TOBB EKONOMİ VE TEKNOLOJİ ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

GRAFİK İŞLEME BİRİMİ ÖNBELLEKLERİNDE YERELLİĞE BAĞLI DİNAMİK YAZMA POLİTİKASI

YÜKSEK LİSANS TEZİ

Çağatay TURGUT

Bilgisayar Mühendisliği Anabilim Dalı

Tez Danışmanı: Doç. Dr. Oğuz ERGİN

AĞUSTOS 2017

Fen Bilimleri Enstitüsü Onayı

……..
Prof. Dr. Osman EROĞUL
Müdür

Bu tezin Yüksek Lisans derecesinin tüm gereksinimlerini sağladığını onaylarım.

..……
Doç. Dr. Oğuz ERGİN
Anabilimdalı Başkanı V.

TOBB ETÜ, Fen Bilimleri Enstitüsü'nün 131111004 numaralı Yüksek Lisans Öğrencisi **Çağatay TURGUT**'un ilgili yönetmeliklerin belirlediği gerekli tüm şartları yerine getirdikten sonra hazırladığı **"GRAFİK İŞLEME BİRİMİ ÖNBELLEKLERİNDE YERELLİĞE BAĞLI DİNAMİK YAZMA POLİTİKASI"** başlıklı tezi 17.08.2017 tarihinde aşağıda imzaları olan jüri tarafından kabul edilmiştir.

Tez Danışmanı :	**Doç. Dr. Oğuz ERGİN** TOBB Ekonomi ve Teknoloji Üniversitesi	.....
Jüri Üyeleri :	**Prof. Dr. Mehmet Önder EFE(Başkan)** Hacettepe Üniversitesi	.....
	Yrd. Doç. Dr. Buğra ÇAŞKURLU TOBB Ekonomi ve Teknoloji Üniversitesi	.....

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.

Çağatay TURGUT

ÖZET

Yüksek Lisans Tezi

GRAFİK İŞLEME BİRİMİ ÖNBELLEKLERİNDE
YERELLİĞE BAĞLI DİNAMİK YAZMA POLİTİKASI

Çağatay TURGUT

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

Danışman: Doç. Dr. Oğuz ERGİN

Tarih: Ağustos 2017

GPU Önbelleklerinde Yerelliğe Bağlı Dinamik Yazma Politikası önerilen yazma yerelliği detektörü kullanarak uygulamaların belli bölgelerindeki yerellik yoğunluklarına bakarak uygun görülen önbellek yazma politikasına aktif geçişi sağlamaktadır. GPU mimarilerinde kullanılan farklı yazma politikalarının birbirlerine göre belli durumlarda avantajları bulunmaktadır. Uygulamaların çalışma anında tercih edilebilecek yazma politikası uygulamanın ve donanımın birçok parametrelerine göre değişiklik gösterebilmektedir. Günümüz GPU mimarilerinde derleyiciler tarafından sabit bir yazma politikası seçilmektedir. Bu çalışmada saat periyot çözünürlüğünde çalışabilen GPU simülatörü üzerinde uygulamaların çalışma anında yerellik yoğunluklarına göre kullanılan yazma politikaları arasında dinamik geçişler yapılarak sabit bir yazma politikasına göre başarım artışı sağlanabildiği gösterilmiştir.

Anahtar Kelimeler: GPU, GPGPU, Önbellek, Yazma politikaları, Yerellik detektörü, Dinamik yazma politikası

ABSTRACT

Master of Science

LOCALITY DRIVEN DYNAMIC CACHE WRITE POLICY
ON GRAPHICS PROCESSING UNITS

Çağatay TURGUT

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

Supervisor: Assoc. Prof. Dr. Oğuz ERGİN

Date: August 2017

Locality Driven Dynamic Write Policy on GPU Caches switches to appropriate write policy by sampling locality densities at intervals of an application with the suggested locality detector. Different write policies being used in GPU architectures has their own advantages compared to other policies. The most appropriate cache write policy at runtime can vary depending on the type of application and hardware resources. Present day GPU architectures use a fixed cache policy which is decided to be used before run time. On this study, we showed that it is possible to increase the performance of applications by using a dynamic cache write policy instead of a fixed cache policy on a cycle level GPU performance simulator.

Keyword : GPU, GPGPU, Cache, Write policies, Locality detector, Dynamic write policy

TEŞEKKÜR

Çalışmalarım boyunca değerli yardım ve katkılarıyla beni yönlendiren hocam Oğuz ERGİN'e, kıymetli tecrübelerinden faydalandığım TOBB Ekonomi ve Teknoloji Üniversitesi Bilgisayar Mühendisliği Bölümü öğretim üyelerine ve destekleriyle her zaman yanımda olan aileme ve arkadaşlarıma çok teşekkür ederim.

İÇİNDEKİLER

Sayfa

ŞEKİL LİSTESİ

Sayfa

TABLO LİSTESİ

Sayfa

KISALTMALAR

GPU	: Graphics Processing Unit
GPGPU	: General Purpose computing on Graphics Processing Units
CUDA	: Compute Unified Device Architecture
OpenCL	: Open Computing Language
SIMD	: Single Instruction Multiple Data
SIMT	: Single Instruction Multiple Thread
DRAM	: Dynamic Random Access Memory
MSHR	: Miss Status Holding Registers
WA	: Write Allocate
NoWA	: No Write Allocate
VTA	: Victim Tag Array
IPC	: Instructions Per Cycle

1 GİRİŞ

1.1 Grafik İşleme Birimi Evrimi

GPU (Grafik İşleme Birimi) bilgisayar görüntülerini islemek ve merkezi işlemci birimi üzerindeki yükü azaltmak için ortaya çıkmış hızlandırıcı elektronik çiptir. Kullanımı 70`lerde oyun salonlarında bulunan oyun konsollarına kadar dayanırken, resmi olarak 1999 yılında Nvidia tarafından üretilen GeForce256 çipiyle beraber GPU terimi ortaya atılmıştır [1]. Günümüzde kişisel bilgisayarlar, cep telefonları ve birçok elektronik devre içerisinde kullanılan önemli bir bilgisayar bileşeni haline gelmiştir. Böyle bir bileşene sahip olunma gereksinimi, grafik yaratımı için takip edilmesi gereken ardışık görevlere bakılarak anlaşılmaktadır.

Örnek olarak 3 boyutlu bir uzaydaki herhangi bir kesitin 2 boyutlu ekranımıza yansımasını isteyelim. Bunun için uzayımızın içerisindeki bütün objelerin koordinatları ve karakteristiklerinin önceden bilinilmesi gerekmektedir. Buna ek olarak koordinatı belli bir ışık kaynağı, bu ışıktan belli doğrultularda çıkan ışık huzmeleri, açılara ve objelerin karakteristiklerine bağlı olarak objelere çarpıp kameraya giren ışık huzmeleri gerekmektedir. Işığın tersi olarak gölgelendirme işlemleri de ışık kaynağının ve objelerin birbirlerine göre pozisyonlarına bakılarak hesaplanmaktadır. Ekrana görüntüyü getirmesi amacıyla uzayımızın içerisinde bir kamera olması ve bu kameranın pozisyonu, bakış açısı ve görüntü genişliğinin bilinmesi gerekmektedir. Bu hesaplamalar sonucunda ekranımıza bir görüntü basmak için ekran piksel sayımız kadar piksel görüntüsünün hesaplanması gerekmektedir. Yine aynı senaryo içerisinde basit bir kamera donuşu için bütün piksel çıkışlarının tekrar hesaplanması amacıyla uzayın kamera üzerine düşen görüntüsünün matris transformasyonu alınması gerekmektedir. Bütün bu hesaplamaların merkezi işlemci biriminde zaman kısıtlaması ile hesaplanması geçmiş dönemlerde oldukça zor iken günümüz uygulamalarının hızlı gelişimi sonucu olanaksız bir hale gelmiştir. Bu yüzden de hızlandırıcı bir donanımın yani GPU'nun gereksinimi doğmuştur. Verilen bu örnek aşamalara ek olarak birçok detay aşama daha bulunmaktadır, bu aşamaların

en büyük ortak yanı ise yapılan büyük vektör ve matris hesaplamalarıdır. Vektör ve matris işlemleri birbirlerinden bağımsız ve paralelleştirilebilir çok sayıda hesaplamalardan oluştuğundan dolayı GPU'lar için ideal iş yüküdür.

İlk GPU teriminin kullanılarak ortaya çıkarıldığı GeForce 256, grafik yaratımı için gerekli dönüşüm, kırpma, aydınlatma ve oluşturma gibi aşamaları tek bir çip içerisinde oluşturabildiği için büyük ilgi görmüştü. Bu sayede daha önce dönüşüm ve aydınlatma gibi grafik yaratım aşamalarının hesaplamaları merkezi işlemci biriminden GPU üzerine taşınarak, merkezi işlemci biriminin grafik yaratım işlemlerindeki hesaplamalarını fazlaca azaltmıştır. GeForce 256 ve sonrasında üretilen GPU örneklerinde de sabit grafik üretim aşamalarına sahip bir donanım mimarisi bulunmaktaydı. Bu sabit aşamalı mimarinin belli bölümlerinin değiştirilememesi ve tasarımcılar tarafından bir gereksinim haline gelmesi üzerine 2002 yılında GeForce 3 ile beraber "nfiniteFX" programlanabilir grafik yaratım mimarisi gelmiştir [2]. Bu mimari ile bazı gölgelendirme aşamalarının programlanabilir hale gelmesi tasarımcıların kendi grafik oluşturma gereksinimleri doğrultusunda gölgelendirme aşamalarını düşük seviye programlama dili ile tasarlama olanağına sahip hale gelmişlerdir. Bu mimari ile ilk defa GPU üzerinde gölgelendirme aşamalarına programlar yazılarak grafik girdileri üzerinde istenilen matematiksel hesaplamalar yapılmış ve GPU entegresinin bir işlemci olarak kullanıldığı döneme girilmiştir. Daha önceden olduğu gibi GeForce3 ailesinde de grafik yaratım aşamalarında kullanılan farklı gölgelendirme aşamalarının donanımsal olarak ayrı kaldığı mimari devam etmekteydi. 2006 yılında donanımsal olarak farklı alanlara sahip olan ve farklı hesaplamalar yapan piksel, geometri ve tepe gölgelendirme gibi aşamaları daha kapsamlı programlanabilir donanımsal tek merkezi bir gölgelendirme aşaması yaratılarak günümüzde modern GPU'larda da kullanılan birleşik gölgelendirme mimarisi ortaya konulmuştur. Grafik üretiminde farklı gölgelendirme aşamalarının kullanılma oranları üretilecek grafiğe göre değişiklik göstermesi ve zaman zaman bazı gölgelendirme aşamalarının hiç kullanılmaması, eski mimarilerde performans tıkanıklığına yol açmaktaydı. Bunun yanı sıra farklı gölgelendirme aşamalarının donanımsal olarak var olması çip boyutları, kullanılan güç ve maliyet açısından da dezavantajlar yaratmaktaydı. Bu sebeplerden dolay programlanabilir tek bir gölgelendirme alanı yaratılarak, grafik yaratılma suresinde birden fazla gölgelendirme

aşamasının aynı anda kaynak paylaşımı ya da zamanda paylaşımı oluşturularak eski mimarilere göre daha verimli grafik üretebilen bir mimari ortaya çıkmıştır.

1.2 Tez Kapsamı

Bilimsel hesaplamalarda elde edilen başarımın artması amacıyla GPU mimarisine gelen çok katmanlı önbellek hiyerarşisinde L2 önbellekleri üzerinde kullanılan önbellek politikalarının başarıma etkisi incelenmiş, dinamik bir önbellek politikası önerilmiştir. Önerilen bu politika GPU'lar üzerinde yapılan akademik araştırmalarda kullanılan GPGPU-Sim simülatöründe tasarlanmış. Sunulan politikayı gerçeklemek için simülatör kaynak kodlarında değişiklikler yapılmıştır. Farklı GPU kütüphanelerinden seçilen uygulamalar GPGPU-Sim içerisinde bulunan önbellek politikaları ve bu çalışmada önerilen dinamik politika ile çalıştırılmış, sonuçlar karşılaştırılarak yorumlanmıştır.

1.3 Tez Organizasyonu

Bölüm 1.1'de kısa GPU tarihçesi, bölüm 1.2'de çalışmanın kapsamı özetle anlatılmıştır.

Bölüm 2.1'de GPU mimarisi gösterilip, kodlama ile GPGPU örneği verilmiştir. Bölüm 2.2'de Fermi ailesine ait GPU'ların mikromimarisi gösterilip çalışma prensipleri gösterilmiştir. Bölüm 2.3'de bu çalışmaya paralel olan ve referans verilen çalışmalar özetlenmiştir.

Bölüm 3'de bu çalışmanın sunulan tasarımın üzerinde gerçekleştirildiği GPU simülatörü olan GPGPU-Sim detaylı bir şekilde anlatılmıştır.

Bölüm 4'de bilgisayar mimarilerinde tanımlı önbellek yazma politikaları tanımlanmış ve birbirlerine göre avantajları ve dezavantajları tartışılmıştır.

Bölüm 5'de sunulan tasarımın çalışma prensipleri anlatılmıştır. Bölüm 5.1'de kullanılan önbellek politikaların durum makineleri gösterilmiş, bölüm 5.2'de ise tasarım için gerekli olan ek donanım ve çalışma durumları gösterilmiştir. Bölüm 5.3'de sunulan dinamik politikanın hangi politika seçmesi gerektiğini gösteren skorlama sistemi anlatılmıştır.

Bölüm 6.1'de farklı uygulamaların karakteristikleri ve önbellek politikalarının bu uygulamalar üzerindeki etkileri gösterilmiştir. Bölüm 6.2'de GPU önbellekleri arası davranış benzerliği ve dinamik politikada kullanılan politikaların eşit yerellikler sayması için yapılan testler gösterilmiştir. Bölüm 6.3'de ise uygulamaların farklı politikalar ile çalıştırılması sonuçları gösterilmiş ve karşılaştırılmıştır. Bölüm 6.4'de dinamik politikanın uygulamalar üzerindeki etkileri yorumlanmıştır.

2 GRAFİK İŞLEME BİRİMİ

2.1 Grafik İşleme Birimi Mimarisi

Birleşik gölgelendirme mimarisi ile ortaya programlanabilir bir işlemci ortaya çıkmıştır [3]. Grafik üretiminde girdi grafiğini bir matris ve gölgelendirme uygulamasını matematiksel işlemler olarak kullanarak genel amaçlı hesaplamaların CPU yerine GPU üzerinde yapılmasına GPGPU (Grafik İşlemcisinde Genel Amaçlı Hesaplama) denmektedir. Merkezi işlemci birimine göre daha düşük frekanslarda çalışsa da fazla sayıda çekirdeğe sahip olması GPU'ların resim ya da grafik verileri gibi bağımsız çok fazla hesaplamaya sahip problemler üzerinde daha efektif çalışabilmektedir. Paralel programlama için oldukça verimli çalışan GPU, seri programlama da oldukça dezavantaja sahiptir. Fazla sayıda çekirdeğe sahip olan GPU, seri programlama ile sadece tek bir çekirdeğini çalıştırarak CPU'ya göre oldukça verimsiz bir başarım elde etmektedir. Bu sebepten dolayı donanım üzerindeki başarımı maksimize etmek için yazılan bir kodun seri kısımlarının CPU üzerinde ve paralel kısımlarının GPU üzerinde çalıştırılması sıkça kullanılan bir tekniktir. Paralel hesaplamalarda CPU'ya göre sahip olduğu bu avantajından dolayı bilimsel hesaplamalar için de uygun bir donanım haline gelmesi, oyun ve animasyon odaklı geliştirilen GPU mimarilerinin GPGPU odaklı da geliştirilmesi için bir motivasyon haline gelmiştir. Bu sayede grafik oluşturma amaçlı performans artışı getirmeyen fakat bilimsel hesaplamalar için büyük avantajlar sağlayan mimari değişikliklerine gidilmiştir. Eski mimarilerin sahip olmadığı dallanma komutlarına destek, bu mimari değişiklikler içerisinde SIMD (Tek komut Çok Veri) mimarisinden SIMT (Tek Komut Çok İş Parçacığı) mimarisine geçişteki önemli gelişmelerden biridir [4]. Aynı şekilde, bilimsel hesaplamaları daha iyi desteklemek amacıyla çok seviyeli önbellek hiyerarşisi de GPU'lar için güncel bir mimari eklentisidir.

Günümüzde GPGPU programlama için CUDA (Compute Unified Device Architecture) ve OpenCL (Open Computing Language) kullanılmaktadır [5] [6]. Mimarilerin mantıksal olarak oldukça benzemesine rağmen CUDA ve OpenCL arasında terminoloji farklılıkları vardır. Bu çalışmadaki terimler CUDA terminolojisi

kullanılarak anlatılmıştır. Eski GPU mimarilerinde gölgelendirme aşamalarını programlamanın tek yolu zor ve karmaşık olan düşük seviyede kullanılan uygulama programlama arayüzünü kullanmaktaydı. Yeni mimariler ile ortaya çıkan CUDA, yüksek seviyede uygulama programlama arayüzü sağlayarak C dili kuralları dahilinde kod yazılmasını desteklemektedir.

Şekil 2.1`de aynı hesaplamayı yapan Standart C ile CUDA kodları gösterilmiştir. 1 milyon uzunlukta iki dizeden biri sabit bir girdiyle çarpılarak diğer girdiyle toplanır. Standart C kodunda bu işlem döngü içerisinde tek iş parçacığıyla hesaplanmaktadır. CUDA kodu için öncelikle girdiler bellek transferi yapılarak GPU'ya yollanır, transferlerden sonra 256 iş parçacığından oluşan 4096 CUDA iş parçacığı bloğu çalıştırılır, bu da her dize girdisi için bir iş parçacığına, toplamda 1 milyon iş parçacığına denk gelmektedir. Çalıştırılan fonksiyon başında __global__ yazarak bu fonksiyonun GPU üzerinde DRAM belleği kullanarak çalışacağı söylenmiş olur. Çalışacak 1 milyon iş parçacığı hesaplamasından sonra sonuç bellek transferi ile GPU'dan geri alınır.

CUDA'nın sağladığı yüksek seviye uygulama programlama arayüzü ile GPU üzerinde kod koşturmanın oldukça kolay hale gelmesine rağmen koşturulacak koddan alınacak başarım, kod ile donanım arasındaki uyuşmaya oldukça bağımlıdır. Bu da CUDA ile çalıştırılacak toplam iş parçacığı, iş parçacığı bloğu, iş parçacığı üzerine kodda düşen değişkenler, kullanılan matematiksel hesaplamalar gibi parametrelerin kodun üzerinde koşacağı GPU donanım mimarisine göre özenle seçilmesini gerektirmektedir.

Standart C Kodu	CUDA eklentili C kodu

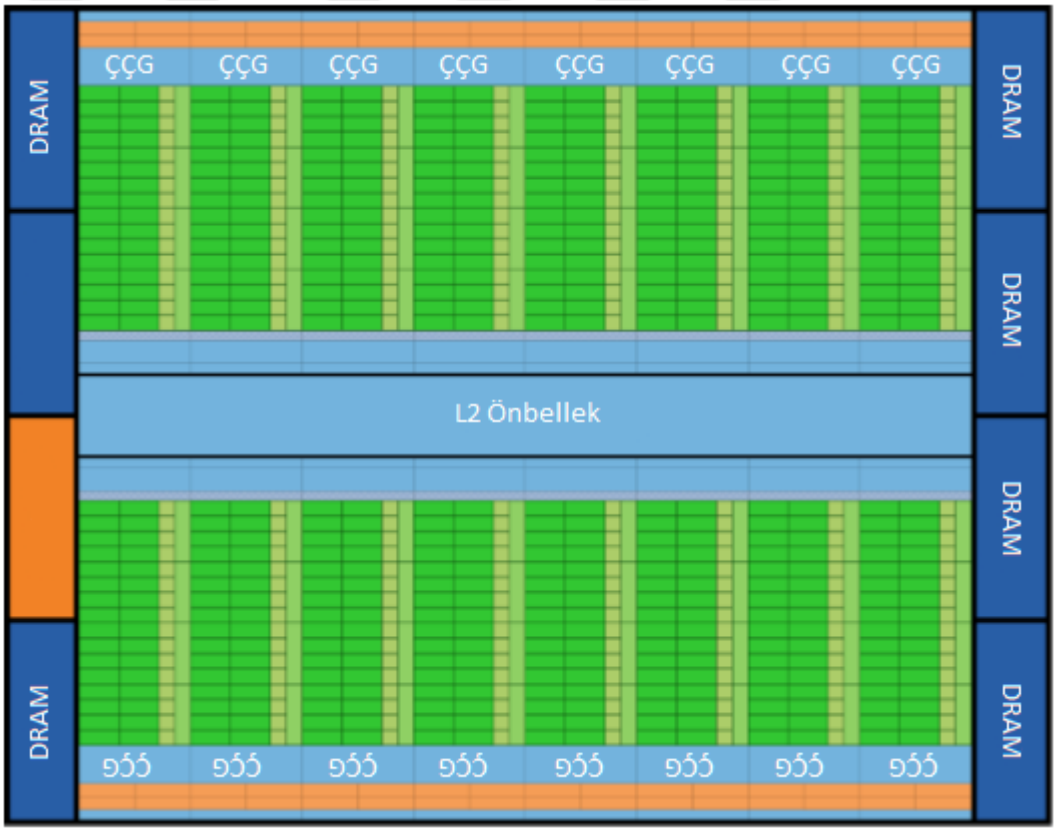
void carpmaToplama(int n,float a,float*x,float*y){
for(int i =0; i < n;++i)
y[i]= a*x[i]+ y[i];
}
 	
int N =1<<20;
 	
carpmaToplama(N,2.0, x, y);
	__global__ void carpmaToplama(int n,float a,float*x,float*y){
	int i = blockIdx.x*blockDim.x + threadIdx.x;
	if(i < n)
	y[i]= a*x[i]+ y[i];
	}
	int N =1<<20;
	cudaMemcpy(x, d_x, N, cudaMemcpyHostToDevice);
	cudaMemcpy(y, d_y, N,cudaMemcpyHostToDevice);
	
	carpmaToplama<<<4096,256>>>(N,2.0, x, y);
	cudaMemcpy(d_y, y, N, cudaMemcpyDeviceToHost);

Şekil 2.1: C ve CUDA eklentili C kodları farklılıkları

2.2 Grafik İşleme Birimi Mikro-mimarisi

Şekil 2.2`de Fermi ailesi ile beraber gelen birleşik gölgelendirme mikro-mimarisi gösterilmiştir [3]. 6 adet 64 bitlik toplamda 384 bit bellek ara yüzüne sahip 6GB bellek hacmine kadar destek sağlayan bu mimari 1848 MHz hızında çalışarak 177.4 Gbps bellek bant genişliğine sahiptir. Ortak erişilebilen 768KB L2 önbellek etrafına yerleştirilmiş 512 adet CUDA çekirdekleri 32`lik gruplara ayrılarak 16 adet ÇÇG (Çoklu Çekirdek Grubu) oluşturulmuştur. Bu mimari ile GTX480 1345 GFLOPS hesaplama gücüne sahiptir.

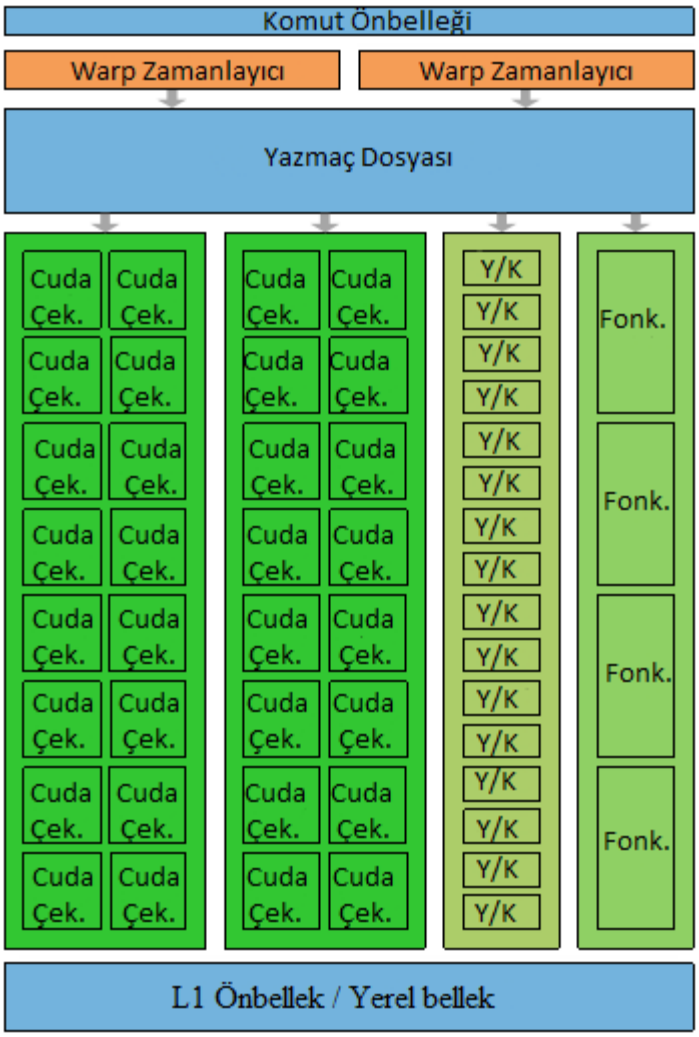
Modern GPU'lardan biri olan GP100 ile bu değerler 1251 MHz 4096 bit bellek arayüzü ile 720 Gbps bellek bant genişliği, 1417 MHz hızında çalışan 3584 CUDA çekirdeği ile 10.6 TFLOPS hesaplama gücüne sahiptir.



Şekil 2.2: Fermi birleşik gölgelendirme mikro mimarisi, GTX480

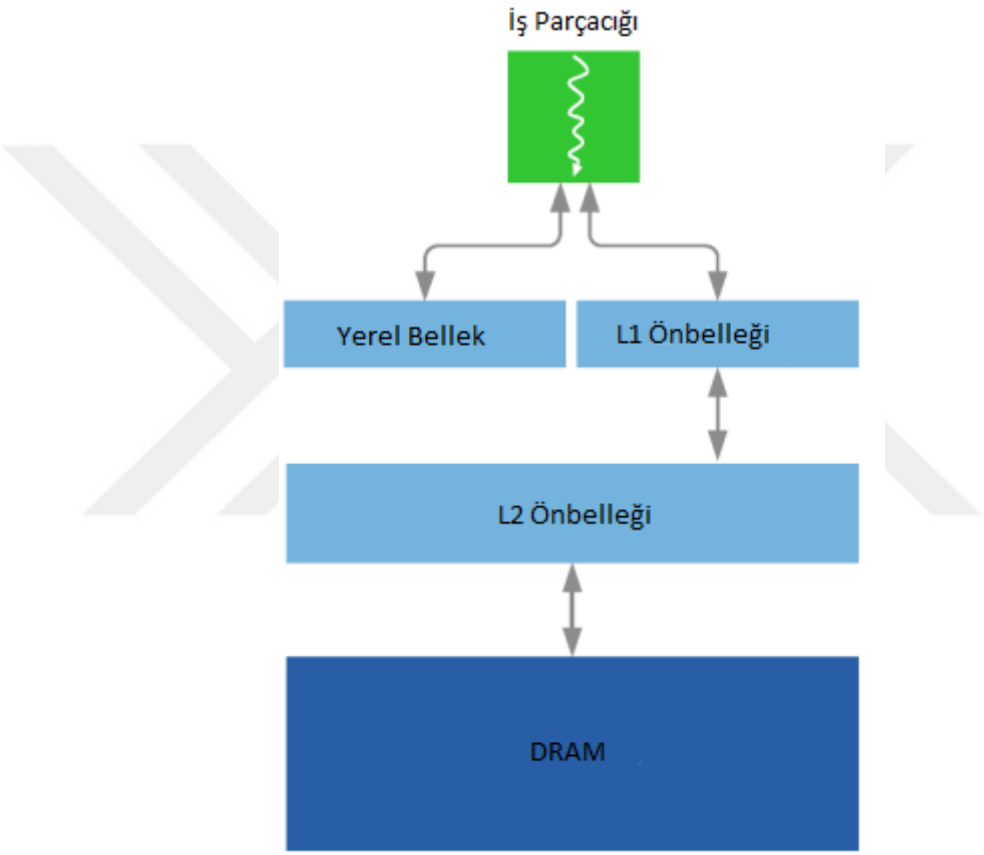
Şekil 2.3`de Fermi ailesinde işlemlerin yapıldığı önemli mimari parçası olan çoklu çekirdek grubunun mimarisi gösterilmiştir. Bu mimari kayan nokta ya da tamsayı işlemlerini her saat periyodunda birleşik çarpma ve toplama ile yapabilen CUDA çekirdekleri, bellek operasyonlarını yürüten yükleme/kaydetme birimleri,

trigonometrik ve karekök gibi işlemleri yapabilen fonksiyon birimleri, çift warp zamanlayıcısı, yazmaç dosyası ve önbellek ile yerel bellek arasında ayrılabilir SRAM belleğinden oluşmaktadır. GPU ile çalıştırılan iş parçacıkları warp adi verilen 32lik iş parçacıkları gruplarda çalıştırılmaktadır. Her saat darbesinde 2 warp zamanlayıcısı tarafından komutları hazır bulunan 2 adet warp seçilir ve komuta göre gerekli birimlere yollanır. Bu birimler 16 CUDA çekirdeğinden oluşan 2 birim, 16 yükleme/kaydetme birimi ve 4 adet özel işlem biriminden oluşur. Bu sayede çoklu çekirdek grubunda ayni anda 2 warp, toplamda 64 iş parçacığı paralel bir şekilde çalışabilmektedir. Çoklu çekirdek grubu içerisinde aynı anda çalıştırılabilecek maksimum warp sayısı, bir warp'ın kullandığı yerel bellek ve yazmaç dosyasına bağlıdır. Warp gereksinimleri ise yazılan koda göre değişiklik göstermektedir.



Şekil 2.3: Fermi çoklu çekirdek grubu mimarisi

Fermi ailesinin bellek hiyerarşisi Şekil 2.4`deki gibidir. Ayni warp içerisindeki iş parçacıkları ya da ayni çoklu çekirdek grubuna atanmış warplar, çoklu çekirdek grubu özelindeki L1 önbelleği ve paylaşımlı bellek sayesinde haberleşmektedirler. Bunun haricindeki bütün iş parçacıklarının en kısa yol ile haberleşmesi GPU genelinde ortak kullanılan L2 önbellekleri üzerinden yapılmaktadır. L2 önbelleği GPU genelinde ortak kullanılan bir önbellek olduğundan dolay senkronizasyon gereksinimi olan tüm önbellek politikalarıyla beraber çalışabilmektedir. L2 önbelleğinin bir alt hiyerarşisi olarak DRAM bulunmaktadır.



Şekil 2.4 : Fermi bellek hiyerarşisi

2.3 Literatürde GPGPU Üzerine Güncel Çalışmalar

2.3.1 Önbellek duyarlı iş parçacığı grubu zamanlayıcısı

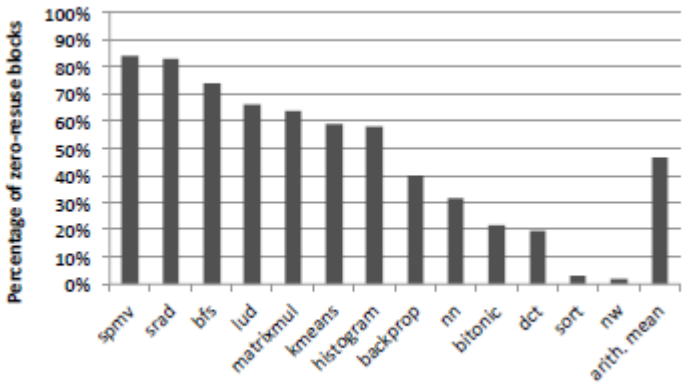
Kaynak kullanımını daha fazla kullanımı için çalışan toplam warp sayısını en yüksek limitte çalıştırmanın uygulama performansı için en uygun durum olmayabildiği olmadığı gösterilmiştir [7]. Warp sayısı ile önbellek kayıp sayısının belli durumlarda bağıntılı olduğu gözlemlenmiştir. Bunun sebebi ise limitli kaynakların kullanılmasında

yarışan warp sayısının artmasıdır. Warp sayısının artması ile kayıp sayısının artması arasında doğru orantı gözükse de sistemin performansı hakkında aynı şey söylenememektedir. Bunun sebebi ise warp başına düşen önbellek boyutunun azalması ve oluşabilecek potansiyel yerellik verilerinin yerellik zamanları gelmeden önbellekten atılmak zorunda kalmalarıdır.

Çalışan uygulamaların başarımını en yüksek noktaya çekebilmek için potansiyel yerellik kayıpları hesaplanarak ideal warp sayısı hesaplanmaktadır. Potansiyel yerellik kayıpları çıkarılan veri önbelleği (Victim Cache) benzeri bir yapı ile bulunmaktadır. Eğer yerellik kaybı belli bir eşiği geçmiyor ise warp sayısı arttırılmaktadır. Eğer yerellik kaybı belli bir eşiği geçiyor ise warp sayısı azaltılmaktadır. Bu sayede uygulamada koşan warp sayısı optimum seviyede tutulmaya çalışılarak uygulama başarımı arttırılmaktadır. Önerilen teknik ile yüksek ve orta önbellek duyarlılığı olan uygulamalarda ortalama %24 başarım artımı sağlanmıştır.

2.3.2 Adaptif GPU önbellek atlama

L1 veri önbellek verimliliği ve enerji kullanımını azaltmak amaçlı bir GPU önbellek yönetim tekniğidir. Tek sefer kullanılarak önbellekten atılan verileri engellemek amacıyla önbelleği atlayan dinamik kestirim yöntemidir.



Şekil 2.5 : Tekrar kullanılmayan L1 önbellek blok istatistikleri

Şekil 2.5'de önbelleğe alındıktan sonra bir daha kullanılmayan blokların tüm bloklara oranları gösterilmiştir [8]. Bu istatistiklere göre toplam blokların averajda %46'sı bir daha hiç kullanılmadan atılmaktadır.

Kestirim yöntemi parametre olarak bellek adresi yerine program sayacını kullanmaktadır. Sebep olarak ise GPU uygulamalarının önbellek blok erişimlerinde kullandıkları farklı adres sayısının, önbellek erişimlerinde kullanılan komutların program sayaçlarından çok daha az olmasıdır. Program sayacının hash değeri 128 satırlık ve her satırında 4 bitlik sayaca sahip olan tahmin tablosunu indeksleme için kullanılmaktadır.

Önbelleğe erişimde bir isabet olursa, isabet edilen satır üzerinde işlem yapan son program sayacı yerelliğe sahip olmuş demektir, bu yüzden satırdaki kayıtlı olan program sayacı tahmin tablosunda bulunur ve girdinin skoru yerelliğe sebep açtığı yönünde güncellenir. Eğer önbelleğe erişimde bir kayıp olursa, kayıp olan girdide kayıtlı olan program sayacının tahmin tablosundaki skoru yerelliğe sahip olmadığı yönde güncellenir. Kayıp olan istek için isteğin program sayacının tahmin tablosundaki skoru belli bir eşik skoruyla karşılaştırılır ve ona göre önbelleğe getirilip getirilmeyeceğine karar verilir.

Bu çalışmada sonuç olarak, önerilen tekniğin başarımı %13'e kadar arttırabildiği ve enerji kaybını %25 azaltabildiği söylenmiştir.

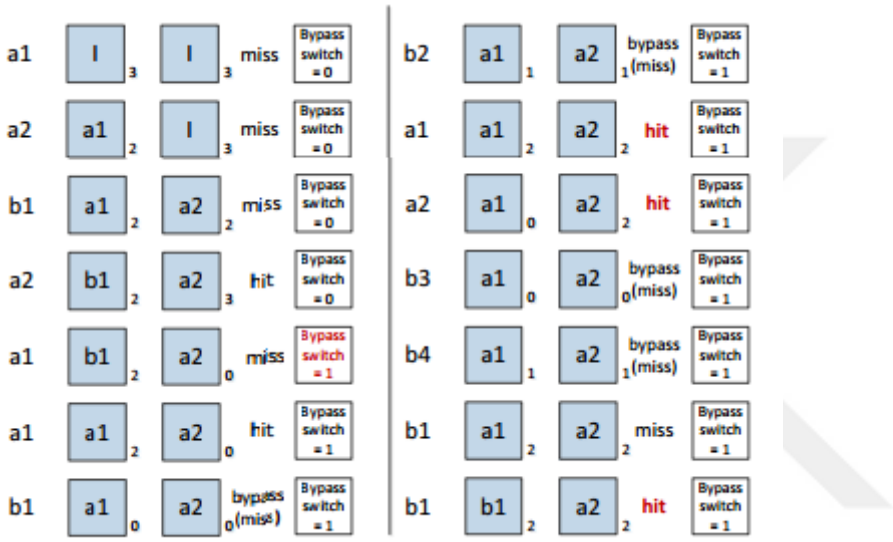
2.3.3 Enerji verimli GPU hesaplama için adaptif önbellek yönetimi

Bu çalışmada önbellek satırlarının tekrar kullanılma istatistikleri kullanılarak bir sonraki isteklerin önbellekte işlem görmesi ya da önbelleği atlama arasında karar verilmektedir [9]. Her önbellek satırına koruma uzaklığı adında sayaç konulur. Bu sayaç, o önbellek satırına olan erişimlerin kaç tanesinden koruncağını göstermektedir. Önbelleğe yeni girilen ya da yerellik oluşturan satırın sayacı daha önceden belirlenen bir eşiğe ayarlanırken diğer erişimlerde bir azaltılır. Önbellek satırları sadece 0'a eşit sayaca sahip olduklarında atılabilir olmaktadır. Eğer önbellekteki bütün satırlar korunuyor ise, gelen istekler önbellekte işeme alınmadan atlanmaktadır. Önerilen tekniğin yüksek derecede önbelleğe duyarlı uygulamalarda başarımı averajda %42 arttırdığı gözlemlenmiştir.

2.3.4 Çok çekirdekli mimariler için adaptif önbellek atlama

GPU'larda paralel çalışan fazla sayıda warplar aynı anda sınırlı L1 önbelleklerinde alanına erişmeye çalıştıklarında, yerellik amaçlı önbelleklere eklenen veriler

kullanılmadan önbellekten atılmaktadır. Bu durumlarda önbelleklerin atlanması başarıma fayda sağlayabilmektedir. Bu çalışmada L1 önbellek setlerine ekstra olarak "bypass" biti, L2 önbelleklerine ise "victim" bitleri eklenmiştir [10]. L2 önbelleğinde bulunan bir veriyi L1 önbelleklerinden biri art arda ister ise L1 önbelleğinde veriler kullanılmadan çıkarıldığı yorumlanarak, L2 "victim" bitlerinden o L1 önbelleğine denk gelen bit kaldırılır. L2'den L1'e gönderilen istek cevabı ile "bypass" bit bilgisi de yollanarak, daha sonra L1 önbelleğine gelecek isteklerin atlanmaları sağlanır.



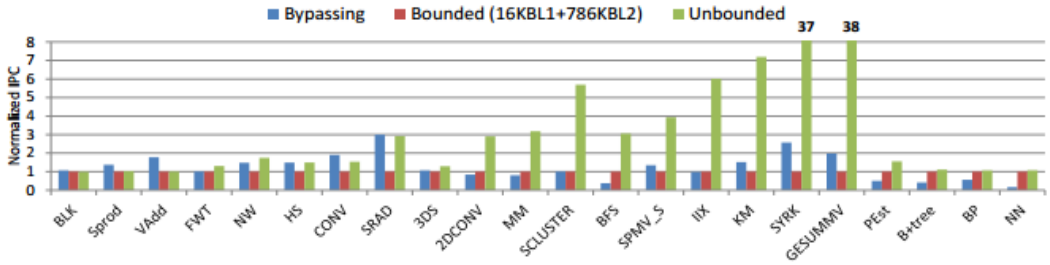
Şekil 2.6 : 2 Yollu L1 önbellek atlama örneği

Şekil 2.6'de önerilen yöntemin çalışma durumundaki örneği verilmiş. Kutuların sol taraflarında bulunanlar önbelleğe gelen erişimler, kutular ise önbelleğin farklı setleridir. Kutuların sol alt köşelerinde bulunan sayı ise önbellek satırındaki verinin en son ne zaman kullanıldığını gösterir. Birinci ve ikinci adımlarda, a1 ve b2 istekleri önbelleğin boş satırına yazılır Üçüncü adımda b1 isteği önbellekte yer olmadığı için a1 isteğinin yazıldığı önbellek satırına yazılır. Dördüncü adımda a2 isteği önbellekte bulunduğu için en son kullanılma zamanı güncellenir. Beşinci adımda a1 isteği daha önce L1 önbelleğinde bulunduğundan dolayı bu bilgi L2 önbelleğinde vardır ve tekrar aynı istek geldiği için L2 tarafından yollanan atlama bilgisiyle atlama bilgisi güncellenir. Bu adımdan sonra gelen her kayıp isteğinde atlama yapılır ve her atlama başına önbellek satırlarının en son kullanılma zamanları bir arttırılır. Eğer en son kullanılma zamanı 2 eşiğini geçer ise atlama yerine en son kullanılan önbellek satırına istek yazılır.

Bu çalışmanın amacı CCWS [7] ile aynı gibi gözükse de çözüme gidişte warp sayısını limitlemeden ve VTA gibi harici bir donanım yerine önbellek satırlarına eklenen bitler kullanılmıştır. Uygulamalarda ortalama %16,1 başarım artışı sağlanmıştır.

2.3.5 GPGPU önbellek mimarisinin verimli kullanımı

Bu çalışmada önbelleklerdeki sıkışma durumlarına 3 farklı tekniğin birleştirilmesiyle çözüm aranmıştır [11].



Şekil 2.7 : Uygulamalarda önbellek duyarlılığı

Şekil 2.7'de görüldüğü üzere sınırsız, sınırlı ve baypas durumlarında önbelleklerin başarıma etkisi gösterilmiştir. Bazı uygulamalarda önbelleklerin olmaması tercih edilirken bazı durumlarda önbelleklerin olması tercih edilmektedir. Önbelleklerin olmamasının tercih edildiği uygulamalardaki gözlemler %99'a varan yüksek seviyelerdeki önbellek kayıp oranıdır. Bu yüzden de dinamik olarak önbelleğin baypas edildiği bir teknik seçilmiştir. Belli örnekleme aralıklarıyla önbellek kayıp oranlarına bakılarak belli bir eşik değeriyle karşılaştırmaktadırlar. Örneklenen kayıp oranının eşik değeri geçtiği zamanlarda önbellek baypas edilirken eşiğin altında kaldığı durumlarda ise önbellek aktif olarak çalıştırılmaktadır.

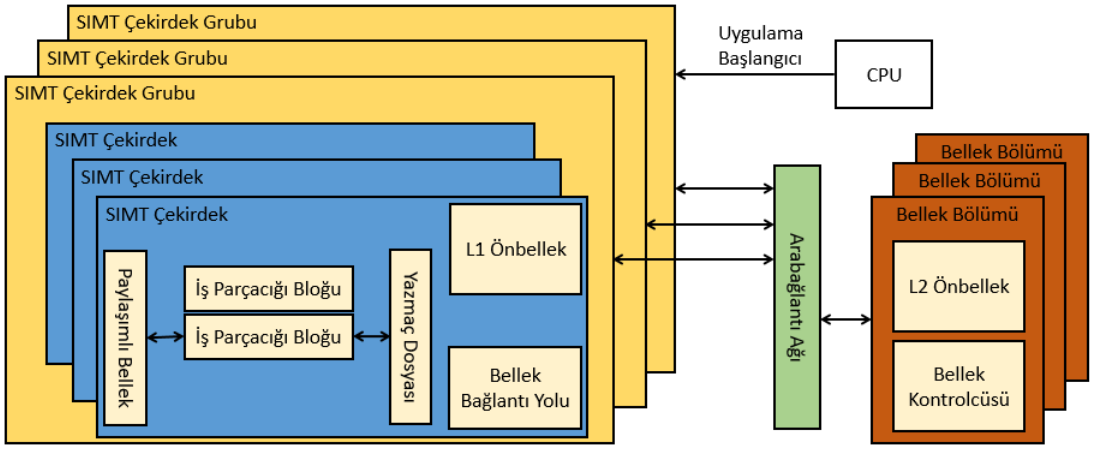
İkinci teknik olarak GPU'larda problem olduğu bilinen önbellek çekişmesi durumuna CCWS [7] çalışmasındaki tekniğin daha basit hali çözüm olarak gösterilmiştir. Bu durumda da belli örnekleme aralıklarıyla L1 önbelleklerinin kayıp oranlarına bakılmaktadır. Eğer N adet örnekleme aralığında alınan kayıp oranları art arda bir eşiği geçiyor ise, her GPU çekirdeğinde farklı sayıda warp çalıştırılmaya başlanır. Belli bir süre sonra farklı sayıda warp çalıştıran çekirdeklerin bu süre içerisinde toplam kaç adet komut çalıştırıldığına bakılarak, en çok komutu çalıştırabilen çekirdeğin warp sayısı diğer çekirdekler için de geçerli kılınır. Bu sayede önbellek çekişmesi durumlarında optimum warp sayısı uygulamanın çalışma anında bulunmaya çalışılmıştır.

3 GPU SİMÜLATÖRÜ: GPGPU-Sim

GPGPU-Sim, detaylı zamanlama raporlamaları yapabilen bir GPU simülatör yazılımıdır [12]. Raporlamalarını saat periyodu çözünürlüğünde yapmaktadır. GPGPU-Sim simülatörü CPU ve GPU kodlarından oluşan programları çalıştırmaktadır. CPU ya da PCI Express zamanlamaları dışındaki bütün GPU mikro mimari hakkındaki zamanlamaları göstermektedir. NVIDIA GeForce 8x, 9x ve Fermi serilerinde kullanılan GPU mikro mimarileri modelleyebilmektedir. Simülatör doğruluğunu yazarları tarafından RODINIA kütüphanesi uygulamaları kullanılarak gerçek NVIDIA GPU'ları ile simülatörün çıkardığı IPC (Instructions per Cycle) karşılaştırılarak %98,3 benzerlikte sonuç çıkardığını gözlemlemişlerdir.

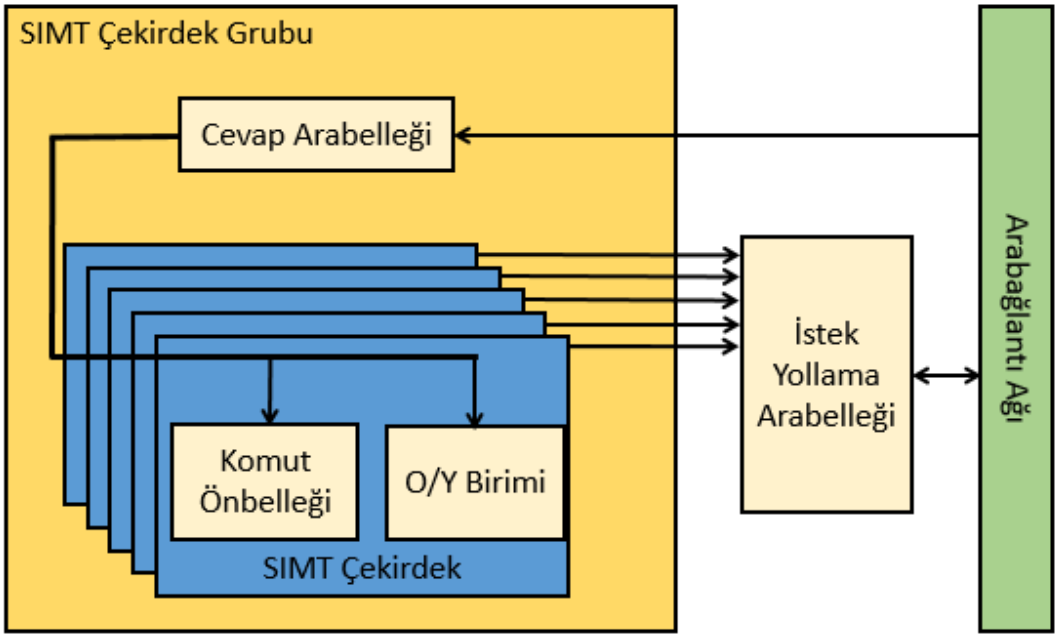
GPGPU-Sim ile benzetimi istenilen GPU özellikleri simülatöre özel bir yapılandırma dosyasıyla gösterilmektedir. Yapılandırma dosyası farklı saat alan frekansları, bellek kontrolcü sayısı, zamanlayıcı tipi, önbellek ve bellek boyutları gibi birçok GPU parametresinin değiştirilmesine izin vermektedir. Farklı özelliklere sahip olan GPU'ların yapılandırma dosyaları ile beraber gelen GPGPU-Sim, farklı GPU'ları (GTX480, Quadro FX5600, Quadro FX5800, Tesla C2050) simülasyonunu yapmamıza olanak sağlamaktadır. Hali hazırda bulunan spesifik GPU yapılandırma dosyaları istekler doğrultusunda değiştirilerek, bu değişikliklerin GPU üzerindeki etkisinin gözlemlenebilmesi akademik araştırmalar için uygun bir ortam oluşturmaktadır. Bu tez kapsamında GTX480 yapılandırma dosyaları kullanılmıştır.

Simülatör mimarisi Şekil 3.1'de verilmiştir. Bu mimaride 4 farklı saat alanı bulunmaktadır: SIMT Çekirdek Grubu saat alanı, arabağlantı ağı saat alanı, L2 önbellek saat alanı ve DRAM saat alanı. Farklı saat alanları birbirlerine göre asenkron olarak çalışmaktadır ve alanlar arası geçişler arabelleklerle sağlanmaktır.



Şekil 3.1: GPGPU-Sim mimarisi

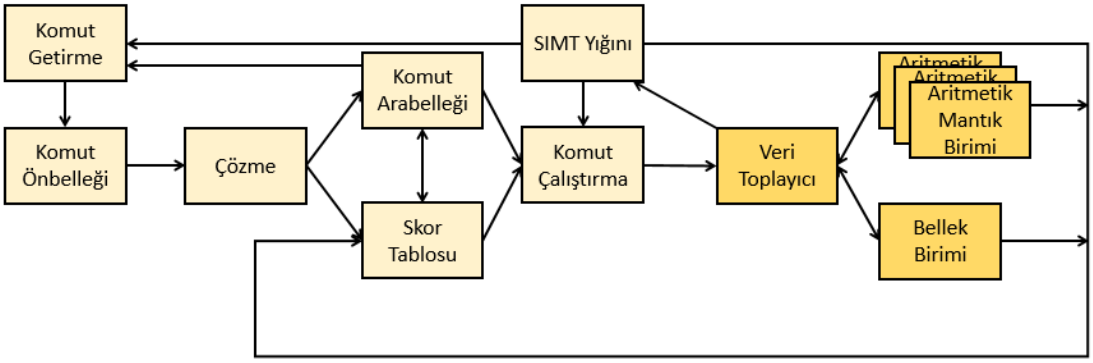
SIMT Çekirdek birimleri gruplara ayrılarak SIMT Çekirdek Grubu birimleri oluşturulmuştur. Her SIMT Çekirdek Grubu biriminin kendine özel Cevap Arabelleği bulunmaktadır. Cevap Arabelleği içerisinde bulunan veriler SIMT Çekirdek birimlerinin komut önbelleği ya da bellek işlemlerinin yapıldığı Okuma/Yazma birimlerine çekilmektedir. Okuma/Yazma birimleri tarafından üretilen bellek istekleri için SIMT Çekirdek Grupları içerisinde SIMT Çekirdekleri tarafından ortak olarak kullanılan İstek Yollama Arabelleği bulunmaktadır.



Şekil 3.2: GPGPU-Sim çekirdek kümesi mimarisi

Şekil 3.2'de SIMT çekirdek mimarisi verilmiştir. Komut Önbelleği bloğu, çalıştırılacak komutları depolamak için kullanılır. Komut Arabelleği girdileri sırasıyla

geçerli biti, çözümlenmiş komut ve hazır bitinden oluşmaktadır. Komut Arabelleğinin içerisinde geçerli bir komut girdisi bulunmadığı durumlarda Komut Önbelleğinde bulunan hazır bir komut çekilir, çözümlenir ve Komut Arabelleğine yollanır. Geçerli biti, girdinin geçerli bir komuta sahip olduğunu gösterirken, hazır biti çözümlenmiş komutun arabellek içerisinde olduğunu ve çalıştırılmak için hazır olduğunu gösterir. Hazır biti işaretli komutların çalıştırılmasından önce herhangi bir çakışma olup olmadığına bakmak için skor tablosuna bakılır. Çakışma bulunmadığı ve yeterli donanım kaynağı bulunduğu durumlarda komut çalıştırılabilir.



Şekil 3.3: SIMT çekirdek mimarisi

Komut arabelleği içerisinde komutu bulunmayan warplar komut önbelleğine erişim için warp zamanlayıcı yardımıyla sırasıyla seçilirler. Zamanlayıcı tarafından seçilen warp'ın Komut Arabelleği içerisindeki geçerli biti aktifleştirilir.

Komut Önbelleği sadece okunabilir, tıkanmasız ve küme ilişkilendirmeli bir önbellektir. Bu önbelleğe gelen istekler isabet durumunda çözümleme birimine yollanmaktadır. İstekler kayıp ile sonuçlandığı durumlarda Miss Status Holding Register(MSHR) biriminin dolu olup olmamasına bakılır. Dolu ise başarısız rezervasyon hatası alınır, dolu değil ise alt bellek hiyerarşisine Komut Önbellek tarafından okuma isteği yollanılır ve bu istek takibi için gerekli bilgiler MSHR birimine yazılır.

MSHR, önbelleğe isabet etmeyen işlemlerin bir sonraki bellek işlemlerine tıkanmadan devam edebilmesi için kullanılmaktadır ve bu donanımı kullanan sistemlerdeki önbelleklere tıkanmasız önbellek denir. Sınırlı sayıda girdiden oluşan bu donanımın girdileri tam ilişkilidir. Girdilerinde önbelleğe gelen kayıp isteklerinin fiziksel adresi ve hangi verilerin geçerli olduğu bayraklar gibi bilgiler tutmaktadır. Aynı blok

adresine gelen istekleri birleştirme mekanizmasına sahip olmakla beraber tamamlanmamış istekleri girdilerinde tuttuğu için komut sırasına uygun şekilde programın çalıştırılmasını da sağlar.

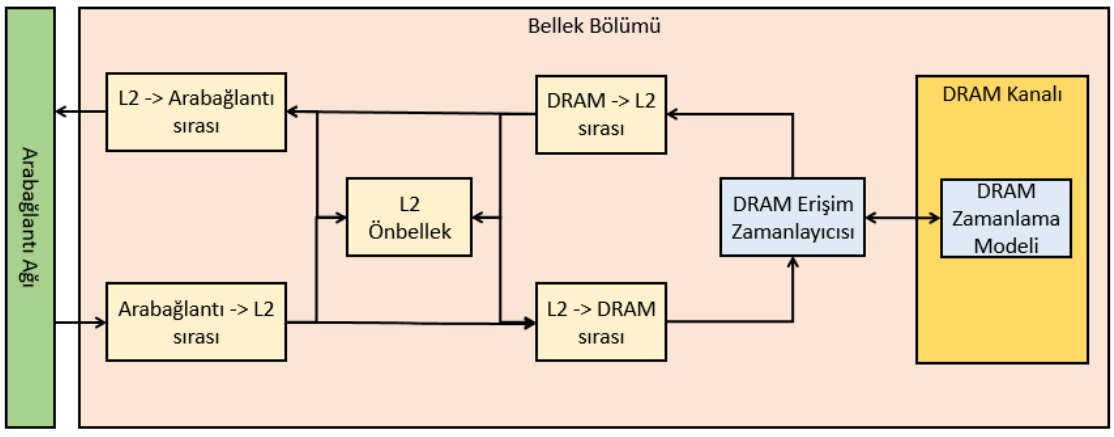
Bir sonraki Komut Çalıştırma kısmında Warp Zamanlayıcı ile Komut Arabelleğinde hazır warplar sırasıyla seçilir. Bir warp'ın zamanlayıcı tarafından seçilebilmesi için bariyerde beklememesi, Komut Arabelleği içerisinde geçerli bir komuta sahip olması, skor tablosu denetlemesinden geçebilmesi ve komutun çalıştırılacağı bellek ya da aritmetik mantık birimlerinin dolu olmaması gerekmektedir.

SIMT Yığını, dallanma durumlarında dallanma ile ilgili bilgilerin yazıldığı yığındır. Bu sayede dallanma ile ilgili bilgiler iş parçacıklarının birleşme zamanında geri sağlar. Dallanma olmadığı durumlarda program bir sonraki program sayacı adresinden devam etmektedir. Dallanma durumlarında bir sonraki program sayacı, aktif iş parçacığı maskelerini ve birleşme sonrasında gidilecek program sayacını yığın içerisine saklamaktadır.

Skor tablosu yakın zamanda üzerine yazılacak fakat hala yazılmamış yazmaçları kayıtlayarak çalıştırılacak komutların yazma sonrası yazma ya da okuma sonrası yazma gibi komut bağımlılık risklerini engellemektedir.

Veri Toplayıcı birimi komutların çalışması için gerekli verilerini getirerek komutu çalıştırmaya hazırlamaktadır.

SIMT Çekirdek Gruplarının bellek üniteleri ile haberleşmesi Arabağlantı Ağı yardımıyla yapılmaktadır.



Şekil 3.4: GPGPU-SIM bellek kontrol birimi

Bellek Kontrol Birimi Şekil 3.4'deki gibidir. Arabağlantı Ağı tarafından gelen paketler Arabağlantı-L2 sırasına konulur. L2 önbelleği ortak son seviye önbellek olduğundan dolayı tüm SIMT Çekirdek Grupları tarafından paylaşılmaktadır ve her saat periyodunda bir isteği işleyebilmektedir. Eğer L2 önbelleğinde istenilen veri bulunursa, gelen isteğe yollanılacak paket L2-Arabağlantı sırasına konulur. Diğer zamanlarda ise paket L2-DRAM sırasına konulur. Sıradan çekilen paketler DRAM Erişim Zamanlayıcısıyla seçilerek DRAM'e yollanır. GDDR3 zamanlamalarıyla modellenen DRAM'den gelen cevaplar DRAM-L2 sırasına yollanır. Kullanılan önbellek politikasına göre DRAM-L2 sırasındaki cevaplar L2 önbelleğine yazılıp Arabağlantı Ağı tarafına geri yollanılır.

Simülatör üzerinde çalıştırılan uygulamalar Virginia Üniversitesinde geliştirilen Rodinia, Nvidia tarafından CUDA geliştirme ortamıyla beraber gelen CUDA SDK ve GPGPU-Sim için üretilmiş 2009 ISPASS konferansında kullanılan kütüphaneler kullanılmıştır.

4 ÖNBELLEK YAZMA POLİTİKALARI

Önbellek yazma politikaları, bellek erişim isteğinin isabet ya da kayıp durumlarında farklı tanımlanırlar. Önbelleğe gelen yazma isteği ile önbellekte aynı adrese sahip bir girdi bulunduğu durumlara isabet durumu denilmektedir ve veri isabet eden girdiye yazılabilir. Eğer gelen yazma isteğinin adresine sahip herhangi bir önbellek girdisi bulunmuyorsa, bu duruma kayıp denmektedir.

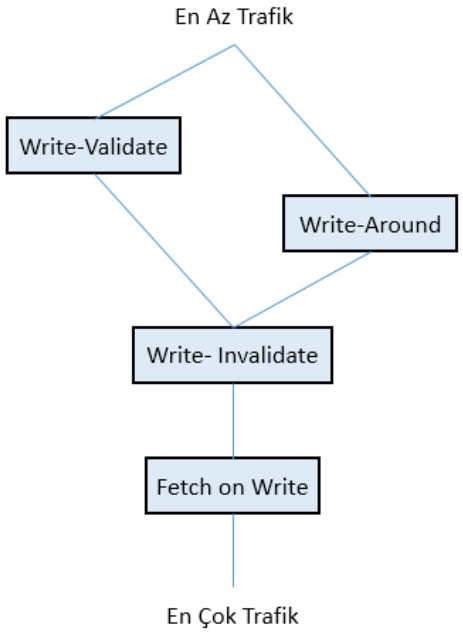
İsabet durumlarında, veri hem önbelleğe hem de alt bellek hiyerarşisine yazılabilir ve buna write-through politikası denir. Verinin sadece önbelleğe yazıldığı durumlara ise write-back politikası denir. Bu durumda önbellekteki girdileri kirli olarak tanımlamak amacıyla fazladan bit yeri gerekmektedir. Bu politikanın write-through politikaya göre avantajı ise yazma trafiğini daha da azaltabilmesidir [13]. Bu yüzden üzerinde çalışılan L2 önbelleklerinde isabet durumunda write-back politikası kullanılmıştır.

Kayıp durumlarında politika seçenekleri Fetch-On-Write, Write-Allocate ve Write-Validate kombinasyonlarından oluşmaktadır. Fetch-On-Write politikası ile kayıp isteklerine ait adresin verisi alt bellek hiyerarşisinden çekilir. Write-Allocate politikası ile kayıp isteği için önbellekten bir girdi ayrılır. Write-Invalidate politikası ile kayıp isteğinin yazılacağı önbellek girdisindeki eski veriler geçersiz olarak işaretlenir. No-Fetch-On-Write, Write-Allocate ve No-Write-Invalidate kombinasyonuna Write-Validate denmektedir. Write-Validate ile verilerin yazılacağı önbellek girdisinin sahip olduğu eski veriler geçersiz olarak işaretlenir ve yeni veriler geçerli olarak işaretlenerek önbellek girdisine yazılır. No-Fetch-On-Write, No-Write-Allocate ve No-Write-Invalidate kombinasyonuna Write-Around denmektedir. Write-Around kullanıldığı zaman veriler önbelleğe yazılmadan bir alt bellek hiyerarşisine yollanmaktadır. Fetch-On-Write ya da Write-Allocate birbirlerinden ayrıl kullanıldıklarında bir anlam ifade etmedikleri için literatürde bu ikisinin kombinasyonu kısaca Fetch-On-Write ya da Write-Allocate olarak geçmektedir. Bu çalışmanın devamında da Write-Allocate olarak kullanılacaktır. Write-Allocate ile önbelleğe yazılacak veri için önbellekte bir girdi ayrılır, eksik veri kısımları bir alt

bellek hiyerarşisinden çekilir ve yazılacak veri ile birleştirilip bu girdiye yazılır. Diğer kombinasyonlar anlamsız oldukları için tanımlı değillerdir.

Write-Invalidate veya Write-Validate politikasının kullanılması için adreslenebilen her veri büyüklüğü için 1-bit geçersiz bayrağı bulunmalıdır, bu da bayt adreslenebilir yapılar için %12,5 ek önbellek alanı demektedir. Günümüz GPU önbelleklerinin altyapısında bu politikalar için gereken geçersiz bitleri bulunmadığından dolayı pratikte kullanılamamaktadır.

No-Fetch-On-Write politikasının kullanıldığı politikaların Fetch-On-Write politikasına göre önemli bir avantajı yazma kayıplarını azaltabilmeleridir. Fetch-On-Write ile isabet etmeyen herhangi bir isteğin verisi ileride kullanılma garantisi olmaksızın bir alt bellek hiyerarşisinden çekilmektedir. Bu yüzden No-Fetch-On-Write politikası ile herhangi bir verinin alt bellek hiyerarşisinden çekilmesine gerek duyulmayan yazma kayıplarına kurtarılmış kayıp denmektedir. Bu tanımdan yola çıkarak Jouppi'nin yaptığı çalışmada yazma kayıplarında toplam oluşturulan trafik aşağıdaki gibi sıralandırılmıştır.



Şekil 4.1: Önbellek yazma politika trafik sıralaması

Bu çalışmada Write-Around ve Write-Allocate politikaları kullanılmıştır. Bu politikalarının birbirlerine göre avantajları belli durumlarda ortaya çıkmaktadır.

Fetch-On-Write politikası ile yazma isteklerinin kısa bir süre sonra okunması gerektiği durumlarda, yazılan verilerin önbellekte bulunmasından dolayı okuma isabetlerini arttırabilmektedir.

Write-Around politikasının Write-Allocate politikasına göre avantajlarından biri yazma kayıplarının oluşturduğu trafiği azaltabilmektir. Diğer bir avantajı ise yazma kayıplarını bir alt bellek hiyerarşisine gönderdikten sonra işlemlerine devam edebilmesidir. Write-Allocate politikasında ise herhangi bir yazma kaybı sonrasında alt bellek hiyerarşisinden gerekli verilerin gelmesi beklenir. Write-Allocate politikasının avantajı ise yazılan verileri ardından okuyan uygulamalarda verilerin önbellekten okunmasından dolayı başarımı arttırmaktadır.

Politikaların birbirlerine göre gösterilen bu avantajları farklı durumlarda ortaya çıkabilmektedir. Bu durumda da donanım ve üzerinde koşan uygulamanın isteklerine göre hangi politikanın daha iyi olduğu değişiklik gösterebilir.

Okuduğu verileri işleyerek belleğe geri yazan ve geri yazdığı verileri bir sonraki döngüde kullanmayan bir programda Write-Around önbellek politikası kullanılması mantıklıdır. Eğer program döngüye sahip ve her döngüde bir önceki döngünün sonuçlarını kullanıyor ve bu verileri depolayabilecek yeterli büyüklüğe sahip bir önbelleğe var ise bu durumda da Write-Allocate politikasının kullanılması uygun olabilir. Verilen bu iki farklı uygulamaların bellek erişim modelleri, bellek erişim spektrumunun uç taraflarını göstermektedir. Günümüzde kullanılan uygulamaların çoğu bu spektrumun orta taraflarında bulunmaktadır. Yani uygulamaların belli fazlarında yolladıkları yazma isteklerini bir daha okumamaktayken, diğer fazlarında yolladıkları yazma isteklerinin ardından aynı verileri okuma istekleri yollayabilmektedirler. Bazı durumlarda ise yollanılan yazma isteklerinin belli bir kısmını okurken diğer kısmını okumamaktadırlar.

Eğer uygulamanın bellek erişim davranışı önceden bilinebilir ise hangi yazma isteklerin Write-Allocate politikası ile önbellekte saklanması gerektiği ve hangilerinin ise Write-Around politikası ile bir alt bellek hiyerarşisine yollanması gerektiği bilinerek mümkün olan en iyi başarım sağlanabilir. Yani bütün program boyunca sabit bir politika yerine, uygulamanın fazına uygun bir önbellek politikası kullanılır ise başarımı arttırmak mümkündür.

Uygulamanın bellek erişim davranışının önceden bilinemediği durumlarda bazı kestirimlere gidilebilir. Örneğin "Exploiting Inter-Warp Heterogeneity to Improve GPGPU performance" çalışmasında farklı uygulamalardan alınan farklı warp'ların isabet oranları bir milyon saat periyotlarıyla örneklenmiştir [14]. Sonuçta warp'ların isabet oranlarının değişiminin yüz bin saat periyodu kadar süre aldığı görülmüştür. Yani warp'ların bellek erişim karakteristiklerinin değişimleri uzun süreler içerisinde gerçekleşmektedir. Bu yüzden uygulamanın belli bir noktasında örneklenen bellek erişim istatistiklerinden, uygulamanın geleceği hakkında bir kestirim yapılabileceği ortaya çıkmaktadır. Örneğin, uygulamanın yazma isteklerinin daha sonradan geri okunulup okunulmadığına bakılarak uygulamanın yakın geleceğinde hangi politikanın kullanılması gerektiği tahmin edinilebilir.

CCWS çalışmasında warp sayısının artması ile önbellek sıkışması arasında doğru bir orantı bulunmuştur [7]. Uygulamaların en iyi verimde çalışabildiği warp sayısını bulabilmek için uygulama farklı sayıda warplar ile çalıştırılmıştır. Warp'ların statik olarak değiştirildiği bu teknik ile warp sayısının optimal sayıdan sonra daha da arttırılmasının önbellek sıkışmasına ve bunun da başarımı azalttığı gözlemlenmiştir. Önbellek sıkışmasının dinamik olarak belirlenerek warp sayısının uygulamanın koşma esnasında değiştirilebileceği gösterilmiştir. Önbellek sıkışması bulunması için VTA (Çıkarılmış Veri Etiket Dizesi, Victim Tag Array) adında bir donanım kullanılmıştır. VTA, önbelleklerde isabeti arttırmak için kullanılan Çıkarılmış Veri Önbelleği donanımının değişime uğramış halidir [15]. Önbellekten çıkarılan girdilerin etiketleri bu donanım içerisinde aranarak yerellik kaybı olup olmadığı bulunarak önbellek sıkışmasının ne kadar olduğu tahmin edinilerek warp sayısını önbellek sıkışması olmadığı noktaya dinamik olarak ayarlanmaya çalışmışlardır.

5 DİNAMİK POLİTİKA

Bu çalışmada sunulan dinamik politikada seçilen önbellek yazma politikalarının çalışma şekli, yerelliklerin yakalandığı VTA donanımı ve bu donanım uygulanan operasyonlar, politika seçimi için referans alınan skorlama sisteminin çalışma prensipleri ayrı ayrı anlatılmıştır.

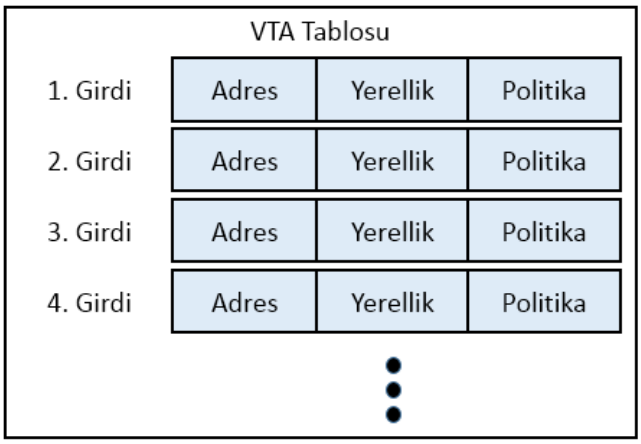
5.1 Önbellek İşlemleri

WA politikası ile istekle eşleşen bir önbellek satırı olup olunmadığına bakılır. Eşleşme yani isabet durumunda istek yazma ise satıra yazılı, okuma ise cevap önbellekten geri yollanır. Bir eşleşme bulunamadığı durumda çıkartılacak bir önbellek satırı aranır. Bütün satırlar rezerve edilmiş ise rezervasyon yapamama durumu döner. Rezerve olmayan çıkarılmaya müsait bir satır olduğu durumunda hem yazma hem de okuma isteklerinde bir alt bellek hiyerarşisinden okuma yapılacağı için MSHR (Miss-Status-Handling-Register) kaynaklarına bakılır [16]. Eğer aynı istek MSHR içerisinde bulunuyorsa istekler birleşir ve isabet rezervasyonu durumu dönülür. Eğer aynı istek MSHR içerisinde yok ise boş MSHR satırı aranır. Boş satır yok ise rezervasyon yapamama durumu dönülür. Boş satır var ise okuma durumunda satır istek bilgileriyle doldurulup okuma isteği bir alt belleğe yollanır. Yazma durumlarında ise simülatörün orijinal davranışı değiştirilmiştir. Orijinal kaynak kodunda yazma isteklerinde veriler bir alt bellek hiyerarşisine yollanır ve bir alt bellek hiyerarşisinde güncellenen veriler geri okunmaktadır. Böyle bir davranış WA politikasının mantığına aykırıdır çünkü WA davranışı ile yazılacak veriler önbellekteki bir satıra yazılarak, alt bellek hiyerarşisine yollanan toplam yazma trafiği azaltılmak amaçlanır. WA politikası ile bir alt bellek hiyerarşisinden yazma komutu içerisindeki sadece eksik bilgiler çekilmelidir. Bu sayede yazma isteğinin içerisindeki veriler MSHR veri tamponlarında tutulurken eksik verilerin çekilmesi için okuma isteği bir alt bellek hiyerarşisine yollanılır ve önbellek yazma işleminde takılı kalmadan bir sonraki isteklere devam edebilir. Eğer yazılan veride bir eksiklik yok ise bir alt bellek hiyerarşisine okuma isteği yollamaya da gerek kalmamalıdır. Hem okuma hem de yazma işlemlerinden sonra önbellekten rezerve edilen satırda daha önce kirli bayrağı kaldırılmış bir veri var ise alt bellek hiyerarşisine yollanır.

NoWA politikasının WA politikasına göre davranış farkı sadece yazma kayıplarında yaşanır. Yazma kaybı durumlarında önbellekten bir satır rezerve edilmek yerine yazılacak veri direk olarak bir alt bellek hiyerarşisine yollanmaktadır.

5.2 VTA Yapısı ve Operasyonları

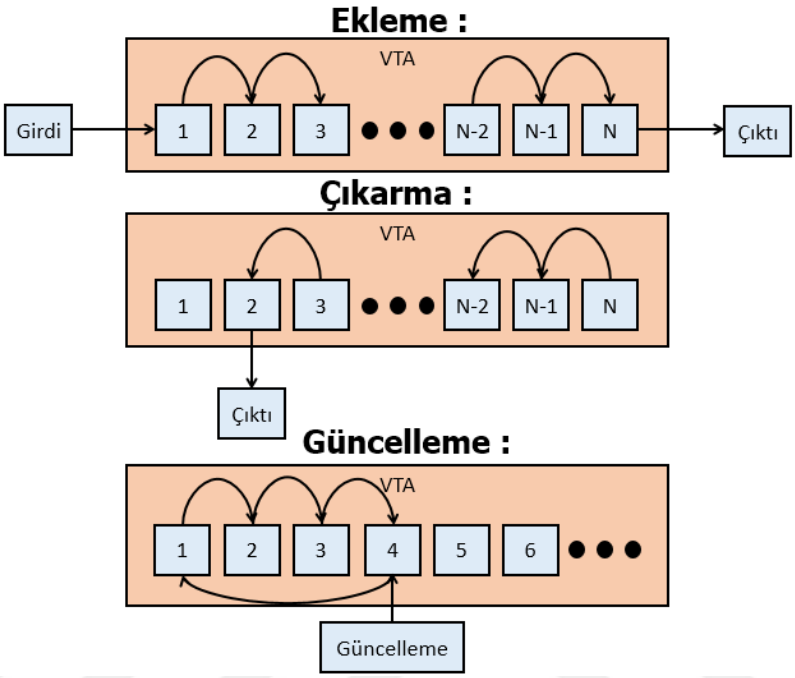
VTA ile program esnasında oluşan ya da oluşabilecek potansiyel yerelliklerin bulunması amaçlanmıştır. VTA tablosunu Çıkarılan Veri Önbelleğinden ayıran en büyük özellik VTA içerisinde verilerin depolanmamasıdır. Çıkarılan Veri Önbelleği içerisindeki girdilerde depolanan verinin boyutu önbellek boyutlarıyla aynıdır. GPU önbelleklerinde depolanan veri paralel çalışacak bütün iş parçacıklarının verilerine eşit olduğundan dolayı, GPU önbelleklerinin satırlarında depolanan veri doğal olarak fazladır. Örnek vermek gerekirse Fermi ve Kepler ailesinde bulunan GPU'lar 32 iş parçacığını paralel olarak çalıştırır, L1 ve L2 önbellek satırları sırasıyla 128 bayt ve 32 bayt veri depolayabilmektedir.



Şekil 5.1: VTA tablo yapısı

VTA tablo yapısı Şekil 5.1'de gösterilmiştir. Adres, yerellik ve politika bayraklarından oluşmaktadır. Adres alanına, bellek isteğinin etiket kısmı yazılmaktadır. Bu sayede 128 ve 32 baytlık önbellekler için kullanılan VTA tablosunun adres kısmında sırasıyla 10 ve 8 bitlik blok ofset bilgilerinin yazılmamasıyla alan kazancı olur. Adres bilgisi ile aynı isteğin kısa süreler içerisinde tekrarlanıp tekrarlanmadığı gözlemlenerek, oluşan ve oluşmayan potansiyel yerellikler kestirilebilir. Yerellik bayrağı ile VTA içerisindeki istek girdisinin tekrarlanıp tekrarlanmadığı gözlemlenir. Tekrarlanan isteklerde bu bayrak kaldırılarak yerellik olduğu gösterilirken, VTA tablosundan çıkarılmaz zorunda kalan ve yerellik bayrağı

düşük olan girdiler ise yerellik olmadığı gösterilir. Politika bayrakları ise girdilerin hangi politika altında girildiğini gösterir, bu bilginin gerekliliği VTA karar akış şemasında gösterilecektir. VTA tablosunun uzunluğunun farklı değerlerle test edilerek, sisteme etkisi gösterilecektir.



Şekil 5.2: VTA tablosu işlemleri

VTA tablosunun 3 ana işlemi Şekil 5.2'de erilmiştir.

VTA tablosuna yeni bir girdi eklenmesi halinde, VTA donanımı FIFO davranışıyla girdilerini oynatır. Yeni girdi ilk pozisyona gelirken diğer girdilerin pozisyonu bir kaydırılır ve en sondaki girdi ise çıkartılmış olur.

VTA tablosundan bir girdi çıkarıldığı durumda, ondan önce eklenen girdiler, yani çıkartılan girdinin pozisyonundan sonraki girdiler bir geri kaydırılır.

VTA tablosunda bir girdi güncellendiği durumda, güncellenen girdi pozisyonundan önceki pozisyonlar bir kaydırılır, güncellenen girdi en başa getirilir ve girdinin yerellik bayrağı kaldırılır.

Yukarıda gösterilen mimariye sahi olan VTA'nın komutları belli okuma ve yazma işlemleri sonrası çağırılmaktadır. Bunun sonucunda belirlenen yerelliklere göre önbellek özelinde bir yerellik skoru güncellenmektedir. Bu skora göre ise uygulama için hangi politikanın daha uygun olduğu kestirilebilmektedir.

Hangi okuma ve yazma durumlarında VTA komutlarının kullanıldığı aşağıda açıklamalarıyla beraber listelenmiştir.

- Yazma Kaybı / WA:
    - MSHR İsabet: Önbellekte olmayıp MSHR içerisinde olması, bu isteğin kısa süre önce yollandığı ve MSHR içerisindeki girdi ile birleştirebileceği anlamına gelir. Daha önceki isteğin her iki politika ile yollanabilme ihtimali olduğundan dolayı VTA içerisinde hem WA hem de NoWA politika bayrakları ile girdi aranır. Girdinin bulunduğu durum yerellik olarak sayılır ve girdi güncellenir. Bulunmadığı çok özel olası durumda ise bu girdi VTA tablosuna WA politika bayrağı ile eklenir.

    - MSHR Kayıp: Önbellekte de MSHR içerisinde de olmayan istek VTA içerisinde her iki politika bayrağı ile aranır, bulunduğu durumda girdi güncellenir. Bulunmadığı durumda ise yeni bir girdi olarak eklenir. WA politikasından dolayı önbellekte bir ayrılan satır içerisinde kirli bayrağı kalkık bir girdi var ise, bu girdi VTA içerisinde aranır, bulunursa çıkartılır. Çünkü önbellekte olmayacağını bildiğimiz bir satırın yerelliğini VTA ile aramak gereksiz olur.

- Yazma Kaybı/NoWA:
    - MSHR İsabet: Önbellekte olmayıp MSHR içerisinde olan bir istek VTA içerisine sadece WA politikası ile alınabilir. Bu yüzden VTA içerisinde WA politika bayrağı ile istek aranır, bulunursa girdi güncellenir ve yerellik durumu sayılır. Girdi VTA içerisinde bulunmazsa, yeni bir girdi olarak VTA'ya eklenir.
    - MSHR Kayıp : Önbellekte ve MSHR içerisinde olmayıp VTA içerisinde sadece NoWA politikası ile olabilir. Bu yüzden VTA içerisinde NoWA politika bayrağı ile istek aranır, bulunursa girdi güncellenir ve yerellik durumu sayılır. Girdi VTA içerisinde bulunmazsa, yeni bir girdi olarak VTA'ya eklenir.

- Yazma İsabeti:
    - Yazma isteklerinin yazma isabeti ile oluşabilmesi için isabet edilen önbellek girdisinin WA politikası ile önbelleğe alınmış olması gerekmektedir. Bu yüzden istek sadece WA politika bayrağı ile VTA içerisinde aranır. Eğer VTA içerisinde bulunursa, VTA girdisi güncellenir.

- Okuma Kaybı:
    - MSHR İsabet: Kısa zaman önce sadece WA politikası ve yazma isteği ile bu durum oluşabilir. VTA içerisinde girdi WA politika bayrağı ile aranır.
    - MSHR Kayıp: Hem okuma kaybı hem de MSHR kaybı olduğu durumlarda girdi sadece NoWA politika bayrağıyla VTA içerisinde aranır WA politika bayrağı ile aranmama sebebi ise eğer WA ile bir yerellik olsaydı, okuma kaybı ve MSHR kaybı durumu olamazdı.
    - Devamında ise MSHR isabet/kayıp işlemlerinden bağımsız olarak eğer VTA içerisinde girdi bulunursa, yerellik bulundu sayılır, yerellik bayrağı kaldırılır ve girdi VTA'dan çıkarılır. Çıkarılma sebebi ise herhangi bir yazmadan sonra okuma yerelliğinin bulunmasından sonra bir daha yazma yerelliği aranmamasıdır. Çünkü o adrese yollanan okuma isteği, gerekli verilerin önbellekte olmasını garantilemektedir. Bu yüzden daha sonra o adres için bulunacak yerellikler en başta yollanan yazma isteğinden dolayı değil okuma isteklerinden dolayı gerçekleşecektir. Bunların üzerine kayıptan dolayı önbelleğe çekilecek veri için seçilen önbellek satırında kirli bir girdi varsa, bu girdi çıkarılacağından dolayı VTA içerisinde hem WA hem de NoWA politika bayrakları ile aranır, bulunur ise çıkarılır.

- Okuma İsabeti:
    - Sadece WA politika bayrakları ile istek VTA içerisinde aranması gerekir çünkü daha önce VTA içerisine konulan bir yazma adresinin okuma isabeti olarak ortaya çıkması bir tek WA politikasıyla verilerin önbelleğe alınmasıyla olabilir. VTA içerisinde bulunursa yerellik sayılır, yerellik bayrağı yukarı çekilir ve okuma kaybında söylenen aynı sebeplerden dolayı VTA'dan girdi çıkarılır.

5.3 Skorlama

Uygun politikanın seçilmesi amacıyla bir skorlama sistemi uygulanmıştır. Yerellik skoru adı verilen bu skorlama sistemi yerelliğin bulunduğu durumlarda arttırılırken, yerelliği bulunmadan VTA tablosundan atılmak zorunda kalan her girdi için ise düşürülür. Yerelliğin olup olmadığı ise yerellik bayrağına bakılarak anlaşılmaktadır. Bu sayede uygulamanın yerellikten yararlandığı fazlarında bu skor artarken, yerellikten faydalanmadığı noktalarda düşmektedir. Fakat bu şekilde kullanılacak olan bir skorlama sisteminde, uygulamanın ilk komutunun skorun etkisi en son komutun skorunun etkisi ile aynı olacağından kümülatif bir puanlamaya bakılmak zorunda kalınır. Bu yüzden de uygulamanın lokal olarak politika değişiklikleri için gerekli bilgi çıkarılamaz. Bu yüzden skor üzerinde yürüyen ortalama uygulanır. Örneğin son 20 güncellemenin toplamı belli bir eşiği geçer ise WA, altında kalırsa NoWA politikalarının yürütüldüğü bir skorlama sistemi uygulanmıştır. Eşik ile karşılaştırılan toplam güncelleme sayısı, uygulamadaki skorlamayı kaç adet isteğin etkileyeceğini tanımlar. Toplam güncelleme sayısı değiştirilerek, uygulamaların politika değişikliklerine olan tepki hızı değiştirilebilir. Aynı şekilde eşik değeri değiştirilerek, politika değişikliğinin kararlılığı değiştirilir. Örneğin her yerellikte 1 skor artarken, yerellik olmadığı durumlarda 1 skor düştüğünü düşünürsek, son 20 skor güncellemesinde net 10 skor artışı bekleniyorsa en az 15 yerellik olmadığı durum gerekir. Bu da skor güncellemelerinin yani isteklerin %75'nin yerelliğe sahip olmasını gerektirmektedir. Sonuç olarak yerellik skor değişimleri, toplam bakılan son güncelleme sayısı ve skor eşiği parametreleri ile jenerik bir kontrol sistemi oluşturulmuştur.

6 SONUÇLAR

6.1 Uygulama Çeşitleri:

GPU mimarisinde uygulamaların başarımını limitleyen iki adet kaynak vardır. Bunlardan biri toplam çekirdek sayısıdır. Eğer uygulamalar GPU çekirdeklerini tam zamanlı bir şekilde kullanıyorsa, bu uygulamaların başarımı çekirdek sayısından dolayı kısıtlanır. Böyle durumlarda eklenecek her çekirdek sayısı başına uygulamanın performansı artmaktadır. Bu uygulamalar hesap yoğunlukludur. Diğer bir yanda ise bellek bant genişliğinden dolayı başarımı kısıtlanan bellek yoğunluklu uygulamalar bulunmaktadır. Bu uygulamalarda GPU çekirdekleri veri beklerken boş durumda durmalarından dolayı çekirdek sayısının artması başarımı etkilememektedir [17]. Bellek hızı ya da paralel arayüzün genişletilmesi bu uygulamalarda başarım artışı sağlamaktadır.

Önbellek politikaları hesap yoğunluklu uygulamaların başarımlarını etkilemezken, bellek yoğunluklu uygulamaların başarımlarını etkileyebilmektedir. Bir uygulamayı hesap yoğunluklu olarak tanımlamak için uygulamanın başarımını bellek bant genişliği tarafından kısıtlanması gerekmektedir. Aşağıdaki tabloda bir uygulamanın hesap yoğunluklu olup olmadığına hem bellek bant genişliği kullanım oranı bellek frekansı değiştirilerek, hem de uygulama bellek izdüşümü girdi sayısı değiştirilerek incelenmiştir.

Tablo 6.1: Girdi sayısı ve DRAM frekansına bağlı BFS IPC değişimi

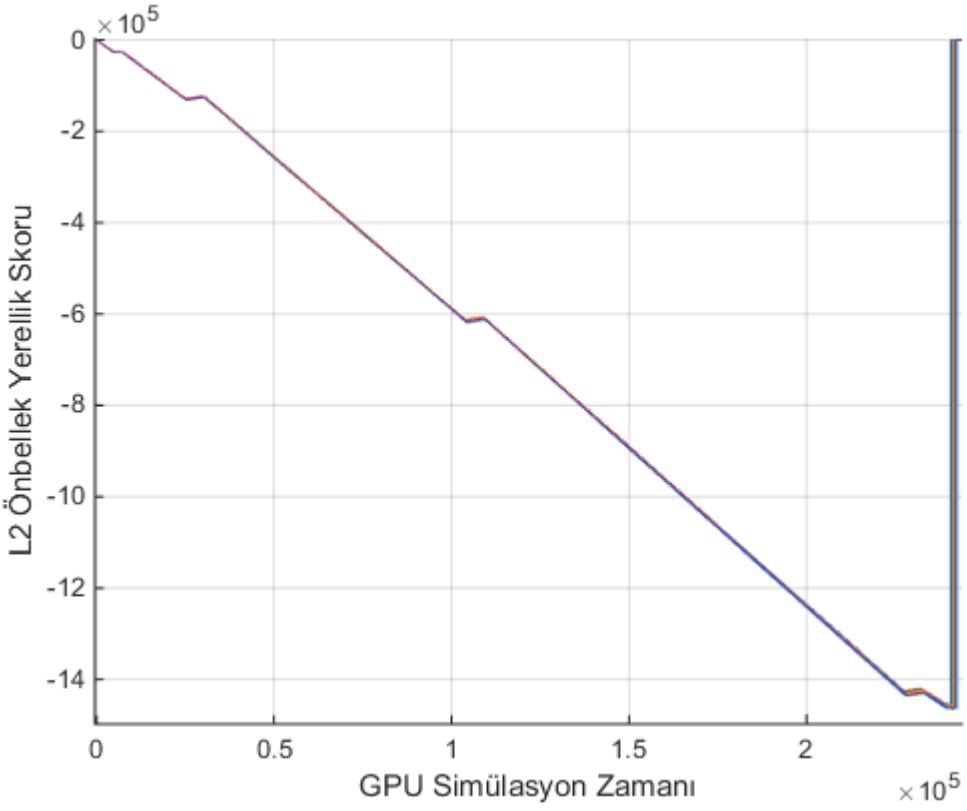
Eleman Sayısı	DRAM frekans	WA IPC	NoWA IPC	WA/NoWA IPC
1m	3600 MHz	51.4087	51,0464	1,007
1m	1800 MHz	51.3129	50.9566	1,0069
1m	900 MHz	47.2744	50.7842	0,9308
1m	100 MHz	6.3141	9.7996	0,6443
64k	100MHz	29.8613	25.2303	1,1835
4k	100MHz	26.6036	21.1262	1,2592

İlk 4 tablo girdisinde 1 milyon elemana sahip BFS uygulaması farklı DRAM frekansları ile koşturulmuştur. Frekansın düşürülmesiyle uygulamanın bellek yoğunluğu arttırılmaya çalışılmıştır. Bellek yoğunluğunun artmasıyla orantılı olarak politikaların başarıma katkı farklarının da artması beklentileri gözlemlenerek doğrulanmıştır. DRAM frekansı 1.8GHz ve üzerinde uygulamanın başarımına politika seçiminin etkisi yoktur çünkü başarımı asıl limitleyen kısım GPU çekirdeklerinin işlemlerden dolayı meşgul olmasıdır. DRAM frekansı 1.8GHz aşağısına indirildikçe politika aralarındaki farklar artmaktadır. Bu performans değişikliğini DRAM verimliliği gözlenerek de görülebilmektedir. DRAM verimliliği toplam çalıştığı saat periyotlarının ne kadarında okuma ya da yazma işlemi yaptığı olarak tanımlanmıştır. 1800MHz DRAM frekansında WA politikası ve NoWA politikası sırasıyla 0.35 ve 0.21 verimliliklere sahiptir. 900MHz DRAM frekansında WA politikası ve NoWA politikası sırasıyla 0.54 ve 0.32 verimliliklere sahiptir. DRAM verimliliği WA politikasında 0.35'den 0.54'e yükselmesi performansını etkilemeye başlamıştır fakat NoWA politikası için 0.32 verimlilikten 0.21 verimliliğe artması performansı etkilememiştir çünkü 0.32 verimlilik hala sistem için bellek yoğunluğu yaratmamaktadır. 100MHz DRAM frekansında WA politikası ve NoWA politikası sırasıyla 0.65 ve 0.55 verimliliklere ulaşmıştır. Bu noktada sistemin bellek bant genişliği fazlaca kullanılmaya başlandığı için politika seçimleri %50 gibi büyük başarım farkları oluşturmaya başlamaktadır.

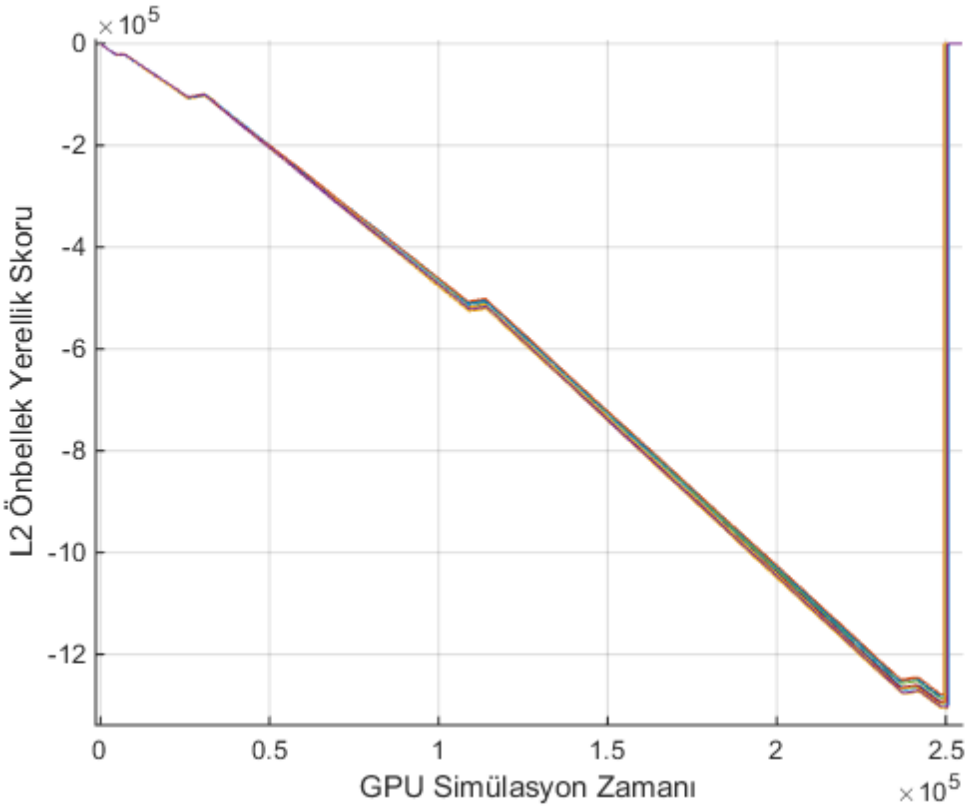
Tablonun son 3 girdisinde ise aynı DRAM frekansında farklı girdi boyutlarıyla BFS uygulamasının üzerinde önbellek politikaları karşılaştırılmıştır. Kullanılan simülasyon ortamında toplam 786KB L2 önbellek boyutuna sahiptir. BFS uygulamasında kullanılan önbellek boyutunun büyüklüğü kullanılması gereken önbellek politikasını değiştirilebilmektedir. Örneğin önbellek boyutunu fazlaca aşan 1 milyon girdi sayılı durumda NoWA politikası fazlaca daha çok başarım elde edebilirken, önbellek boyutlarına sığabilen girdilerle BFS uygulaması WA politikasında daha çok başarım elde edebilmektedir.

6.2 Önbellekler ve Politika Skorları Arası Davranış Benzerliği

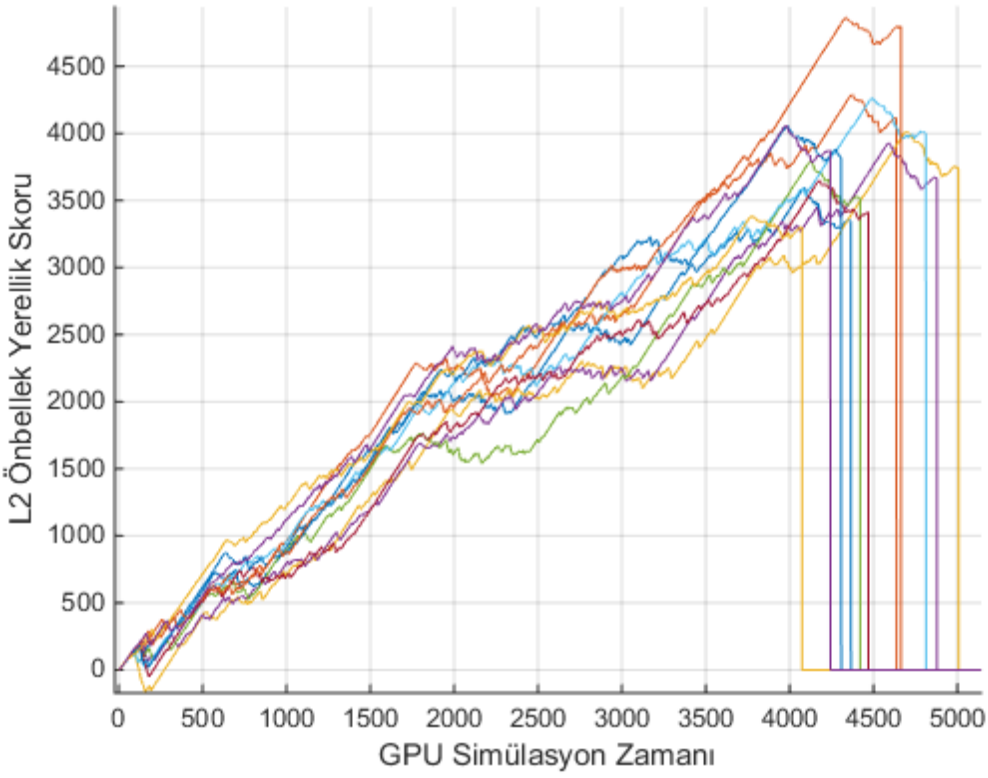
Aşağıdaki şekillerde 1M ve 64K elemanla beraber yürütülen BFS uygulamasının sırasıyla WA ve NoWA politikalarıyla toplanan L2 önbelleği başına yerellik skorları üst üste çizdirilmiştir. Yerelliğin eğilimini kümülatif olarak daha iyi görebilmek için politika seçimi için kullanılan yürüyen ortalama kullanılmamıştır.1M elemanla yürütülen BFS uygulamasında önbellekler neredeyse aynı hareket eden yerellik skorları gösterirken 64K elemanla yürütüldüğünde farklılık gösterebilmektedir. Bunun iki sebebi vardır. Birinci sebebi 1M ile yürütülen BFS uygulaması 250bin simülasyon saat döngüsünde tamamlanırken 64K ile yürütülen BFS uygulaması 5bin simülasyon saat döngüsü sürmektedir. Bu yüzden çekirdekler arasındaki küçük farklılıklar kısa süren uygulamada göze daha çok çarpmaktadır. İkinci sebep ise 64K elemana sahip BFS uygulaması önbelleklere sığarak, 1M elemana sahip önbelleklere sığmayan uygulamadan çok daha farklı bir yerellik davranışını göstermesidir. Sonuç olarak her iki durumda da önbellekler arası fazla fark olmaması, iki politikada da daha önce belirlenen aynı bir L2 önbelleğinin skorlarının karşılaştırılmasını mümkün kılmaktadır.



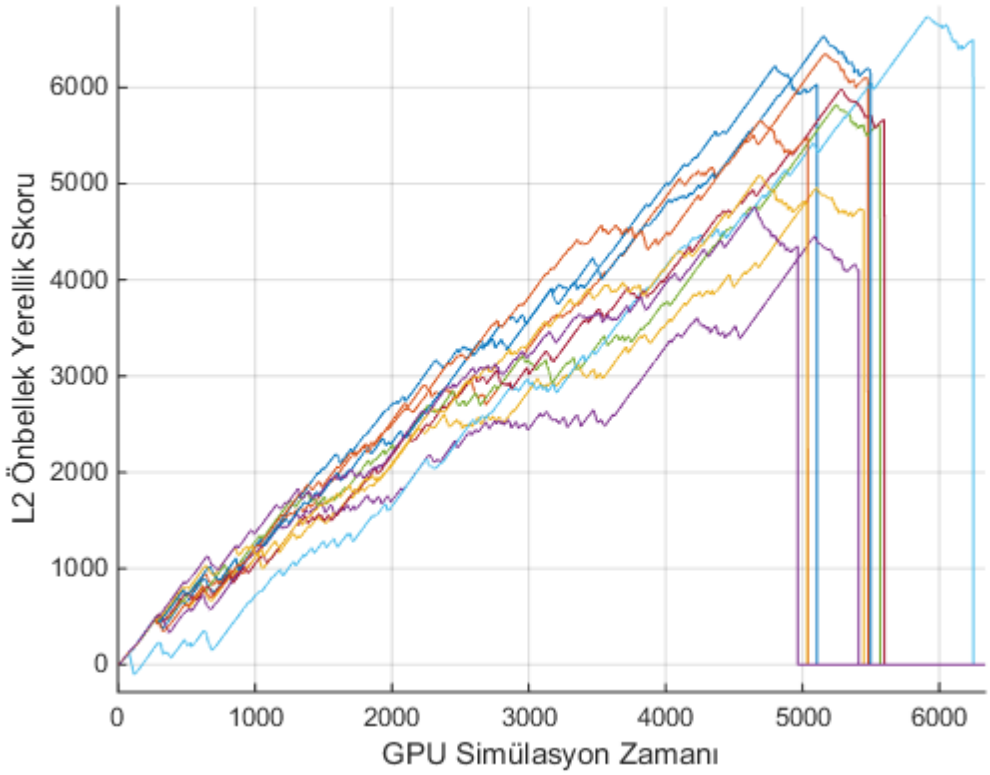
Şekil 6.1: 1M BFS WA L2 önbellek yerellik skorları



Şekil 6.2: 1M BFS NoWA L2 önbellek yerellik skorları



Şekil 6.3: 64K BFS WA L2 önbellek yerellik skorları



Şekil 6.4: 64K BFS NoWA L2 önbellek yerellik skorları

WA politikasında VTA ile bulunan yerellikler uygulama esnasında kesinlikle olan yerelliklerdir, çünkü herhangi bir yazma kaybı ile VTA içerisine eklenen girdi önbellekten atıldığı zaman VTA içerisinden de atılır. Aynı şekilde VTA içerisinde yakalanan herhangi bir yerellik önbellekte de olmuş demektir. Sonuç olarak WA politikasında VTA girdileri, önbellek girdileri ile tutarlıdır. NoWA politikasında ise VTA ile önbellek girdileri arasında bir tutarlılık bulunmamaktadır, çünkü yazma kayıpları VTA içerisine konulurken önbellekte bu istek için bir girdi oluşturulmadan alt bellek hiyerarşisine yollanmaktadır. Bu yüzden VTA içerisine konulan girdinin atılması için VTA'nın giriş pozisyonundan son pozisyonuna gitmesi gerekmektedir. WA politikasında bütün yerellikleri takip edebilecek bir boyutta VTA donanımı düşünüldüğünde, NoWA politikasında VTA tablosundan bir girdinin girişinden çıkışına geçecek süre fazla olduğundan dolayı gerçekte olmayacak yerellikleri de yakalayabilmektedir. Örneğin yazma isteklerinin çok az olduğu bir uygulamanın NoWA ile çalıştırıldığı düşünülürse, VTA içerisine eklenen bir yazma kayıp isteğinin atılması çok uzun sürebilmektedir.

Sonuç olarak WA politikasında yakalanan yerellikler sayılırken, NoWA politikasında yakalanan yerellik sayısı yakınsanmaktadır. Yakınsanmanın doğruluğu ise uygulamanın bellek trafik karakteristiği ve VTA boyutundan etkilenmektedir. Uygulamaların bellek trafik karakteristiği kontrol edilemediğinden dolayı sadece VTA boyutu ile ilgili testler yapılabilmektedir.

VTA boyutunun yeterli olup olmadığının testi WA politikasında yapılabilmektedir. Eğer yeterli VTA boyutumuz var ise VTA içerisinden çıkarılan girdilerin hiçbirinin VTA doluluğundan olmaması gerekmektedir. Çıkarma sebepleri WA ile yazma kaybı zamanında önbellekten çıkarılan bir girdinin VTA içerisinde olması, okuma kaybında önbellekten çıkarılan bir girdinin VTA içerisinde olması ya da okuma yerelliğinden sonra yerelliğin olduğu adrese sahip VTA girdisinin çıkarılmasıyla olmaktadır. Bu durumların istatistikleri toplanarak 128 bin elemanlı, 100MHz DRAM frekansı ile BFS uygulamasında VTA boyutları değiştirilerek testler yapılmıştır.

Tablo 6.2: BFS WA VTA girdi çıkarılma istatistiği

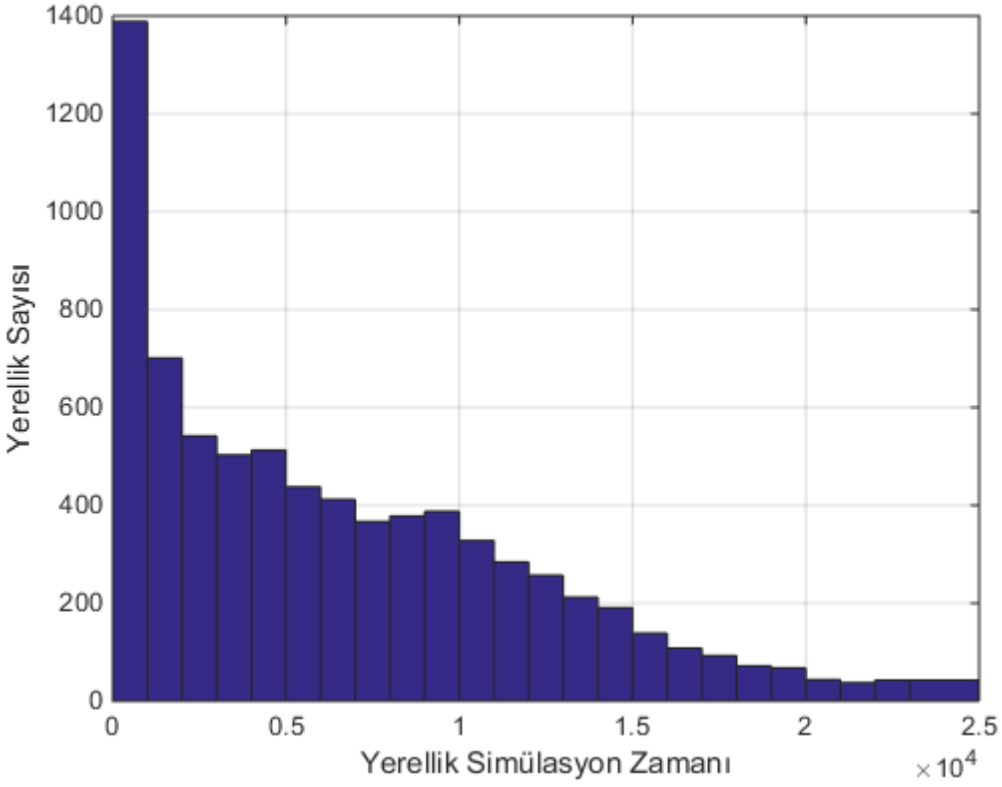
VTA Girdi Çıkarılma Sebebi	VTA 512 Girdi	VTA 128 Girdi	VTA 64 Girdi	VTA 16 Girdi
Yazma Kaybı	12170	12153	5823	8
Okuma Kaybı	1069	1050	704	23
Okuma Yerelliği	412	411	302	91
Dolu VTA	0	51	6829	13562

Tablo 6.3 : BFS WA VTA yerellik istatistiği

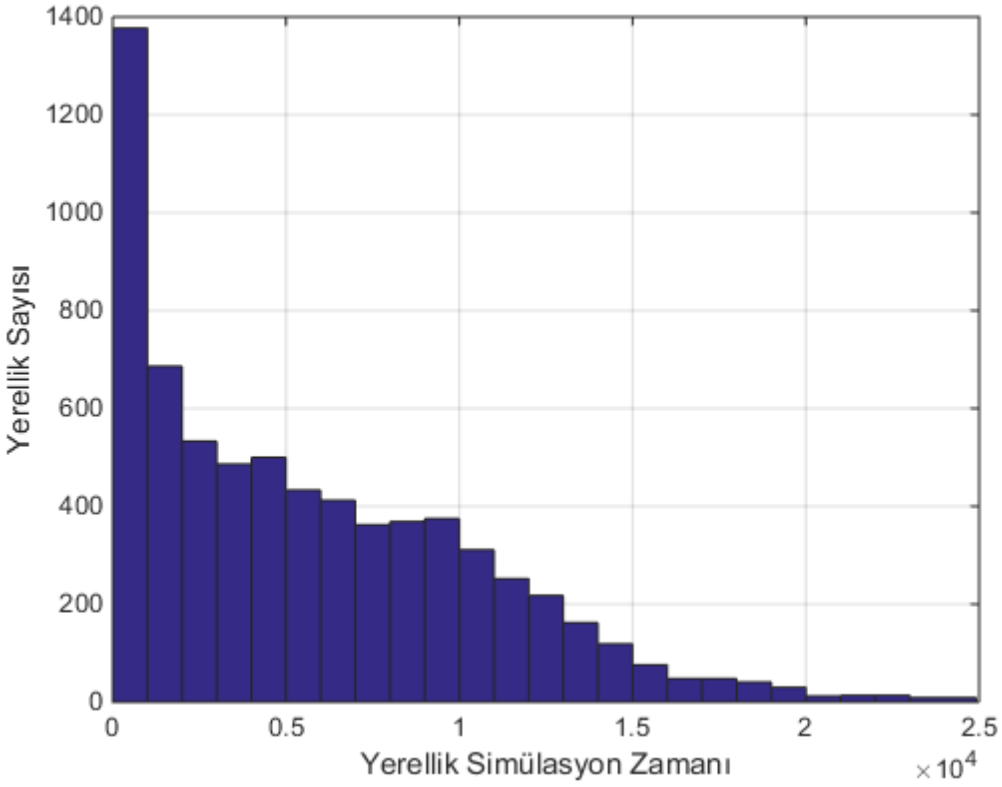
VTA Yerelliği	VTA 512 Girdi	VTA 128 Girdi	VTA 64 Girdi	VTA 16 Girdi
Yazma Yerelliği	8161	8161	7142	2510
Okuma Yerelliği	412	411	302	91

Tablo 6.2: BFS WA VTA girdi çıkarılma 'de VTA'den girdilerin çıkarılma sebepleri ile sayıları verilmiştir. 512 girdiye sahip olan VTA denemesinde VTA dolu olması sebebiyle çıkarılan hiçbir girdi olmamıştır. Bu da uygulama içerisinde tüm yerelliklerin kaçırılmadan yakalanabildiklerini ve hatasız bir yerellik grafiğinin

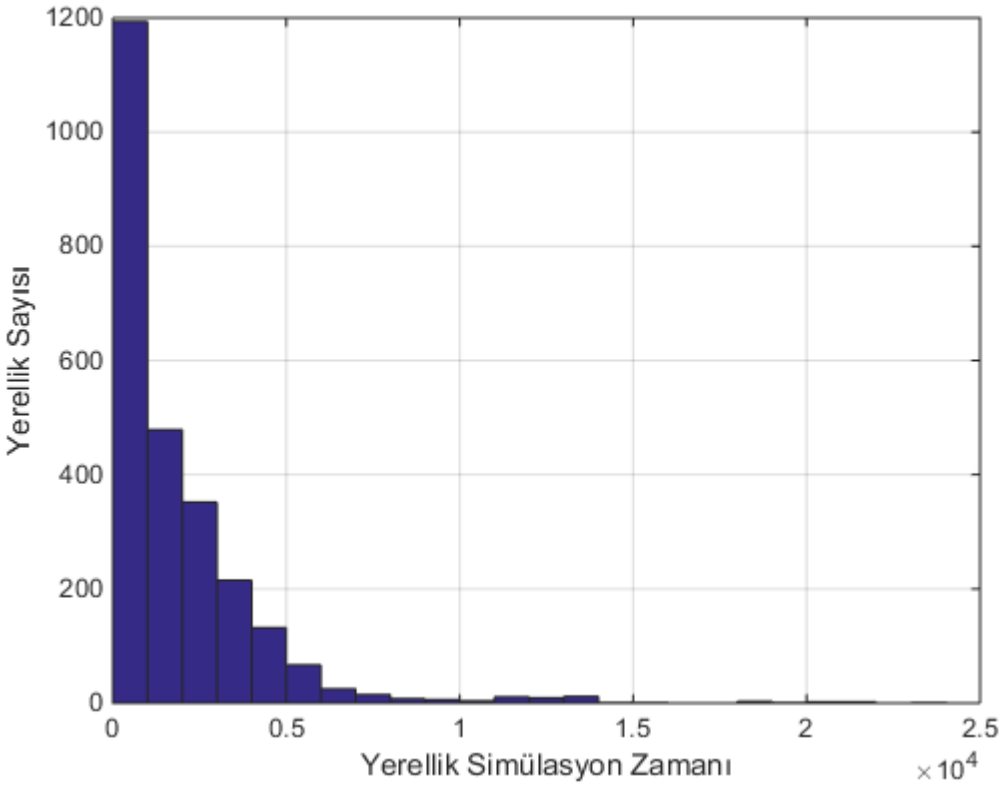
oluşturulabildiğini göstermektedir. VTA girdi sayısı azaltıldıkça VTA dolu olma sebebiyle erken çıkarılan girdilerin sayıları artmaktadır. Bu durumun dezavantajı önbellekten çıkarılmayan girdilerin VTA'den çıkarılması ve bu yüzden de oluşabilecek potansiyel yerelliklerin kaçırılmasıdır. Tablo 6.3 : BFS WA VTA yerellik istatistiği'de farklı VTA boyutları ile BFS uygulamasında yakalanan yerellik istatistikleri verilmiştir. VTA boyutu 512 olduğu zaman hiçbir yerellik kaçırılmadığı durumu kabul edilirse uygulamada WA sayesinde toplam 8573 yerellik bulunmaktadır. Boyut 128'e indirildiği zaman hala yerellik kaybı bulunmamaktadır. Boyut 64'e indirildiğinde VTA girdilerinin yarısından fazlası dolu olma sebebiyle çıkarılmasına rağmen Tablo 6.3'de görüldüğü üzere yerelliklerin %86'sı yakalanabilmektedir. Bunun sebebi ise önbelleğe konulan yeni girdiler üzerine yerellik olması ile bu girdilerin önbellekten atılması arasında zaman farkı olmasıdır. Bu zaman aralığındaki VTA boyutuna ve içerisine eklenen toplam yazma isteklerine bağlı olarak yerelliğin kaybedilme ihtimali vardır. Önbelleğe verilerin girilmesinden sonra aynı verilere gelen yerellik zamanları ölçülmüştür. Şekil 6.5, Şekil 6.6 ve Şekil 6.7'de bu zamanlamaların histogramı çizilmiştir. Hiçbir yerelliği kaybetmeden kaydedebilen 512 boyutlu VTA ile daha düşük boyutlu VTA'ların kaydedebildikleri yerellik zamanları karşılaştırıldığında, VTA boyutu ile yakalanabilecek uzak zamanlı yerellikler orantılı gözükmektedir. 64 boyutlu VTA'nın yerellik kayıpları 512 boyutlu VTA ile karşılaştırıldığında 12.500 saat dilimi farkına kadar oluşabilen yerelliklerde kayıp göstermemektedir fakat 12.500 saat diliminden sonraki yerelliklerde ise az kayıplar başlamaktadır. 16 boyutlu VTA ise en yakındaki yerellikleri yakalayabilirken, 2.500 saat diliminden sonra yerellikleri kayıplı yakalayabilirken 5.000 saat diliminden sonra ki yerellikleri yakalayamamaktadır.



Şekil 6.5: 128k BFS, 512 VTA, WA yerellik zaman histogramı



Şekil 6.6: 128k BFS, 64 VTA, WA yerellik zaman histogramı



Şekil 6.7: 128k BFS, 16 VTA, WA yerellik zaman histogramı

WA politikasında toplanan bilgiler ile NoWA politikasında toplanan bilgiler karşılaştırılarak VTA boyutuna karar verilmiştir. NoWA politikasında VTA boyutu yerellik skorunun doğrulu üzerinde büyük etkisi vardır. NoWA politikasında toplanan istatistikler Tablo 6.4'de verilmiştir. Düşük bir VTA boyutu WA politikasında olduğu gibi birçok yerelliğin bulunamamasına yol açmaktadır. Büyük VTA bir boyutu ise WA politikasında aksine NoWA politikasında fazlaca olmayan yerelliklerin bulunmasına yol açmaktadır. Bunun sebebi ise VTA girdilerinin NoWA politikasında ne zaman çıkarılması gerektiği bilinmediğinden dolayı girdilerin sadece VTA dolduğu zaman çıkartılmasındandır. Bu sorunu aşmanın birkaç yöntemi vardır fakat bu yöntemler donanımı pratik olmayacak bir seviyeye çıkarmaktadır. Örneğin önbelleğe ve VTA içerisindeki bütün girdilere zaman damgası eklenerek önbellekten çıkarılan herhangi bir girdinin zaman damgası VTA içerisindeki girdilerle karşılaştırılarak daha eski girdiler tespit edilerek VTA'dan çıkarılabilir fakat bunun için hem zaman girdisi için alan gerekirken hem de girdi zamanlarının paralel karşılaştırılması için her VTA girdisine bir karşılaştırıcı eklenmesi gerekmektedir. Bu yüzden de benzeri çözümler pratikte kullanılamamaktadır.

WA politikasında büyük bir VTA boyutu ile hatasız bir şekilde toplam 8161 yerelliğin bulunabildiği görülmüştür. NoWA politikasında ise aynı büyük boyutta bir VTA ile Tablo 6.4'de görüldüğü üzere toplam 17890 yerellik bulunmuştur, bu sayı ise olması gerektiğinin iki katından fazladır. 64 girdiye sahip olan bir VTA ile yaklaşık olarak WA politikası ile aynı sayıda yerellik bulunduğundan dolayı dinamik politika için bu VTA boyutu kullanılmıştır.

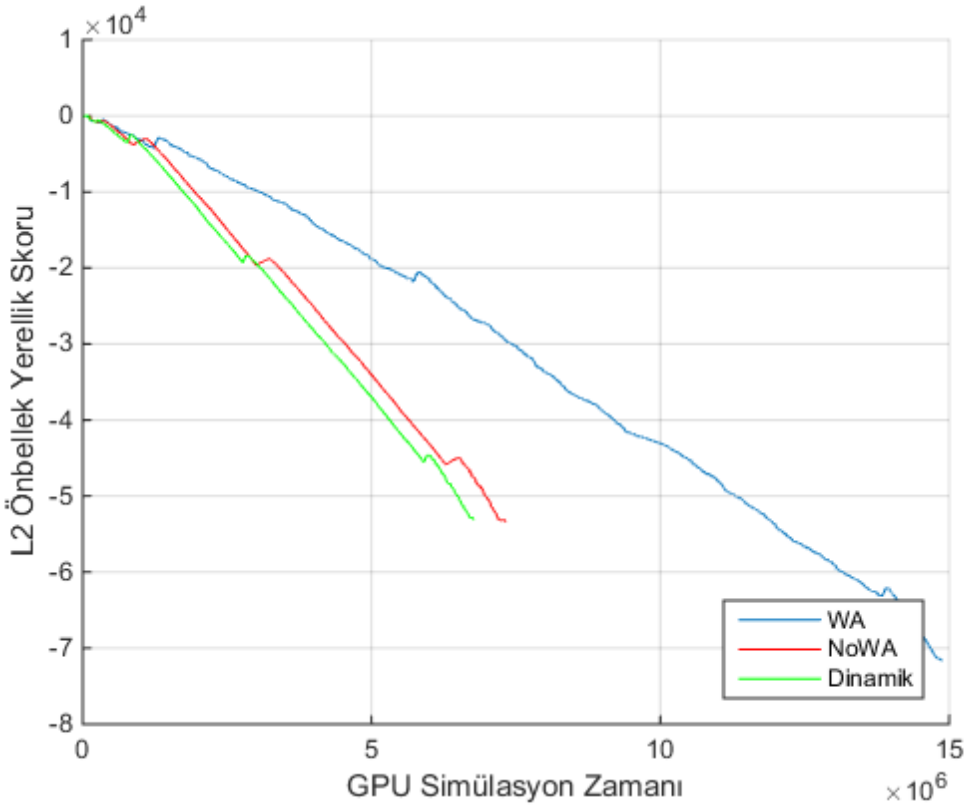
Tablo 6.4 : BFS NoWA VTA yerellik istatistiği

VTA Potansiyel Yerelliği	VTA 512 Girdi	VTA 128 Girdi	VTA 64 Girdi	VTA 16 Girdi
Yazma Potansiyel Yerelliği	17890	14061	8338	2818
Okuma Potansiyel Yerelliği	790	609	319	78

6.3 Politika Başarım Karşılaştırımı

6.3.1 BFS (%3 WA, %97 NoWA)

Şekil 6.8'de 256k elemanlı BFS uygulamasının her politika ile ortaya çıkan skorlamaları gösterilmiştir. Dinamik politikada yürüyen averaj kullanılırken gösterim amaçlı şekilde kümülatif skorlar çizdirilmiştir. Şekildeki farklı simülasyon bitiş zamanları, politikaların uygulama üzerindeki başarımı değiştirdiklerini göstermektedir. Farklı yerellik bitiş skorları ise WA ve NoWA politikaları ile farklı yerellik kestirim sayılarının olduğunu göstermektedir. Uygulamayı en hızlı sonuca ulaştıran politika yeşil renkli dinamik politika iken dinamik politikayı kırmızı renkle NoWA ve mavi renkle WA politikaları takip etmektedir. Her üç politika ile yerellik skoru eksi yönde gitmesi, 256k BFS uygulamasında yerelliğin çok olmadığını göstermektedir.



Şekil 6.8: 256k BFS politika skorları

BFS uygulaması, girdi olan bir ağacın bütün komşularını bulana kadar ağaç üzerinde arama yapan bir algoritmadır. Döngülü çalışan bu uygulama her döngüde bir önceki sefer keşfedilen düğümlerin komşuluklarına bakmaktadır. CUDA dilinde bu uygulama 2 adet fonksiyon ile gerçekleştirilmiştir. Şekil 6.9'da gösterilen ilk fonksiyon daha önce keşfedilen bütün düğümlerin komşulukları bulunur ve bulunan komşulukların sayısına göre ana düğümün komşuluk sayısı arttırılır. Şekil 6.10'da gösterilen ikinci fonksiyonda ise ilk fonksiyonda keşfedilen düğümlerin bir sonraki döngüde başlangıç düğümleri olarak seçilmesi için yeni bulunan düğümler işaretlenir. Bu yüzden de birinci fonksiyondan ikinci fonksiyonda geçerken üzerinde dolaşılan ağacın ve donanımsal önbelleğin büyüklüğüne göre belli bir oranda yerellik bulunmaktadır. 256k elemana sahip olan bu uygulamada ağacın büyüklüğünden dolayı birinci fonksiyonda yerellik bulunamazken, girdi olan ağacın birinci fonksiyondan ikinci fonksiyona geçişlerinde yerellik bulunma ihtimali vardır.

```
__global__ void
Kernel( Node* g_graph_nodes, int* g_graph_edges, bool* g_graph_mask,
bool* g_updating_graph_mask, bool *g_graph_visited, int* g_cost, int no_of_nodes)
{
    int tid = blockIdx.x*MAX_THREADS_PER_BLOCK + threadIdx.x;
    if( tid<no_of_nodes && g_graph_mask[tid])
    {
        g_graph_mask[tid]=false;
        for(int i=g_graph_nodes[tid].starting; i<(g_graph_nodes[tid].no_of_edges + g_graph_nodes[tid].starting); i++)
            {
            int id = g_graph_edges[i];
            if(!g_graph_visited[id])
                {
                g_cost[id]=g_cost[tid]+1;
                g_updating_graph_mask[id]=true;
                }
            }
    }
}
```

Şekil 6.9: BFS uygulaması fonksiyon 1

```
__global__ void
Kernel2( bool* g_graph_mask, bool *g_updating_graph_mask, bool* g_graph_visited, bool *g_over, int no_of_nodes)
{
    int tid = blockIdx.x*MAX_THREADS_PER_BLOCK + threadIdx.x;
    if( tid<no_of_nodes && g_updating_graph_mask[tid])
    {

        g_graph_mask[tid]=true;
        g_graph_visited[tid]=true;
        *g_over=true;
        g_updating_graph_mask[tid]=false;
    }
}
```

Şekil 6.10: BFS uygulaması fonksiyon 2

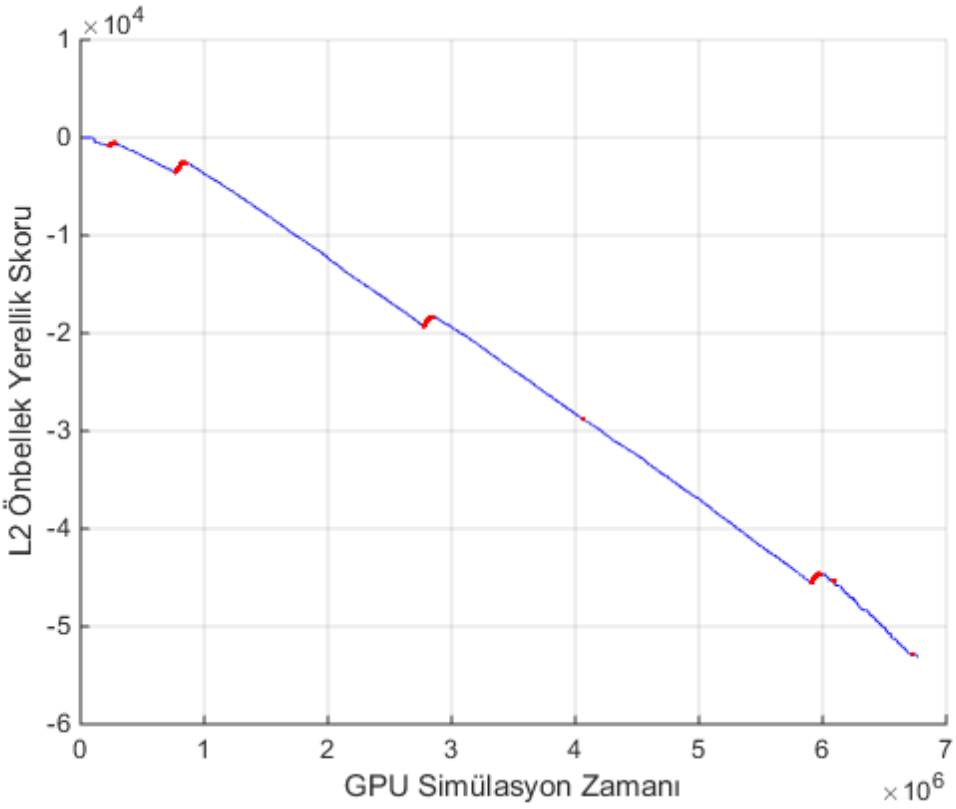
Tablo 6.5'de görüldüğü üzere dinamik politika IPC olarak WA politikasına göre %118, NoWA politikasına göre %8 daha hızlı çalışmaktadır. Politikalar L2 önbelleklerine uygulandığından dolayı L1 önbelleklerinin kayıp oranı değişmemektedir. L2 önbellek kayıp oranı WA ile en az oranda olmasına rağmen uygulamayı en yavaş çalıştıran politikadır. Bunun sebebi ise WA politikasının yarattığı fazla trafiğin yeterli olarak yerellik için kullanılamamasıdır. Bu da DRAM'den işlemler için gerekli verilerin çekilmesini geciktirerek sistemin boşa zaman harcamasına sebep vermektedir. NoWA politikası DRAM'den verileri WA politikası gibi önceden çekmediği için L2 kayıp oranı WA politikasına göre iki katı olmasına rağmen performans olarak iki kat daha hızlıdır. Bu da WA politikası ile DRAM'den önceden çekilen verilerin L2 yerellik oranını arttırmasına fakat bunun dışında çekilen ve kullanılmayan verilerin sistemi ekstra beklettiğini göstermektedir. Dinamik politika ise gerektiği zaman DRAM'den ekstra bilgileri çekerek performansın NoWA politikasından da yüksek olmasını sağlamıştır. DRAM gecikmesi olarak bakıldığında dinamik politika ile NoWA politikasının aynı gecikmelere sahip olduğunu görürken, uygulamanın belli bölgelerinde gerekli bilgileri DRAM'den L2 önbelleğine önceden çekmesinden dolayı

Paylaşımlı Bellek Gecikmesini gerekli verileri DRAM'den çekmek yerine L2'den çekerek çekirdeklere daha erken yollamasıyla azaltmaktadır.

Tablo 6.5 : 256k BFS istatistikleri

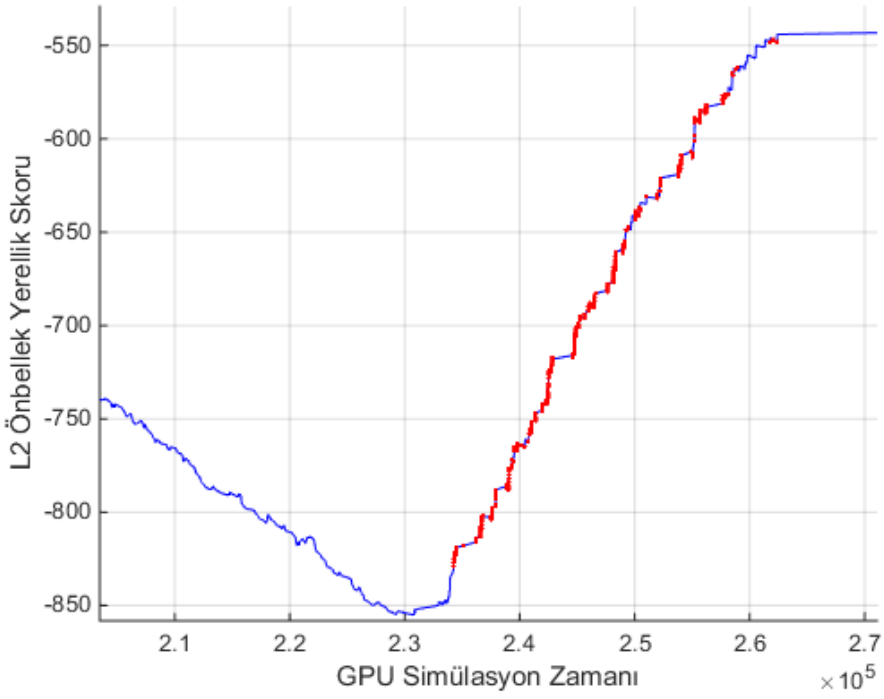
	WA	NoWA	Dinamik
IPC	7.08	14.29	15.45
L1 Kayıp Oranı	0.8361	0.8341	0.8336
L2 Kayıp Oranı	0.1711	0.3271	0.3138
DRAM Gecikmesi	53575034	20195592	19977539
Arabağlantı Ağı Gecikmesi	6675204	10977294	10944273
Paylaşımlı Bellek Gecikmesi	206557567	97074283	89906717

Şekil 6.11'da dinamik politika ile uygulamanın hangi taraflarında WA ve NoWA politikalarının seçildiği gösterilmiştir. Mavi alanlarda yerellik skoru düşmesi sonucu NoWA politikası, kırmızı taraflarda yerellik artışından dolayı WA politikası seçilmiştir. L2 önbellek erişimlerinin %97'sinde NoWA, %3'ünde ise WA politikası uygulanmıştır. Sadece NoWA politikasının kullanılması yerine uygulamanın yalnızca %3'lük kısmında WA politikası kullanılması uygulamayı WA politikasına göre %118 ve NoWA politikasına göre %8 hızlandırabilmektedir.



Şekil 6.11: 256K BFS, dinamik politika (Mavi: NoWA, Kırmızı: WA)

Şekil 6.12'da dinamik politikanın WA ile NoWA politikaları arasındaki geçişleri görebilmek için BFS uygulamasının skor grafiğinin yakınlaştırılmış hali konulmuştur. Yerellik skoruna yazma yerellikleri için 2, okuma yerellikleri için 1 ve yerellik olmadan atılan girdiler için -1 skor eklenmiştir. Politika değişimi için 20 güncelleme önceki skor ile güncel skor arasında pozitif 15 puan fark olduğunda WA, diğer durumlarda ise NoWA politikası seçilmiştir. Bu yüzden de şekilde mavi ile çizilen NoWA politikası kullanılan alandan kırmızı ile gösterilen WA politikasına geçişlerinde sistemin yerellik olduğundan emin olması için belli bir süre beklemesi gerekmektedir. Aynı şekilde WA politikasından NoWA politikasına geçiş için de skor artışında belli bir azalma olması gerekmektedir.



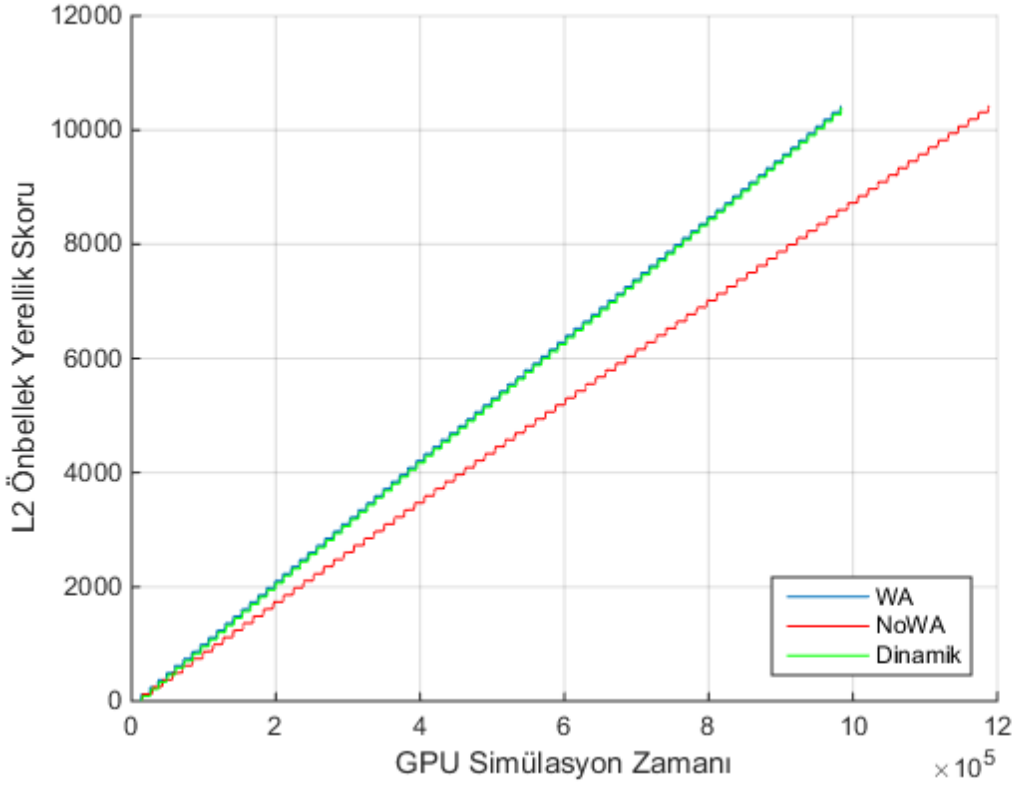
Şekil 6.12: Dinamik politika geçiş davranışı

6.3.2 NQU (%99 WA, %1 NoWA)

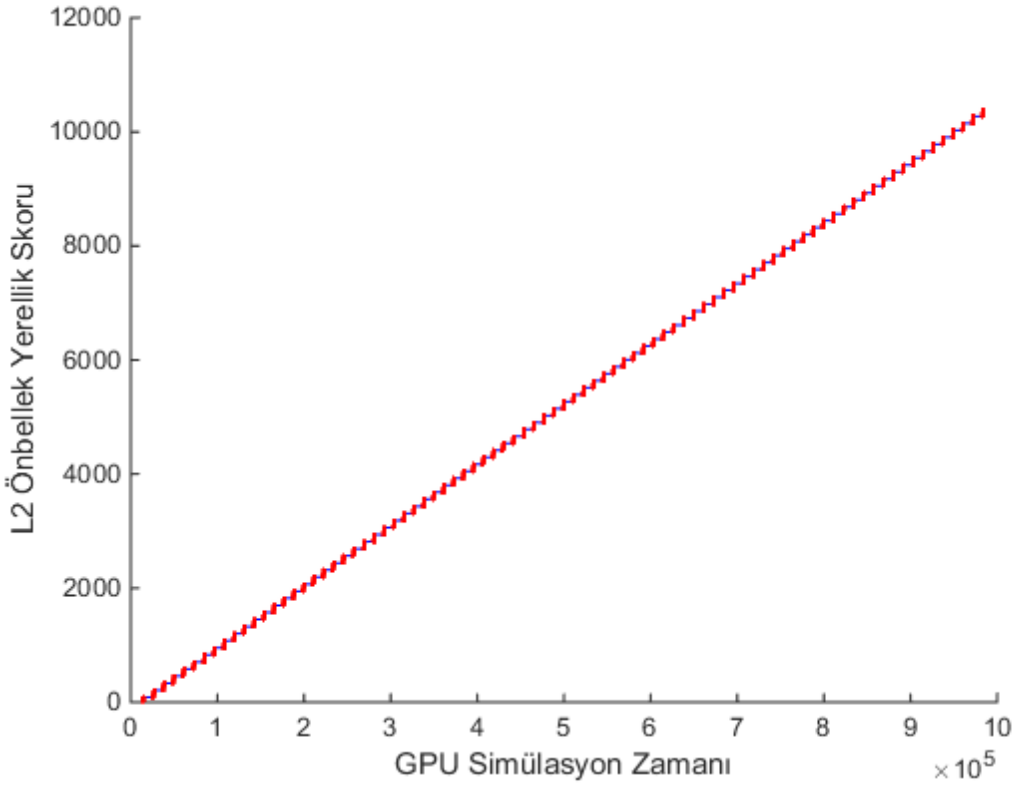
N-Queens Problem olarak bilinmektedir. Satranç tablosunda N-adet vezirleri birbirlerini tehdit etmeyecek şekilde koymayı amaçlayarak, bu pozisyonlardan toplam kaç adet olabileceğini araştırmaktadır. Bu problemin çözümü backtracking algoritması ile çözülmektedir. Algoritma satranç tablosunun son durumuna bakarak herhangi bir pozisyona yeni bir veziri eklemeye çalışmaktadır. Bu yüzden de her adımda satranç tablosunun son durumu üzerine eklemeler yaptığından dolayı, satranç tablosunun boyutu önemli olmaktadır. Eğer boyut önbelleğe sığıyor ise programda hızlandırma sağlanabilir. Bu yüzden de Tablo 6.6'da görülen önbellek politikaları arasında başarım farkı bulunmaktadır. Politikalardan bağımsız olarak NQU çözüm kümesinin boyutu L1 önbelleklerine sığamadığı içim hiçbir yerellik bulunamamıştır. Bunun aksine L2 önbelleğine sığarak WA politikasında uygulamanın en başında okunan problem girdileri haricinde hiçbir kayıp olmamıştır. Bu durumda da başlangıç durumu NoWA olarak başlayan dinamik politika ilk fırsatta WA politikasına geçiş sağlamış ve tamamen WA olarak devam etmiştir.

Tablo 6.6 : NQU istatistikleri

	WA	NoWA	Dinamik
IPC	394.3504	326.1031	393.9809
L1 Kayıp Oranı	1.0000	1.0000	1.0000
L2 Kayıp Oranı	0.0001	0.9981	0.0020
DRAM Gecikmesi	270319	2519	270729
Arabağlantı Ağı Gecikmesi	508	508	508
Paylaşımlı Bellek Gecikmesi	7380	7380	7380



Şekil 6.13: NQU politika skorları

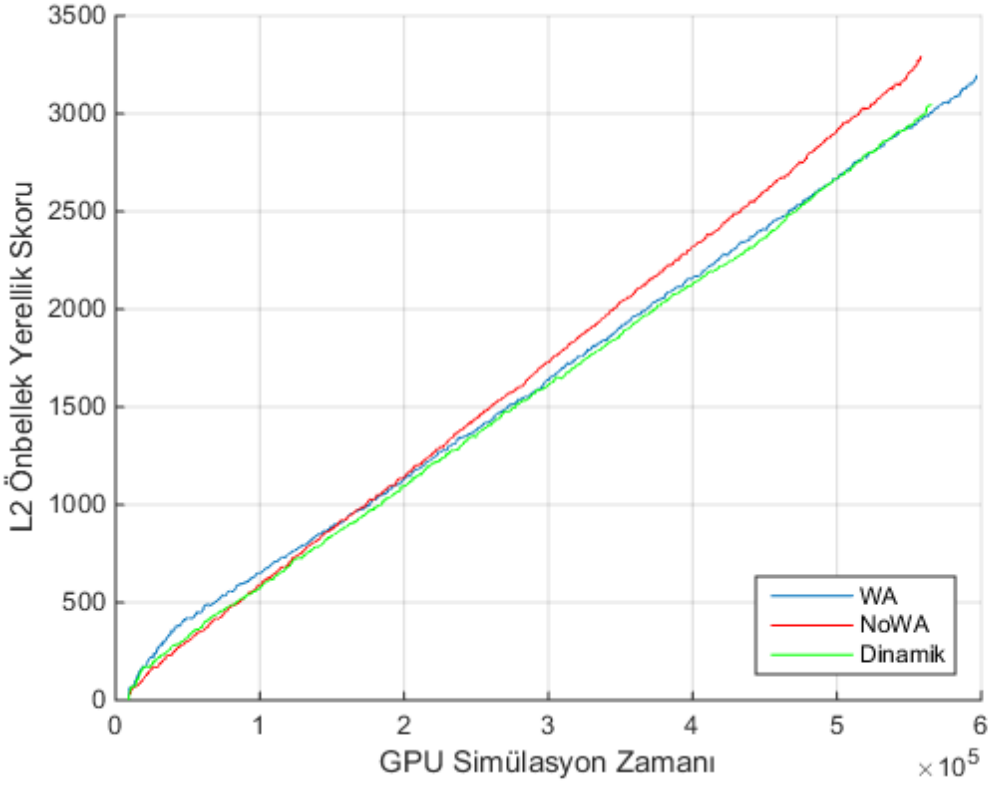


Şekil 6.14: NQU dinamik politika seçimi (Mavi: NoWA, Kırmızı: WA)

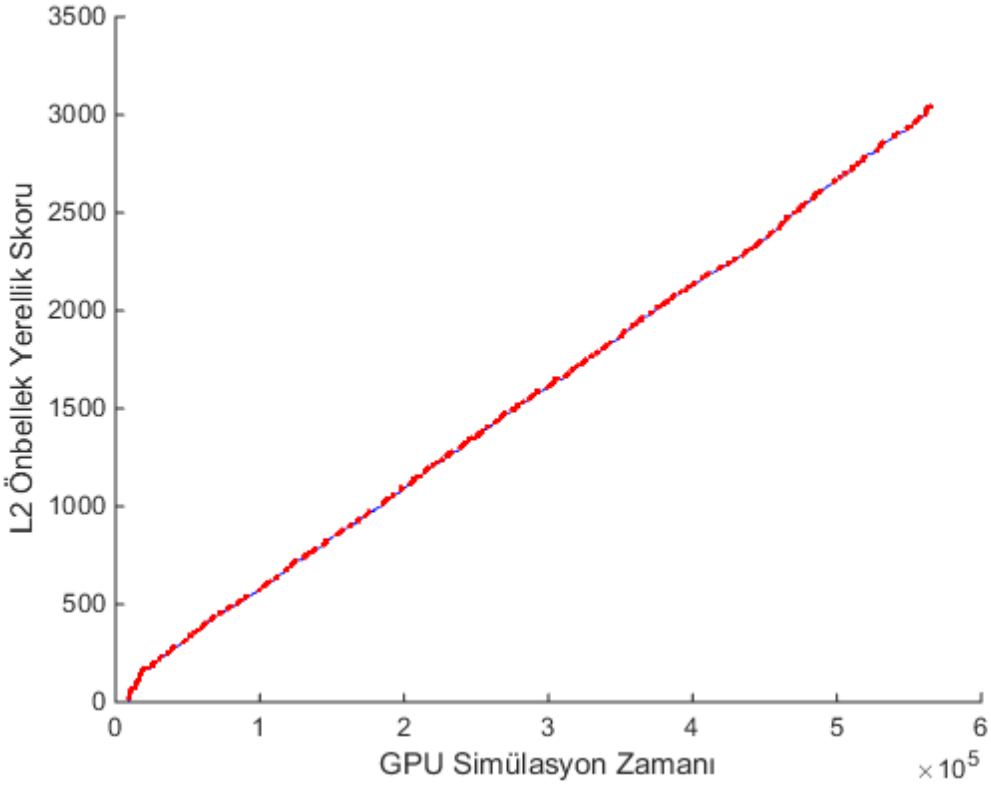
6.3.3 LPS (%25 WA, %75 NoWA)

Tablo 6-7 : LPS istatistikleri

	WA	NoWA	Dinamik
IPC	138.76	147.92	146.48
L1 Kayıp Oranı	0.7098	0.7086	0.7109
L2 Kayıp Oranı	0.1751	0.5470	0.4074
DRAM Gecikmesi	2747272	3352012	2445196
Arabağlantı Ağı Gecikmesi	65393	58110	55843
Paylaşımlı Bellek Gecikmesi	3267793	3101785	2706930



Şekil 6.15: LPS politika skorları

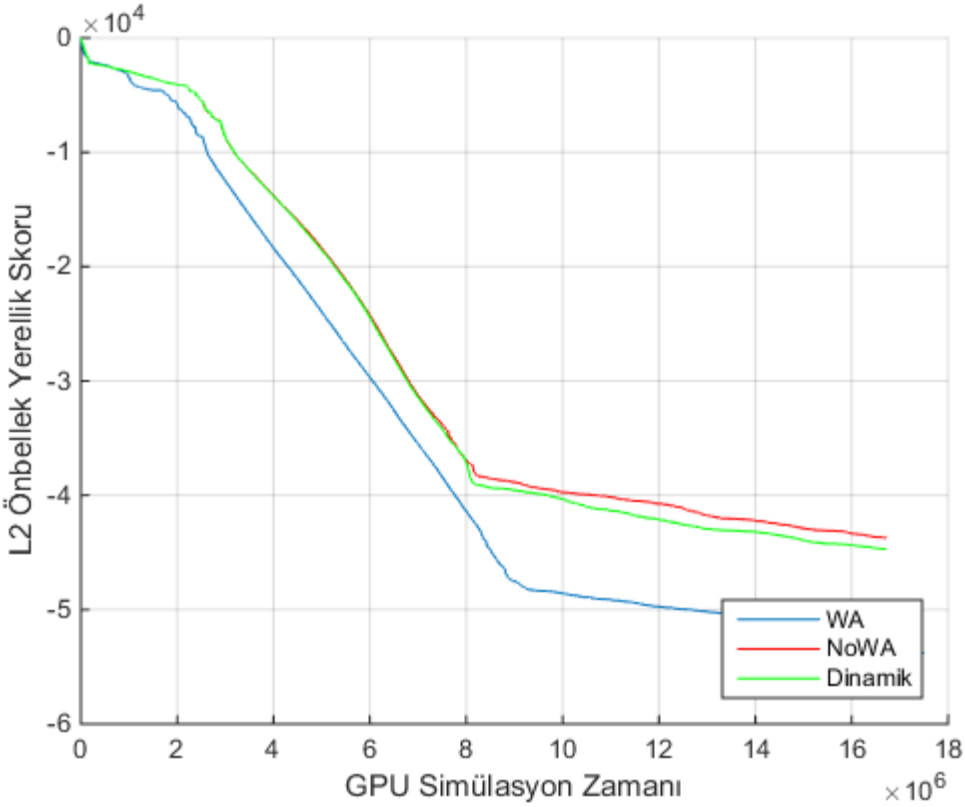


Şekil 6.16: NQU dinamik politika seçimi (Mavi: NoWA, Kırmızı: WA)

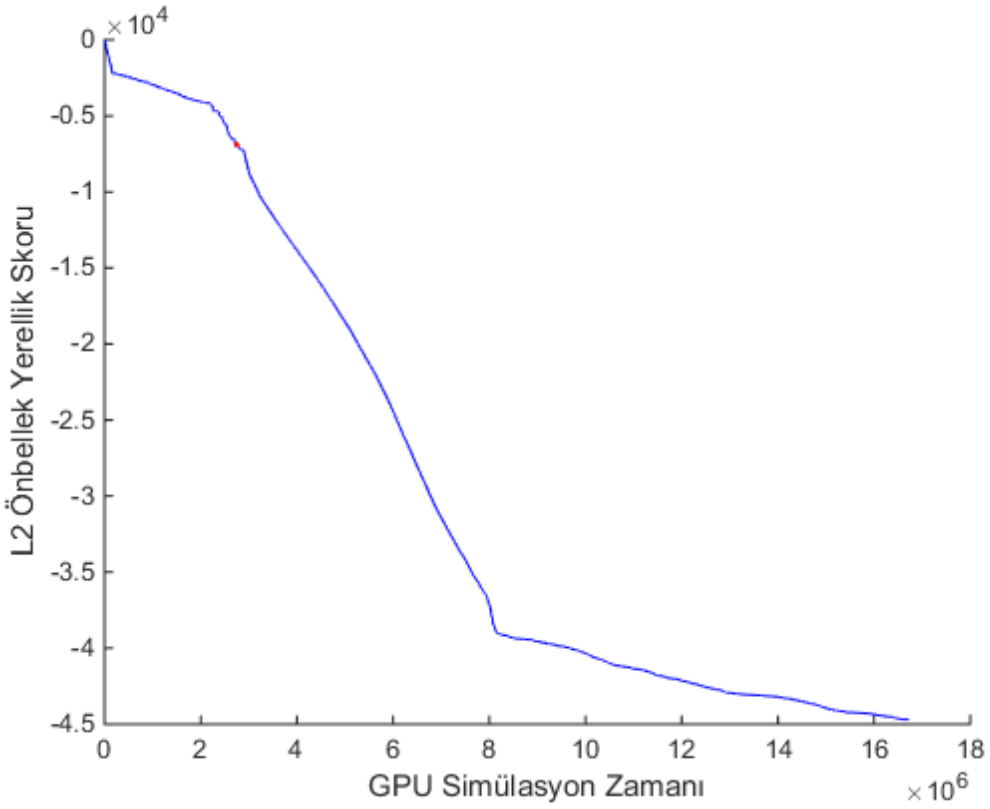
6.3.4 LIB (%1 WA, %99 NoWA)

Tablo 6.8 : LIB istatistikleri

	WA	NoWA	Dinamik
IPC	49.65	51.08	50.59
L1 Kayıp Oranı	0.6664	0.6625	0.6626
L2 Kayıp Oranı	0.5986	0.5997	0.5986
DRAM Gecikmesi	13053258	12891776	13053258
Arabağlantı Ağı Gecikmesi	2576	2547	2576
Paylaşımlı Bellek Gecikmesi	4162096	4229454	4162096



Şekil 6.17: LIB politika skorları

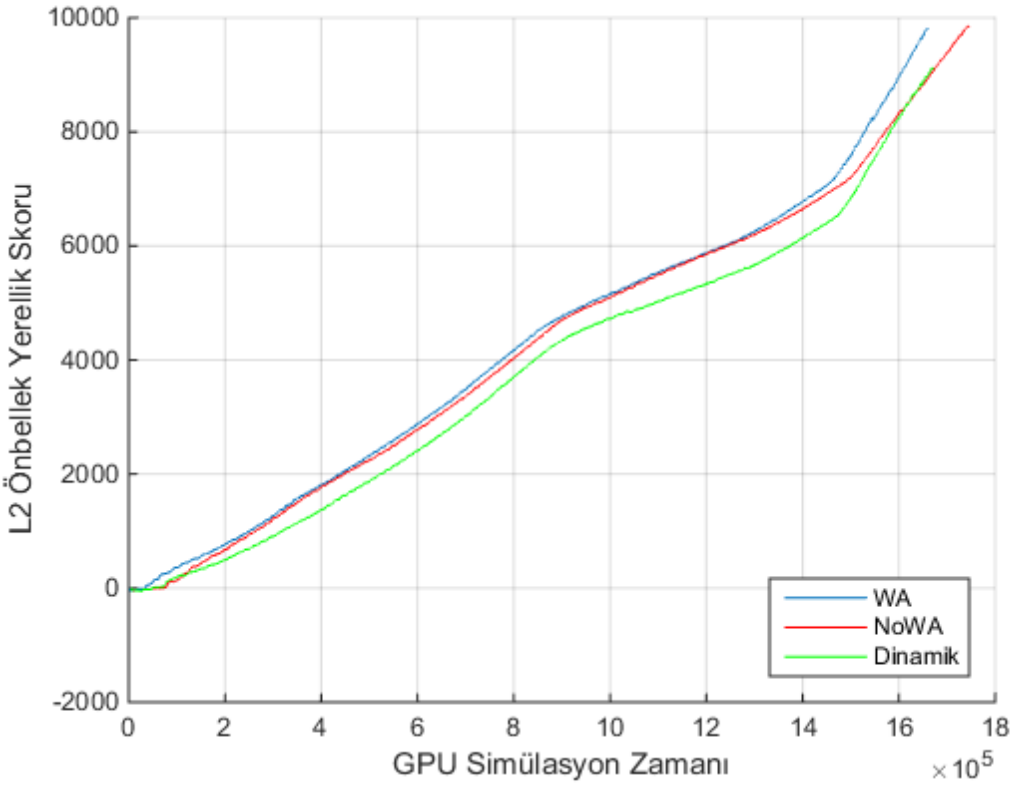


Şekil 6.18: LIB dinamik politika seçimi (Mavi: NoWA, Kırmızı: WA)

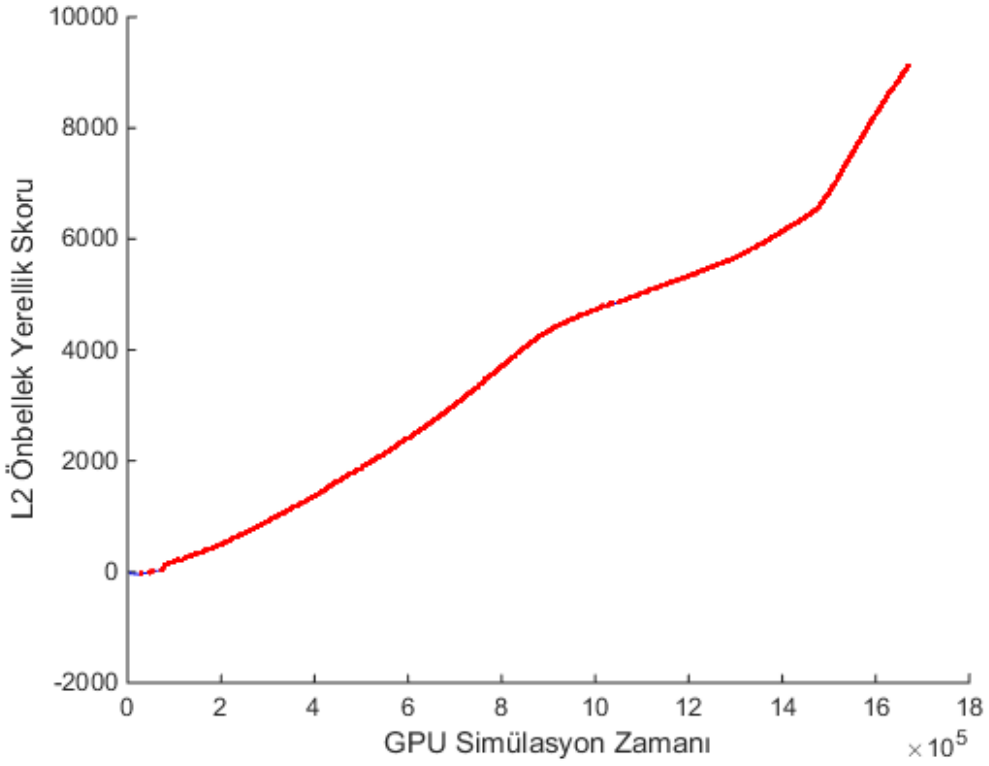
6.3.5 RAY 1k x 1k (%92 WA, %8 NoWA)

Tablo 6.9 : RAY istatistikleri

	WA	NoWA	Dinamik
IPC	450.1270	427.6683	447.2946
L1 Kayıp Oranı	0.4190	0.4208	0.4196
L2 Kayıp Oranı	0.5986	0.5997	0.5986
DRAM Gecikmesi	15039	271398	76191
Arabağlantı Ağı Gecikmesi	53765	9834	53087
Paylaşımlı Bellek Gecikmesi	588751	980658	643561



Şekil 6.19 : RAY politika skorları

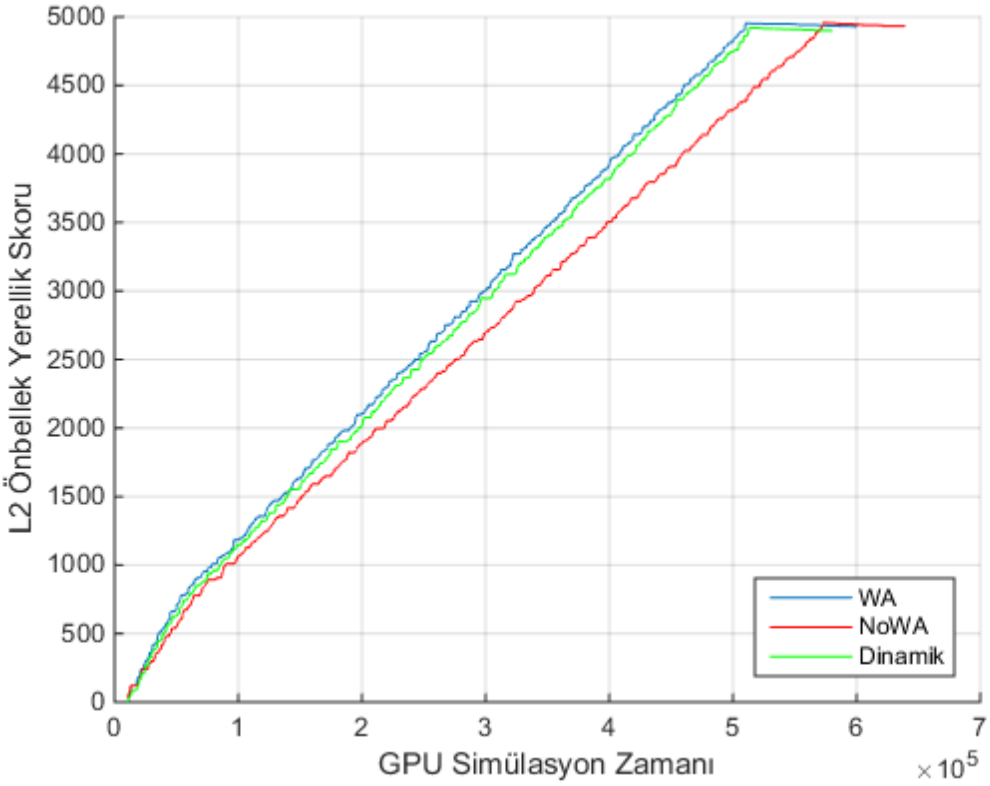


Şekil 6.20 : RAY dinamik politika seçimi (Mavi: NoWA, Kırmızı: WA)

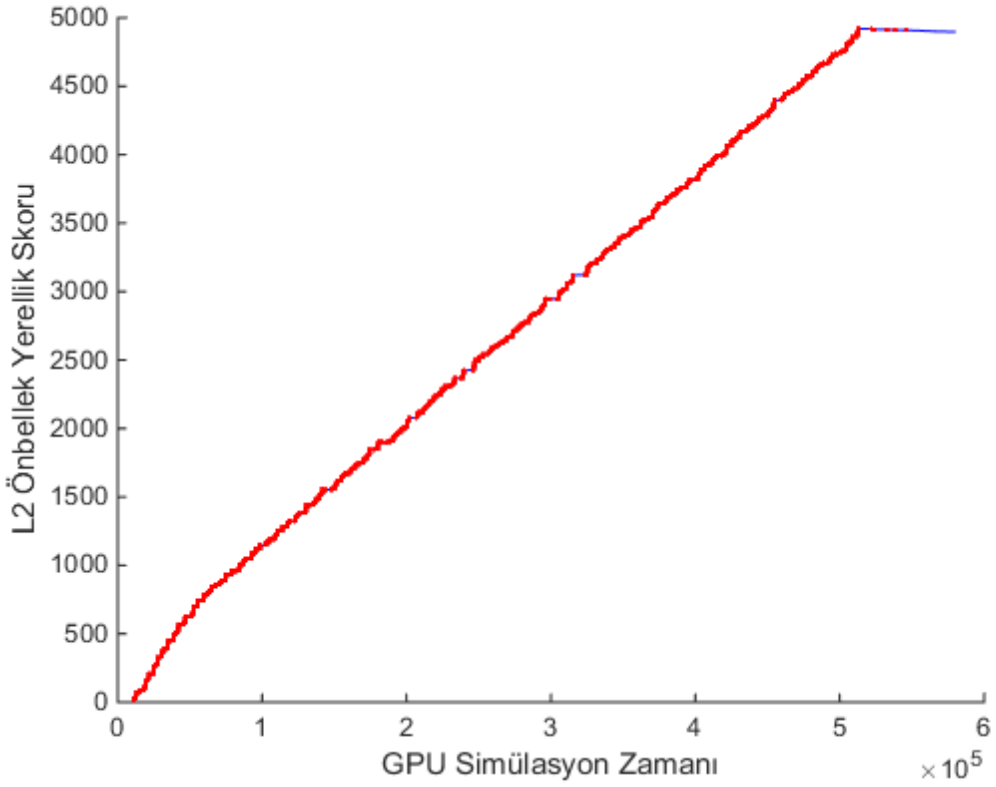
6.3.6 BP (%99 WA, %1 NoWA)

Tablo 6.10 : BP istatistikleri

	WA	NoWA	Dinamik
IPC	116.2108	111.8704	116.0748
L1 Kayıp Oranı	0.3688	0.3689	0.3689
L2 Kayıp Oranı	0.2959	0.3856	0.2965
DRAM Gecikmesi	8737934	10346065	9217007
Arabağlantı Ağı Gecikmesi	5283	7581	5177
Paylaşımlı Bellek Gecikmesi	17544246	16969184	17493721



Şekil 6.21 : BP politika skorları

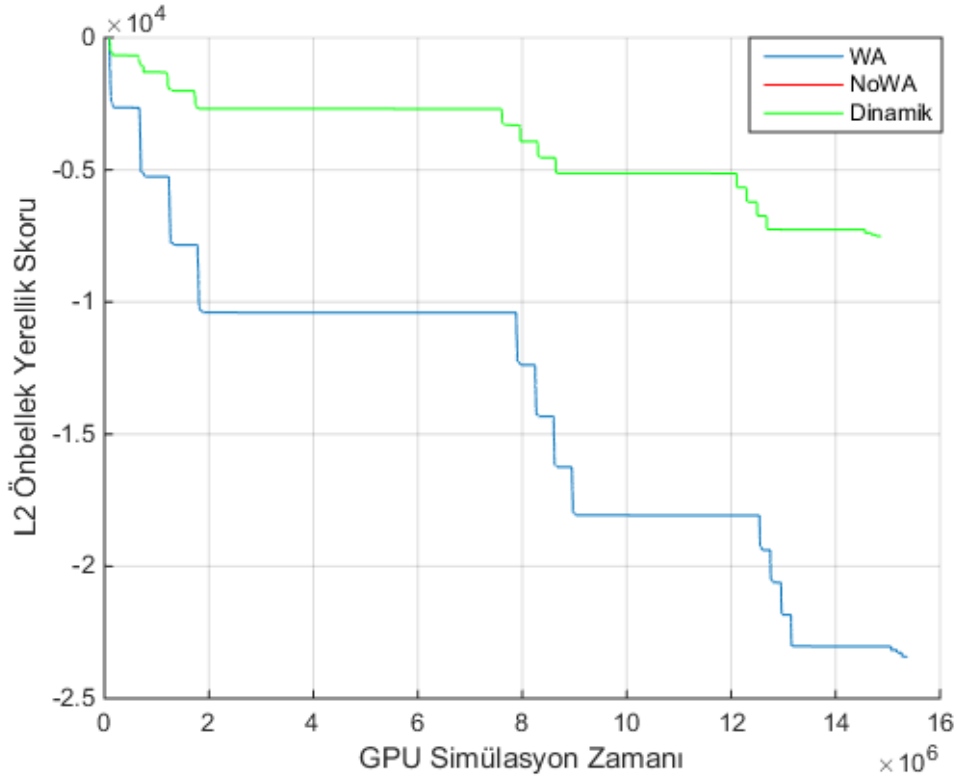


Şekil 6.22 : BP dinamik politika seçimi (Mavi: NoWA, Kırmızı: WA)

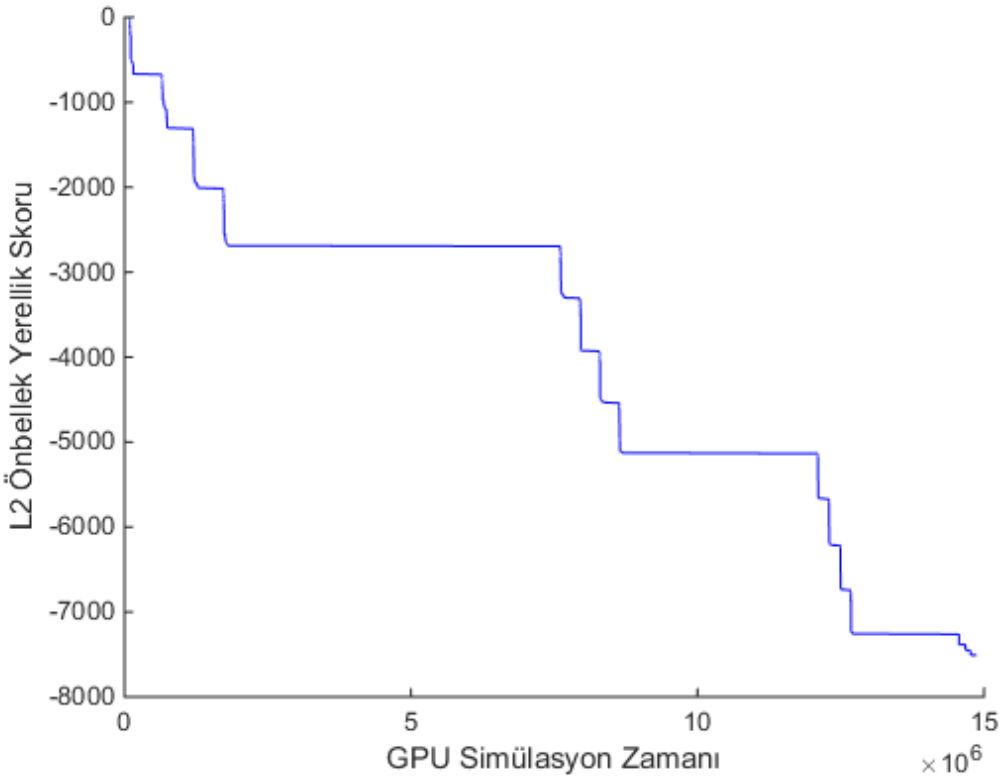
6.3.7 LUD 1k (%0 WA, %100 NoWA)

Tablo 6.11 : LUD istatistikleri

	WA	NoWA	Dinamik
IPC	145.9333	150.6163	150.6163
L1 Kayıp Oranı	0.5813	0.5813	0.5813
L2 Kayıp Oranı	0.1624	0.1673	0.1673
DRAM Gecikmesi	39165021	37658358	37658358
Arabağlantı Ağı Gecikmesi	7000667	6988232	6988232
Paylaşımlı Bellek Gecikmesi	104696040	103884153	103884153



Şekil 6.23 : LUD politika skorları

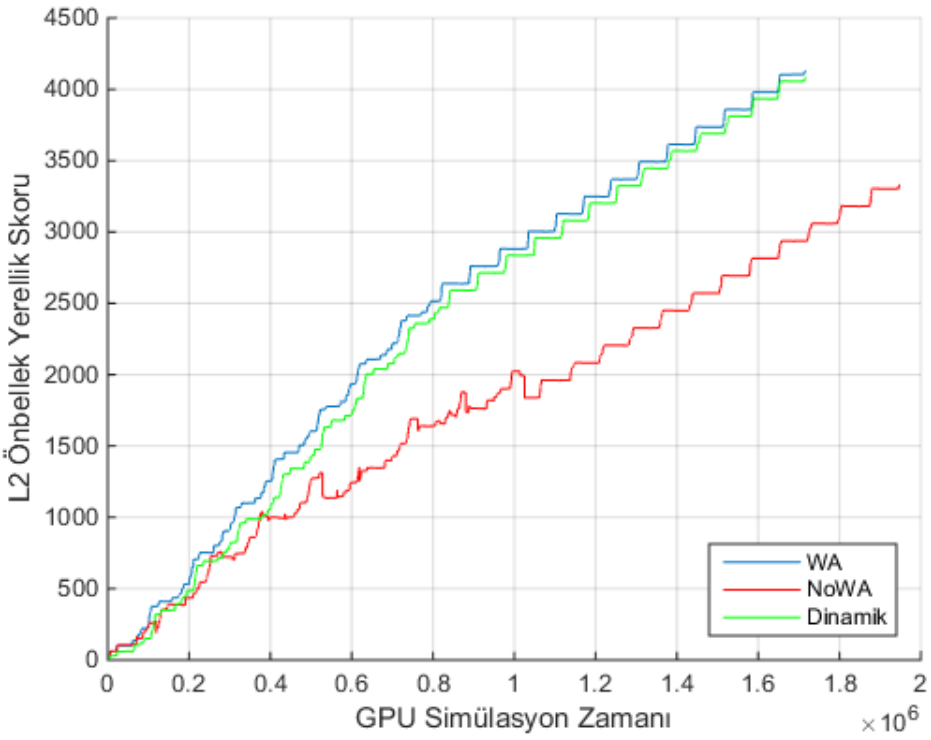


Şekil 6.24 : LUD dinamik politika seçimi (Mavi: NoWA, Kırmızı: WA)

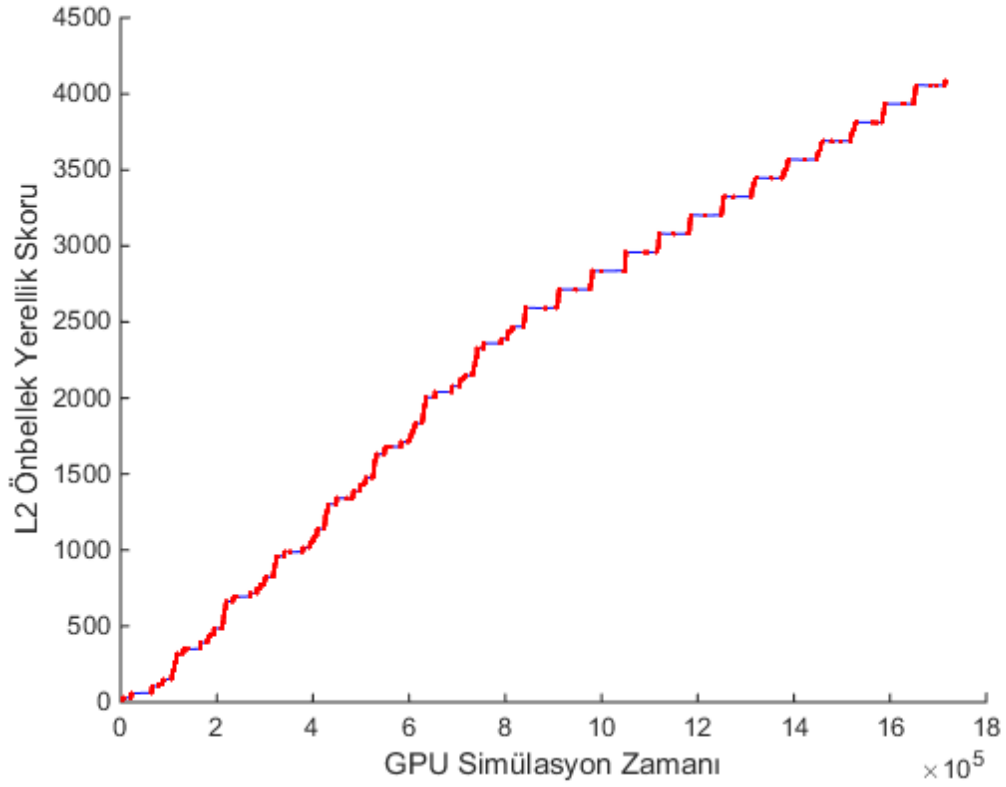
6.3.8 B+Tree (%99 WA, %1 NoWA)

Tablo 6.12 : B+Tree istatistikleri

	WA	NoWA	Dinamik
IPC	145.9333	150.6163	150.6163
L1 Kayıp Oranı	0.2274	0.2275	0.2270
L2 Kayıp Oranı	0.3575	0.4040	0.3582
DRAM Gecikmesi	1468754	1712089	1480045
Arabağlantı Ağı Gecikmesi	922529	953858	889304
Paylaşımlı Bellek Gecikmesi	8584103	8259809	8527112



Şekil 6.25 : B+Tree politika skorları

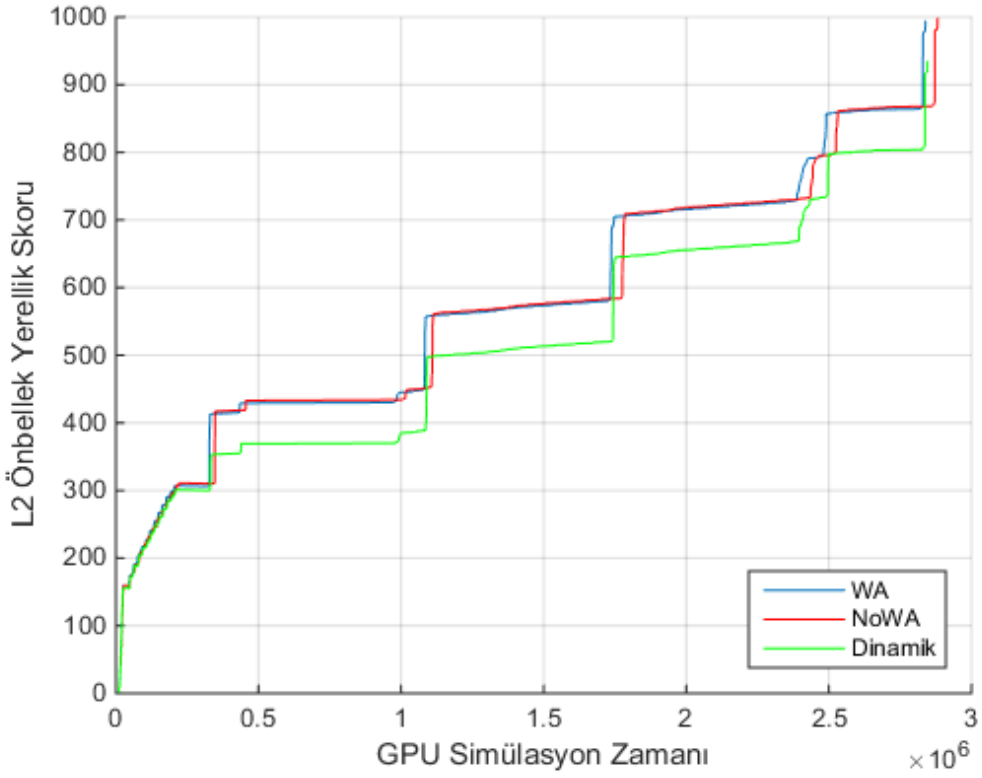


Şekil 6.26 : B+Tree dinamik politika seçimi (Mavi: NoWA, Kırmızı: WA)

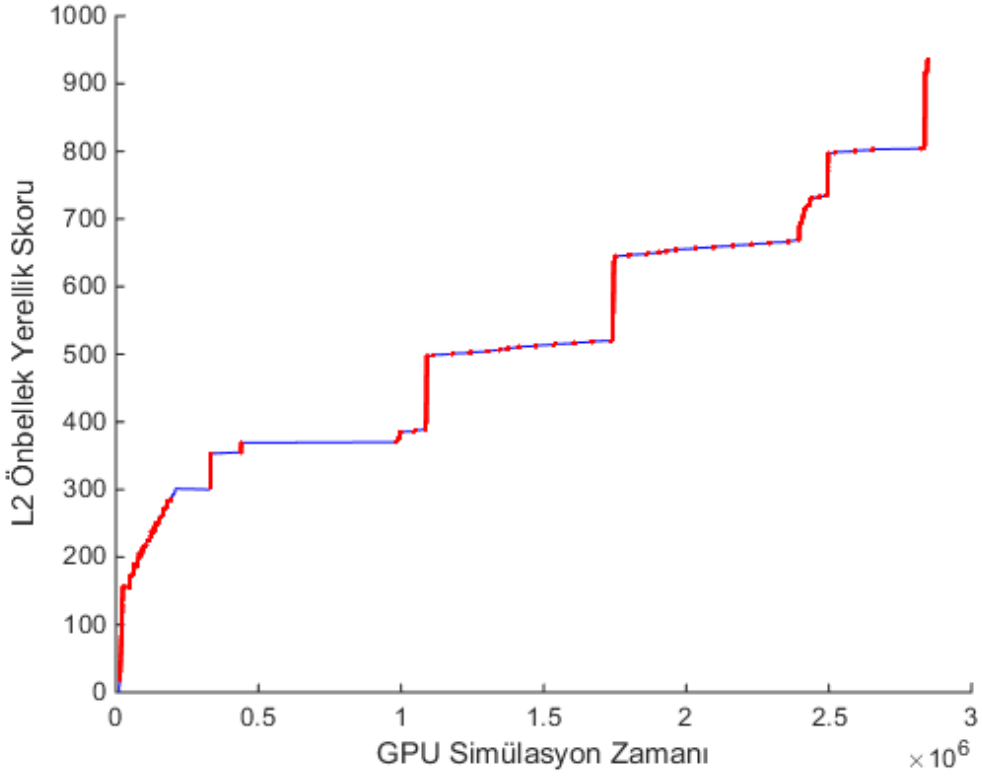
6.3.9 NN 28 (%80 WA %20 NoWA)

Tablo 6.13 : NN istatistikleri

	WA	NoWA	Dinamik
IPC	35.7722	35.2375	35.6803
L1 Kayıp Oranı	0.0212	0.0212	0.0213
L2 Kayıp Oranı	0.0280	0.0809	0.0375
DRAM Gecikmesi	34049	83909	41457
Arabağlantı Ağı Gecikmesi	85793	118395	85505
Paylaşımlı Bellek Gecikmesi	946823	961360	943276



Şekil 6.27 : NN politika skorları

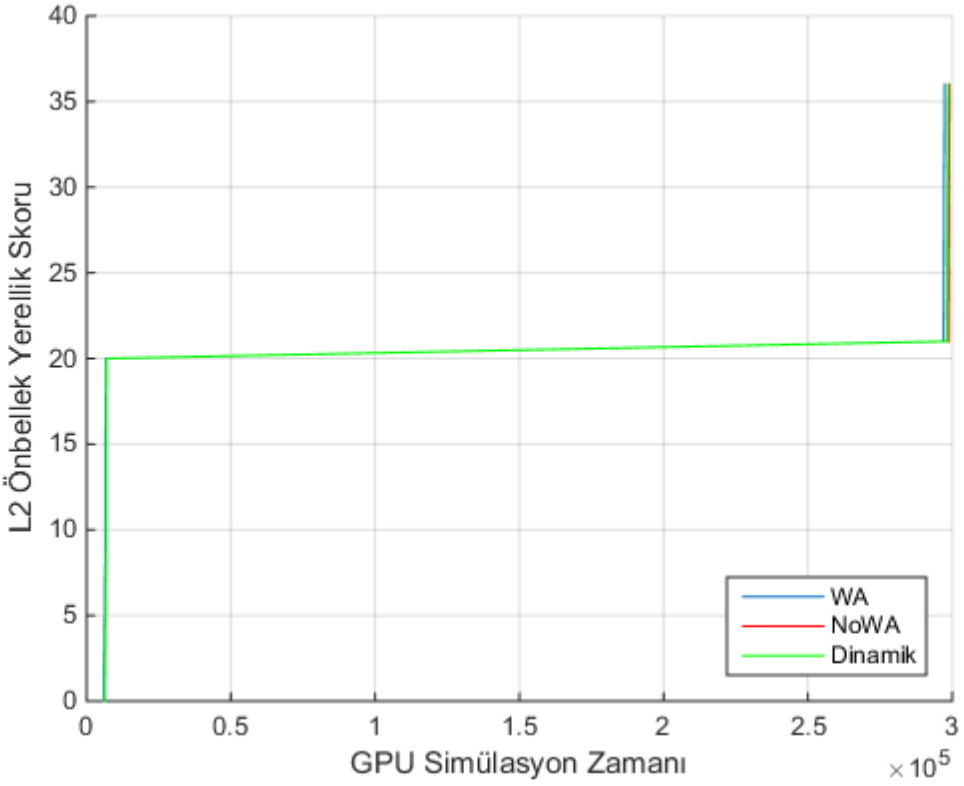


Şekil 6.28 : NN dinamik politika seçimi (Mavi: NoWA, Kırmızı: WA)

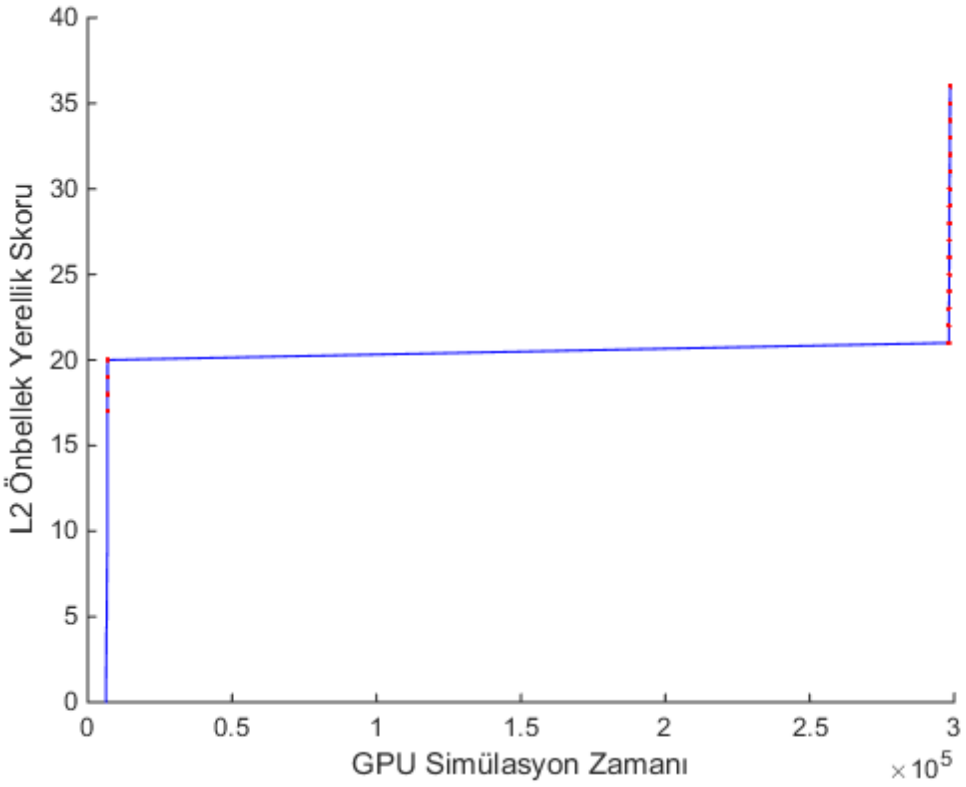
6.3.10 Gaussian 128 (%54 WA, %46 NoWA)

Tablo 6.14 : Gaussian istatistikleri

	WA	NoWA	Dinamik
IPC	91.5471	91.2301	91.2450
L1 Kayıp Oranı	0.4189	0.4190	0.4195
L2 Kayıp Oranı	0.0012	0.0026	0.0022
DRAM Gecikmesi	777388	779685	781234
Arabağlantı Ağı Gecikmesi	1387321	1389072	1387769
Paylaşımlı Bellek Gecikmesi	1418957	1441070	1444711



Şekil 6.29 : Gaussian politika skorları

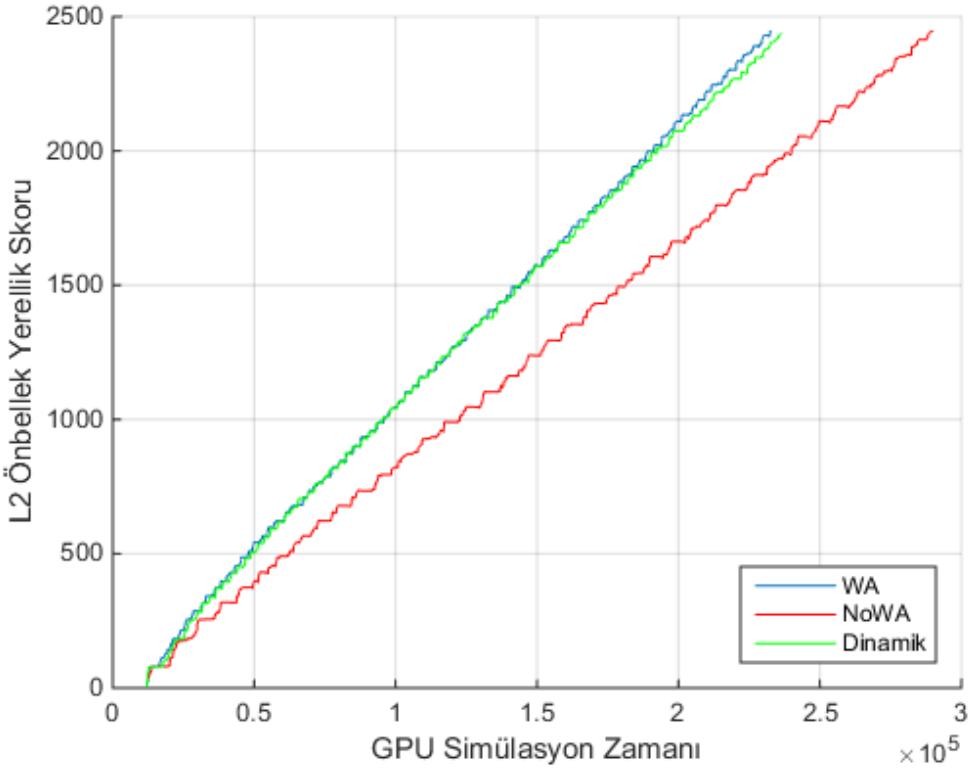


Şekil 6.30 : Gaussian dinamik politika seçimi (Mavi: NoWA, Kırmızı: WA)

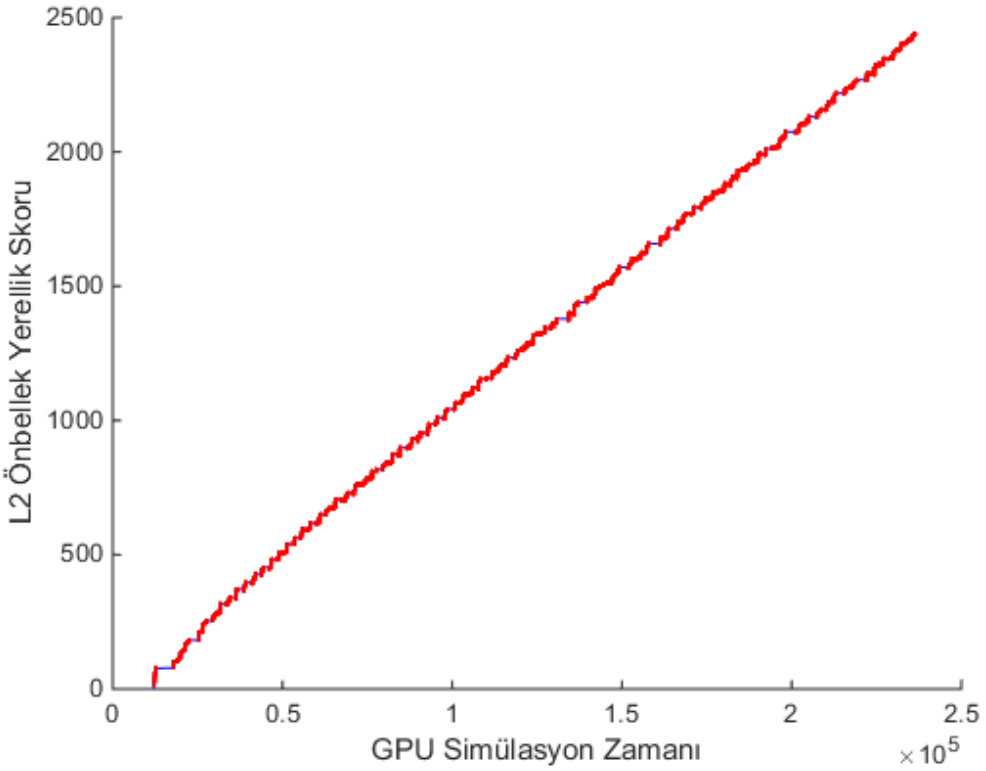
6.3.11 Hotspot (%98 WA, %2 NoWA)

Tablo 6.15 : Hotspot istatistikleri

	WA	NoWA	Dinamik
IPC	471.6226	376.2828	471.6226
L1 Kayıp Oranı	0.9690	0.9625	0.9672
L2 Kayıp Oranı	0.1497	0.4001	0.1536
DRAM Gecikmesi	66974	34663	60389
Arabağlantı Ağı Gecikmesi	124957	144197	124010
Paylaşımlı Bellek Gecikmesi	238057	528924	256251



Şekil 6.31 : Hotspot politika skorları

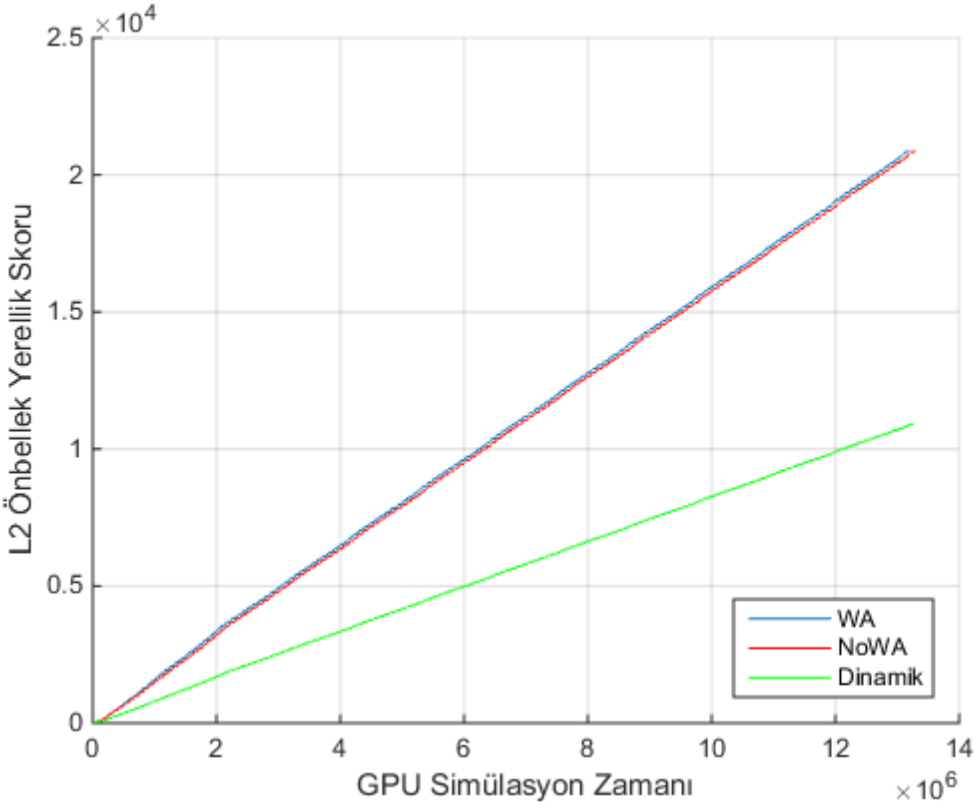


Şekil 6.32 : Hotspot dinamik politika seçimi (Mavi: NoWA, Kırmızı: WA)

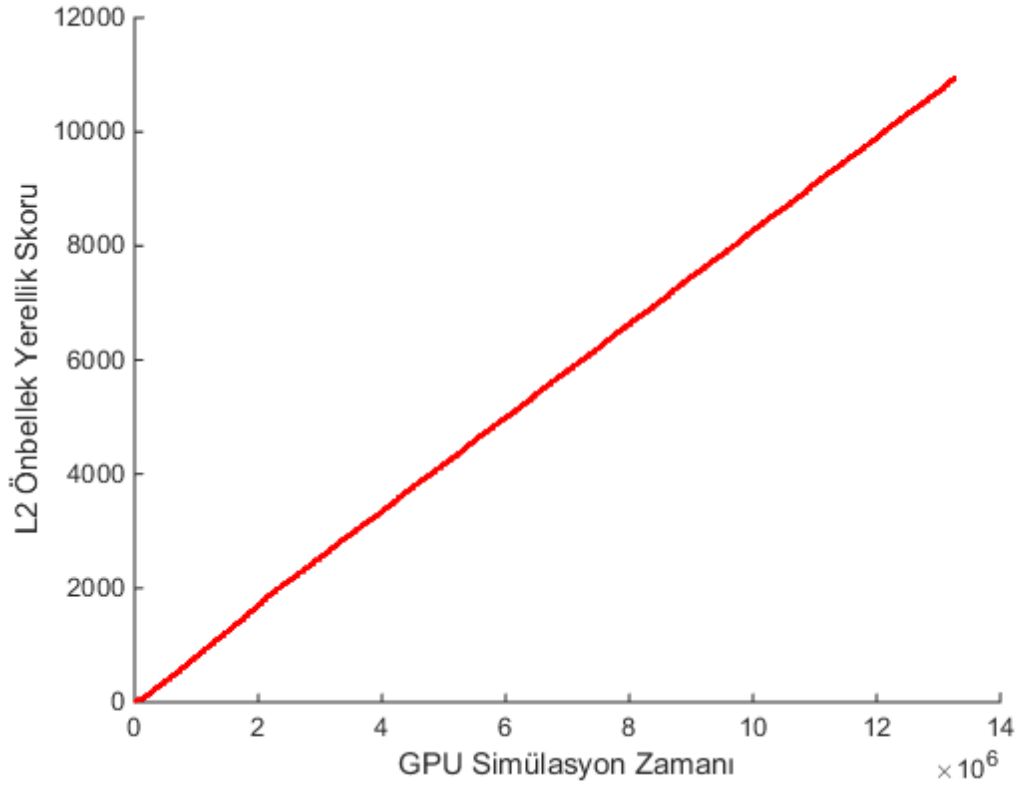
6.3.12 Myocyte (%99 WA, %1 NoWA)

Tablo 6.16 : Myocyte istatistikleri

	WA	NoWA	Dinamik
IPC	0.0917	0.0914	0.0916
L1 Kayıp Oranı	0.4719	0.4719	0.4719
L2 Kayıp Oranı	0.0059	0.1733	0.0061
DRAM Gecikmesi	0	0	0
Arabağlantı Ağı Gecikmesi	0	0	0
Paylaşımlı Bellek Gecikmesi	2311	2305	2305



Şekil 6.33 : Myocyte politika skorları

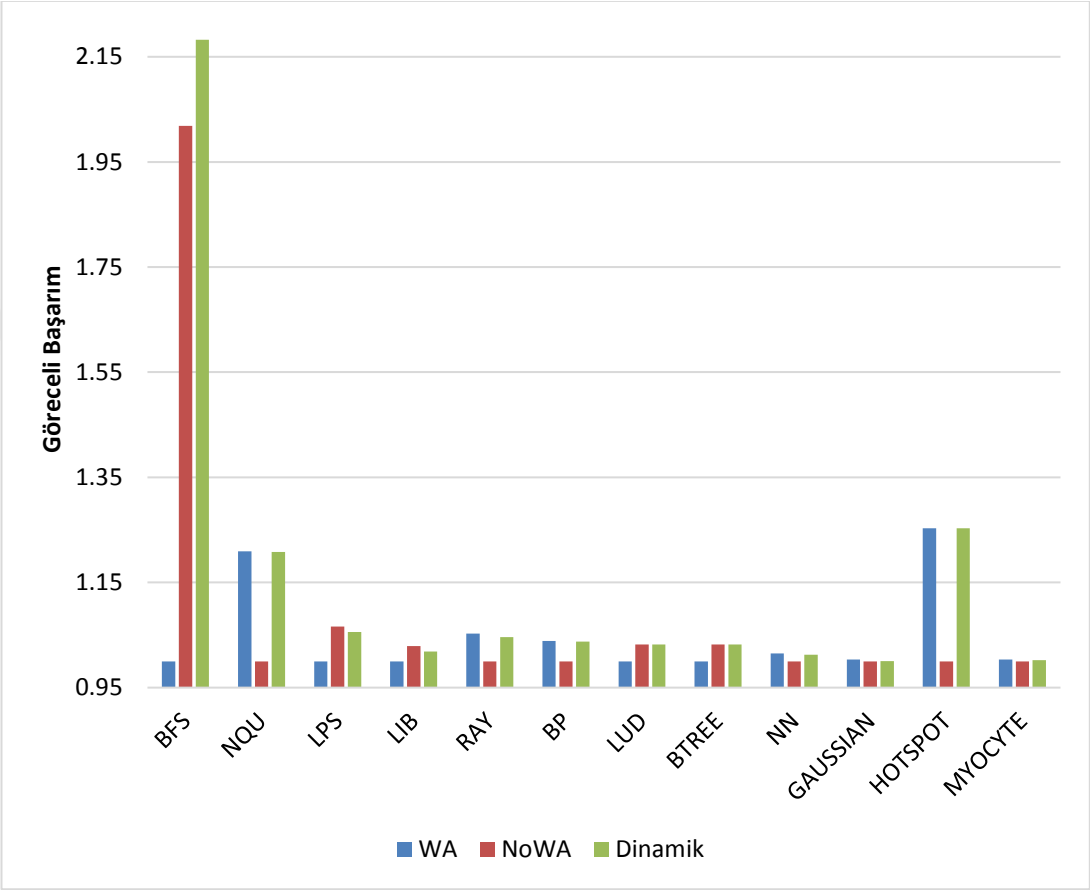


Şekil 6.34 : Myocyte dinamik politika seçimi (Mavi: NoWA, Kırmızı: WA)

6.4 Yorumlar

Farklı kütüphanelerden alınan uygulamalar WA, NoWA ve dinamik politikaları ile koşturulmuş ve başarımları gözlemlenmiştir. Dinamik politikanın başarım üzerindeki etkisinin VTA boyutu, skorlama katsayıları ve skor eşiği gibi parametrelerden etkilendiği görülmüştür. Şekil 6.35'de politikaların farklı uygulamalar üzerindeki başarıma etkileri gösterilmiştir. Başarım yüzdeleri en yavaş çalışan politikanın değeri %100 olarak normalize edilmiştir. Bazı uygulamalarda WA politikası en yavaş politika olurken diğer uygulamalarda NoWA politikası en yavaş politika olmuştur. Dinamik politika ise hiçbir uygulamada en yavaş politika olmamıştır. Dinamik politika uygulamaların çoğunda hızlı çalışan NoWA ya da WA uygulamasına yakınsarken bazı uygulamalarda hem WA hem de NoWA politikalarından daha iyi başarımlar elde etmiştir. 2 farklı kernelden oluşan BFS uygulamasının kernellerinin farklı politikalara verdiği tepki daha iyi olduğu gözlemlenmiş ve dinamik politika ile çalıştırılması ile diğer politikalara göre daha iyi başarım elde edilmiştir. Dinamik politikanın uygulamalar üzerindeki başarımı uygulama girdilerine göre de oldukça değişmektedir. Şekil 6.35'de BFS uygulamasının dinamik politika ile WA politikasına göre %16,

NoWA politikasına göre %218 daha hızlı çalışması 256K eleman girdili durumda görülmüştür. Başka boyutlardan oluşan girdilerle dinamik politikanın daha iyi ya da daha kötü başarım elde ettiği durumlar da gözlemlenmiştir. Bu yüzden de uygulama başarım karşılaştırma grafiğinin farklı girdiler ile tamamen değişebileceği de göz önünde bulundurulmalıdır.



Şekil 6.35 : Uygulamaların politika başarım karşılaştırması

KAYNAKLAR

[1] Nvidia, “GeForce 256” , http://www.nvidia.com/page/geforce256.html.

[2] Nvidia, “GeForce 3” , http://www.nvidia.com/page/geforce3.html.

[3] **P. N. Glaskowsk**y, “NVIDIA’s Fermi: The First Complete GPU Computing Architecture” Nvidia, 2009.

[4] **E. Lindholm, J. Nickolls, S. Oberman ve J. Montrym**, “NVIDIA Tesla: A unified graphics and computing architecture” *IEEE Micro* , cilt 28, no. 2, sf. 39-55, 2008.

[5] **J. Nickolls, I. Buck, M. Garland ve K. Skadron**, “Scalable Parallel Programming with CUDA” *Queue - GPU Computing*, cilt 6, no. 2, sf. 40-53, 2008.

[6] **J. E. Stone, D. Gohara ve G. Shi**, “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems” *Computing in Science & Engineering*, cilt 12, no. 3, sf. 66-73, 2010.

[7] **T. G. Rogers, M. O'Connor ve T. M. Aamodt**, “Cache-Conscious Wavefront Scheduling” *MICRO-45*, Vancouver, 2012.

[8] **Y. Tian, S. Puthoor, J. L. Greathouse, B. M. Beckmann ve D. A. Jimenéz**, “Adaptive GPU cache bypassing” *GPGPU-8*, San Francisco, 2015.

[9] **X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang ve W.-M. Hwu**, “Adaptive Cache Management for Energy-Efficient GPU Computing” *MICRO-47*, Cambridge, 2014.

[10] **X. Chen, S. Wu, L.-W. Chang, W.-S. Huang, C. Pearson, Z. Wang ve W. M. W. Hwu**, “Adaptive Cache Bypass and Insertion for Many-core Accelerators” *MES '14*, Minneapolis, 2014.

[11] **M. Khairy, M. Zahran ve A. G. Wassal**, “Efficient Utilization of GPGPU Cache Hierarchy” *GPGPU-8*, San Francisco, 2015.

[12] **A. Bakhoda, G. Yuan, W. W. L. Fung, H. Wong ve T. M. Aamodt**, "Analyzing CUDA Workloads Using a Detailed GPU Simulator" *ISPASS*, Boston, 2009.

[13] **N. P. Jouppi**, "Cache write policies and performance" *ISCA '93*, San Diego, 1993.

[14] **R. Ausavarungnirun, S. Ghose, O. Kayiran, G. H. Loh, C. R. Das, M. T. Kandemir ve O. Mutlu**, "Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance" *PACT '15*, San Francisco, 2015.

[15] **N. P. Jouppi**, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers" *17th Annual International Symposium on. IEEE,* Seattle, 1990.

[16] **D. Kroft**, "Lockup-Free Instruction Fetch/Prefetch Cache Organization" *ISCA '81*, Minnesota, 1981.

[17] **S. Song, M. Lee, J. Kim, W. Seo, Y. Cho ve S. Ryu**, "Energy-efficient scheduling for memory-intensive GPGPU workloads" *Design, Automation and Test in Europe Conference and Exhibition*, Dresden, 2014.

ÖZGEÇMİŞ

Ad-Soyad : Çağatay TURGUT
Uyruğu : T.C.
Doğum Tarihi ve Yeri : 21.04.1990 / İZMİR
E-posta : cagatayturgut@gmail.com

ÖĞRENİM DURUMU:

- **Lisans :** 2012, Bilkent Üniversitesi, Elektrik Elektronik Mühendisliği
- **Yüksek Lisans :** 2017, TOBB Ekonomi ve Teknoloji Üniversitesi, Bilgisayar Mühendisliği

MESLEKİ DENEYİM VE ÖDÜLLER:

Yıl	Yer	Görev
2013-2017	METEKSAN SAVUNMA	Sayısal Tasarım Mühendisi

YABANCI DİL: İngilizce

TEZDEN TÜRETİLEN YAYINLAR, SUNUMLAR VE PATENTLER:

- **Turgut Ç. , Ergin O.** "GPU Önbelleklerinde Yerelliğe Bağlı Dinamik Yazma Politikası", 19. Akademik Bilişim Konferansı (AB17) Sunumları, Aksaray, Şubat 2017.