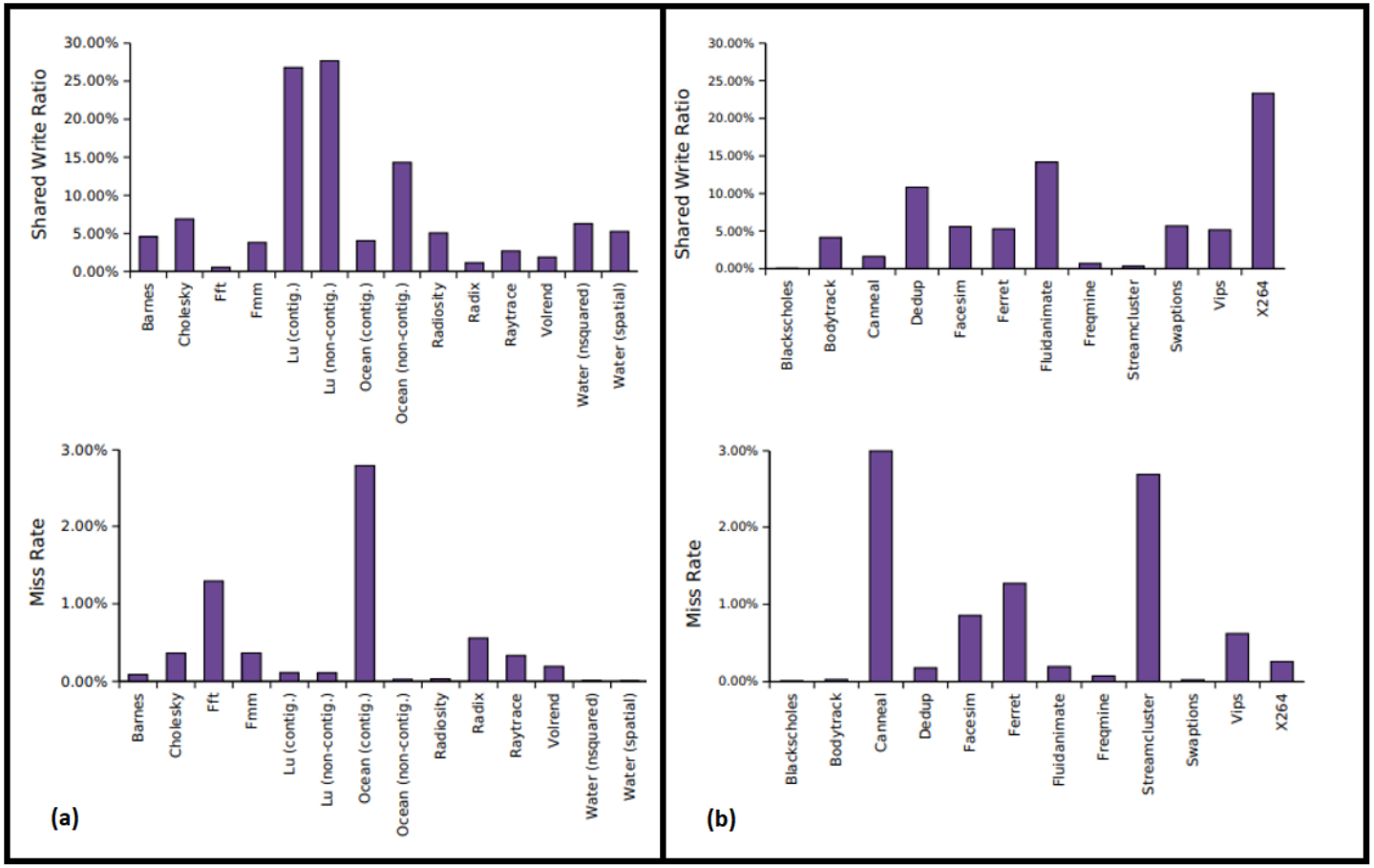
Program	Application Domain	Problem Size
barnes	High-Performance Computing	65,536 particles
cholesky	High-Performance Computing	tk29.O
fft	Signal Processing	4,194,304 data points
fmm	High-Performance Computing	65,536 particles
lu	High-Performance Computing	1024×1024 matrix, 64×64 blocks
ocean	High-Performance Computing	514×514 grid
radiosity	Graphics	large room
radix	General	8,388,608 integers
raytrace	Graphics	car
volrend	Graphics	head
water	High-Performance Computing	4096 molecules

(a)

Program	Application Domain	Problem Size
blackscholes	Financial Analysis	65,536 options
bodytrack	Computer Vision	4 frames, 4,000 particles
canneal	Engineering	400,000 elements
dedup	Enterprise Storage	184 MB data
facesim	Animation	1 frame, 372,126 tetrahedra
ferret	Similarity Search	256 queries, 34,973 images
fluidanimate	Animation	5 frames, 300,000 particles
freqmine	Data Mining	990,000 transactions
streamcluster	Data Mining	16,384 points per block, 1 block
swaptions	Financial Analysis	64 swaptions, 20,000 simulations
vips	Media Processing	1 image, 2662×5500 pixels
x264	Media Processing	128 frames, 640×360 pixels

(b)

Şekil 4.1: (a) Splash2 ve (b) PARSEC uygulamaları ve açıklamaları



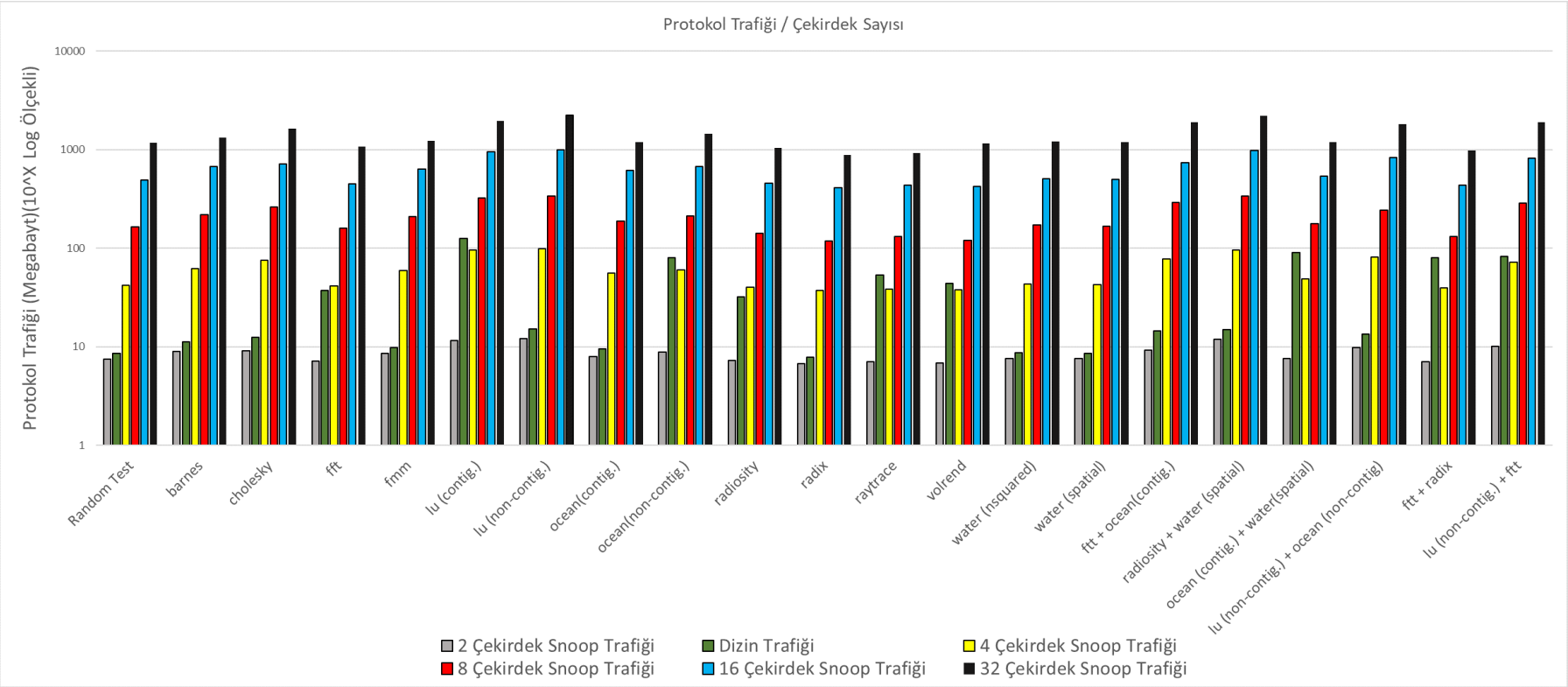
Şekil 4.2: Splash2(a) ve PARSEC(b) için önbellekte bulamama ve aynı adrese yazma analizi (64KB 4 Yollu 64 bayt satırlı L1 Önbellek ve 8 Çekirdek)

4.1.1 Hata düzeltimi için deney düzeni

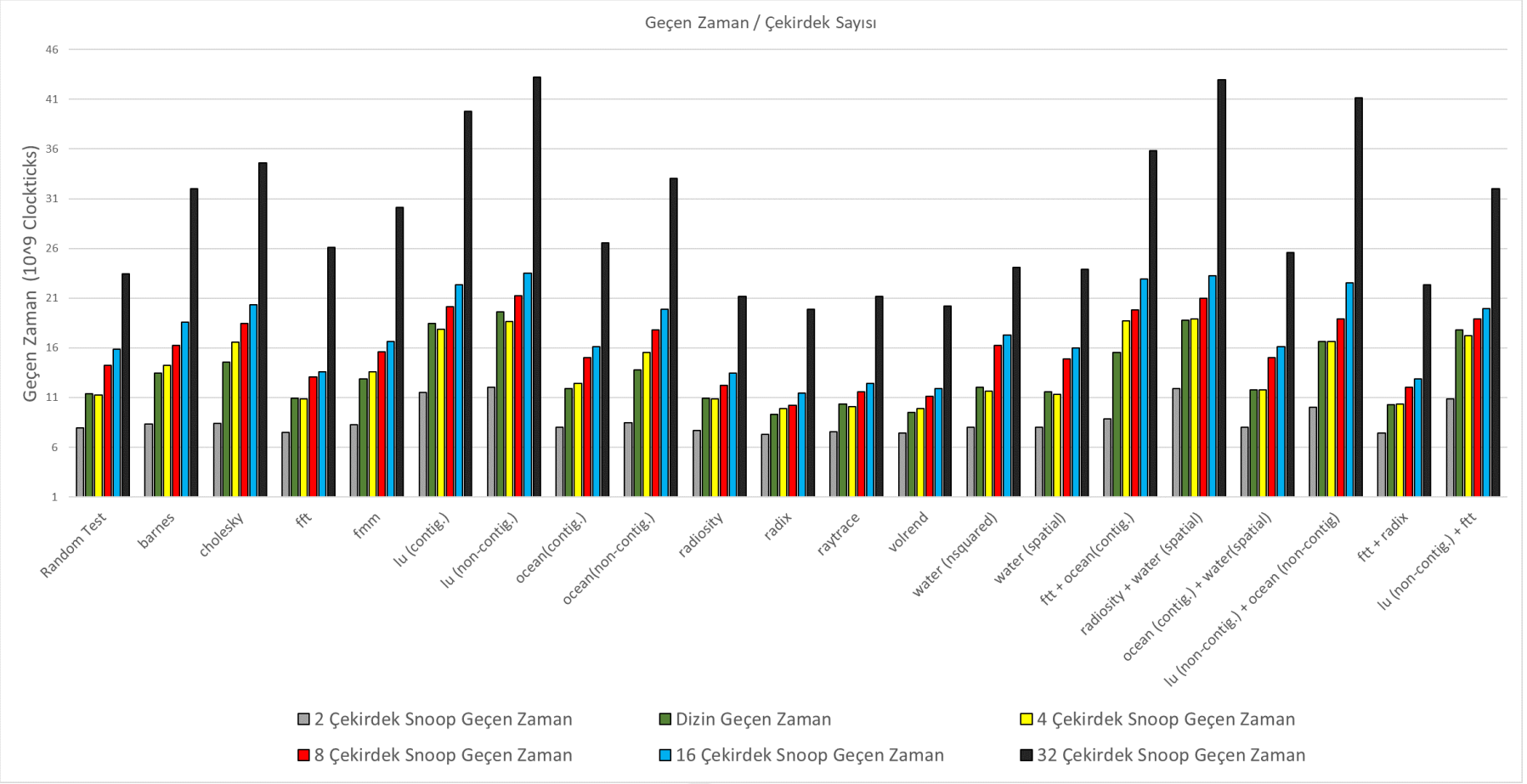
Hata düzeltimi deneyleri kapsamında öncelikle Splash2 kullanarak tutarlılık protokollerinin analizi yapıldı. Hangi uygulamaların nasıl kullanılacağı kararını Şekil 4.2'deki analize dayanarak verildi. Buna göre 2-32 çekirdek sayıları için tüm uygulamalar olabildiğince paralel (örneğin 32 çekirdek için 32 iş parçası); ftt ve ocean(contig.) (en yüksek iki bulamama oranı), radiosity ve water (spatial)(en düşük iki bulamama oranı), ocean (contig.) ve water(spatial) (en yüksek ve en düşük bulamama oranları), lu (non-contig.) ve ocean (non-contig) (en yüksek iki ortak yazma yapan farklı uygulamalar), ftt ve radix (en düşük iki ortak yazma oranları) ile lu (non-contig.) ve ftt(en yüksek ve en düşük ortak yazma oranları) kaynakları eşit paylaşacak şekilde (örneğin 32 çekirdek için 16 çekirdek bir uygulamaya diğer 16 çekirdek diğer uygulamaya ayrılacak şekilde) çalıştırılmıştır. Bu noktada tek çok iş parçacıklı uygulama ve aynı anda çalıştırılan iki uygulamanın karakterizasyon için yeterli olacağı düşünülmüştür. Şekil 4.3 ve 4.4'te bu uygulamalar için alınan mesajlaşma trafiği ve çalışma zamanı sonuçlarını görebilirsiniz. Analiz için Gem5 simülatörü "syscall emulation" modunda kullanıldı. Her çekirdeğin 32 KiB 4 yollu 1 seviye özel önbelleği olduğu varsayıldı. Tüm uygulamalar toplam 500000 yükleme yapana kadar çalıştırıldı.

4.1.2 Donanım güvenliği için deney düzeni

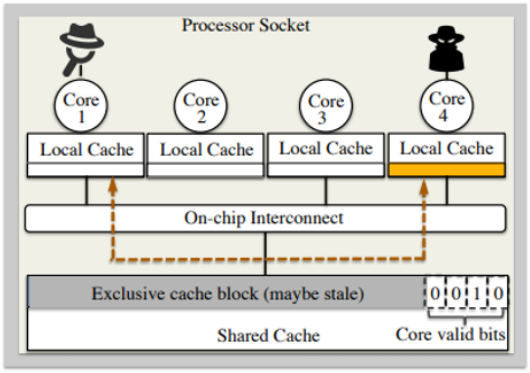
Yao et al.'ın belirttiği atak modeli [1] GEM5 simülatörü kullanılarak oluşturuldu. Bu atak modeline göre Şekil 4.5 'te gösterildiği gibi sistem üzerinde bir çekirdekte çalışan ajan ve bir ya da daha çok çekirdekte çalışabilen truva atı programları bulunmaktadır. KSM yardımıyla senkronize edilmiş ortak fiziksel paylaşımlı bir önbelleğe erişimli olan bu iki program arasında bölüm 3'te anlatılan şekilde iletişim sağlanmaktadır. Bu iletişim Şekil 4.6'da gösterildiği şekilde S ve E etiketlerinde olan adreslere ajan programın erişip zamanlama ölçümlerine göre bu iki satırın etiketlerini ayırt edebilmesine bağlıdır. Ayırt edilebilen tutarlılık etiketleri Şekil 4.7'deki akışa göre kullanılarak seri haberleşme için kullanılabilmektedir.



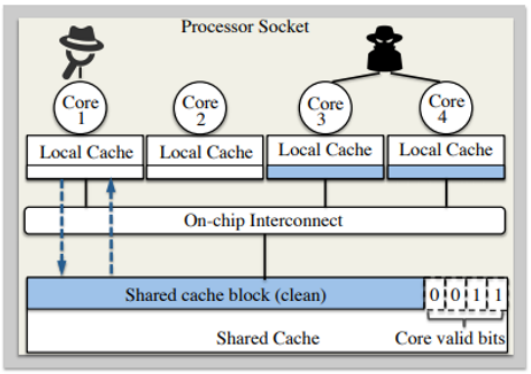
Şekil 4.3: Splash2 uygulamaları ve random test uygulamasının çekirdek sayısına göre protokol trafiği (log ölçekli)



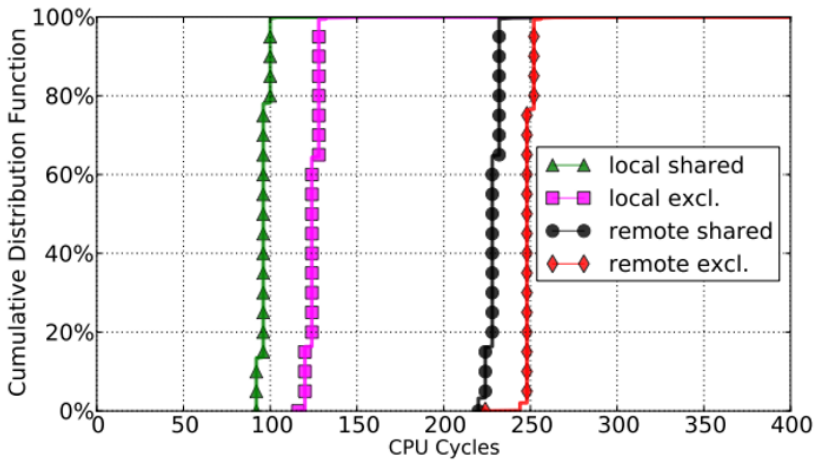
Şekil 4.4: Splash2 uygulamaları ve random test uygulamasının çekirdek sayısına göre geçen zaman (log ölçekli)



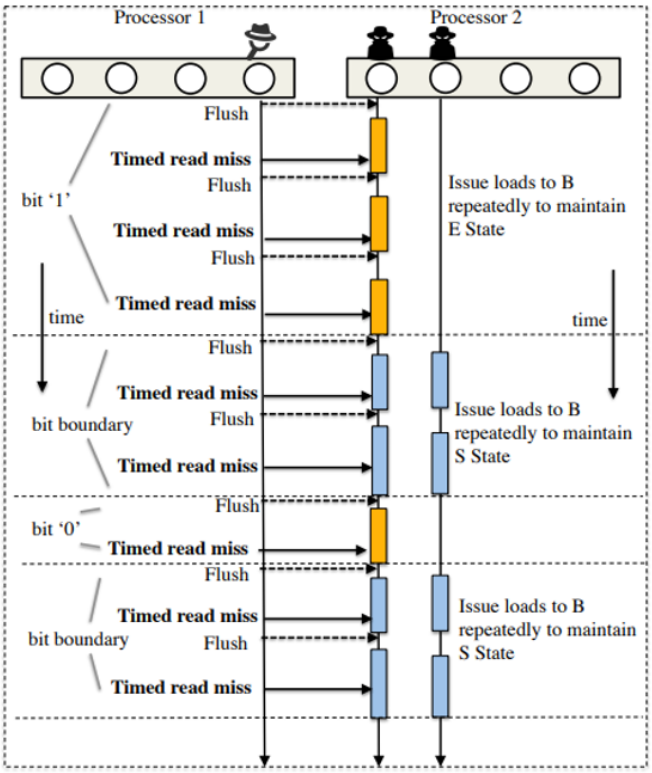
(a) Cache block in E state



Şekil 4.5: Ajan ve truva atının haberleşeceği altyapı ve S, E etiketlerine yapılan erişimlerin sistemde izlediği yol



Şekil 4.6: "S" ve "E" etiketlerine olan erşimler için geçen süre (çevrim olarak)



Şekil 4.7: "S" ve "E" etiketleri kullanılarak oluşturulan yan kanal üzerinden erişim

GEM5 simülatöründe yer alan MESI protokolünün "SLICC" ile tanımlanmış durum makinesi değiştirilerek Bölüm 3'te önerilen mekanizma gerçeklendi. Bunun için yazma ve okuma işlemleri ile beraber E ve S durumları için zaman aşımı da durum değişimini tetikleyecek şekilde güncellendi. E ve S durumlarında olan satırlara yapılan arka arkaya erişimler birinci düzey paylaşımlı önbelleklerde ilgili satırın zaman aşımı için kullanılan sayacını artıracak şekilde düzenlemeler yapıldı. Aynı zamanda bu satırlar önbellekten çıkarılırsa veya başka bir etikete sahip hale gelirse sayaç sıfırlandı.

Ajan ve truva atı uygulamalarının çalışmalarını simüle etmek için GEM5 içerisindeki "ruby_random_test.py" konfigürasyonu değiştirilerek rastgele adreslere erişimler yerine aynı adrese arka arkaya erişimler tanımlandı. Bu konfigürasyon ile truva atının çalışması simüle edildi. Erişimlerden sonra, ajan davranışını gözlemlemek için zaman ölçümleri yapılarak sonuçlar elde edildi. Çalışma ortamını devam etmekte olan bir sistem için gözlemleyebilmek için önbellekler truva atı kodu çalıştırılmadan önce rastgele erişimler ile ısındırıldı.

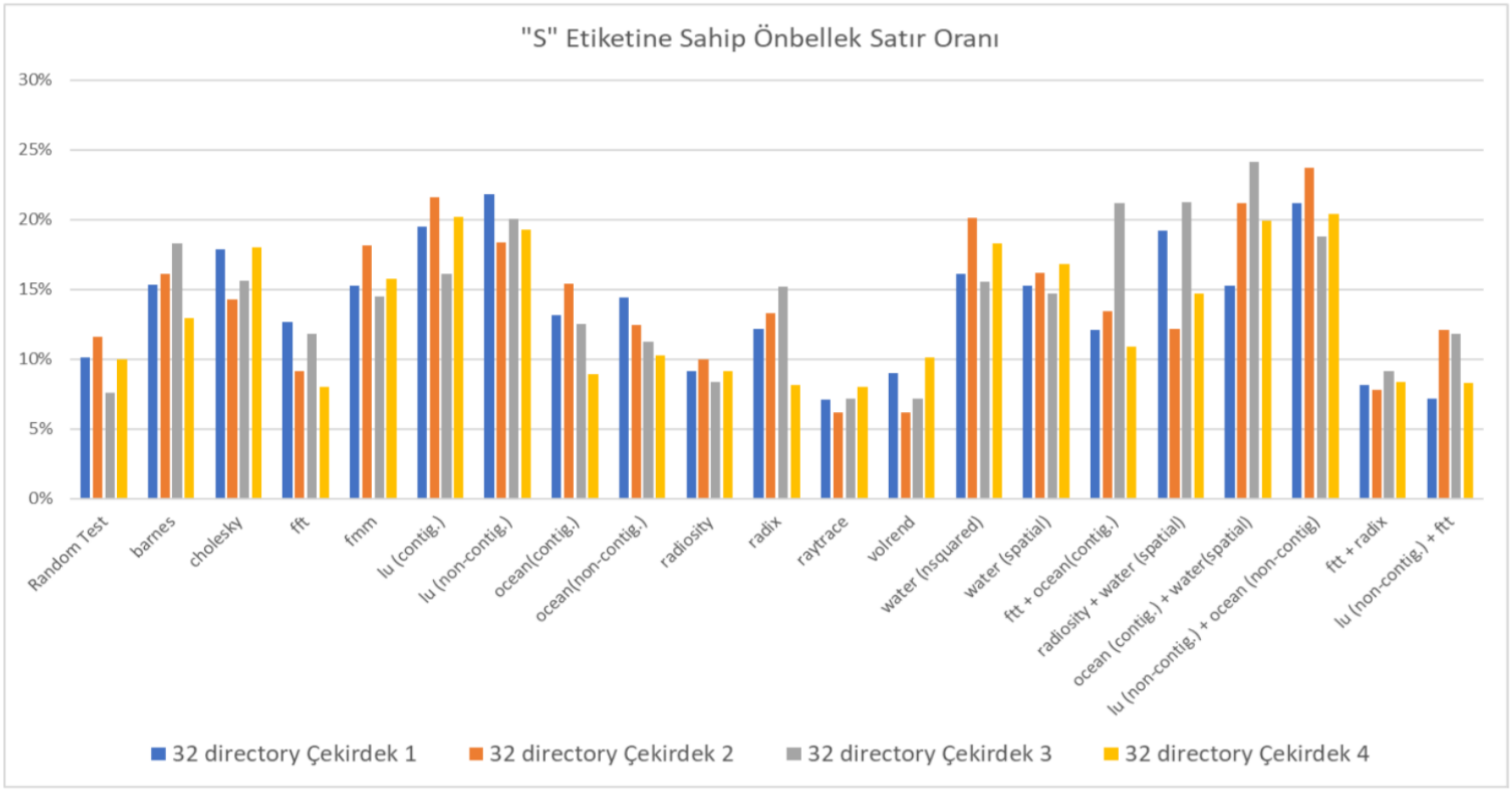
4.2 Sonuçlar

4.2.1 Hata düzeltim mekanizmasına ilişkin sonuçlar ve tartışma

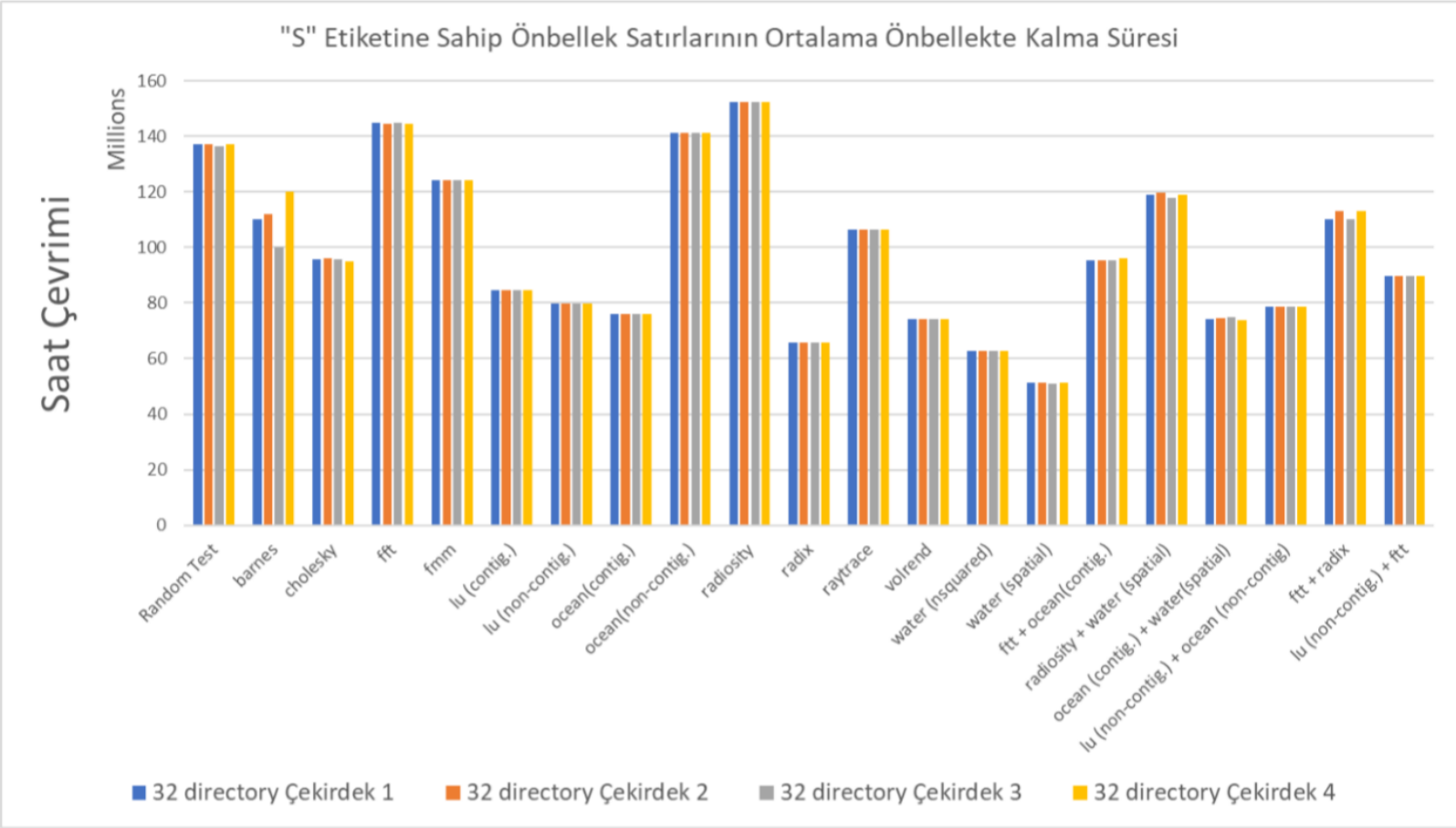
Belirtilen deney düzeni ile uygulamalarda paylaşımlı önbellek satırlarının tüm önbellek satırlarına oranı her bir çekirdek için bulundu. Bu oran, çekirdeklerde yer alan tüm önbellek satırlarının hata düzeltme için kullanılabilirliğini göstermektedir. Başka bir deyişle sonuçlardan, birden fazla kopyası bulunan verilerin tüm verilere oranı görülebilmektedir. Belirtilen uygulamaların Şekil 4.8'e göre tüm aplikasyonlarda bu oran her çekirdekte ortalama %15 olarak görülmektedir. Ancak bazı uygulamalar için bu oranın %22 civarına çıkabildiği gözlemlenmektedir. Buna göre eğer bu mekanizma uygun aplikasyonlar ile çalıştırılırsa, paylaşımlı önbellek satırlarının neredeyse dörtte birini korumak mümkündür. Ayrıca bu oran programların nasıl optimize edildiği ile doğrudan ilişkili olduğu için oranı artıracak şekilde yazılım geliştirmek mümkündür. İkinci olarak "S" etiketine sahip paylaşımlı önbellek satırlarının kaç saat çevrimi boyunca kullanabilir olduğunu gösteren bir deney yapıldı. Böylece hata düzeltme penceresinin ortalama kaç çevrim olduğu saptandı. Şekil 4.9'da tüm program ve program çiftlerinin paylaşımlı önbellek satırlarının kaç milyon saat çevrimi çekirdeklerin önbelleklerinde durduğu gösterilmiştir. Buna göre her bir satır ortalama 98 milyon çevrim boyunca ait olduğu önbellekte kalmaya devam etmektedir. Bu pencere akış diyagramındaki tüm işleri yapmak için gereken süreden oldukça fazladır. Yani; herhangi bir anda paylaşımlı önbellek satırlarının hata düzeltmeye hazır olmaması durumu neredeyse görülmeyecek düzeydedir.

Bu deneylerin dışında gerçek bir çalışma sırasında önerilen sistemin kapasitesini görebilmek için SPLASH-2 uygulamaları ve uygulama ikilileri çalıştırılırken rastgele zamanlarda her biri için beş bin geçici hata oluşturulmuştur. Deneyler sırasında şekil 4.8'de görülene uyumlu bir şekilde raytrace uygylamasında beş bin hatadan 281'ü(en düşük), ocean(contig.) ve water(spatial) uygulamaları birlikte çalıştırılırken beş bin hatadan 1426 (en yüksek) tanesi saptanabilmiştir. Toplama bakıldığında 20 adet yürütmede enjekte edilen yüz bin hatadan 18436 tanesi saptanabilmiştir.

Bu deneylerin dışında gerçek bir çalışma sırasında önerilen sistemin kapasitesini görebilmek için SPLASH-2 uygulamaları ve uygulama ikilileri çalıştırılırken rastgele zamanlarda her biri için beş bin geçici hata oluşturulmuştur. Deneyler sırasında şekil 4.8'de görülene uyumlu bir şekilde raytrace uygylamasında beş bin hatadan 281'ü(en düşük), ocean(contig.) ve water(spatial) uygulamaları birlikte çalıştırılırken beş bin hatadan 1426 (en yüksek) tanesi saptanabilmiştir. Toplama bakıldığında 20 adet yürütmede enjekte edilen yüz bin hatadan 18436 tanesi saptanabilmiştir.



Şekil 4.8: Splash2 uygulamaları için paylaşımlı önbellek satırlarının tüm satırlara oranı



Şekil 4.9: Paylaşımlı önbellek satırlarının çekirdeklerin birinci düzey önbelleklerinde kaldığı saat çevrimi sayısı

Çok çekirdekli sistemlerde, paylaşımlı birinci düzey önbelleklerde etkili bir şekilde kullanılabilecek bir yapı bulunmamaktadır. Var olan yapılar ECC'ye dayandığından yer ve zaman israfı oluşturmaktadır. Yapılan çalışmalar önbelleklerde görülen hata miktarını azaltmak için ovalama (cache scrubbing) kullanmata ya da ECC mekanizmalarının harcadığı enerjiyi ve düzeltme oranını optimize etmeye çalışmaktadır [39]. Bilinen çalışmalar göz önüne alındığında birinci düzey önbellekler için önerilen bu mekanizma alanında bir ilktir. Sonuçlar incelendiğinde

Splash2 programları için uygulamaların dörtte birinin bu mekanizma ile korunabileceği görülmüştür. Ayrıca bu oranı yazılımsal olarak artırmak ve daha çok veriyi koruyabilmek mümkündür. Bunun yanında, zaman analizi göz önünde bulundurulduğunda herhangi bir zaman aralığında bu mekanizmanın kullanılamayacak olmasının olasılığının çok az olduğu görülmektedir. Yani düzeltilebilecek bir veri bulma oranı yüksek ve kullanılamama zamanı oldukça düşüktür. Önerilen mekanizmanın ihtiyaç duyduğu iki işlem bulunmaktadır. Bu işlemler, eşlik biti kontrolü ve dizinde paylaşımlı veriler için yeni bellek erişimi oluşturmaktır. Eşlik biti var olan sistemlerde birinci düzey önbellekler için varsayılan özelliktir ve bu sebeple ekstra bir alana gerek duyulmamaktadır. Fakat eşlik biti kontrolünden sonra dizine gidilecek şekilde sistemde bir düzenleme yapılmalıdır. Bu düzenlemede dizinden dönen paylaşımlı önbellek satırı sorgusuna karşılık yeni bir bellek erişimi oluşturma işlemi kesme gibi çalışacak şekilde düzenlenebilir. Bunlar göz önüne alındığında, önerilen mekanizmanın ekstra gereksinimlerinin minimal düzeyde olduğu görülmektedir.

4.2.1.1 Gerçekleme

Önerilen hata düzeltme şemasıyla HDK kullanılması durumlarını karşılaştırmak için Verilog kullanılarak iki ayrı önbellek tasarımı FPGA üzerinde gerçeklendi. Temel önbellek tasarımı için Ariane [40] tasarımının birinci düzey veri önbelleği tek başına çalışacak şekilde konfigüre edildi. Kontrol grubu için HDK modeli olarak Reed-Solomon [41] hata düzeltme şeması önbellek tasarımına eklendi. Deney grubununun gerçeklemesi ise bölüm 4.1'de detaylandırıldığı gibi çalışacak şekilde önbellek tasarımı güncellenerek yapıldı. İki grup da Xilinx VCU108 model FPGA için sentezlendi ve sonuçlar elde edildi. Deneyler için rotalama veya yerleştirme için bir optimizasyon yapılmadı.

Önerilen algoritma HDK kullanan şemaya göre toplamda %42.93 daha az kaynak kullanımı oluşturmuştur. Bunun başlıca sebebi HDK için gerekli ekstra verilerin de saklanması için her önbellek satırında fazladan alan kullanımıdır. Sonuçlara göre bu fazlalık sadece eşlik biti bulundurmaya göre BRAM kullanımında %40.24 oranında artış ile göze çarpmaktadır. Diğer kaynakların kullanımındaki fazlalık ise HDK algoritmasının

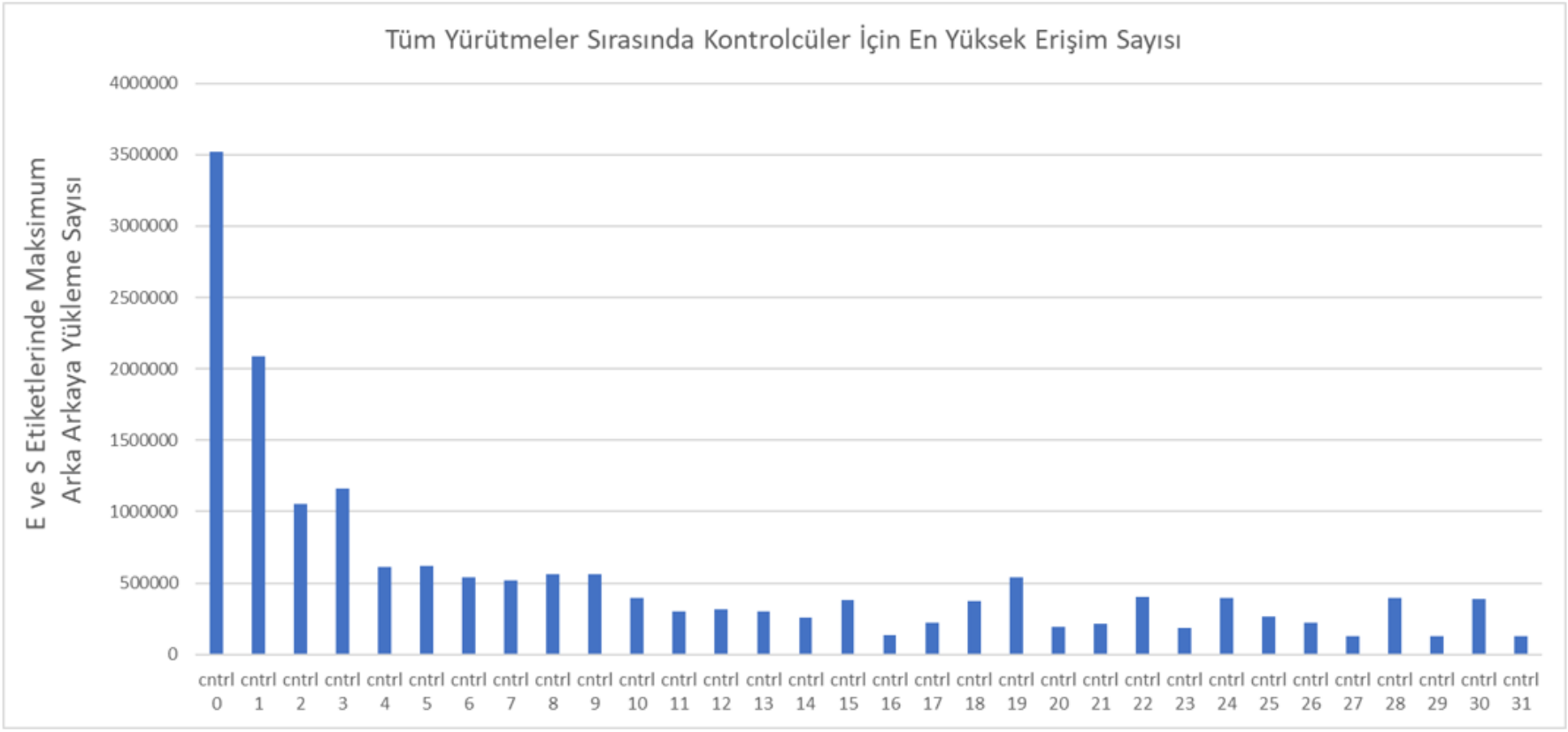
kodlama ve kod çözme adımlarının devresinin geliştirilen algoritmada tekrar yürütmeyi sağlama devresine göre daha karmaşık olmasından kaynaklanmaktadır.

İki tasarım arasında kritik yol analizi de yapılmıştır. Hata oluşmadığı durumda, normal çalışma koşulları altında, HDK kullanılan tasarımın kritik yol gecikmesi 4.819 nanosaniye olarak, önerilen metodun kritik yol gecikmesi 3.032 olarak ölçülmüştür. Buna göre önerilen algoritmanın kullanılmasıyla gecikme %37.06 düşmektedir. Bu durumda kontrol tasarımı yaklaşık 200MHz frekanslı bir saat ile uyumlu çalışabilecek iken önerilen tasarım yaklaşık 320MHz frekanslı bir saat ile kullanılabilmektedir. Bunun sebebi kodlama ve kod çözmenin eşlik biti kontrolünden önemli ölçüde fazla zaman almasıdır.

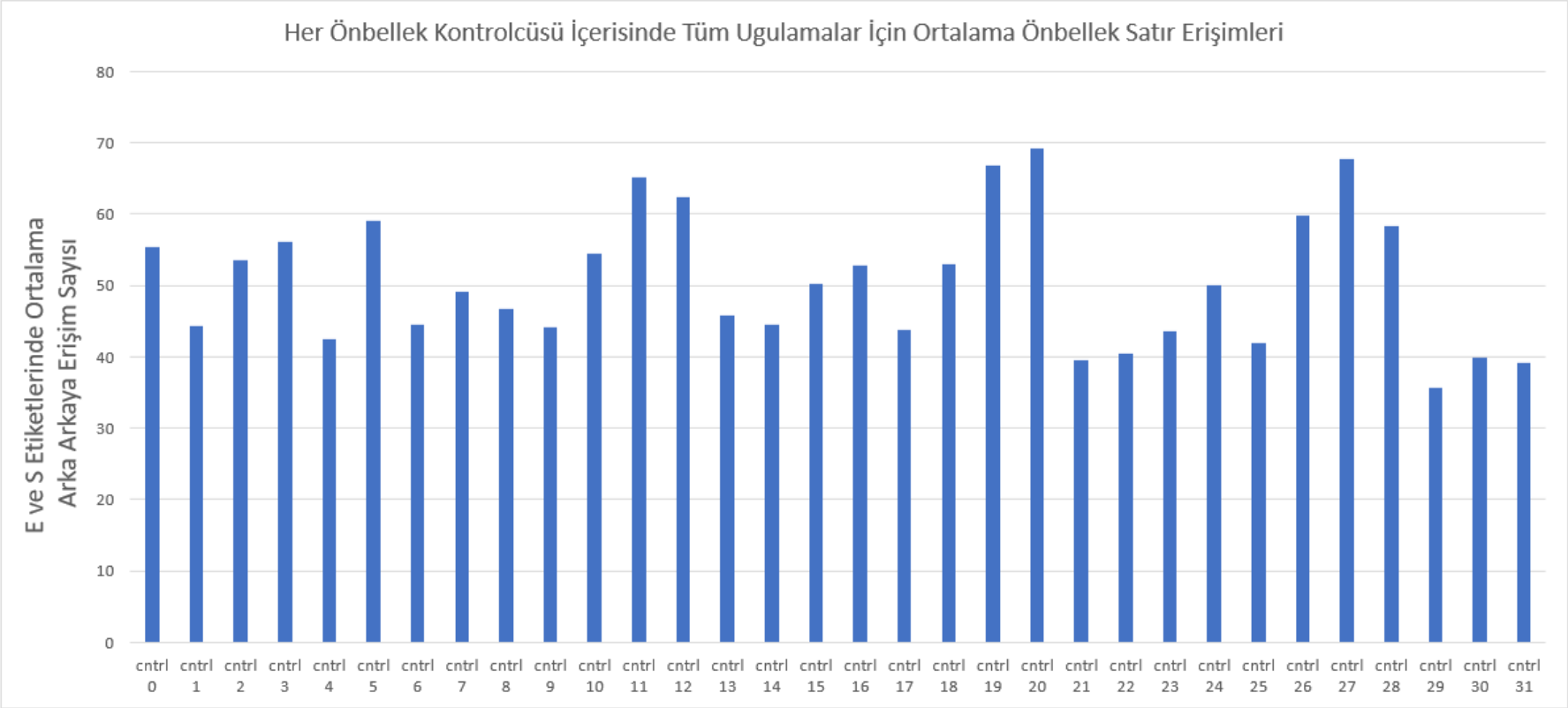
4.2.2 Güvenlik mekanizmasına ilişkin sonuçlar ve tartışma

Öncelikle SPLASH2 ve PARSEC program bütünlerinin uygulamaları profillendi. Bunun içn her uygulama için arka arkaya aynı adrese yapılan yükleme işlemlerine bakıldı. Bunun için belirtilen deney düzeninde 1, 2, 4, 8, 16 ve 32 iş parçacığıyla uygulamalar çalıştırılarak maksimum ve ortalama arka arkaya erişim sayıları elde edildi. Şekil 4.10, deneylerde tüm uygulamalar için elde edilen değerlerin maksimumunu, şekil 4.11 ortalamalarını göstermektedir. Her kontrolcünün işlediği arka arkaya yükle buyruğu sayılarının maksimum değerleri, zaman aşımı uygulandığında performansın ciddi bir şekilde kötü etkileneceğine işaret etmektedir. Öte yandan her kontrolcü için tüm erişimlerin ortalamasına bakıldığında bu değerlerin maksimum değerden onlarca kat az olduğu ve zaman aşımının performansı ortalama durumda düşürmeyeceği gözlemlenmiştir. Şekil 4.12 , en yüksek yüz erişimin uygulamaların çalışma süreci boyunca tüm erşimlerin en fazla %1'ini oluşturduğunu göstermektedir.

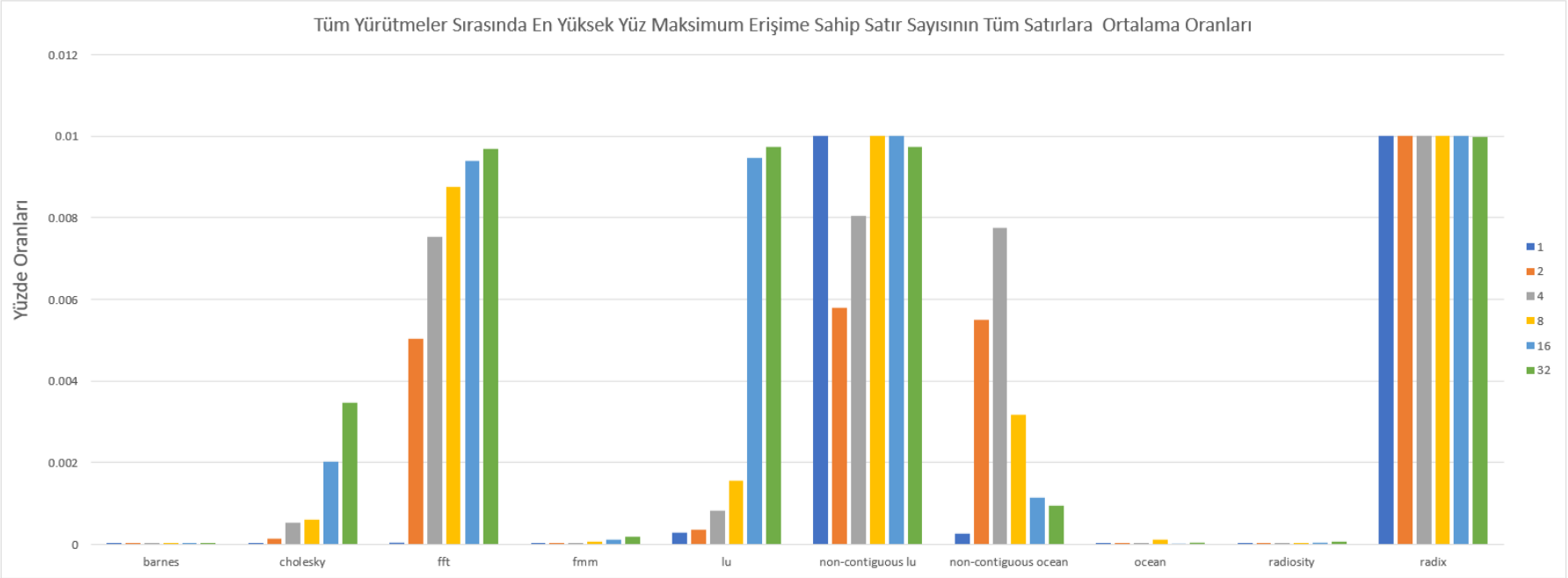
Bölüm 3'te anlatılan mekanizmanın normal bir sistem üzerindeki etkisini ölçmek için SPLASH2 ve PARSEC program bütünleri kullanıldı ve meknizmanın oluşturduğu performans düşüşü gözlemlendi. Deneyler zaman aşımı için tanımlanan sayacın farklı maksimum değerleri için tekrarlandı ve uygulamalar bitene kadar çalıştırıldı. Farklı maksimum değerlere ek olarak bu maksimum değerlerin bulunarak ajan-truva atı arasındaki iletişime eklenen gürültünün temizlenmesini önlemek amacıyla dinamik olarak değişen maksimum değerler için de sonuçlar elde edildi. Buna göre 10 ve 20 ile 10 ve 50 arasında rastgele zaman aşımı sayaçlarının kullanıldığı durumlar değerlendirildi. Şekil 4.13, SPLASH2 uygulamaları için oluşan gecikmeleri, Şekil 4.14, tüm uygulamalarda yaşanan ortalama performans kayıplarını göstermektedir. Grafikler incelendiğinde yürütme zamanının en kötü durumda yaklaşık %15 arttığı gözlemlenmiştir.



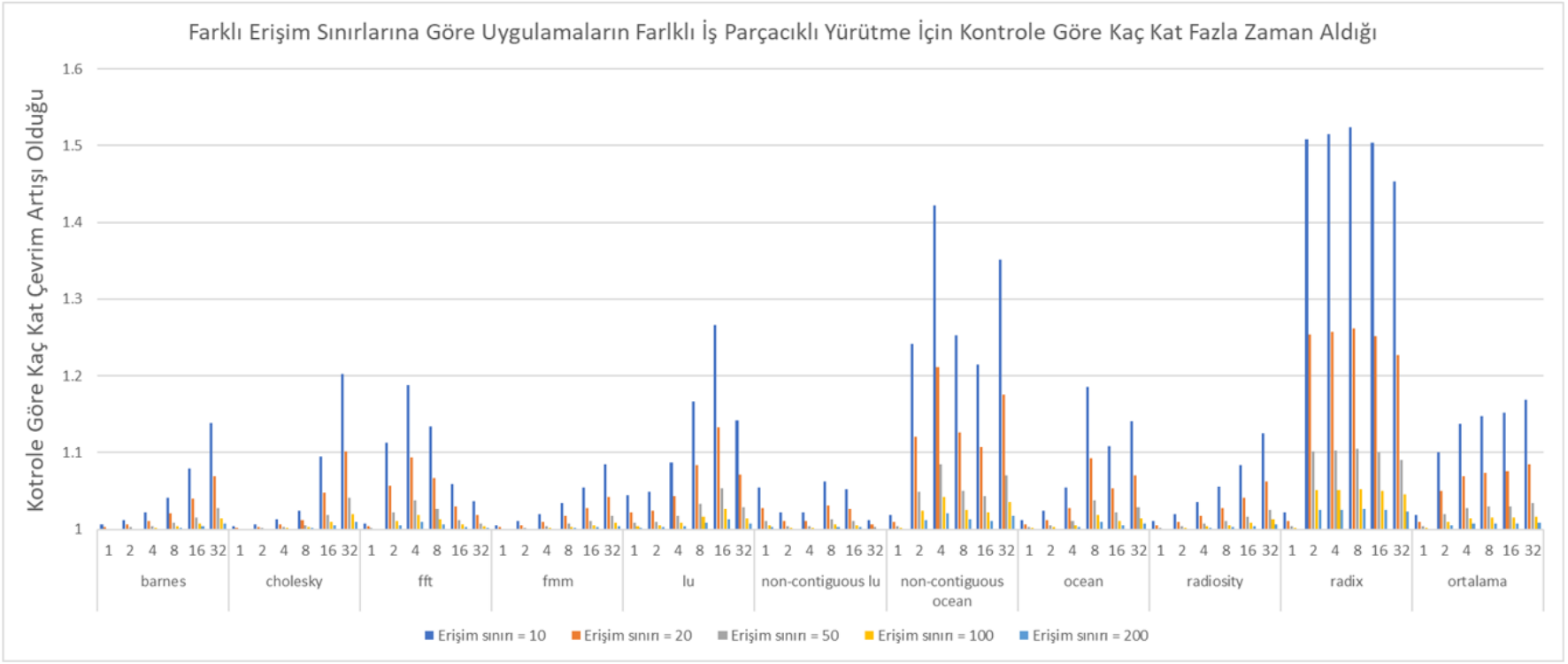
Şekil 4.10: Tüm kontrolcüler için gözlemlenen aynı adrese arka arkaya yapılan maksimum yükle buyruğu sayısı



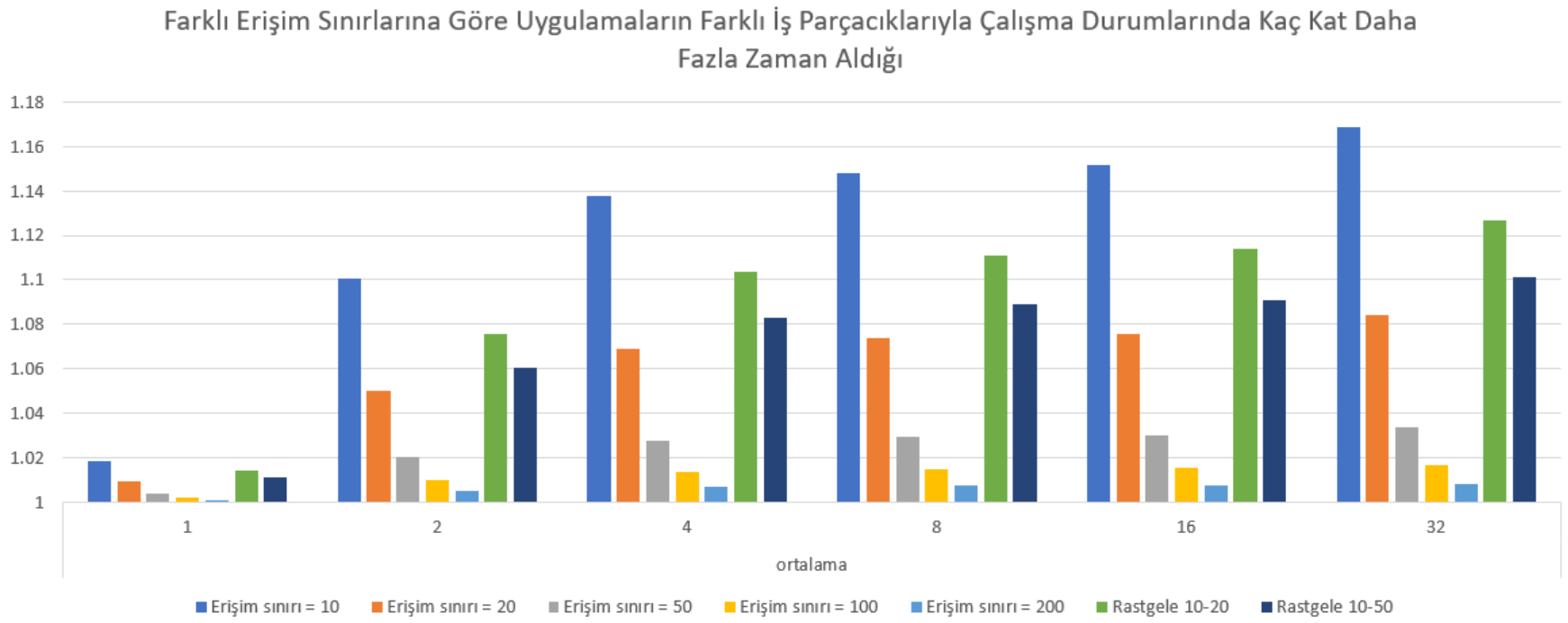
Şekil 4.11: Tüm kontrolcüler için gözlemlenen aynı adrese arka arkaya yapılan ortalama yükle buyruğu sayısı



Şekil 4.12: En yüksek arka arkaya erişim sayısına sahip 100 önbellek satırının uygulama sırasında kullanılan tüm satırlara oranı



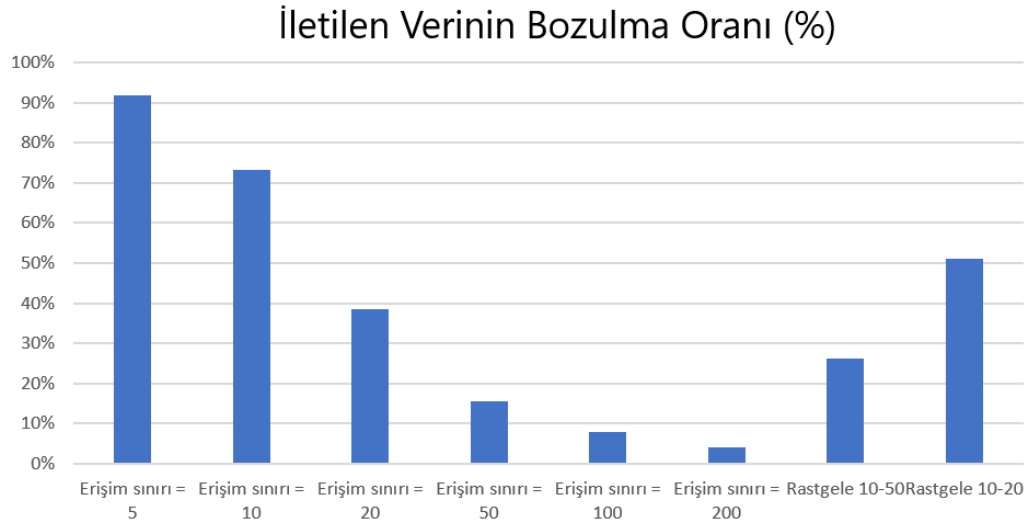
Şekil 4.13: SPLASH2 uygulamaları için 1, 2, 4, 8, 16 ve 32 iş parçacıklı çalışma sırasında ortalama performans kayıpları



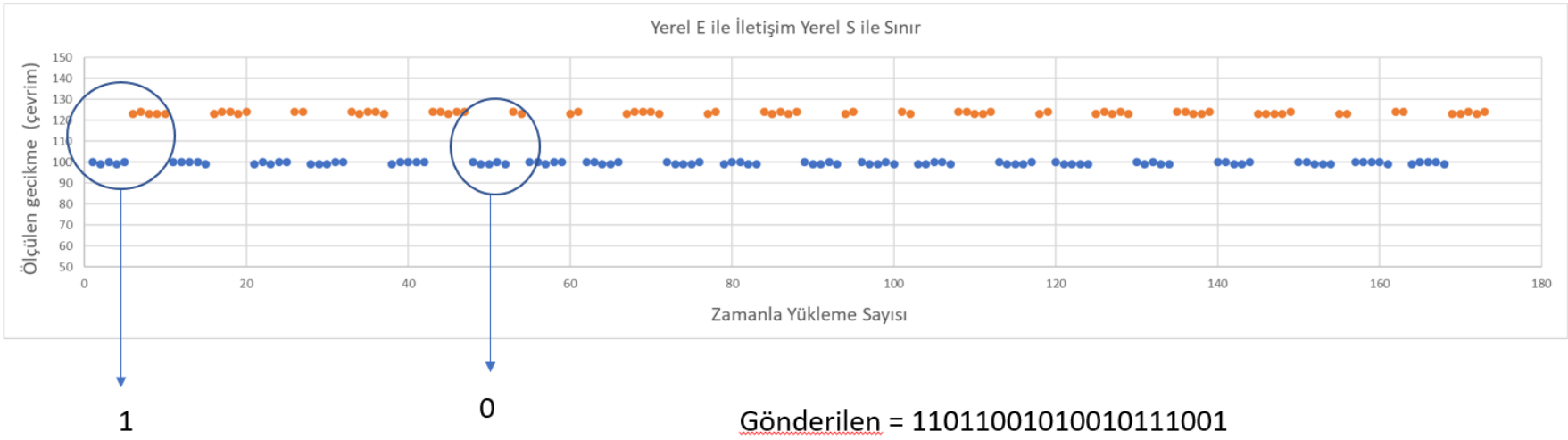
Şekil 4.14: 1, 2, 4, 8, 16 ve 32 iş parçacıklı çalışma sırasında ortalama performans kayıpları

Zamanlama yan kanalıyla gönderilen bitlerin ajan tarafından anlamlandırılmasını Şekil 4.16 açıklamaktadır. Bu grafiğe göre bitlerin anlamlandırılmasının E ve S etiketlerine olan erişimlerin sabit şekilde ayrıştırılabilir olmasına bağlı olduğu görülmektedir. Önerilen mekanizmanın etkisini görmek için zaman aşımının bu iletişimi nasıl bozduğu Şekil 4.17'da gösterilmiştir. Bu durumda iletişimde kullanılan senkronizasyon ve gönde- rim zamanlamaları ajan tarafından anlaşılamayacak şekilde değişmektedir. Bu durumda bitlerin bir bölümü iki uygulama arasıdaki iletişimde kaybedilecektir. Yaşanan kaybın ölçüsü olarak ajan ve truva atı arasındaki bit gönderiminde gönderilen bitin anlaşılamaması ya da yanlış gönderildiği durumların tüm iletişime olan yüzdesini Şekil 4.15 ortaya koymaktadır. Ölçümler 256 KB veri transferi için yapılmıştır.

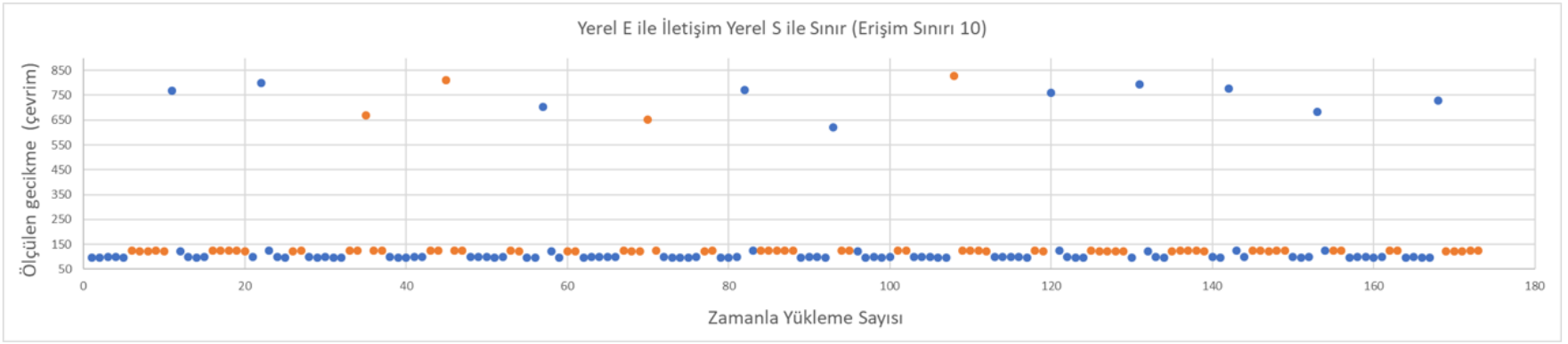
Sonuçlara göre farklı koşullarda uygulanabilecek şemalar öne sürülebilir. Grafikler incelendiğinde performans kaybı başına bit bozumu erişim sınırı 5 iken oldukça yüksek iken gönderilen verinin %90'ından çoğunun ajan tarafından anlaşılamaz durumda olduğu gözlemlenmektedir. Benzer şekilde eşik değeri arttıkça performans kaybı başına bit bozumunun arttığı fakat tüm iletişimin bozulmasının git gide azaldığı görülmektedir. Rastgele eşik değerleri durumlarına bakıldığında iletişimin yarısından fazlasının geri mühendislikle anlaşılamayacak şekilde bozulabildiği ortaya konulmuştur. Bu veriler ışığında seçilecek eşik değerinin ve rastgeleliğin performansa ve bit kaybına etkisi ciddi şekilde değişim göstermektedir. Bu noktada sistem ihtiyacına göre istenilen değer tercih edilebilir. Eğer resim ya da ses dosyaları gibi verilerle çalışan bir sistem üzerinde bu iletişim sağlanmaya çalışılıyorsa bit kaybının çok olduğu eşik değerlerini seçmek daha güvenli bir sistem oluşturacaktır. Benzer şekilde eğer hata toleransı olmayan verilerle çalışılıyorsa, sistemin güvenliği için seçilecek eşik değerlerinde performans kaybını en aza indirgemek faydalı olabilir.



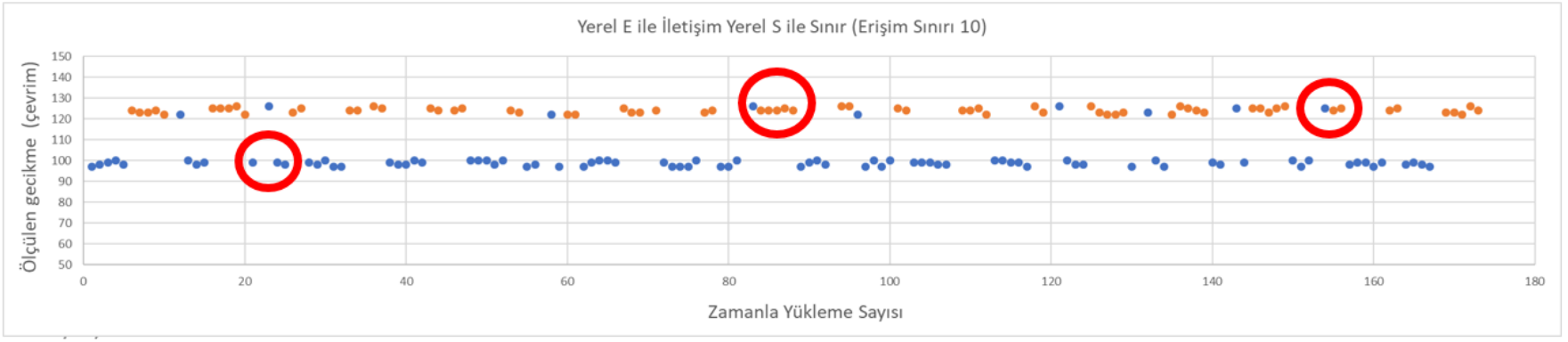
Şekil 4.15: Farklı eşik değerleri için verilerin bozulma oranı



Şekil 4.16: Normal koşullarda ajan'ın zamama bağlı olarak yaptığı zamanlama ölçümleri ve gönderilen bitler



(a)



(b)

Şekil 4.17: Geçen zamana bağlı olarak ajan uygulamasının zamanlama ölçümleri (Erişim sınırı 10 için) (a) ve zamanlama grafiğinin yakınlaştırılmış hali (b)

5. DEĞERLENDİRME VE GELECEK ÇALIŞMALAR

Tez kapsamında tutarlılık protokolleri kullanılarak iki ana konu ele alınmıştır. Bu konulardan ilki olan hata düzeltimi için HDK'ya alternatif olarak birinci düzey paylaşımlı önbelleklerde kullanılabilecek bir mekanizma önerilmiştir. İkinci olarak gün geçtikçe önemi hızla artan donanım güvenliği konusunda [1]'ın ortaya koyduğu saldırı ele alınara tutarlılık protokolleri kullanılarak oluşturulan zamanlama yan kanalları engellenmeye çalışılmıştır. Bunun için iki mekanizma gösterilmiş ve detaylı analizleri yapılmıştır.

Çok çekirdekli sistemlerde, paylaşımlı birinci düzey önbelleklerde etkili bir şekilde kullanılabilecek bir yapı bulunmamaktadır. Var olan yapılar ECC'ye dayandığından yer ve zaman israfı oluşturmaktadır. Yapılan çalışmalar önbelleklerde görülen hata miktarını azaltmak için ovalama (cache scrubbing) kullanmata ya da ECC mekanizmalarının harcadığı enerjiyi ve düzeltme oranını optimize etmeye çalışmaktadır [2, 10, 39]. Bilinen çalışmalar göz önüne alındığında birinci düzey önbellekler için önerilen bu mekanizma alanında bir ilktir. Sonuçlar incelendiğinde Splash2 programları için uygulamaların dörtte birinin bu mekanizma ile korunabile- ceği görülmüştür. Ayrıca bu oranı yazılımsal olarak artırmak ve daha çok veriyi koruyabilmek mümkündür. Bunun yanında, zaman analizi göz önünde bulundurulduğun- da herhangi bir zaman aralığında bu mekanizmanın kullanılamayacak olmasının olasılı- ğının çok az olduğu görülmektedir. Yani düzeltile- bilecek bir veri bulma oranı yüksek ve kullanılamama zamanı oldukça düşüktür. Önerilen mekanizmanın ihtiyaç duyduğu iki işlem bulunmaktadır. Bu işlemler, eşlik biti kontrolü ve dizinde paylaşımlı veriler için yeni bellek erişimi oluşturmaktır. Eşlik biti var olan sistemlerde birinci düzey önbellekler için varsayılan özelliktir ve bu sebeple ekstra bir alana gerek duyulmamaktadır.

Fakat eşlik biti kontrolünden sonra dizine gidilecek şekilde sistemde bir düzenleme yapılmalıdır. Bu düzenlemede dizinden dönen paylaşımlı önbellek satırı sorgusuna karşılık yeni bir bellek erişimi oluşturma işlemi kesme gibi çalışacak şekilde düzenlenebilir. Bunlar göz önüne alındığında, önerilen mekanizmanın ekstra gereksinimlerinin minimal düzeyde olduğu görülmektedir.

Güvenlik açısından elde edilen sonuçlar incelendiğinde zaman aşımı eşik değerlerinin seçiminin en önemli nokta olduğu görülmektedir. Bu değer arttıkça iletişime eklenen gürültü değerinin azaldığı fakat performansın arttığı gözlemlenmiştir. Bu noktada, bu mekanizmanın kullanılacağı sistemin özelliklerine göre eşik değeri seçimi yapılmasının gerekliliği açıktır. 5 gibi düşük eşik değerleri kullanıldığında ajan ve truva atı uygulamalarının iletişiminin %90'ından fazlasının engellenebileceği görülmüştür. Eğer hata

toleransı yüksek olan görev kritik sistemlerde bu mekanizma kullanılıyorsa yaşanacak %15'lik bir çalışma zamanı artışı önemli olmayabilir. Ancak performansın önemli olduğu sistemler için iletişim bozumu ve performans arasında istenilen ödünleşmeyi seçmek daha uygun olabilir. Hata toleransı olmayan veriler ile çalışılıyorsa, çalışma zamanını %1 etkileyip iletişimi %10 bozarak güvenlikten ödün vermemek mümkün olabilir.

Önerilen sistemin güvenliğini artırmak için seçilen eşik değerinin dinamik olarak değiştirmek mümkündür. Bu rastgelelik iletişime eklenen gürültünün ters mühendislikle çözülmesinin önüne geçecektir. Rastgelelik kullanmayan eşik değerleri için ajan, eğer kullanılan mekanizmanın farkına varırsa eşik değerini bulabilir. Bu değer bulunduktan sonra tüm olasılıklar denenerek iletişimdeki gürültü yok edilebilir. Bu da güvenliğin sağlanamayacağını gösterir. Ancak, Bölüm 4'te gösterildiği gibi önbellek satırlarına eğer rastgele değerlerle eşik değereri atanırsa belirli bir şablona göre iletişime gürültü katılmıyor olacağından ters mühendislik yapma yetisi ortadan kalkmış olacaktır. Rastgele değerlerin oluşturulduğu aralık performansı ve bit kaybını etkileyeceğinden, eşik değerinin sabit olduğu duruma benzer olarak, bu ödünleşmenin sisteme en uygun olduğu sayılar seçilmelidir.

Hata düzeltimi mekanizması tez kapsamında potansiyel olarak incelenmiş olsa da gerçek bir sistem üzerinde performansı HDK şemaları ile karşılaştırmalı olarak ölçülebilir. Benzer şekilde düzeltim için harcanacak enerji karşılaştırması yapmak mümkündür. Tez kapsamında mekanizmanın ortalama hata oluşma zamanına etkisi incelenmiş ancak somut olarak ortaya konulmamıştır. Ortalama hata oluşma zamnının analizi mekanizmanın kullışlılığını ortaya koyacaktır. Bunlara ek olarak devre alanı ve düzeltim gecikmesi açısından analizler yapılabilir.

Güvenlik mekanizması için daha farklı saldırılar için önerilen mekanizmanın etkisini gözlemlemek mümkündür. Bununla beraber enerji ve alan analizi yapılarak bu mekanizmanın gerçeklenmesinin etkileri daha açık görülebilir.

Tez, iki ana konu için de önerilen mekanizmaların teorik olarak kullanılabilir olduğunu gösterir niteliktedir. Gerçek sistemlerde kullanışlılığı görmek için çalışmalara yol göstermektedir.

KAYNAKLAR

[1] **Yao**, **F.**, **Doroslovacki**, **M.**, and **Venkataramani**, **G.** (2018). Are Coherence Protocol States Vulnerable to Information Leakage? In: *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*. IEEE Computer Society, pp. 168–179.

[2] **Govindaraju**, **V.** et al. (2008). Toward a multicore architecture for real-time ray-tracing. In: *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pp. 176–187.

[3] **Sadler**, **N.** and **Sorin**, **D. J.** (2006). Choosing an Error Protection Scheme for a Microprocessor's L1 Data Cache. In: *2006 International Conference on Computer Design*, pp. 499–505.

[4] **Shivakumar**, **P.** et al. (2002). Modeling the effect of technology trends on the soft error rate of combinational logic. In: *Proceedings International Conference on Dependable Systems and Networks*, pp. 389–398.

[5] **Abu-Ghazaleh**, **N.**, **Ponomarev**, **D.**, and **Evtyushkin**, **D.** (2019). How the spectre and meltdown hacks really worked. In: *IEEE Spectrum* 56.3, pp. 42–49.

[6] **Yan**, **M.** et al. (May 2019). Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. In: *2019 2019 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society.

[7] **Liu**, **F.** et al. (2015). Last-Level Cache Side-Channel Attacks are Practical. In: *2015 IEEE Symposium on Security and Privacy*, pp. 605–622.

[8] **Yarom**, **Y.** and **Falkner**, **K.** (Aug. 2014). FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, pp. 719–732.

[9] **Kim**, **J.** et al. (2007). Multi-bit Error Tolerant Caches Using Two-Dimensional Error Coding. In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pp. 197–209.

[10] **Yoon**, **D. H.** and **Erez**, **M.** (2009). Flexible cache error protection using an ECC FIFO. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–12.

[11] **Ergin**, **O.** et al. (2006). Exploiting Narrow Values for Soft Error Tolerance. In: *IEEE Computer Architecture Letters* 5.2, pp. 12–12.

[12] **Alameldeen**, **A. R.** et al. (2011). Energy-Efficient Cache Design Using Variable-Strength Error-Correcting Codes. In: *Proceedings of the 38th Annual International Symposium on Computer Architecture*. ISCA '11. San Jose, California, USA: Association for Computing Machinery, pp. 461–472.

[13] **Farbeh**, **H.** and **Miremadi**, **S. G.** (2014). PSP-Cache: A low-cost fault-tolerant cache memory architecture. In: *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1–4.

[14] **Wilkerson**, **C.** et al. (2010). Reducing Cache Power with Low-Cost, Multi-Bit Error-Correcting Codes. In: *Proceedings of the 37th Annual International Symposium on Computer Architecture*. ISCA '10. Saint-Malo, France: Association for Computing Machinery, pp. 83–93.

[15] **BanaiyanMofrad**, **A.**, **Homayoun**, **H.**, and **Dutt**, **N.** (2011). FFT-Cache: A Flexible Fault-Tolerant Cache Architecture for Ultra Low Voltage Operation. In: *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. CASES '11. Taipei, Taiwan: Association for Computing Machinery, pp. 95–104.

[16] **Mohr**, **K. C.** and **Clark**, **L. T.** (2006). Delay and Area Efficient First-level Cache Soft Error Detection and Correction. In: *2006 International Conference on Computer Design*, pp. 88–92.

[17] **Irazoqui**, **G.**, **Eisenbarth**, **T.**, and **Sunar**, **B.** (2016). Cross Processor Cache Attacks. In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ASIA CCS '16. Xiapos;an, China: Association for Computing Machinery, pp. 353–364.

[18] **Patterson**, **D. A.** and **Hennessy**, **J. L.** (2008). Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design). 4th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

[19] **Stallings**, **W.** (2002). Computer Organization and Architecture. 6th. Prentice Hall Professional Technical Reference.

[20] **Adve**, **S. V.** et al. (1991). Comparison of Hardware and Software Cache Coherence Schemes. In: *Proceedings of the 18th Annual International Symposium on Computer Architecture*. ISCA '91. Toronto, Ontario, Canada: Association for Computing Machinery, pp. 298–308.

[21] **Sorin**, **D. J.**, **Hill**, **M. D.**, and **Wood**, **D. A.** (2011). A Primer on Memory Consistency and Cache Coherence. 1st. Morgan Claypool Publishers.

[22] **Agarwal**, **A.** et al. (1988). An Evaluation of Directory Schemes for Cache Coherence. In: *Proceedings of the 15th Annual International Symposium on Computer Architecture*. ISCA '88. Honolulu, Hawaii, USA: IEEE Computer Society Press, pp. 280–298.

[23] **Archibald**, **J.** and **Baer**, **J.-L.** (Sept. 1986). Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. In: *ACM Trans. Comput. Syst.* 4.4, pp. 273–298.

[24] **Yang**, **Q.**, **Bhuyan**, **L. N.**, and **Liu**, **B. .-.-.** (1989). Analysis and comparison of cache coherence protocols for a packet-switched multiprocessor. In: *IEEE Transactions on Computers* 38.8, pp. 1143–1153.

[25] **Martin**, **M. M. K.**, **Hill**, **M. D.**, and **Wood**, **D. A.** (2003). Token Coherence: decoupling performance and correctness. In: *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.* Pp. 182–193.

[26] **Kelm**, **J. H.** et al. (2010). WAYPOINT: Scaling Coherence to Thousand-Core Architectures. In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. PACT '10. Vienna, Austria: Association for Computing Machinery, pp. 99–110.

[27] **Goodman**, **J. R.** (1983). Using Cache Memory to Reduce Processor-Memory Traffic. In: *Proceedings of the 10th Annual International Symposium on Computer Architecture*. ISCA '83. Stockholm, Sweden: Association for Computing Machinery, pp. 124–131.

[28] **Stenstrom**, **P.** (1990). A survey of cache coherence schemes for multiprocessors. In: *Computer* 23.6, pp. 12–24.

[29] **Katz**, **R. H.** et al. (1985). Implementing a Cache Consistency Protocol. In: *Proceedings of the 12th Annual International Symposium on Computer Architecture*. ISCA '85. Boston, Massachusetts, USA: IEEE Computer Society Press, pp. 276–283.

[30] **Papamarcos**, **M. S.** and **Patel**, **J. H.** (1984). A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In: *Proceedings of the 11th Annual International Symposium on Computer Architecture*. ISCA '84. New York, NY, USA: Association for Computing Machinery, pp. 348–354.

[31] **Sweazey**, **P.** and **Smith**, **A. J.** (1986). A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus. In: *Proceedings of the 13th Annual International Symposium on Computer Architecture*. ISCA '86. Tokyo, Japan: IEEE Computer Society Press, pp. 414–423.

[32] **Randell**, **B.**, **Lee**, **P.**, and **Treleaven**, **P. C.** (June 1978). Reliability Issues in Computing System Design. In: *ACM Comput. Surv.* 10.2, pp. 123–165.

[33] **Mukherjee**, **S. S.**, **Emer**, **J.**, and **Reinhardt**, **S. K.** (2005). The soft error problem: an architectural perspective. In: *11th International Symposium on High-Performance Computer Architecture*, pp. 243–247.

[34] **Martínez**, **J. A.**, **Maestro**, **J. A.**, and **Reviriego**, **P.** (2017). A Scheme to Improve the Intrinsic Error Detection of the Instruction Set Architecture. In: *IEEE Computer Architecture Letters* 16, pp. 103–106.

[35] **Atamaner**, **M.** et al. (2017). Detecting errors in instructions with bloom filters. In: *2017 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pp. 1–4.

[36] **Seifert**, **N.**, **Xiaowei Zhu**, and **Massengill**, **L. W.** (2002). Impact of scaling on soft-error rates in commercial microprocessors. In: *IEEE Transactions on Nuclear Science* 49.6, pp. 3100–3106.

[37] **Goodman**, **B. L.** et al. (Oct. 2016). ECC bypass using low latency CE correction with retry select signal.

[38] **Bienia**, **C.**, **Kumar**, **S.**, and **Li**, **K.** (2008). PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on Chip-Multiprocessors. In: *IISWC*. Ed. by **Christie**, **D.** et al. IEEE Computer Society, pp. 47–56.

[39] **Slayman**, **C. W.** (2005). Cache and memory error detection, correction, and reduction techniques for terrestrial servers and workstations. In: *IEEE Transactions on Device and Materials Reliability* 5.3, pp. 397–404.

[40] **Zaruba**, **F.** and **Benini**, **L.** (Nov. 2019). The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11, pp. 2629–2640.

[41] **Keirn**, **Z. A.** et al. (2004). Use of redundant bits for magnetic recording: single-Parity codes and Reed-Solomon error-correcting code. In: *IEEE Transactions on Magnetics* 40.1, pp. 225–230.

ÖZGEÇMİŞ

Ad-Soyad : Mert Atamaner

Uyruğu : T.C.

Doğum Tarihi ve Yeri : 23.05.1994, İzmir

E-posta : atamanermert@gmail.com

ÖĞRENİM DURUMU:

- **Yüksek Lisans** : 2018, TOBB ETÜ, Bilgisayar Müh.
- **Lisans** : 2014, TOBB ETÜ, Bilgisayar Müh.

MESLEKİ DENEYİM VE ÖDÜLLER:

Yıl	**Yer**	**Görev**
2018 - Halen	TOBB ETÜ	Özel Başarı Burslu Yüksek Lisans Öğrencisi
Ekim 2017 - Haz 2018	ETH Zürich, SAFARI Lab.	Stajyer
Oca 2017 - Nis 2017	University of Rome Tor Vergata	Stajyer
May 2016 - Ağu 2016	Kasırga Bilişim Elektronik	Stajyer

YABANCI DİL: İngilizce (İyi), Almanca (Başlangıç)

TEZDEN TÜRETİLEN YAYINLAR, SUNUMLAR VE PATENTLER:

- **Atamaner, M.**, & Ergin, O. (2019, Eylül). Paylaşımlı Önbelleklerde Tutarlılık Protokolleri Kullanılarak Hata Düzeltimi, *İşlemci Tasarım Çalıştayı*