TOBB EKONOMİ VE TEKNOLOJİ ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

DRAM ÜZERİNDE GERÇEK RASTGELE SAYI ÜRETME MEKANİZMALARI İÇİN SİSTEM TASARIMI

YÜKSEK LİSANS TEZİ

Fatma Nisa BOSTANCI

Bilgisayar Mühendisliği Anabilim Dalı

Tez Danışmanı: Prof. Dr. Oğuz ERGİN

TEMMUZ 2022

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.

Fatma Nisa BOSTANCI

İMZA

ÖZET

Yüksek Lisans Tezi

DRAM ÜZERİNDE GERÇEK RASTGELE SAYI ÜRETME MEKANİZMALARI İÇİN SİSTEM TASARIMI

Fatma Nisa BOSTANCI

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

Tez Danışmanı: Prof. Dr. Oğuz ERGİN

Tarih: TEMMUZ 2022

Rastgele sayı üretimi kriptografik algoritmalar, bilimsel simülasyonlar ve endüstriyel araçlar gibi birçok uygulama için önemlidir. Gerçek Rastgele Sayı Üreteçleri (GRSÜler) fiziksel entropi kaynaklarını örnekleyerek kriptografik olarak güvenli rastgele sayılar üretmektedir. GRSÜler bu nedenle genellikle fazladan donanıma ihtiyaç duymaktadır ve uzun gecikme sürelerine sahiptir. GRSÜler üzerine yapılan çalışmalarda, yüksek hızla ve düşük gecikmeyle gerçek rastgele sayı üretimini yaygın kullanılan cihazlarda gerçekleştirebilmek için DRAM aygıtlarını entropi kaynağı olarak kullanan mekanizmalar önerilmiştir. Bu çalışmalar, DRAM aygıtları örneklenerek rastgele sayı üretilebileceğini göstermektedir. Fakat önerilen mekanizmaların gerçek sistemlerde kullanılabilmesi için gereken uçtan uca sistem entegrasyonu birçok zorluğa sahiptir.

DRAM tabanlı GRSÜlerin güncel sistemlerde kullanılabilmesi için aşılması gereken üç temel zorluk tespit etmekteyiz: (1) DRAM tabanlı GRSÜler ile rastgele sayı üretimi, bellek denetleyicisinde asıl bellek istekleri ile rastgele sayı üretimi (RSÜ) istekleri arasında bir çatışma (*RSÜ çatışması*) oluşturarak tüm sistemin başarımını düşürebilmektedir, (2) bu çatışma yoğun bir biçimde rastgele sayı kullanan uygulamaların (*GRSÜ uygulamalarının*) önceliklendirilmesine sebep olarak sistem adilliğini düşürebilmektedir ve (3) DRAM tabanlı GRSÜlerin gecikmeleri nedeniyle GRSÜ uygulamalarında ciddi başarım kayıpları gözlemlenebilmektedir.

Bu zorlukları aşmak için DR-STRaNGe'i geliştirmekteyiz. DR-STRaNGe, (1) RSÜ istekleri ile bellek isteklerini bellek denetleyicisinde ayırarak RSÜ çatışmasını azaltan,

(2) RSÜ isteklerinin farkında olan bir bellek istek planlayıcısı ile sistem adilliğini artıran ve (3) DRAM kanallarında atıl çevrimleri öngören bir mekanizma kullanılarak doldurulan bir rastgele sayı arabelleği ile yüksek GRSÜ gecikmelerini saklayan bir sistem tasarımıdır.

DR-STRaNGe'i, 186 farklı çoklu programlanmış iş yükü kullanarak değerlendirmekteyiz. Deneysel değerlendirmelerimiz sonucunda, DRAM aygıtlarında rastgele sayı üretimini yoksayan sistemlere kıyasla uçtan uca sistem tasarımımızın, GRSÜ ve GRSÜ olmayan uygulamalarda ortalama sistem performansını sırasıyla %25.1 ve %17.9 artırdığını ve $5Gb/s$ hızla gerçek rastgele sayı üretirken ortalama sistem adilliğini %32.1 artırdığını göstermekteyiz. DR-STRaNGe'in GRSÜ ve GRSÜ olmayan bellek işlemlerinde harcanan süreyi %15.8 azalttığı için DRAM aygıtlarında rastgele sayı üretimini yoksayan sistemlere kıyasla enerji tüketimini %21 azalttığını göstermekteyiz.

Anahtar Kelimeler: Bellek, DRAM, Gerçek rastgele sayı.

ABSTRACT

Master of Science

END-TO-END SYSTEM DESIGN FOR DRAM-BASED TRUE RANDOM NUMBER GENERATORS

Fatma Nisa BOSTANCI

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

Supervisor: Prof. Dr. Oğuz ERGİN

Date: JULY 2022

Random number generation is an important task in a wide variety of critical applications including cryptographic algorithms, scientific simulations, and industrial testing tools. True Random Number Generators (TRNGs) produce cryptographically-secure truly random data by sampling a physical entropy source that typically requires custom hardware and suffers from long latency. To enable high-bandwidth and low-latency TRNGs on widely-available commodity devices, recent works propose hardware TRNGs that generate random numbers using commodity DRAM as an entropy source. Although prior works demonstrate promising TRNG mechanisms using DRAM, practical integration of such mechanisms into real systems poses various challenges.

We identify three key challenges for using DRAM-based TRNGs in current systems: (1) generating random numbers with DRAM-based TRNGs can degrade overall system performance by slowing down concurrently-running applications due to the interference between RNG and regular memory operations in the memory controller (i.e., *RNG interference*), (2) this RNG interference can degrade system fairness by causing unfair prioritization of applications that intensively use random numbers (i.e., *RNG applications*), and (3) RNG applications can experience significant slowdown due to the high latency of DRAM-based TRNGs.

To address these challenges, we propose DR-STRaNGe, an end-to-end system design for DRAM-based TRNGs that (1) reduces the RNG interference by separating RNG requests from regular memory requests in the memory controller, (2) improves fairness

across applications with an RNG-aware memory request scheduler, and (3) hides the large TRNG latencies using a random number buffering mechanism combined with a new DRAM idleness predictor that accurately identifies idle DRAM periods.
We evaluate DR-STRaNGe using a comprehensive set of 186 multi-programmed workloads. Compared to an RNG-oblivious baseline system, DR-STRaNGe improves the performance of non-RNG and RNG applications on average by 17.9% and 25.1%, respectively. DR-STRaNGe improves system fairness by 32.1% on average when generating random numbers at a 5 Gb/s throughput. DR-STRaNGe reduces energy consumption by 21% compared to the RNG-oblivious baseline design by reducing the time spent for RNG and non-RNG memory accesses by 15.8%.

Keywords: Memory, DRAM, True random number.

TEŞEKKÜR

Çalışmalarım boyunca değerli yardım ve katkılarıyla beni yönlendiren hocam Prof. Dr. Oğuz Ergin'e, kıymetli tecrübelerinden faydalandığım TOBB Ekonomi ve Teknoloji Üniversitesi Bilgisayar Mühendisliği Bölümü öğretim üyelerine, destekleriyle her zaman yanımda olan annem Mihriban Bostancı, babam Ali Namık Bostancı ve kardeşlerim Elif Ceren ve Ahmet Eyüp Bostancı'ya, sundukları üretken ve keyifli çalışma ortamı için Kasırga Mikroişlemciler Laboratuvarındaki çalışma arkadaşlarım Ataberk Olgun, İsmail Emir Yüksel, Alperen Bolat, Yahya Can Tuğrul, Esra Ayaz'a ve tezimi dikkatlice okuyup düzenlememe yardım eden Oğuzhan Canpolat ve Şevval İzmirli'ye teşekkürlerimi sunarım.

İÇİNDEKİLER

ŞEKİL LİSTESİ

Sayfa

ÇİZELGE LİSTESİ

Sayfa

KISALTMALAR

DRAM : Dynamic Random Access Memory
GRSÜ : Gerçek Rastgele Sayı Üreteci
PRNG : Pseudo Random Number Generator
SRSÜ : Sözde Rastgele Sayı Üreteci
TRNG : True Random Number Generator

SEMBOL LİSTESİ

Bu çalışmada kullanılmış olan simgeler açıklamaları ile birlikte aşağıda sunulmuştur.

Simgeler	**Açıklama**
Gb/s	gigabit saniye (aktarım hızı)
Mb/s	megabit saniye (aktarım hızı)
Gb/s	gigabit saniye (aktarım hızı)
ns	nanosaniye
μs	mikrosaniye
ms	milisaniye

1. GİRİŞ

Rastgele sayılar, kriptografik anahtar oluşturma, kimlik doğrulama, bilimsel simülasyonlar, Monte Carlo yöntemleri ve endüstriyel testler gibi birçok uygulamada kullanılmaktadır [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]. Bu uygulamalar genellikle yüksek performans elde etmek için yüksek verimli bir rasgele sayı üreteci gerektirmektedir [12, 13, 14].

Rastgele sayı üreteçleri iki sınıfa ayrılmaktadır [15, 16, 17, 18]. Birincisi, gerçek rasgele sayı üreteçleri (GRSÜ, *-ing. TRNG*) [19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 5, 43, 44, 4, 45] rastgele sayı üretmek için doğası gereği rastgele olan fiziksel süreçlerden (örneğin elektriksel gürültü, saat titreşimi, atmosferik gürültü ve Brownian hareketi) kaynaklanan entropiyi kullanmaktadır. İkincisi, sözde rasgele sayı üreteçleri (SRSÜ, *-ing. PRNG*) [46, 47, 48, 49, 50] rastgele sayı üretmek için bir çekirdek değeri (*-ing. seed value*) kullanarak deterministik bir sayı dizisi oluşturmaktadır. Bu sayı dizisi çekirdek değeri hakkında bilgi sahibi olmayan bir gözlemci için rastgele görünmektedir.

Kimlik doğrulama ve anahtar oluşturma gibi güvenlik uygulamaları için rastgele sayı kalitesi çok önemlidir [51, 5, 6, 7, 8, 9, 3, 10, 11, 52, 53, 54, 55, 56, 57]. SRSÜler deterministik sayı dizileri verdiğinden, güvenlik açısından kritik uygulamalar için tercih edilmemektedir [58, 6, 3]. SRSÜler yerine bu uygulamalar GRSÜleri kullanılmaktadır. SRSÜlerden farklı olarak GRSÜler, uygulama güvenliğini tehlikeye atabilecek öngörülebilir çekirdek değerlere bağlı rastgele sayılar üretmez.

DRAM hemen hemen tüm bilgisayar sistemlerinde yaygın olarak bulunmaktadır ve ana bellek olarak mobil ve IoT cihazlarına kolayca entegre edilebilmektedir. Neredeyse tüm sistemlerde ve yeni tasarlanmakta olan bellekte işleme (*-ing. processing-in-memory*) mimarilerinde düşük maliyetli ve yüksek verimli gerçek rastgele sayı üretimini sağlamak için önceki çalışmalar [20, 21, 22, 23, 59, 12, 60, 13], DRAM aygıtlarında gözlemlenen üretim sürecinden kaynaklanan değişiklikleri ve bu değişikliklerin ortaya çıkardığı entropiyi kullanarak rastgele sayı üretme mekanizmaları önermektedir. Ancak, hiçbir çalışma bu mekanizmaların günümüz sistemlerinde kullanılabilmesi için gerekli olan uçtan uca sistem entegrasyonu tasarımını sağlamamaktadır.

DRAM tabanlı GRSÜlerin günümüz sistemlerine entegrasyonunda aşılması gereken üç temel zorluk tespit etmekteyiz. İlk olarak, DRAM aygıtlarında rastgele sayı üretimi, DRAM aygıtının ana bellek olarak kullanıldığı sistemlerde asıl bellek isteklerinin gerçekleştirilmesine müdahalede bulunabilmektedir. Bunun sonucunda RSÜ ve diğer bellek istekleri arasında bir çatışma ortaya çıkmaktadır (*RSÜ çatışması*). Bu çatışma, aynı anda çalışmakta olan uygulamalarda önemli yavaşlamalara sebep olabilmektedir. İkinci olarak, var olan bellek istek planlayıcısı (*-ing. memory scheduler*) tasarımları bellek isteklerini yüksek sistem adilliği ve yüksek performansı

yakalamayı hedefleyerek planlamaktadır. Ancak bu tasarımlar istekleri planlarken DRAM aygıtlarında rastgele sayı üretimini ve GRSÜ ile diğer bellek istekleri arasındaki farklı karakteristik özellikleri yok saymaktadır. GRSÜ istekleri bellek istek planlayıcısına çoklu olarak gelmektedir. GRSÜ istekleri ve diğer istekleri birbiri ardına tamamlamak zamanlama parametrelerini değiştirmeyi gerektirerek ek bir gecikme yaratmaktadır. Bu nedenle RSÜ istekleri kendi aralarında gruplanarak, art arda tamamlanmaktadır. Son olarak, DRAM aygıtlarında rastgele sayı üretimi yüksek gecikmeye sahiptir ve bu gecikme nedeni ile rastgele sayı kullanan uygulamalar (GRSÜ uygulamaları) bellekte uzun süre boyunca isteklerinin karşılanmasını bekleyebilmektedir. DRAM tabanlı GRSÜler rastgele sayı üretmek için yeterli sayıda rastgele biti DRAM aygıtlarında birden çok okuma işlemi gerçekleştirerek toplamaktadır. Rastgele sayı isteğinden sonra gelen buyruklar rastgele sayıya bağlı ise GRSÜ işlemcideki buyruk penceresini durdurabilmektedir. Bu nedenle bir GRSÜ uygulaması sistemde yürüyen tek uygulama olsa dahi bellek isteklerinin karşılanmasını uzun bir süre bekleyebilmektedir.

Bu tezde bahsi geçen çalışmamızdaki amacımız, düşük maliyetli ve yüksek başarıma sahip uçtan uca bir sistem tasarımı geliştirerek günümüz sistemlerinde DRAM tabanlı GRSÜlerin kullanılabilmesini sağlamaktır. Bu tasarımın (1) GRSÜ çatışmalarını azaltarak ve kontrol ederek GRSÜ ve GRSÜ olmayan uygulamaların yaşadığı gecikmeleri azaltması, (2) GRSÜ isteklerinin farkında olan bir bellek istek planlayıcısı kullanarak sistem adilliğini artırması ve (3) GRSÜ uygulamalarının yüksek GRSÜ gecikmeleri nedeniyle yaşadığı performans kaybını engellemesi amaçlanmaktadır. Bu amaç doğrultusunda, DRAM tabanlı GRSÜler için yeni bir uçtan uca sistem tasarımı olan DR-STRaNGe'i (*-ing. End-to-End **S**ystem design for **DR**AM-based **T**rue **Ra**ndom **N**umber **Ge**nerators*) tasarlamaktayız. DR-STRaNGe üç temel parçadan oluşmaktadır. İlk olarak DR-STRaNGe GRSÜ çatışmasını engellemek ve GRSÜ gecikmelerini azaltmak için bir rastgele sayı depolama mekanizması *(Rastgele Sayı Depolama Mekanizması)* gerçeklemektedir. Rastgele Sayı Depolama Mekanizması, (1) DRAM kanallarında herhangi bir istek olmadığı için boş geçirilen çevrimleri öngören ve bu çevrimleri kullanarak GRSÜ çatışmasını azaltan bir öngörücü ve (2) rastgele sayıları bellek denetleyicisinde saklayarak daha sonra gelen GRSÜ isteklerini düşük gecikme ile karşılayan bir arabellekten oluşmaktadır. İkinci olarak DR-STRaNGe GRSÜ isteklerinin farkında olan bir bellek istek planlayıcısı *(GRSÜ Bellek İstek Planlayıcısı)* gerçeklemektedir. GRSÜ Bellek İstek Planlayıcısı sistem adilliğini artırır ve isteklerin GRSÜ nedeni ile bellekte bekleme süreyi azaltmaktadır. GRSÜ Bellek İstek Planlayıcısı, GRSÜ istekleri için ikinci bir istek kuyruğu oluşturmaktadır, bu sayede GRSÜ istekleri ile bellek isteklerini birbirinden ayırmaktadır. Daha sonra bu istek kuyrukları arasında işletim sistemi tarafından atanan program önceliklerine göre önceliklendirme yapmaktadır. Üçüncü olarak DR-STRaNGe rastgele sayı kullanan uygulamalar için bir arayüz gerçeklemektedir. Bu arayüz sayesinde uygulamalar düşük gecikme ve yüksek hızla DRAM aygıtları üzerinde rastgele sayı üretebilmektedir. DR-STRaNGe seçilen DRAM tabanlı GRSÜ mekanizmasından bağımsızdır ve önerilen tüm DRAM tabanlı GRSÜ mekanizmaları ile uyumludur.

DR-STRaNGe'in başarımını çeşitli uygulamalar kullanarak ölçmekteyiz. Deneysel değerlendirmelerimiz sonucunda, DRAM aygıtlarında rastgele sayı üretimini yok sayan sistemlere kıyasla DR-STRaNGe'in (1) GRSÜ ve GRSÜ olmayan uygulamaları sırası ile %25.1 ve %17.9 hızlandırdığını, (2) $5\,Gb/s$ hızla rastgele sayı üretirken sistem

adilliğini %32.1 artırdığını, (3) GRSÜ ve GRSÜ olmayan bellek işlemleri için bellekte %15.8 daha az zaman harcayarak enerji tüketimini %21 düşürdüğünü göstermekteyiz. DR-STRaNGe 22*nm* teknolojisi ile 0.0022mm^2 alan kaplamaktadır (bir Intel Cascade Lake CPU çekirdeğinin [61] %0.00048'i kadardır). Bu tezde sunduğumuz katkılar aşağıda verilmiştir:

1. DR-STRaNGe DRAM tabanlı GRSÜ mekanizmaları için uçtan uca sistem tasarımının en önemli üç zorluğunu aşan ilk sistem tasarımıdır.
2. Etkili rastgele sayı depolama teknikleri ile GRSÜ gecikmesinin ve GRSÜ çatışmasının azaltılabileceğini göstermekteyiz. DRAM kanallarında kullanılmayan çevrimleri yüksek başarı ile öngören (%80) bir öngörücü ile rastgele sayı depolayan bir mekanizmayı ilk kez ortaya koymaktayız.
3. GRSÜ isteklerinin istek planlamada dikkate alındığı, sistem adilliğini artıran ve GRSÜ nedenli bellek gecikmelerini azaltan bir bellek istek planlayıcısı önermekteyiz.
4. DR-STRaNGe'in başarım, adillik ve enerji tüketimini farklı uygulamalardan oluşan program paketleri ile değerlendirmekteyiz.
5. DR-STRaNGe'i daha önce önerilen en yüksek başarıma sahip iki DRAM tabanlı GRSÜ mekanizması ile değerlendirmekteyiz: D-RaNGe [12] ve QUAC-TRNG [13]. DR-STRaNGe'in bu iki mekanizma ile uyumlu olduğunu ve ikisinin de başarımını ve sistem adilliğini artırdığını ve enerji tüketimini düşürdüğünü göstermekteyiz.

2. TEMEL BİLGİLER

Bu bölümde DRAM organizasyonu, bellek istek planlayıcıları ve DRAM tabanlı GRSÜler ile ilgili temel bilgileri açıklamaktayız.

2.1 DRAM Organizasyonu

DRAM tabanlı ana belleklere bir bellek kanalı (*-ing. memory channel*) aracılığı ile erişilmektedir. Her bir kanal birden çok DRAM grubuna (*-ing. DRAM bank*) bağlıdır. Her grup DRAM hücrelerinden (*-ing. DRAM cell*) oluşan iki boyutlu bir dizi içermektedir ve bu dizi satırlar ve sütunlar olarak düzenlenmiştir. Bir istek yapıldığında, DRAM hücreleri satır büyüklüğünde getirilir ve getirilen satır, satır arabelleğinde (*-ing. row buffer*) saklanır. Bir istek, satır arabelleğinde bulunan satıra yapılıyorsa daha hızlı karşılanır. Bu duruma arabellekte bulma [62] (*-ing. row buffer hit*) denmektedir. DRAM organizasyonu ile ilgili daha fazla bilgi için, okuyucuyu geniş çaplı olarak DRAM organizasyonu üzerine araştırma yapan önceki çalışmalara [63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93] yönlendirmekteyiz.

2.2 Bellek İstek Planlayıcısı

FR-FCFS (First Ready First Come First Serve) bellek istek planlayıcısı [94, 95], ilk olarak satır arabelleğinde bulunacak istekleri diğer isteklerden, sonrasında ise daha yaşlı istekleri genç isteklerden yüksek öncelikle planlamaktadır. Bu şekilde FR-FCFS planlayıcısı satır arabelleğini kullanarak en yüksek verimi elde etmeyi amaçlamaktadır. Ancak yoğun bellek isteklerine sahip uygulamalar ve satır arabelleğinden yüksek oranda yararlanan uygulamalar, bu planlayıcı tarafından haksız bir biçimde önceliklendirilmektedir [96, 97].

Önceki çalışmalar [98, 99, 100, 101, 102, 103, 97, 104, 105, 106, 107, 108] uygulamaların bellek istek örüntülerini dikkate alarak hem performansı hem de sistem adilliğini tüm uygulamalar için artıran bellek istek planlayıcıları önermektedir. Bu planlayıcılar ya uygulamaları inceleyerek bir sıralama oluşturmakta [98, 99, 100, 101, 97, 102] ya da uygulamaları kara listeye alma metotlarını [108, 103] kullanmaktadır. Sıralama tabanlı planlayıcılar düşük sıralamaya sahip uygulamaları adil olmayan bir biçimde yavaşlatabilmektedir ve yüksek donanım karmaşıklığına sahiptir. Kara liste metodu ile planlama yapan BLISS [108, 103] ise basit bir bellek istek planlayıcısıdır. Uygulamaları iki gruba ayırarak planlama yapmaktadır. Hiçbir geçmiş çalışma bellekte rastgele sayı üretimini dikkate alarak bellek isteklerini planlamamaktadır ve rastgele sayı üretiminin yüksek gecikmelerini ve farklı zamanlama parametrelerine sahip isteklerin yarattığı farklı gecikmeleri planlama için göz önünde bulundurmamaktadır.

2.3 DRAM Tabanlı Gerçek Rastgele Sayı Üreticileri

Önceki çalışmalar birçok DRAM tabanlı GRSÜ mekanizması önermektedir. Bu çalışmalar DRAM zamanlama hatalarını (*-ing. DRAM timing failures*), yenileme işleminin yapılmamasına bağlı sızdırma hatalarını (*-ing. retention failures*) ve başlangıç değerlerini (*-ing. start-up values*) entropi kaynağı olarak kullanmaktadır.

Sızdırma hataları ve başlangıç değerlerine bağlı GRSÜler sağlayabildikleri rastgele sayı üretme hızı açısından kısıtlıdır [12] çünkü sızdırma hataları oda sıcaklığında birkaç dakikaya varan gecikmelere sahiptir [89, 87, 90, 86, 109, 110] ve rastgele başlangıç değerleri yalnızca pahalı güç döngüleri sonucunda oluşturulabilmektedir [111]. Bu nedenle bu mekanizmalar, yüksek hızda ve sürekli rastgele sayı üretimi için uygun değildir.

DRAM zamanlama hatalarını kullanarak rastgele sayı üreten mekanizmalar sürekli olarak kullanılabilmektedir ve bu mekanizmalar yüksek hıza ($> 100\ Mb/s$) sahiptir. Bu GRSÜler DRAM üreticileri tarafından hücrelerin düzgün çalışmasını sağlamak amacı ile belirlenen zamanlama parametrelerini bilinçli olarak ihlal ederek çalışmaktadır. Rastgele DRAM zamanlama hataları, dikkatle tasarlanmış geçerli DRAM komutları aracılığı ile DRAM aygıtlarında hızlı bir biçimde oluşturulabilmektedir [12, 60, 13].

DRAM aygıtlarında rastgele sayı üretimi için önceki çalışmalar [12] rastgele sayı üretimi için seçilmiş hücreleri (*-ing. RNG cells*), zamanlama parametreleri ihlal edildiğinde rastgele sayı veren hücreleri, içeren satırları ayırmaktadır. GRSÜ hızı ayrılan satırdaki GRSÜ hücrelerinin sayısına bağlıdır. GRSÜ gecikmesi ise rastgele sayı üretimi için kullanılan geçerli DRAM komutlarının toplam gecikmesine bağlıdır. Rastgele sayı üretimi, özellikle istenen GRSÜ hızı çok yüksek ise, sistemde yürütülmekte olan diğer uygulamaların performansını etkileyebilmektedir. Bunun iki temel sebebi vardır. İlk olarak GRSÜ birden çok ayrılmış satıra birden çok okumaya ihtiyaç duymaktadır. İkinci olarak ise, standart olmayan zamanlama parametreleri sebebi ile DRAM, rastgele sayı üretildiği sırada başka uygulamaların yanlış veri okumasını ya da yazmasını önlemek amacı ile meşgul olmaktadır.

3. MOTİVASYON

GRSÜler kriptografik anahtar oluşturma, kimlik doğrulama, ilk değer ve dolgu bitleri oluşturma ve donanım saldırıları için savunma mekanizmaları gibi birçok güvenlik uygulamasında kullanılır [5, 6, 7, 8, 9, 10, 11, 3, 81]. Rastgele sayıların kalitesi kullanıcı bilgilerini ele geçirmeye yönelik saldırılara karşı sistem güvenliğini sağlamak için önemlidir [6, 3]. Yeni tasarlanan güvenlik protokolleri (örneğin, kuantum anahtar dağıtımı protokolleri [52, 53]) zayıf rastgele sayıları hedef alan saldırılara karşı daha yüksek güvenlik garantisi sunar. Bu protokoller yüksek hızlı (Gb/s seviyelerinde) rastgele sayı üretimini gerektirir [14].

Yüksek hızlı DRAM tabanlı GRSÜler genellikle özelleştirilmiş donanıma ihtiyaç duyan diğer GRSÜ tasarımlarına göre daha geniş bir ölçekte kullanılabilir çünkü DRAM aygıtları genellikle tüm bilgisayar sistemlerinde ve yeni geliştirilmekte olan bellekte işleme mimarilerinde [112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 66, 125, 92, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 71, 140, 116, 141, 142, 91, 114, 143, 144, 145, 146, 147, 148, 149, 150, 151] kullanılmaktadır. DRAM tabanlı GRSÜler bu sistemlerde yürütülen güvenlik kritik ugulamalara yüksek hızla gerçek rastgele sayı üretebilir. Bu sistemlerde DRAM aygıtlarında rastgele sayı üretiminin iki temel faydası vardır. Birincisi, DRAM tabanlı GRSÜler rastgele sayı üretimi için fazladan donanım sağlanmayan sistemlerde (örneğin, mobil ve IoT cihazlarında) güvenlik uygulamalarının yürütülebilmesini sağlar. İkincisi, bellekte işleme mimarilerinde DRAM tabanlı GRSÜler (1) büyük kod bölümlerinin tamamen bellekte yürütülebilmesini sağlayarak sistem verimini artırır ve (2) güvenlik görevlerinin tamamen bellekte yürütülebilmesini sağlayarak güvenliği artırır.

Önceki çalışmalar yüksek hızlı DRAM tabanlı GRSÜler önermiştir ancak hiçbir çalışma bu mekanizmaların kullanılabilmesi için uçtan uca bir sistem tasarımı sağlamamıştır. DRAM tabanlı GRSÜlerin günümüz sistemlerine entegrasyonunda aşılması gereken üç temel zorluk tespit ediyoruz: (1) GRSÜ ve GRSÜ olmayan uygulamalar arasında bellek denetleyicisinde ortaya çıkan ve sistemi yavaşlatan GRSÜ çatışması, (2) GRSÜ isteklerinin bellek istek planlayıcısında haksız yere önceliklendirilmesi ile sistem adilliğinin düşmesi ve (3) GRSÜ uygulamalarını yavaşlatan yüksek DRAM tabanlı GRSÜ gecikmeleri.

DRAM tabanlı GRSÜlerin rastgele sayı üretimini yok sayan bir sistem üzerindeki etkisini göstermek için, iki çekirdekli gerçekçi bir sistemi ve DRAM tabanlı bir GRSÜ mekanizmasını benzetim aracı ile modellemekteyiz. Bu sistem rastgele sayı üretmek için DRAM aygıtında önceden belirlenmiş satırları zamanlama parametrelerini düşürerek okumaktadır [12]. Bu okumalar sonucunda gözlemlenen rastgele hatalar 64 bit uzunluğunda rastgele sayılar oluşturmak için kullanılmaktadır. DRAM aygıtlarında rastgele sayı üretimi sırasında sistem diğer bellek isteklerini hatalı okumamak için bekletmektedir. Rastgele sayı üretimini en kısa sürede yapmak için sistem tüm bellek

kanallarını aynı anda kullanır ve diğer bellek isteklerinin bekletildiği zamanı azaltır. Aynı anda tek kanal kullanılması da mümkündür ancak bu durumda rastgele sayı üretimi daha çok zaman almaktadır ve seçilen kanala yapılan diğer bellek istekleri daha uzun bir süre bekletilmektedir. Bu nedenle tüm DRAM kanallarının ve kümelerinin kullanılması GRSÜ çatışmasını azaltmak ve en kısa sürede rastgele sayı isteklerini karşılamak için önemlidir.

Deneylerimizde 172 adet GRSÜ ve GRSÜ olmayan uygulamadan oluşan iki çekirdekli iş yükü kullanmaktayız. GRSÜ uygulamaları için dört farklı sentetik uygulama oluşturmaktayız. Sentetik GRSÜ uygulamalar sırayla $640Mb/s$, $1280Mb/s$, $2560Mb/s$, ve $5120Mb/s$ hızla rastgele sayı isteği göndermektedir. GRSÜ olmayan uygulamalar için sık kullanılan 4 farklı kıyaslama ugulama paketi içinden 43 adet tek çekirdekli uygulama kullanmaktayız (SPEC CPU2006 [152], TPC [153], MediaBench [154] ve YCSB [155]). Oluşturulan iş yükleri Çizelge 3.1'da gösterilmektedir.

Çizelge 3.1: Çok Çekirdekli İş Yükleri.

İş Yükleri	**GRSÜ Olmayan Uygulamalar**	**GRSÜ Uygulamaları**
2 Çekirdek	43 uygulama	x1 GRSÜ uygulaması (640 Mb/s GRSÜ hızı) x1 GRSÜ uygulaması (1280 Mb/s GRSÜ hızı) x1 GRSÜ uygulaması (2560 Mb/s GRSÜ hızı) x1 GRSÜ uygulaması (5120 Mb/s GRSÜ hızı)

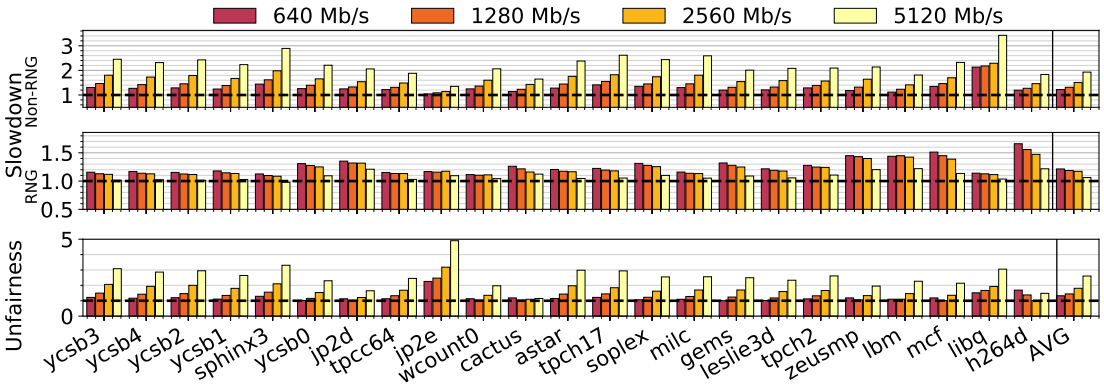
Şekil 3.1 (üst ve orta) GRSÜ ve GRSÜ olmayan uygulamaların toplam yürütme zamanını, her uygulamanın tek başına yürütülmesine göre normalize ederek göstermektedir. Şekil 3.1 (alt) sistemin aynı iki çekirdekli iş yükü için adaletsizlik endeksini göstermektedir. Adaletsizlik endeksi, yaşanan en büyük bellek kaynaklı yavaşlamanın yaşanan en az bellek kaynaklı yavaşlamaya oranı ile hesaplanır [97, 156, 96] ve 7. Bölümde ayrıntılı olarak açıklanmaktadır. Adaletsizlik endeksinin 1 olması tüm uygulamalarda eşit oranda bellek kaynaklı yavaşlama gözlemlendiğini göstermektedir. Yüksek adaletsizlik endeksi bir ya da birden fazla uygulamanın bellek istek planlayıcısı tarafından adil olmayan bir biçimde önceliklendirildiğini göstermektedir.

GRSÜ Çatışmasının Etkisi. Şekil 3.1 (üst ve orta) hem GRSÜ hem de GRSÜ olmayan uygulamaların bellek denetleyicisindeki GRSÜ çatışması sebebi ile ciddi yavaşlamalara maruz kaldığını göstermektedir.

GRSÜ olmayan uygulamaların yavaşlama miktarı GRSÜ uygulamasının gerektirdiği rastgele sayı üretim hızı arttıkça artmaktadır. Gereken GRSÜ hızı $5\,Gb/s$ olduğunda GRSÜ olmayan uygulamalar ortalama %93.1 yavaşlamaktadır. Şekil 3.1 (orta) GRSÜ uygulamalarının belleği başka bir uygulama ile paylaşması nedeniyle yavaşladığını göstermektedir. Bu deney sonucunda, GRSÜ uygulamalarından en fazla ve en az hızla rastgele sayı isteğinde bulunan uygulamaların sistemde tek başına yürütüldüklerine kıyasla sırası ile ortalama %21.4 ve %6.2 yavaşladığını göstermekteyiz.

Adil Olmayan Önceliklendirme. Şekil 3.1 (alt) adaletsizlik endeksini gösterir. Bu şekilde uygulamaların adaletsizlik endeksinin gereken GRSÜ hızı ile arttığını göstermekteyiz. Ortalamada, $640Mb/s$ hızı için iş yüklerinin 1.32 adaletsizlik endeksine sahip olduğunu ve bu endeksin GRSÜ hızı $5120Mb/s$ çıktığında 2.61'e yükseldiğini göstermekteyiz.

GRSÜ Gecikmesinin Etkisi. GRSÜ uygulamalarının yüksek GRSÜ gecikmesi sebebi ile, yüksek GRSÜ hızı gerektiği durumlarda yürütme zamanlarının %58.8'ini rastgele sayı üretimine harcadığını gözlemlemekteyiz.



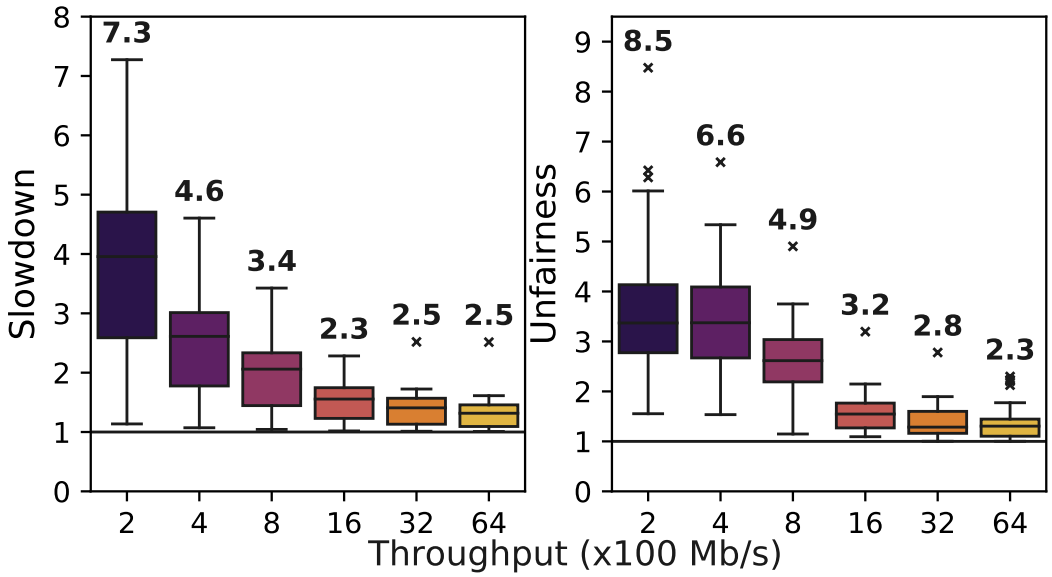
Şekil 3.1: Çeşitli gereken GRSÜ hızlarına göre GRSÜ olmayan (üst) ve GRSÜ uygulamalarının (orta) birlikte yürütüldüğü sistemdeki yavaşlama ve adaletsizlik oranı (alt).

GRSÜ Hızının Etkisi. GRSÜ hızının etkisini görmek için 6 farklı DRAM tabanlı GRSÜ mekanizmasını deneylerimizde kullanmaktayız. Bu mekanizmaların GRSÜ hızları 200 Mb/s'den 6.4 Gb/s'a artmaktadır. Önerilen en yüksek başarıma sahip iki DRAM tabanlı GRSÜ mekanizmasının, D-RaNGe [12] ve QUAC-TRNG [13]'nin GRSÜ hızları sırası ile ortalama $\sim 563 Mb/s$ ve $\sim 3.44 Gb/s$ hızla rastgele sayı üretebildiği gözlemlenmektedir. Bu değerler GRSÜ için önerilen en yüksek başarıma sahip sistem düzeni [13] varsayılarak hesaplanmaktadır.[1] Deneyimizde 43 iki çekirdekli, GRSÜ ve GRSÜ olmayan uygulamalardan oluşan iş yükleri kullanılmaktadır. Şekil 3.2 solda GRSÜ olmayan uygulamaların yavaşlama dağılımlarını ve sağda sisetmin adaletsizlik endeksini 43 iş yükü için göstermektedir.

Şekildeki her kutu, x-ekseninde verilen hızla rastgele sayı üreten GRSÜ mekanizmasına sahip iki çekirdekli temel sistem için gözlemlenen yavaşlama ve adaletsizlik endekslerinin çeyrekler açıklığını göstermektedir. Her bir çeyrekler açıklığının medyan değeri onu temsil eden kutunun orta noktası olarak verilmektedir. Kutuların üst kısmında yer alan işaretlemeler ($\times$) çeyrekler açıklığının dışında kalan aykırı değerleri göstermektedir. Aykırı değerler üst çeyrekten 1.5 kat daha fazla olan değerler olarak tanımlanmaktadır. Her bir kutu için gözlemlenen en yüksek yavaşlama ve adaletsizlik endeksini kutu üzerine eklenen etiket ile gösterilmektedir.

Bu deney sonuçlarından hareketle iki gözlemde bulunmaktayız. İlk gözlemimiz, en yüksek hıza ve en düşük gecikmeye sahip DRAM tabanlı GRSÜ mekanizmalarının kullanıldığı sistemlerde iş yüklerinde önemli yavaşlamalar ve sistem adilliği düşüşü görüldüğüdür. En yüksek hızlı GRSÜ ile bile ortalama %39.9 yavaşlama ve %28.5 sistem adilliği düşüşü görülmektedir. İkinci gözlemimiz, gelecekte geliştirilebilecek daha yüksek hıza sahip varsayımsal DRAM tabanlı GRSÜ mekanizmaları ile bile

[1] Tüm tasarımlar yalnızca GRSÜ hızının etkisini gösterebilmek için, D-RaNGe'in raporladığı düşük gecikme değerleri varsayılarak tasarlanmaktadır [12]. Ancak QUAC-TRNG [13]'nin GRSÜ gecikmesi bu şekilde varsayılan gecikmeden daha yüksek raporlanmıştır. Bu nedenle, QUAC-TRNG mekanizmasına sahip bir sistemin gerçek yavaşlama ve adaletsizlik endeksi değerleri daha yüksek olacaktır.



Şekil 3.2: DRAM tabanlı GRSÜ hızının GRSÜ olmayan uygulamaların hızına ve sistem adilliğine etkisi.

sistem başarımı ve adilliği önemli ölçüde iyileştirilememektedir. Deneylerimizde gözlemlediğimiz maksimum yavaşlama ve adaletsizlik endeksi değerleri GRSÜ hızının 3.2 Gb/s seviyesine gelmesi ile denge noktasına ulaşmaktadır.

Amacımız DRAM tabanlı GRSÜler için düşük maliyetli ve yüksek başarımlı uçtan uca bir sistem tasarımı geliştirmektir. Bu sistem tasarımı (1) GRSÜ ve GRSÜ olmayan uygulamalardaki GRSÜ çakışmasını azaltmalı, (2) sistem adilliğini GRSÜ ve GRSÜ olmayan uygulamalar için artırmalı ve (3) GRSÜ uygulamalarının başarımını düşüren DRAM tabanlı GRSÜlerin yüksek gecikmelerini saklamalıdır.

4. TASARIM FIRSATLARI

Bu bölümde DRAM tabanlı GRSÜler için uçtan uca bir sistem için tasarım fırsatlarını tartışmaktayız.

4.1 Rastgele Sayı Depolama

GRSÜ tasarımında daha önce yapılan çalışmalarda [12, 13] bellek denetleyicisinin DRAM aygıtlarında rastgele sayı üretimini belleğin istek yoğunluğunun düşük olduğu zamanlarda yapılabildiği ve üretilen rastgele sayıların bir arabellekte tutulabildiği varsayılmaktadır. DRAM kanallarına gelecek istekler önceden bilinemediği için bu varsayım DRAM kanallarında atıl zamanı öngören bir öngörücü olmadan tamamlanamaz. DRAM atıllık (*-ing. idleness*) öngörücüsü ile beraber çalışan bir depolama mekanizması, DRAM kanallarındaki atıl çevrimleri yüksek başarım ile öngörerek bu çevrimleri rastgele sayı üreterek doldurmak için gereklidir. Bu sayede mekanizma, düşük performans etkisi ile rastgele sayı üretebilmektedir.

4.2 Bellek İstek Planlayıcısı

GRSÜ gecikmesi sistem performansı için önemli bir faktördür çünkü hem GRSÜ hem de GRSÜ olmayan uygulamalardaki bellek işlemlerinin bellekte beklediği zamanı artırmaktadır. Bellek istek planlayıcısı, bellekte rastgele sayı üretimi durumunda iki tür uygulama için de bellekte bekleme sürelerini azaltmak ve denetlemek için gereklidir.

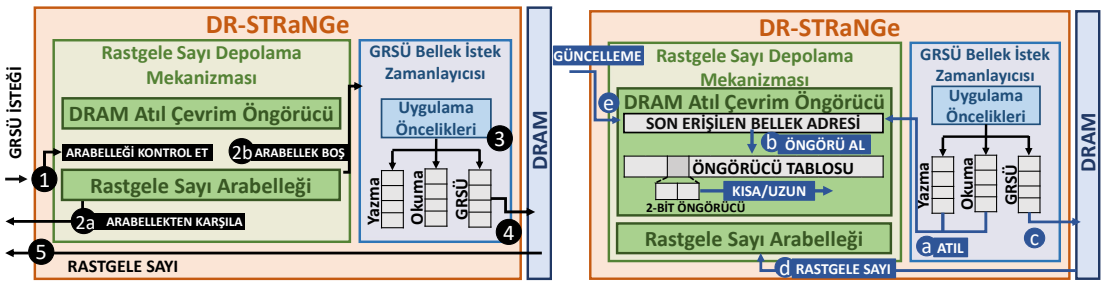
4.3 Uygulama Arayüzü

Sistem rastgele sayı kullanmak isteyen uygulamalara bir arayüz sağlayarak bu uygulamaların DRAM tabanlı GRSÜ ile haberleşmesini sağlamalıdır. Bu arayüz birden çok şekilde gerçeklenebilmektedir. Örneğin, bellek eşlemeli denetim durum yazmaçları [157], sistemde var olan diğer giriş çıkış arayüzleri (x86 IN ve OUT buyrukları gibi) ya da özelleştirilmiş buyruk kümesi mimarisi buyrukları ile bu arayüz sağlanabilir. Uygulama arayüzü, sistem çağrıları ya da durağan ya da devingen uygulama programlama arayüzü (*-ing. application programming interface, API*) fonksiyonları ile uygulamaya açılabilmektedir.

5. DR-STRaNGe

DR-STRaNGe DRAM tabanlı GRSÜler için yüksek performansa ve düşük maliyete sahip uçtan uca bir sistem tasarımıdır. RSÜ çatışmasını düşürerek sistem başarımını artırmakta, GRSÜ ve GRSÜ olmayan uygulamalar arasındaki adaletsizliği düşürerek sistem adilliğini iyileştirmekte ve yüksek GRSÜ gecikmelerini saklayarak GRSÜ uygulamalarının başarımlarını artırmaktadır. DR-STRaNGe 3 temel bileşenden oluşmaktadır. İlk olarak, Rastgele Sayı Depolama Mekanizması düşük RSÜ çatışması ile rastgele sayı üretmek için atıl DRAM çevrimlerini kullanarak yüksek GRSÜ gecikmesini gizlemeyi amaçlamaktadır. Rastgele sayı depolama mekanizması bunu DRAM kanallarında az sayıda rastgele bit üretimi için kullanılabilecek atıl çevrimleri öngörerek ve rastgele sayı üretimi için kullanarak başarmaktadır. Rastgele Sayı Depolama Mekanizması, daha sonra üretilen rastgele sayıları bellek denetleyicisindeki küçük bir arabellekte depolamakta ve gelen rastgele sayı isteklerini bu arabellekten karşılamaktadır. İkinci olarak, GRSÜ Bellek İstek Planlayıcısı GRSÜ isteklerini RSÜ çatışmasını azaltacak ve sistem başarımını artıracak şekilde zamanlayarak bellek isteklerinin yaşadığı bellek kaynaklı bekleme zamanını düşürmeyi amaçlamaktadır. GRSÜ Bellek İstek Planlayıcısı GRSÜ ve GRSÜ olmayan bellek isteklerini ayrı bellek istek kuyruklarında biriktirmekte ve önceliklendirilecek kuyruğu uygulama önceliklerine uygun bir biçimde seçmektedir. Üçüncü olarak, uygulamalar DR-STRaNGe'in uygulama arayüzü aracılığı ile sistemdeki DRAM tabanlı GRSÜ mekanizmasını kullanabilmektedir. DR-STRaNGe tasarım hedeflerini bu üç temel bileşeni birleştirerek sağlamaktadır.

DR-STRaNGe'e genel bakış. Şekil 5.1 DR-STRaNGe'e genel bakışı göstermektedir. Şekil 5.1 (a) GRSÜ istek akışını, Şekil 5.1 (b) ise rastgele sayı depolama akışını açıklamaktadır. Şekil 5.2 bu iki akış için akış şemasını göstermektedir.

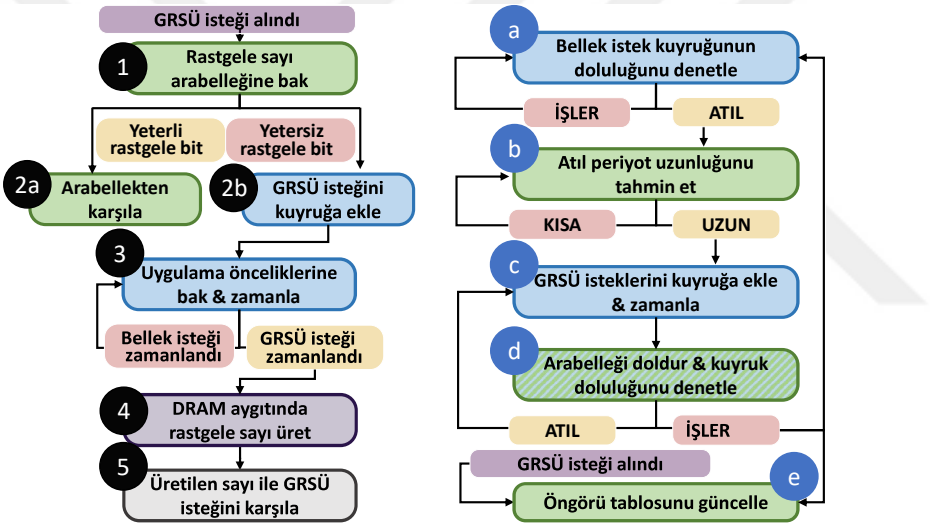


(a) GRSÜ istek akışı (b) Rastgele sayı depolama akışı

Şekil 5.1: DR-STRaNGe genel bakış.

Bellek denetleyicisinin iki çalışma modu vardır: 1) Olağan Çalışma Modu ve 2) GRSÜ Çalışma Modu. *Olağan Çalışma Modu* bellek işlemlerini işlerken *GRSÜ Çalışma Modu* yalnızca GRSÜ isteklerini işlemektedir ve rastgele sayı üretiminden sorumludur. Bellek denetleyici başlangıçta olağan çalışma modundadır.

Bellek denetleyicisi bir GRSÜ isteği aldığı zaman DR-STRaNGe ilk olarak rastgele sayı arabelleğini kontrol eder (❶). Eğer arabellekte hazır bir rastgele sayı varsa DR-STRaNGe GRSÜ isteğini arabellekten düşük gecikme ile karşılar (❷a). Arabellekte hazır bir rastgele sayı yoksa, DR-STRaNGe GRSÜ isteğini kuyruğa ekler (❷b). GRSÜ Bellek İstek Planlayıcısı uygulama önceliklerine bakarak bellek isteklerini sıralar (❸). DR-STRaNGe GRSÜ isteklerini belleğe göndermeden önce ilk olarak GRSÜ isteğinden önce gelen ve daha yüksek önceliğe sahip GRSÜ olmayan uygulamalardan gelen bellek isteklerini belleğe gönderir. Bu istekler bittikten sonra bellek denetleyicisi GRSÜ çalışma moduna geçer ve GRSÜ isteklerini belleğe göndererek DRAM aygıtında rastgele sayı üretir (❹). Yeterli rastgele bit toplandıktan sonra DR-STRaNGe GRSÜ isteğini üretilen rastgele sayı ile karşılar (❺) ve bellek denetleyicisi olağan çalışma moduna geçer.



(a) GRSÜ isteği akış şeması.

(b) Rastgele sayı depolama akış şeması.

Şekil 5.2: DR-STRaNGe akış şeması.

DR-STRaNGe her çevrim başında tüm DRAM kanallarının kullanım oranını bellek istek kuyruklarında bulunan istek sayısına bakarak denetler (ⓐ). Eğer bir kanalın bellek istek kuyrukları boş ise, DR-STRaNGe atıl çevrimlerden oluşan periyodun uzunluğunu tahmin eder (ⓑ). DRAM atıllık öngörücüsü bir DRAM kanalı için bulunulan periyodu rastgele sayı üretimi için yeterli uzunlukta öngörürse, DR-STRaNGe GRSÜ isteklerini istek kuyruğuna ekler ve belleğe gönderir (ⓒ). DR-STRaNGe GRSÜ çalışma moduna geçer ve DRAM kanalındaki tüm grupları kullanarak rastgele sayı üretir (ⓓ). Eğer bir kanal raastgele sayı üretimi sonrasında atıl kalmaya devam ederse, DR-STRaNGe rastgele sayı üreterek arabelleği doldurmaya devam eder. Rastgele sayı üretimi DRAM kanalına ait bellek işlem kuyruğuna yeni bir bellek isteği geldiğinde ya da arabellek tamamen doldurulduğunda durur. Bu durumda, DR-STRaNGe olağan çalışma moduna döner. DRAM kanalına yeni bir istek geldiğinde DRAM atıllık öngörücüsünü günceller (ⓔ).

5.1 Rastgele Sayı Depolama Mekanizması

Rastgele sayı depolama mekanizmasının amacı, rastgele sayıları uygun zamanlarda üretip saklayarak gelen istekleri düşük gecikme ile karşılayabilmektir. Bu sistem başarımını iki şekilde artırmaktadır: (1) GRSÜ uygulamalarının yüksek GRSÜ gecikmesi nedeniyle maruz kaldığı bellekte uzun bekleme sürelerinin kısaltmaktadır ve (2) bellek denetleyicisindeki RSÜ çatışmasını azaltmaktadır. Mekanizmanın ana fikri DRAM kanallarında istek bulunmadığı için kullanılamayan çevrimleri (atıl çevrimleri) rastgele sayı üretimi için kullanmak ve üretilen rastgele sayıları bellek denetleyicisindeki küçük bir arabellekte saklamaktır. Bu sayede rastgele sayı arabelleği boş olmadığı zamanlarda, DR-STRaNGe gelen GRSÜ isteklerini düşük gecikme ile karşılayabilmektedir. Birçok uygulama DRAM bant genişliğinin tamamını kullanmamaktadır [158, 159, 160, 161] ve bu nedenle DR-STRaNGe atıl çevrimleri rastgele sayı üretimi için kullanabilmektedir. Herhangi bir DRAM kanalında art arda iki istek arasında geçirilen atıl sürenin (*atıl periyot*) çevrim cinsinden uzunluğu yürütülen uygulamaların bellek istekleri örüntüleri ile ilişkilidir.

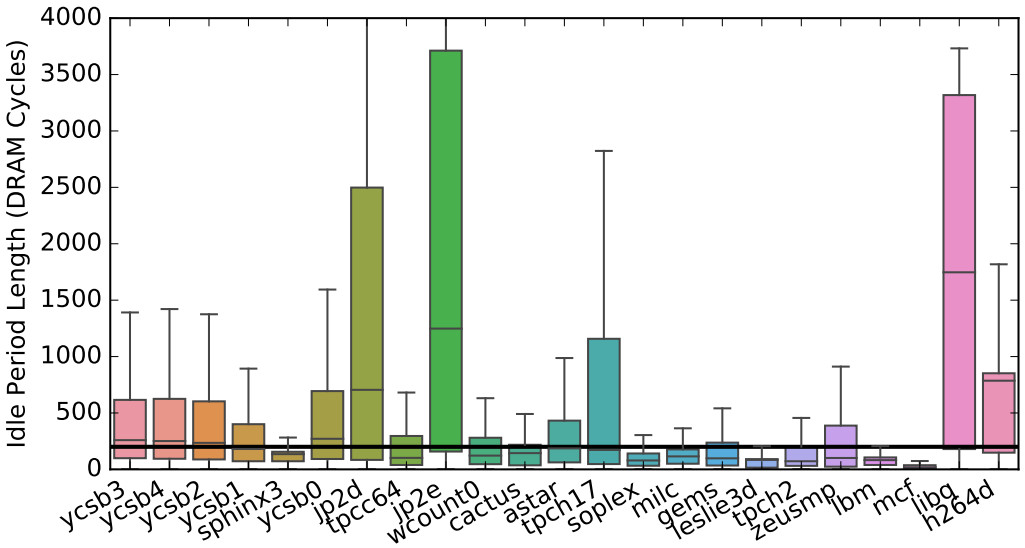
Rastgele sayı üretimine başlamak için uygun zamanı iki ölçüt kullanarak belirlemekteyiz:

1. Düşük DRAM Kullanımı: DRAM kanallarının tamamının kullanılmamasından doğan düşük DRAM kullanım oranı
2. DRAM Atıl Periyot Uzunluğu: Bellek kuyruklarında herhangi bir istek olmamasından kaynaklanan art arda atıl geçen çevrimlerin sayısı

Düşük DRAM Kullanımı. Bir DRAM kanalının düşük kullanım oranına sahip olduğunu belirlemek için bellek istek planlayıcısındaki kuyruklarda bulunan bellek isteklerinin sayısını kullanmaktayız. Bellek istek sayısı tasarım sırasında belirlenmiş bir eşik değeri olan *düşük kullanım eşiği*nden düşük ise DRAM kanalının düşük kullanım oranına sahip olduğuna karar vermekteyiz. Düşük kullanım eşiği, aynı zamanda rastgele sayı üretimi başladığında kuyruklarda bulunabilecek istek sayısını göstermektedir ve bu nedenle RSÜ çatışmasını etkilemektedir. Eşik değeri artırıldığında, rastgele sayı üretme fırsatları artmaktadır. Ancak yüksek eşik değerleri, kuyruklarda daha fazla istek bekletilmesine izin verdiğinden RSÜ çatışmasını artırabilmektedir. Bu nedenle RSÜ çatışması ile GRSÜ fırsatları arasında bir ödünleşme vardır. RSÜ çatışmasının düşük olduğu ve GRSÜ fırsatlarının yüksek olduğu bir düşük kullanım eşik değeri seçilmesi sistem performansı için önemlidir. Düşük kullanım eşik değerini, 32 istek tutan bir kuyruk için deneysel olarak 4 olarak belirlemekteyiz. Bu şekilde, kuyruklarda az sayıda istek bekleterek GRSÜ fırsatlarını artırmakta ve daha fazla rastgele sayı isteğini düşük gecikme ile karşılayabilmekteyiz.

DRAM Atıl Periyot Uzunluğu. Uygulamalar genellikle DRAM kanallarının tamamını kullanmamaktadır ve bu, kanallarda atıl periyotlarla sonuçlanmaktadır. Atıl periyotların uzunluğu uygulamadan uygulamaya bellek kullanım özelliklerine bağlı olarak değişmektedir. Şekil 5.3, orta ya da yüksek oranda bellek isteklerine sahip uygulamalar için DRAM atıl periyot uzunluklarının dağılımını göstermektedir. Kutu grafiğindeki y ekseni gözlemlenen atıl periyotların uzunluklarını DRAM çevrimleri cinsinden göstermektedir. Şekildeki düz yatay çizgi, 64 bitlik rastgele sayı üretmek

için Bölüm 7'de açıklamış olduğumuz deney ortamımızda gerekli olan süreyi, 198 DRAM çevrimi (ortalama 990 *ns*), göstermektedir.



Şekil 5.3: DRAM Atıl Periyot Uzunluklarının Dağılımı.

Her kutu gözlemlenen atıl periyotların çeyrekler açıklığını göstermektedir. Atıl periyotların medyanı ise her bir kutunun orta noktası olarak işaretlenmiştir. Deney sonucunda, birçok uygulama için atıl periyotların büyük bir kısmının 198 çevrim olan 64 bitlik rastgele sayı üretimi süresinden kısa olduğunu gözlemlemekteyiz. Bu nedenle, rastgele sayıların tek seferde üretilmesindense her DRAM grubu için en az birer bit halinde üretilerek saklanmasının daha avantajlı olduğu görülmektedir.

Rastgele sayıları parçalar halinde üretmek atıl periyot uzunluğu sorununu tamamen çözmemektedir çünkü bazı atıl periyotlar her DRAM grubu için bir bit olmak üzere 8 bitlik rastgele sayı üretimi için gerekli olan süreden (40 DRAM çevrimi) daha kısa sürmektedir. Bu atıl periyotlarda rastgele sayı üretmek, bellek isteklerini geciktirerek sistem performansını düşürebilmektedir. Bu nedenle, mekanizmanın kısa atıl periyotlardan kaçınması gereklidir.

DRAM atıl periyot uzunluğunu tahmin etmek için farklı karmaşıklığa sahip üç mekanizma önermekteyiz.

5.1.1 Basit depolama mekanizması

Basit depolama mekanizması her atıl periyodun rastgele sayı üretimi için yeterli uzunlukta olduğunu öngörmektedir. DRAM kanallarındaki her atıl çevrimde rastgele sayı üretimini başlatır ve rastgele sayı arabelleğini doldurur.

Basit depolama mekanizması ile DR-STRaNGe okuma ve yazma kuyruklarında bellek isteği bulunmayan DRAM kanallarını rastgele sayı üretimi için seçer. Bir kanal seçildiği zaman, DR-STRaNGe kanalı GRSÜ çalışma moduna alır ve en az 8 rastgele bit üretir. 8 bit üretildikten sonra kanal hala atıl durumdaysa ve rastgele sayı arabelleği dolu değilse rastgele sayı üretimine devam eder. Ancak, kanala yeni bir bellek isteği

gelmişse ya da rastgele sayı arabelleği tamamen doldurulmuşsa kanal normal çalışma moduna alınır ve bellek isteklerini karşılamaya devam eder. Bellek istekleri, yalnızca kanal normal çalışma modundayken karşılanmaktadır.

5.1.2 DRAM atıllık öngörücüsü

DRAM atıllık öngörücüsünün amacı rastgele sayı üretimi için yeterli uzunlukta olan atıl periyotları tahmin etmektir. Bunun için erişilen son bellek adreslerini kullanan bir mekanizma önermekteyiz. Öngörücümüz her DRAM kanalı için çift doruklu sayaçlardan oluşan bir öngörücü tablosu, son erişilen bellek adresi için bir yazmaç ve atıl periyot uzunluğu için bir sayaç kullanmaktadır. Bir kanal atıl olduğu zaman kanalın öngörücü tablosuna o kanalda erişilen son bellek adresi ile erişilmektedir. Mekanizmamız atıl periyotları ikiye ayırmaktadır: (1) *uzun* (çevrim sayısı $\geq$ *Periyot Eşiği*) ve (2) *kısa* (çevrim sayısı $<$ *Periyot Eşiği*). Öngörücü atıl periyotları, son erişilen bellek adresi ile eriştiği çift doruklu sayaç değeri 2 ya da daha yüksekse uzun, değilse kısa olarak gruplandırmaktadır.

Atıl periyotlar sırasında, DR-STRaNGe atıl periyot uzunluğu sayacını her çevrim artırmaktadır. Bir kanal yeni bir bellek isteği aldığında, DR-STRaNGe o kanala ait öngörücü tablosunu güncellemektedir. İlk olarak, erişilen son bellek adresi ile öngörücü tablosuna erişilir ve çift doruklu sayaç değerine bakılır. İkinci olarak, gözlemlenen atıl periyot uzunluğu periyot eşiğine eşit ya da ondan büyükse çift doruklu sayaç bir artırılır ya da daha küçükse sayaç bir azaltılır. Son olarak, en son erişilen bellek adresi yeni bellek isteğinin eriştiği adres ile güncellenir.

Öngörücünün başarımı, DR-STRaNGe'in performansını iki şekilde etkilemektedir. İlk olarak, eğer öngörücü kısa bir atıl periyodu uzun olarak öngördüğü takdirde RSÜ çatışması artmaktadır. İkinci olarak, eğer öngörücü uzun bir atıl periyodu kısa olarak öngörürse rastgele sayı üretme fırsatını kaçırmaktadır ve bu nedenle bazı rastgele sayı istekleri yüksek gecikme ile karşılanabilmektedir. Öngörücü herhangi bir koşul altında GRSÜ uygulamalarının yürütülememesine sebep olamaz. Çünkü öngörücü tüm atıl periyotları kısa olarak öngörse bile DR-STRaNGe rastgele sayı isteklerini arabellekten değil, istek üzerine rastgele sayı üreterek karşılayabilmektedir.

5.1.3 Pekiştirmeli öğrenme ile DRAM atıllık öngörüsü

DRAM atıl periyot uzunluğu tahminini pekiştirmeli öğrenme (*-ing. reinforcement learning*) problemi olarak tanımlayarak bir pekiştirmeli öğrenme etmeni tasarlamaktayız.

Pekiştirmeli öğrenme, mimari iyileştirme problemlerini çözmek için sıkça kullanılan bir yöntemdir. Örneğin, dallanma öngörüsü [162], bellek istekleri planlayıcıları [163, 164], verileri bellek isteklerinden önce getirme teknikleri [165, 166] ve giriş çıkış iletişimi için dinamik voltaj kontrolü [167] gibi birçok soruna pekiştirmeli öğrenme teknikleri ile çözümler önerilmiştir. Pekiştirmeli öğrenme teknikleri içerisinde Q-öğrenme [168] (*-ing. Q-learning*) basitliği ile tercih edilen bir tekniktir. Q-öğrenme tabanlı bir pekiştirmeli öğrenme etmeni bir durum makinesi aracılığı ile çalışmaktadır.

Bu durum makinesine göre, her durum (s) için her eylem (a) bir değere (Q değeri) sahiptir ve bu değer $Q(s,a)$ ile gösterilir. Bu Q değerleri eylemlerin gelecekte getireceği kümülatif yararı göstermektedir ve Q değerleri hızlı erişim için bir tabloda tutulmaktadır. Herhangi bir s durumunda, algoritma en yüksek $Q(s,a)$ değerine sahip a eylemini gerçekleştirmektedir. İki olası eylem tipi tanımlamaktayız: (1) rastgele sayı üretimini başlatmak ve (2) beklemek. DRAM atıllık problemi için durum son erişilen bellek adresinin en anlamsız 10 biti ve 10 atıl periyot uzunluğunu içeren geçmiş verisinin XOR işleminden geçirilmesi ile oluşturulmaktadır. Geçmiş periyotlardan uzun olarak gruplandırılanlar mantık-1 ile, kısa olarak gruplandırılanlar ise mantık-0 ile gösterilmektedir.

Pekiştirmeli öğrenme etmeni, bir s durumundayken bir a eyleminde bulunduğunda bu eylem ve durum ikilisine ait Q değeri ($Q(s,a)$) ödül belirlendiğinde güncellenmektedir. Ödül (r) atıl periyodun uzunluğuna ve öğrenme etmeninin öngörüsüne bağlı olarak gerçekleştirilen eylemin tipine bağlıdır ve Q değerine eklenmektedir. Etmen doğru öngörüde bulunarak uzun periyotlarda rastgele sayı üretip kısa periyotlarda beklediği durumda, pozitif ödüller kullanılmaktadır. Ancak öngörü yanlış olduğunda ve etmen RSÜ çatışmasını artırdığında ya da bir GRSÜ fırsatını kaçırdığında negatif ödüller uygulanmaktadır. Bir eylemin ödülü atıl periyodun sonunda belirlenmektedir ve atıl periyot sona erdiğinde toplam çevrim sayısına bakarak periyodun uzun ya da kısa olduğu belirlendikten sonra öngörünün doğruluğuna göre Q değeri güncellenmektedir.

Pekiştirmeli öğrenme etmeni, durumu yalnızca bir DRAM kanalı atıl olduğu zaman gözlemleyebilmektedir. Bu nedenle, gelecek durum herhangi bir eylem gerçekleştirilmeden belirlenememektedir. Bu nedenle, güncelleme fonksiyonunda gelecek durumun getireceği potansiyel ödülün etkisi bulunmamaktadır. Son olarak, Q değerlerinin güncellenme fonksiyonu $Q(s,a) = (1-\alpha)Q(s,a) + \alpha * r$ olarak belirlenmiştir ve α öğrenme oranını göstermektedir. Yüksek bir öğrenme oranı değişen bellek erişim örüntülerine daha hızlı adaptasyonu sağlamaktayken öngörüyü gürültüye karşı daha duyarlı bir hale getirebilmektedir. Deneylerimiz sonucunda pekiştirmeli öğrenme etmeninin en yüksek performansa 0.05 öğrenme oranı ile eriştiğini gözlemlemekteyiz.

5.2 GRSÜ Bellek İstek Planlayıcısı

DRAM atıl periyotlarının dağılımı sebebi ile DR-STRaNGe her zaman rastgele sayı isteklerini arabellekten karşılayamayabilmektedir. Bellek istek planlaması, rastgele sayı istekleri geldiğinde rastgele sayı arabelleği boşsa ve kuyruklarda bekleyen bellek istekleri varsa oldukça önemli hale gelmektedir. GRSÜ bellek istek planlayıcısının amacı GRSÜ isteklerini, bellek isteklerini uzun süreler bekletmeden karşılayabilmek ve sistem adilliğini artırmaktır.

GRSÜ istekleri, aynı istek kuyrukları hem GRSÜ hem de bellek istekleri için kullanıldığında bellek isteklerinin kuyruklara eklenmesini engelleyebilmekte ve bellek denetleyicisinin GRSÜ ve normal çalışma modu arasında sık sık çalışma modunu değiştirmesine sebep olabilmektedir. DR-STRaNGe, GRSÜ istekleri için ek bir kuyruk (GRSÜ kuyruğu) kullanarak kuyruk alanı için çekişmeyi ortadan kaldırmaktadır.

5.2.1 Uygulama önceliği ile istek planlama

İşletim sistemi donanım kaynaklarını uygulama önceliklerini göz önünde bulundurarak paylaştırmaktadır. Uygulamaların öncelikleri farklı seviyeler olarak tanımlanabilmektedir ve bu öncelik seviyeleri bellek isteklerini planlamak için kullanılabilir. GRSÜ Bellek İstek Planlayıcısı, uygulama öncelik seviyelerini kullanarak farklı istek kuyruklarını önceliklendirmekte ve farklı tip bellek isteklerini öncelikli olarak gerçekleştirmektedir. Ancak, uygulama öncelik seviyeleri tek başına uygulama seviyesinde adil istek planlama için yeterli değildir. Bunun sebebi, rastgele sayı isteklerinde bulunan uygulamaların hem bellek istek kuyruklarını hem de rastgele sayı istek kuyruğunu aynı anda kullanabilmesidir. Bu durum haksız bir önceliklendirmeye sebep olmaktadır. Bu nedenle, DR-STRaNGe aynı anda hem rastgele sayı hem de bellek istek kuyruklarını kullanan uygulamaları tespit etmelidir. Bu uygulamaları, GRSÜ uygulamalarını, tespit etmek için DR-STRaNGe bir uygulama ilk defa rastgele sayı istediğinde bu uygulamayı işaretlemektedir. GRSÜ ve GRSÜ olmayan uygulamalar belirlendikten sonra DR-STRaNGe uygulama öncelik seviyelerini kullanarak kuyrukları önceliklendirmektedir. Bu durumda, düşük önceliğe sahip olan kuyruk uzun bellek bekleme sürelerine maruz kalabilmektedir. GRSÜ Bellek İstek Planlayıcısı uygulamaların isteklerinin her zaman karşılanması için aşağıda açıkladığımız üzere çeşitli kurallar kullanmaktadır.

GRSÜ Öncelikli. Bir GRSÜ uygulaması diğerlerinden daha yüksek önceliğe sahipse, DR-STRaNGe ilk olarak GRSÜ uygulamasının GRSÜ isteklerini planlamaktadır. Bu sayede (1) rastgele sayı üretimi sebebiyle yaşanan yavaşlamalar önlenmekte ve (2) RSÜ çatışması azaltılmaktadır. Bellek istek planlayıcısı, GRSÜ ve bellek istek kuyrukları dolu iken ve bir GRSÜ isteğine sahip GRSÜ uygulaması diğer tüm GRSÜ olmayan uygulamalardan daha yüksek önceliğe sahipse GRSÜ kuyruğunu seçer ve GRSÜ kuyruğunda istek kalmayana kadar oradan istekleri planlamaya devam eder. Planlayıcı istekler bittiğinde, bellek istek kuyruklarındaki istekleri karşılamaya devam eder. Ancak, gerekli GRSÜ hızı çok yüksekse, planlayıcı sık sık GRSÜ kuyruğunu önceliklendirecektir.

Bellek İstekleri Öncelikli. Bellek istek planlayıcısı GRSÜ olmayan bellek isteğine sahip bir uygulama diğer tüm uygulamalardan daha yüksek önceliğe sahipse, bellek istek kuyruklarını GRSÜ kuyruğuna kıyasla daha öncelikli seçmekte ve bellek isteklerinin kuyrukta bekleme sürelerini en aza indirmeyi amaçlamaktadır. Planlayıcı yalnızca bellek istek kuyruğundaki en eski istek bir GRSÜ uygulamasına aitse ve GRSÜ kuyruğundaki isteklerden daha eskiyse GRSÜ kuyruğundan istekleri seçmektedir. Bu durumda, planlayıcı GRSÜ kuyruğundaki istekler tamamlanana kadar bu kuyruktaki istekleri karşılamaktadır. GRSÜ istekleri bittiğinde, bellek istek kuyruklarına tekrar öncelik vermektedir.

Eşit Öncelik Seviyeleri Durumunda. Eğer iki GRSÜ uygulaması aynı öncelik seviyesine sahipse ve GRSÜ isteklerinde bulunuyorlarsa, planlayıcı ilk olarak daha eski olan isteği karşılamaktadır. Eğer bu iki GRSÜ uygulaması bellek isteklerinde bulunuyorsa planlayıcı temel planlayıcı algoritmasını takip ederek istekleri planlamaktadır. Benzer olarak, GRSÜ olmayan ve aynı öncelik seviyesine sahip uygulamaların istekleri temel planlayıcının kurallarına göre sıralanmaktadır. Ancak eğer bir GRSÜ uygulaması ile GRSÜ olmayan bir uygulama aynı öncelik

seviyesine sahipse, eğer varsa GRSÜ istekleri önceliklendirilmektedir. Bu sayede RSÜ çatışması azaltılmaktadır. Bölüm 8.5'te gösterdiğimiz üzere, DR-STRaNGe bu planlama algoritması ile herhangi bir şekilde GRSÜ olmayan uygulamaların performansını ya da genel sistem adilliğini düşürmemektedir.

Uç Durumlar. Açıkladığımız GRSÜ bellek istek planlayıcısı kurallarına rağmen bazı uygulamaların isteklerinin uzun süre karşılanmadığı uç senaryolar gerçekleşebilmektedir. Bir GRSÜ uygulaması yüksek hızda rastgele sayı isteğinde bulunarak GRSÜ kuyruğunu sık sık doldurabilmektedir. Bu durumda bu uygulama önceliklendirildiği takdirde GRSÜ olmayan uygulamalar uzun süre bellek kuyruklarında bekleyebilmektedir. Benzer olarak yüksek bellek istek yoğunluğuna sahip GRSÜ olmayan uygulamalar bellek istek kuyruklarını doldurabilmektedir. Bu durumda GRSÜ uygulamalarına ait istekler uzun süre bellekte bekletilmektedir. Bu durumlarda, GRSÜ bellek istek planlayıcısı kuralları yeterli gelmeyebilir. Herhangi bir uygulamanın performansının bu nedenle çok fazla düşmesini engellemek için aşağıda açıklamakta olduğumuz mekanizmayı önermekteyiz.

DR-STRaNGe ilk olarak bellek istek kuyruklarının hangisinin daha az önceliklendirildiğini gözlemlemekte ve düşük önceliğe sahip kuyruğun çok sayıda çevrim boyunca duraklatılmasını engellemektedir. GRSÜ bellek istek planlayıcı daha yüksek öncelikli kuyruktan isteklerin karşılandığı her çevrim bir sayaç değerini artırmaktadır. Daha sonra bu sayaç değerini bir eşik değeri ile karşılaştırmaktadır. Bu eşik değerine *bekleme eşik değeri* denmektedir. Eğer duraklatılma sayacı, bekleme eşik değerine eşit ya da ondan yüksek ise daha az öncelikli olan kuyruktan istekler karşılanmaktadır. Bu şekilde daha düşük önceliğe sahip kuyruktaki isteklerin devamı ve düşük önceliğe sahip uygulamaların da devamlılığı sağlanmaktadır. Daha düşük önceliğe sahip kuyruklardan karşılanan her istek ile bekleme sayacı sıfıra eşitlenmektedir.

5.3 Uygulama Arayüzü

Sistem tasarımının tamamlanması için yazılım ve DRAM tabanlı GRSÜ mekanizması arasında bir arayüz bulunması gerekmektedir. Linux tabanlı sistemlerde bu arayüz, çekirdek rastgele sayı üretecinin var olan arayüzü [5] değiştirilerek oluşturulabilmektedir. Rastgele sayı üreteci, aygıt denetleyicilerinden kaynaklanan çevresel gürültüleri bir entropi havuzunda toplamaktadır. getrandom() sistem çağrısı, bir uygulama rastgele sayı istediğinde kullanılır. Sistem çağrısı aracılığı ile sistem, bu entropi havuzundaki rastgele bitleri kullanarak bir arabellek doldurur ve bu arabelleği bir işaretleyici ile çağrıyı yapan uygulamaya iletir.

getrandom() sistem çağrısını değiştiren ve sistem rastgele sayı üreteci yerine DR-STRaNGe'i kullanmayı sağlayan bir arayüz önermekteyiz. Sistem çağrısının DR-STRaNGe ile iletişime geçmesi bellek eşlemeli konfigürasyon durum yazmaçları ya da hedef sistemde var olan diğer giriş çıkış veri yolları kullanılarak sağlanabilmektedir. Bir rastgele sayı isteği yapıldığında DR-STRaNGe bu isteği eğer rastgele sayı arabelleğinde rastgele sayı varsa arabellekten, değilse de rastgele sayı üreterek karşılamaktadır.

Rastgele sayı arabelleğinin boş olmadığı durumlarda, DR-STRaNGe'in sistem çağrısı temel sistem çağrısına kıyasla fazladan herhangi bir gecikme yaratmamaktadır. Ancak, rastgele sayı arabelleği boş olduğu takdirde, oluşacak gecikme DRAM tabanlı GRSÜ mekanizması ile sistemde bulunan temel rastgele sayı üretecinin hızları farkı ile ilişkili olacaktır. Sistemde bulunan rastgele sayı üreteci (örneğin, Linux tabanlı sistemler için çekirdeğin rastgele sayı üreteci [5]), sistemdeki aygıtlara ve bu aygıtlardan örneklenebilecek rastgeleliğe bağlıdır, bu nedenle gecikme hedef sisteme göre değişiklik göstermektedir.

6. GÜVENLİK ANALİZİ

6.1 Güvenli Rastgele Sayılar

Güvenlik-kritik uygulamalarda rastgele sayıların kullanılabilmesi için GRSÜlerin iki ana özelliğe sahip olması gerekmektedir. İlk olarak, GRSÜ mekanizması rastgele sayıları isteği yapan uygulama dışında herhangi bir uygulamaya sızdırmamalıdır. DR-STRaNGe bu özelliği rastgele sayı arabelleğini yalnızca sistem çağrısı aracılığı ile erişilebilir şekilde tasarlayarak gerçekleştirmektedir. Sistem çağrısı rastgele bitleri yalnızda çağrının sahibi olan uygulama ile paylaşmaktadır ve rastgele sayı arabelleği istek karşılandıktan sonra bu bitleri arabellekten çıkarmaktadır. İkinci olarak GRSÜ mekanizması her rastgele sayı isteğine farklı bir rastgele sayı vermelidir. DR-STRaNGe bu özelliği, rastgele sayıları kullanıldıktan sonra arabellekten çıkararak sağlamaktadır. Sonuç olarak, basit ancak kısıtlayıcı uygulama arayüzü ile DR-STRaNGe bu iki özelliği de sağlayarak güvenliği önemli olan uygulamalara güvenli rastgele sayılar sağlamaktadır.

6.2 Rastgele Sayı Arabelleği ile Yan Kanal Saldırıları

Yan kanal saldırıları (*-ing. side-channel attacks*) [169] yan kanal bilgilerini gözlemleyerek ve kullanarak uygulama davranışı ile ilgili bilgi edinmeye ya da gizli bilgileri sızdırmaya [169] yarayan bir saldırı sınıfıdır. Uygulamalar arasında paylaşılan diğer donanımsal kaynaklar (örneğin, önbellekler [86, 170, 171]) herhangi bir veri hakkında zamanlama bilgisi verebildiği halde, DR-STRaNGe bir saldırgan tarafından hedef olarak alındığında yalnızca bir rastgele sayının üretiminin aldığı süre bilgsini vermektedir. Bu zamanlama bilgisi rastgele sayı arabelleğinin dolu ya da boş olduğunu ve bir başka uygulamanın rastgele sayı kullanıp kullanmadığı bilgisini ortaya çıkarabilmektedir. Bu yan kanal, iki sebepten saldırganlar tarafından kullanılması zor bir kanaldır. İlk olarak, DR-STRaNGe sürekli olarak rastgele sayı arabelleğini rastgele sayılarla doldurmaktadır ve bu mekanizma uygulamalarla asenkron çalışmaktadır. Bu nedenle rastgele sayı arabelleği nadir olarak boş kalmaktadır. İkinci olarak, eğer iki ve daha fazla uygulama rastgele sayı kullanıyorsa, arabelleği hangi uygulamanın boşalttığını tespit etmek oldukça zordur. Analizimiz sonucunda, DR-STRaNGe'in zamanlama yan kanal saldırıları için halihazırda diğer paylaşımlı kullanılan donanımsal kaynaklardan yararlanan yan kanallara göre daha zor kullanılacağı sonucuna ulaşmaktayız.

6.3 Rastgele Sayı Arabelleği ile Gizli Kanal Saldırıları

Saldırganlar iletişim için tasarlanmayan kanalları bir iş parçacığından diğerine veri aktarmak için kullanabilmektedir. Rastgele sayı arabelleği, belirli şartlar altında

iletişim için gizli bir kanal [172] olarak kullanılabilir. Bunun için saldırgan haricinde uygulamaların rastgele sayı kullanmaması gerekmektedir. Rastgele sayı arabelleği temel olarak önbellek tabanlı gizli kanallar gibi kullanılabilmektedir. Bu alanda yapılan çalışmalar önbelleklerin gizli iletişim kanalları olarak kullanılması için birçok teknik [173, 174, 175, 176, 177, 178, 179] gösterdiği gibi, bu saldırılara karşı savunma teknikleri [170, 171, 180, 181, 182, 183] de önermiştir. Bu savunma teknikleri rastgele sayı arabelleğine uygulanabilmektedir. İlk olarak, rastgele sayı arabelleği uygulamalar için farklı bölümlere ayrılabilir. Bu durumda, uygulamalar birbrilerinin rastgele sayı arabelleği bölümlerinin dolu ya da boş olma durumunu gözlemleyememektedir. Bu yöntem performansı düşük seviyede etkiler, çünkü deneysel sonuçlarımızda gösterdiğimiz üzere küçük arabellek boyutları birçok uygulama için yeterlidir. İkinci olarak, sistem arabelleğe erişimi aynı anda tek bir uygulamaya verebilir. Bu durumda, arabelleğin tamamı bir uygulama tarafından kullanılabilmektedir. Bu çözüm, arabelleğe erişme iznine sahip olmayan GRSÜ uygulamalarının performansını düşürürken bu uygulamalar yine de DR-STRaNGe'in GRSÜ bellek istek planlayıcısından yararlanarak düşük RSÜ çatışması ile rastgele sayı üretebilmektedir.

6.4 Hizmet Reddi Saldırıları

Bir saldırgan DRAM bant genişliğini rastgele sayı istekleri ile doldurmaya ve diğer uygulamarın isteklerinin karşılanmasını engellemeye çalışabilir. Bu saldırılar, GRSÜ bellek istek planlayıcısı ile önlenmektedir. Uygulamalaın isteklerinin karşılandığından emin olan bir dizi kural kullanan istek planlayıcısı bu tarz saldırılar halinde bile sistem adilliğini korumayı hedeflemektedir. Bu kuralların yanında, hizmet reddi saldırıları işletim sistemi seviyesinde alınabilecek kararlar aracılığı ile sistem adilliği odaklı çözüm metotları [184] kullanılarak engellenebilmektedir.

7. METODOLOJİ

DR-STRaNGe'in performans ve adilliğini Ramulator [82, 185] sistem benzetim aracını genişleterek değerlendirmekteyiz. Ramulator, çevrim bazında çalışan bir DRAM benzetim aracıdır. Bu benzetim aracını günümüzde en yüksek hıza ve en düşük gecikmeye sahip olan iki DRAM tabanlı GRSÜ (D-RaNGe [12] modeli ve QUAC-TRNG [13]) ve rastgele sayı istekleri desteği ile genişletmekteyiz. Deneylerimizde DR-STRaNGe ve sistem için Çizelge 7.1'de verilen konfigürasyonları kullanmaktayız. Rastgele sayı üretimini, aksi belirtilmediği takdirde D-RaNGe [12] mekanizması ile önerilen şekilde simüle etmekteyiz.

İş Yükleri. Simülasyon düzeneğimizde yürütmek üzere 4 farklı uygulama paketinden 43 tek çekirdekli uygulama kullanmaktayız. Bu uygulamalar SPEC CPU2006 [152], TPC [153], MediaBench [154], ve YCSB [155] uygulama paketlerinden seçilmektedir. Son seviye önbellekte bulamama oranlarına (*-ing misses-per-kilo-instruction (MPKI)*) göre bu uygulamaları üç kategoriye ayırmaktayız: (1) L (düşük bellek istek yoğunluğu, MPKI $<$ 1), (2) M (orta bellek istek yoğunluğu, 1 $\leq$ MPKI $<$ 10), ve (3) H (yüksek bellek istek yoğunluğu, MPKI $\geq$ 10). Uygulamaların MPKI değerleri için, her uygulamanın SimPoint [186] aracılığı ile oluşturduğumuz 200 milyon buyruktan oluşan bellek istek izlerini analiz etmekteyiz.

GRSÜ uygulamaları için sentetik bir dizi farklı rastgele sayı isteği sıklığına sahip GRSÜ uygulaması oluşturmaktayız. Rastgele sayı istek yoğunluğunu iki istek arasında yer alan buyruk sayısı ile kontrol etmekteyiz. Sentetik uygulamalar DRAM kanallarının hepsine erişim yapmakla beraber yüksek bellek istek yoğunluğuna sahip değildir. En düşük GRSÜ istek yoğunluğuna sahip sentetik uygulama 640 Mb/s hızla rastgele sayı isteği yapmaktadır ve bu hız günümüz DRAM tabanlı GRSÜ mekanizmalarının maksimum hızından daha düşüktür. En yüksek GRSÜ istek yoğunluğuna sahip sentetik uygulama ise 5 Gb/s hızla rastgele sayı

Çizelge 7.1: Sistem Simülasyon Konfigürasyonları

İşlemci	1-2-4-8-16 çekirdek, 4GHz saat sıklığı, 128 buyruk genişliğinde buyruk penceresi
DRAM	DDR3-1600 [111], 800MHz bus sıklığı, 4 kanal, 1 rank/kanal, 8 grup/rank, 64 bin satır/grup
Bellek Denetleyicisi	32 satırlı okuma/yazma kuyruğu, FR-FCFS [94, 95] column cap of 16 [97]
DR-STRANGE	32 satırlı GRSÜ kuyruğu, GRSÜ bellek istek planlayıcısı, 256 satırlı öngörücü tablosu/kanal, 16 satırlı rastgele sayı arabelleği

isteğinde bulunmaktadır. Aksi belirtilmediği takdirde deneylerimizde yüksek GRSÜ yoğunluğuna sahip sentetik GRSÜ uygulaması kullanılmaktadır.

Deneylerimiz için oluşturduğumuz iş yükleri Çizelge 7.2'de verilmektedir. Toplamda 43 tane iş yükünü iki çekirdek üzerinde yürütülmek üzere oluşturmaktayız. Bu iş yükleri bir adet GRSÜ ve bir adet GRSÜ olmayan uygulamadan oluşmaktadır. Buna ek olarak, dört çekirdek üzerinde yürütülmek üzere her biri 10 iş yükünden oluşan 4 farklı kategoride iş yükleri oluşturmaktayız. Her grup farklı bellek istek yoğunluklarına sahip 3 farklı uygulama ile 1 adet GRSÜ uygulaması içermektedir. Örneğin, LLHS grubu rastgele seçilmiş 2 adet L (düşük bellek istek seviyesine sahip) uygulama, 1 adet H (yüksek bellek istek yoğunluğuna sahip) uygulama ve 1 adet GRSÜ uygulaması içermektedir. Sekiz ve on altı çekirdekli iş parçacıkları için ise her biri onar iş yükünden oluşan düşük, orta ve yüksek bellek isteği yoğunluğunda uygulamalardan oluşan iş yükü grupları oluşturulmaktadır.

Çizelge 7.2: Çok Çekirdekli İş Yükleri

Çekirdek Sayısı	**GRSÜ olmayan uygulamalar/iş yükleri**	**GRSÜ uygulamaları**
2 Çekirdek	43 GRSÜ olmayan uygulama	x1 GRSÜ uygulaması (5120 Mb/s GRSÜ hızı) x1 GRSÜ uygulaması (640 Mb/s GRSÜ hızı)
4 Çekirdek	40 GRSÜ olmayan uygulamadan oluşan iş yükü (4 bellek isteği yoğunluğu grubu, her biri 10 iş yükünden oluşmaktadır.)	x1 GRSÜ uygulaması (5120 Mb/s GRSÜ hızı)
8 Çekirdek	30 GRSÜ olmayan uygulamadan oluşan iş yükü (3 bellek isteği yoğunluğu grubu, her biri 10 iş yükünden oluşmaktadır.)	x1 GRSÜ uygulaması (5120 Mb/s GRSÜ hızı)
16 Çekirdek	30 GRSÜ olmayan uygulamadan oluşan iş yükü (3 bellek isteği yoğunluğu grubu, her biri 10 iş yükünden oluşmaktadır.)	x1 GRSÜ uygulaması (5120 Mb/s GRSÜ hızı)

Karşılaştırma Noktaları. DR-STRaNGe'i iki tasarım ile karşılaştırmaktayız: (1) DRAM aygıtlarında rastgele sayı üretimini yok sayan bir temel sistem tasarımı olan *RNG-oblivious* sistemi, (2) açgözlü atıllık tespiti algoritmasına sahip olan *Greedy Idle* sistem tasarımı. Önerdiğimiz GRSÜ bellek istek planlayıcısını ise (1)FR-FCFS [94, 95] (sütun kapasitesi 16) [97] ve (2) BLISS [108, 103] bellek istek planlayıcıları ile karşılaştırmaktayız. Önerdiğimiz DRAM atıllık öngörücüsünü, pekiştirmeli öğrenme ile atıllık öngörüsü yapan Q-öğrenme etmeni ile karşılaştırmaktayız.

Greedy Idle sistem tasarımı rastgele sayı arabelleğini hiçbir gecikme ya da performans kaybı olmadan dolduran bir mekanizma ile çalışmaktadır. [2] Bir atıl periyodun uzunluğu *periyot eşiği*ne ulaştığı zaman, rastgele sayı arabelleğinin 8 bit ile doldurulduğunu varsaymaktayız. Bu sayede arabellek için rastgele sayı üretimi herhangi bir gecikmeye sebep olmamaktadır. Adil bir karşılaştırma noktası olması için, *Greedy Idle* sistem tasarımı, DR-STRaNGe gibi GRSÜ ve bellek istekleri için farklı kuyruklara sahiptir. Aynı zamanda GRSÜ isteklerini göz önünde bulundurarak uygulama seviyesinde öncelik tabanlı bellek istek planlaması yapabilmektedir.

Ölçütler. GRSÜ ve GRSÜ olmayan uygulamaların performanslarını aynı anda çift çekirdekli bir sistemde aynı sayıda buyruk süresince yürüterek ölçmekteyiz. Çok çekirdekli iş yükleri için ise, GRSÜ olmayan uygulamalar için geçmiş çalışmalarda çok çekirdekli iş yüklerinin performansı için iyi bir ölçüt olduğu gösterilmiş [187] olan ağırlıklı hızlanma ölçütünü [188] kullanmaktayız.

[2] *Greedy Idle* sistem tasarımı performans ve sistem adilliği için bir üst sınır oluşturmaktadır. Açgözlü (*-ing. greedy*) algoritmayla GRSÜ için yeterli atıl periyotları seçerek bu periyotlarda rastgele sayı üretmektedir. Ancak, bu sistemin performans ve sistem adilliği iyileştirmeleri kısıtlıdır çünkü en iyi dinamil bellek istek sıralaması açgözlü algoritma yaklaşımı ile polinom zamanda bulunamamaktadır.

Sistem adilliği için, adaletsizlik endeksini [97, 156, 96] kullanmaktayız. Adaletsizlik endeksi, yaşanan en yüksek bellek kaynaklı gecikmenin en düşük bellek kaynaklı gecikmeye oranı ile hesaplanmaktadır. Bir uygulamanın bellek kaynaklı gecikmesini hesaplamak için, bellek paylaşıldığı zaman uygulama isteklerinin bellek kuyruklarında bekleme süresini ölçmekte ve tek başına yürütülüyorken yaşadığı gecikmeye göre normalize etmekteyiz:

$$MemSlowdown_i = \frac{MCPI_i^{shared}}{MCPI_i^{alone}}, Unfairness = \frac{max_i MemSlowdown_i}{min_i MemSlowdown_i} \tag{7.1}$$

Adaletsizlik endeksi bire eşitse bu, tüm uygulamalar benzer oranda performans kaybı yaşıyor anlamına gelmektedir. Daha yüksek bir adaletsizlik endeksi ise, bir uygulamanın diğerlerine kıyasla daha fazla önceliklendirildiğini ve uygulamalar arasında büyük performans kaybı farklılıkları olduğunu göstermektedir.

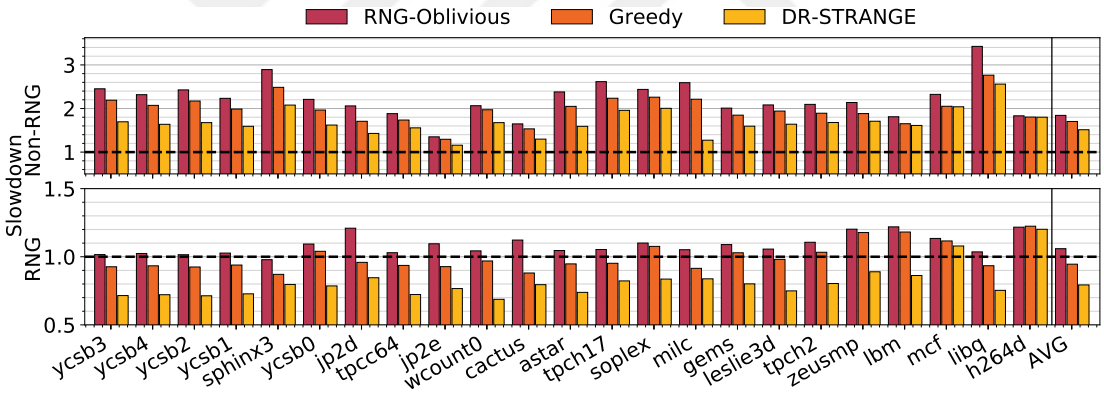
8. DR-STRaNGe'İN DEĞERLENDİRİLMESİ

DR-STRaNGe'in performans, sistem adilliği, enerji verimliliği ve alan masrafını değerlendirmekteyiz.

8.1 DR-STRaNGe'in Performans Üzerine Etkisi

8.1.1 Çift çekirdekli sistem performansı

Şekil 8.1 üç farklı tasarımın performans sonuçlarını karşılaştırmaktadır: (1) DRAM aygıtlarında rastgele sayı üretimini yok sayan bir temel sistem tasarımı olan *RNG-oblivious* sistemi, (2) açgözlü atıllık tespiti algoritmasına sahip olan *Greedy Idle* sistem tasarımı ve (3) DR-STRaNGe. Şekil, GRSÜ olmayan (üst) ve GRSÜ (alt) uygulamalarının çift çekirdekli sistemde çalıştırıldıklarında gözlemlenen yavaşlamalarını iş yüklerindeki uygulamaların tek çekirdekli sistemde tek başlarına çalışmalarına kıyasla göstermektedir. İki gözlemde bulunmaktayız.



Şekil 8.1: GRSÜ olmayan (üst) ve GRSÜ uygulamalarının (alt) çift çekirdekli sistemde yürütüldüklerinde tek çekirdekte yürütülmelerine kıyasla gözlemlenen yavaşlamalar.

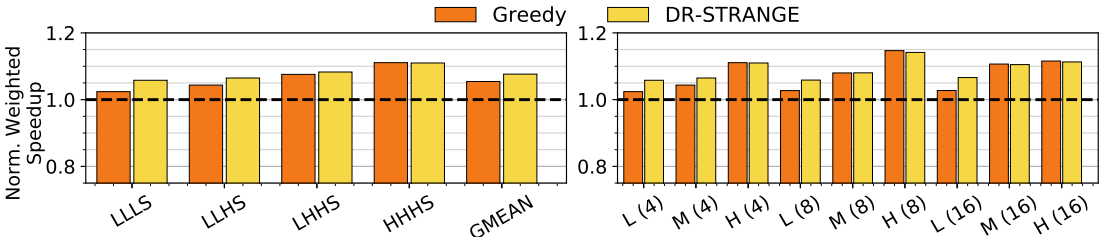
İlk olarak, DR-STRaNGe hem GRSÜ hem de GRSÜ olmayan uygulamaların performanslarını artırmaktadır. DR-STRaNGe, *RNG-oblivious* sisteme kıyasla, GRSÜ olmayan uygulmaların yürütme zamanınını ortalamada %17.9, GRSÜ uygulamalarının yürütme zamanını ise ortalamada %25.1 oranda azaltmaktadır. DR-STRaNGe aynı zamanda GRSÜ uygulamalarının ortalama yürütme zamanını uygulamaların tek çekirdekte ve tek başlarına yürütülmesine kıyasla %20.6 oranda azaltmaktadır. Bunun sebebi GRSÜ gecikmesinin düşürülmesidir.

İkinci olarak, *Greedy Idle* sistem, GRSÜ ve GRSÜ olmayan uygulamaların ortalama performansını %10.7 ve %7.6 artırmaktadır. DR-STRaNGe, *Greedy Idle* sistem tasarımına kıyasla çoğu uygulama için daha iyi performansa sahiptir. Bunun sebebi

ise, *Greedy Idle* sistem tasarımı rastgele sayı arabelleğini yalnızca yeterince uzun atıl periyotlarda doldurmakta ve bu nedenle arabelleği DR-STRaNGe'e kıyasla daha az dolu tutmaktadır. DR-STRaNGe ise arabelleği düşük DRAM kullanımı ölçütünü kullanarak daha sık doldurmaktadır, bu sayede daha fazla rastgele sayı isteğini arabellekten karşılayabilmektedir.

8.1.2 Çok çekirdekli sistem performansı

GRSÜ olmayan uygulamalar. Şekil 8.2 GRSÜ olmayan 4 çekirdekli iş yükleri (sol) ve 4, 8, 16 çekirdekli iş yükleri için (sağ) *RNG-oblivious* tasarıma göre normalize edilmiş ağırlıklı hızlanma oranlarını göstermektedir. Bu şekli temel alarak iki gözlemde bulunmaktayız. İlk olarak, 4 çekirdekli iş yükleri için, DR-STRaNGe ortalama performansı %7.6 artırmaktadır. İş yüklerindeki bellek işlemlerinin ağırlıklı olduğu uygulamalar arttıkça DR-STRaNGe'in sağladığı performans artışı yükselmektedir. *Greedy Idle* sistem *RNG-oblivious* sisteme göre %5.4 daha yüksek performans göstermektedir. DR-STRaNGe, daha fazla rastgele sayı isteğini rastgele sayı arabelleğinden karşılayarak *Greedy Idle* sisteme kıyasla daha fazla performans artışı sağlamaktadır. İkinci olarak, DR-STRaNGe yüksek, orta ve düşük bellek isteği ağırlığına sahip olan uygulamalar için ortalama performansı sırasıyla %12.1, %8.2 ve %6.1 artırmaktadır. DR-STRaNGe, *Greedy Idle* sistem tasarımını performans artışı açısından iki iş yükü grubu haricinde tüm uygulamalarda daha iyi sonuç vermektedir. Bellek işlemlerinin ağırlıklı olduğu 8 ve 16 çekirdekli sistemlerde *Greedy Idle* sistem tasarımı yanlış atıl periyot öngörüsünden kaynaklanan gecikmelere ve RSÜ çatışmasına sahip olmadığı için daha yüksek performansa sahiptir.



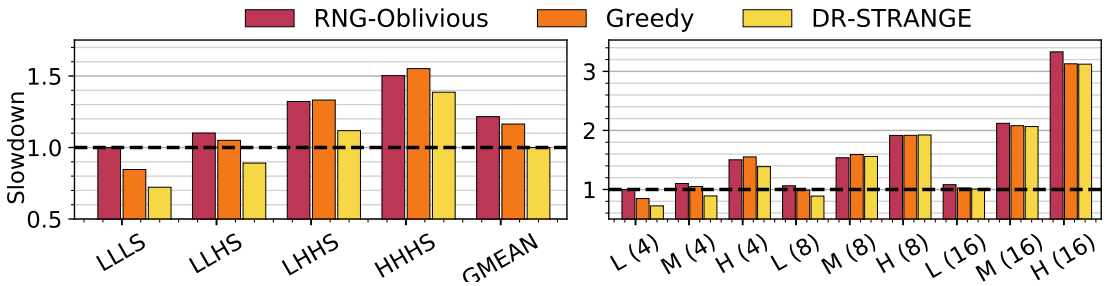
(a) 4 çekirdekli

(b) 4, 8, 16-çekirdekli

Şekil 8.2: GRSÜ olmayan uygulamaların (a) 4 çekirdekli ve (b) 4, 8, 16 çekirdekli bellek istek yoğunluğuna göre gruplandırılmış iş yükünde gözlemlenen normalize edilmiş ağırlıklı hızlanmaları.

GRSÜ uygulamaları. Şekil 8.3 GRSÜ uygulamalarının performansını (1) *RNG-oblivious*, (2) *Greedy Idle* sistem tasarımları ve (3) DR-STRaNGe için GRSÜ uygulamalarının tek başlarına yürütüldükleri sistem performansına kıyasla göstermektedir. Şekil, GRSÜ uygulamalarının farklı iş yükleri ve sistemler ile yavaşlamalarını göstermektedir. Sol taraftaki şekilde 4 çekirdekli sistemde yavaşlamalar gösterilirken sağ taraftaki şekilde 4, 8, 16 çekirdekli sistemlerdeki yavaşlamalar gösterilmektedir. DR-STRaNGe 4 çekirdekli sistemde GRSÜ uygulamalarının performansını %17.8 artırmaktadır. 4, 8 ve 16 çekirdekli sistemlerde ise DR-STRaNGe ortalama performansı %4.5, %6.7 ve %16.9 oranında artırmaktadır.

Tüm iş yükleri için DR-STRaNGe ortalama performansı en az *Greedy Idle* sistem kadar artırmaktadır.

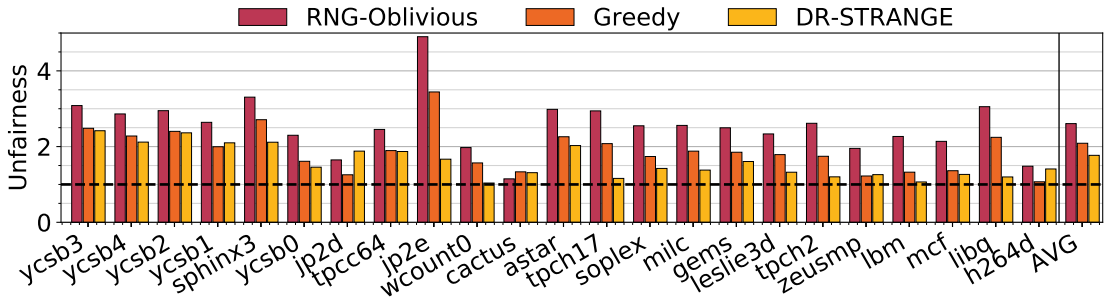


(a) 4 çekirdekli (b) 4, 8, 16 çekirdekli

Şekil 8.3: GRSÜ uygulamalarının (a) 4 çekirdekli ve (b) 4, 8, 16 çekirdekli bellek istek yoğunluğuna göre kıyaslanmış iş yükünde gözlemlenen yavaşlamaları.

8.2 DR-STRaNGe'in Sistem Adilliği Üzerine Etkisi

Şekil 8.4 çift çekirdekli sistemde, sistem adilliğini adaletsizlik metriğini [97, 156, 96] kullanarak göstermektedir. Üç gözlemde bulunmaktayız. İlk olarak, DR-STRaNGe ortalama sistem adilliğini, *RNG-oblivious* sisteme kıyasla %32.1 artırmaktadır. İkinci olarak, DR-STRaNGe'in sistem adilliği *Greedy Idle* sisteme kıyasla %15.2 daha yüksektir. Üçüncü olarak, *jp2d* ve *cactusADM* gibi GRSÜ olmayan uygulamalara sahip bazı iş yüklerinde *Greedy Idle* sisteme kıyasla daha yüksek sistem adaletsizliği gözlemlenmektedir. Çünkü DR-STRaNGe, GRSÜ uygulamalarının performansını etkili rastgele sayı depolama tekniği ile GRSÜ olmayan uygulamalardan daha fazla artırmaktadır. Sonuç olarak, DR-STRaNGe diğer sistem tasarımlarına kıyasla ortalamada daha yüksek sistem adilliğine sahiptir.



Şekil 8.4: Çift Çekirdekli Sistemde Sistem Adilliği.

8.3 Rastgele Sayı Depolama Mekanizmasının Etkisi

Rastgele sayı arabelleğinden rastgele sayı istekleri karşılamak, GRSÜ gecikmesini azaltarak toplam yürütme süresini azaltmaktadır. Rastgele sayı arabelleğinin etkisi, rastgele sayı arabelleğinden karşılanan rastgele sayı isteklerinin tüm rastgele sayı isteklerine oranına bağlıdır. Bu orana *arabellek hizmet oranı* adını vermekteyiz. *Arabellek hizmet oranı*, (1) DRAM kanalı kullanımına, (2) gerekli GRSÜ aktarım

hızına ve (3) rastgele sayı arabellek boyutuna bağlıdır. Bu bölümde, basit rastgele sayı depolama mekanizmasında farklı rastgele sayı arabellek boyutlarıyla DR-STRaNGe'i değerlendirmekteyiz.

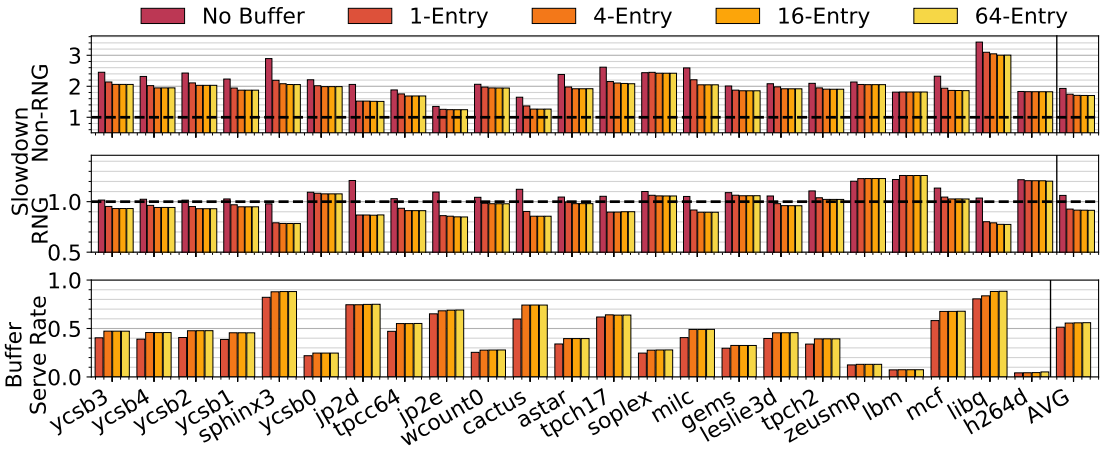
Şekil 8.5 çift çekirdekli bir sistemde yürütülen GRSÜ olmayan uygulamaların ve GRSÜ uygulamalarının tek başına yürütüldükleri zamana kıyasla yavaşlamasını (üst ve orta) ve sistemin arabellek hizmet oranını (alt) göstermektedir. Bu şekilden hareketle iki gözlem yapmaktayız.

İlk olarak, 16 elemanlı bir rastgele sayı arabelleği ortalama performansı GRSÜ ve GRSÜ olmayan uygulamalar için sırasıyla %13.8 ve %11.7 artırmaktadır. Uygulamaların performansı (üst ve orta), 16 elemana kadar artan eleman sayısı ile beraber artmaktadır ve 16 elemanlı arabellek ile önemli bir performans artışı gözlemlenmektedir.

İkinci olarak, 16 elemanlı bir rastgele sayı arabelleğinin ortalama arabellek hizmet oranı 0.55 olarak gözlemlenmektedir. Artan eleman sayısı ile arabellek hizmet oranı yalnızca birkaç uygulama için (*jp2, cactusADM, libquantum*) artmaktadır.

Son olarak, GRSÜ olmayan bellek isteklerinin yoğun olduğu uygulamalar (*zeusmp, lbm*) az sayıda uzun atıl periyot gözlemlenmesi sebebi ile yavaşlamalar yaşamaktadır. Bunun sebebi RSÜ çatışmasının arabellek için rastgele sayı üretimi ile artmasıdır. Bu iş yükleri arabellekten yararlanmamaktadır ve alt şekilde görüldüğü üzere düşük arabellek hizmet oranlarına sahiptir.

Sonuç olarak, rastgele sayı depolama mekanizması ortalamada GRSÜ ve GRSÜ olmayan uygulamaların performansını RSÜ çatışmasını azaltarak ve GRSÜ gecikmesini düşürerek artırmaktadır.



Şekil 8.5: Rastgele sayı arabellek boyutunun GRSÜ olmayan uygulamaların ve GRSÜ uygulamalarının yürütme zamanına (üst ve orta) ve arabellek hizmet oranına (alt) etkisi

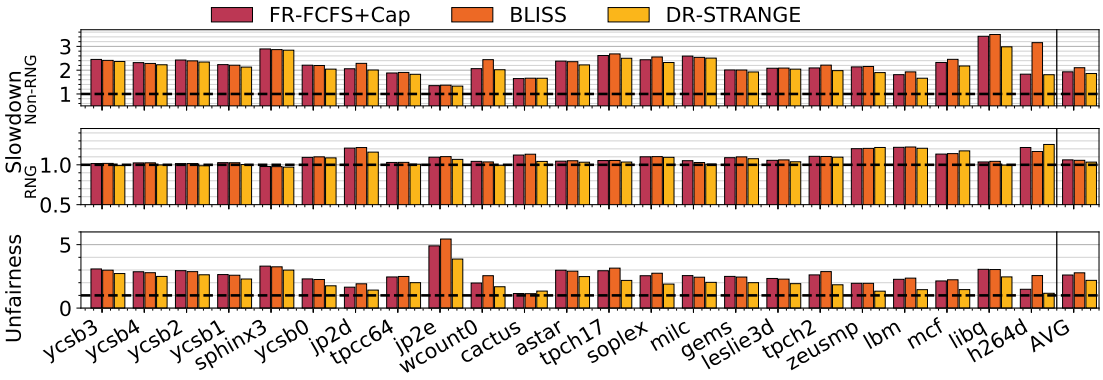
8.4 GRSÜ Bellek İstek Planlayıcısı

Bu bölümde (1) DR-STRaNGe, (2) BLISS [108, 103][3], (3) *RNG-oblivious* sistem tasarımı ile beraber FR-FCFS'i [94, 95] 16 sütun kapasitesi ile (FR-FCFS+Cap [97]) karşılaştırmaktayız.

Şekil 8.6 GRSÜ bellek istek planlayıcısının GRSÜ (orta) ve GRSÜ olmayan (üst) uygulamaların performanslarına etkisini uygulamaların tek çekirdekte tek başlarına yürütülmesine kıyasla göstermektedir. İki gözlem yapmaktayız.

İlk olarak, GRSÜ bellek istek planlayıcısı, FR-FCFS+Cap ve BLISS plalayıcılarının sistem adilliğine kıyasla daha yüksek sistem adilliği sağlamaktadır. Ortalama performansı GRSÜ ve GRSÜ olmayan uygulamalar için %1.6 ve %5.6 artırmaktadır. GRSÜ bellek istek zamanlayıcısının performansı FR-FCFS+Cap ve BLISS planlayıcılarından neredeyse her iş yükü için daha yüksektir.

İkinci olarak, BLISS'in sistem adilliği FR-FCFS+Cap planlayıcısına kıyasla daha düşüktür ve sistem adaletsizliği metriğini %6.6 artırmaktadır. Bellek istek yoğunluğu yüksek GRSÜ olmayan uygulamaların olduğu bazı iş yüklerinde (*jp2e, wcount0, tpch17, soplex, tpch2, lbm, mcf*, ve *h264d*), BLISS'in adaletsizlik metriği FR-FCFS+Cap ve GRSÜ bellek istek planlayıcısından daha yüksektir, bunun sebebi BLISS'in bellekte rastgele sayı üretimine uygun planlama yapmaması ve bu nedenle GRSÜ olmayan uygulamaları kara listeye alarak GRSÜ uygulamalarını daha fazla önceliklendirmesidir. GRSÜ olmayan uygulamaların kara listeye alınması sebebi ile BLISS ortalama GRSÜ olmayan uygulama performansını FR-FCFS+Cap tasarımına göre %8.9 düşürmektedir.



Şekil 8.6: Bellek istek planlayıcısının GRSÜ olmayan (üst) ve GRSÜ uygulamalarının (orta) performansına ve sistemin adilliğine (alt) etkisi.

8.5 Önceliğe Dayalı Bellek İstek Planlayıcının Etkisi

Önceliğe dayalı bellek istek planlamanın etkisini göstermek için üç farklı tasarımı karşılaştırmaktayız: (1) *RNG-Oblivious* sistem tasarımı, (2) GRSÜ uygulamalarını önceliklendiren DR-STRaNGe ve (3) GRSÜ olmayan uygulamaları önceliklendiren DR-STRaNGe.

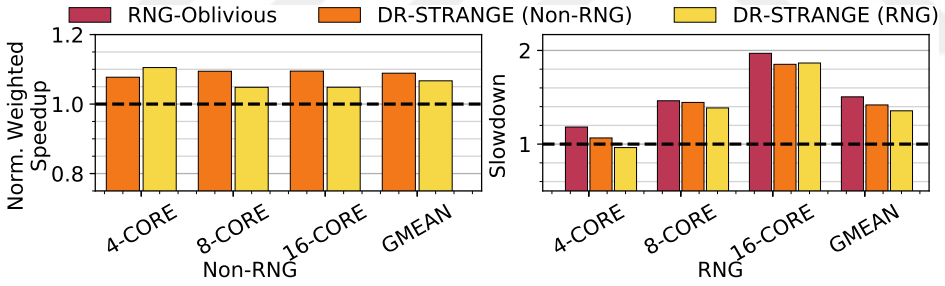
[3]BLISS'i modellerken bildiride önerildiği üzere *Blacklisting Threshold* parametresi için 4 değerini ve *Clearing Interval* parametresi için ise 10000 çevrim değerini kullanmaktayız.

Şekil 8.7 GRSÜ olmayan uygulamaların ağırlıklı hızlanma oranlarını (sol) ve GRSÜ uygulamalarının yavaşlama oranlarını (sağ) göstermektedir. Sonuçlar uygulamaların tek çekirdekte tek başlarına yürütülmeleri durumundaki performanslarına normalize edilmiştir. Bu şekilden hareketle üç gözlem yapmaktayız.

İlk olarak önceliklendirmeye dayalı GRSÜ bellek istek planlayıcısı önemli oranda daha yüksek performans artışı göstermektedir ve bunun sebebi uygulamaların DRAM bant genişliğini daha verimli kullanmasıdır. GRSÜ ve GRSÜ olmayan uygulamalar GRSÜ bellek istek planlamadan yararlanmaktadır ve performans artışları yüksek öncelik seviyesi ile artmaktadır.

İkinci olarak, Şekil 8.7 (sol) GRSÜ olmayan uygulamaların ortalama ağırlıklı hızlanma oranları DR-STRaNGe kullanıldığında *RNG-Oblivious* sistem tasarımına kıyasla %8.9 artmaktadır. Şekil 8.7 (sağ) GRSÜ uygulamalarını daha yüksek seviyede öncelikle yürüttüğümüz durumda DR-STRaNGe ortalama performansı %9.9 artırmaktadır.

Üçüncü olarak, GRSÜ uygulamalarının daha yüksek öncelik seviyesinde çalıştırıldığı durumda DR-STRaNGe hem GRSÜ hem de GRSÜ olmayan uygulamaların ortalama performansını 4 çekirdekli sistemlerde artırmaktadır. Düşük ve orta bellek istek yoğunluklarına sahip GRSÜ olmayan uygulamaların olduğu bazı iş yüklerinde, GRSÜ olmayan uygulamalar yüksek önceliğe sahip olduğunda sistem GRSÜ ve normal çalışma modu arasında sık sık geçiş yaptığı için performans düşmektedir. Bu iş yükleri GRSÜ uygulamaları daha yüksek önceliğe sahip olduğu zaman düşük RSÜ çatışmasından yararlanmaktadır.

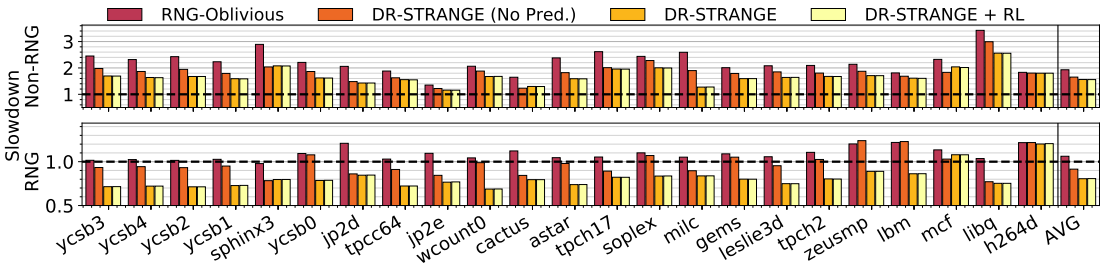


Şekil 8.7: GRSÜ bellek istek planlayıcının GRSÜ (sağ) ve GRSÜ olmayan (sol) uygulamalara etkisi.

8.6 DRAM Atıllık Öngörücünün Etkisi

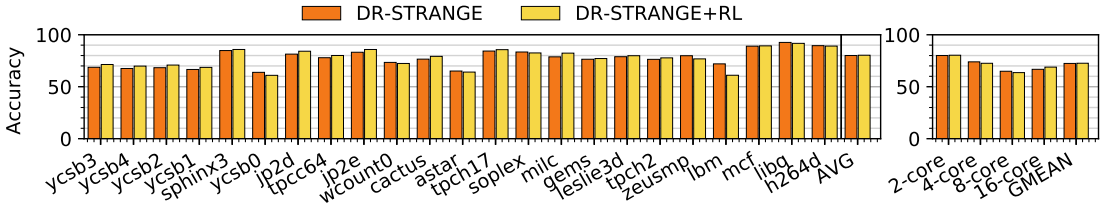
Şekil 8.8 dört sistem tasarımının performansını göstermektedir: (1) *RNG-Oblivious* temel sistem tasarımı, (2) DR-STRaNGe (öngörüsüz), (3) DR-STRaNGe + DRAM atıllık öngörücü ve (4) DR-STRaNGe + pekiştirmeli öğrenme ile DRAM atıllık öngörüsü. Şekil 8.8 GRSÜ olmayan (üst) ve GRSÜ uygulamalarının (alt) yavaşlama oranlarını göstermektedir. Üç gözlem yapmaktayız. İlk olarak, DR-STRaNGe + DRAM atıllık öngörücüsü *RNG-oblivious* temel sistem tasarımına kıyasla GRSÜ ve GRSÜ olmayan uygulamaların ortalama performanslarını %25.1 ve %17.9 artırmaktadır. İkinci olarak, DR-STRaNGe + DRAM atıllık öngörücüsü, öngörü mekanizmasına sahip olmayan DR-STRaNGe tasarımına kıyasla GRSÜ ve GRSÜ olmayan uygulamalar için %13.8 ve %12.4 daha yüksek performans göstermektedir.

Üçüncü olarak, DR-STRaNGe ve basit DRAM atıllık öngörücüsü DR-STRaNGe ve pekiştirmeli öğrenme ile DRAM atıllık öngörüsü yapan sistem tasarımına benzer perfomans göstermektedir. DR-STRaNGe ve pekiştirmeli öğrenmeye dayalı DRAM atıllık öngörücüsü GRSÜ ve GRSÜ olmayan uygulamaların ortalama performansını %23.9 ve %19.3 artırmaktadır.



Şekil 8.8: DRAM atıllık öngörücünün GRSÜ olmayan (üst) ve GRSÜ uygulamalarının (alt) performansına etkisi.

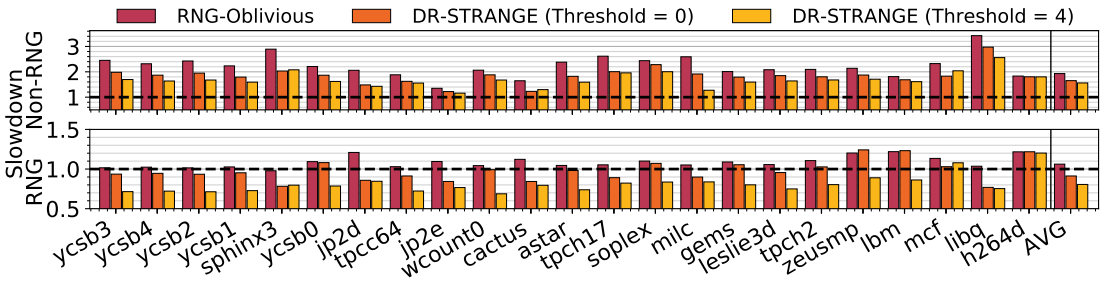
Şekil 8.9 çift çekirdekli sistemlerde (sol) ve 2, 4, 8 ve 16 çekirdekli sistemlerde öngörücünün doğruluk oranlarını göstermektedir. Bu sonuçlardan yola çıkarak üç gözlem yapmaktayız. İlk olarak, basit DRAM atıllık öngörücüsü ve pekiştirmeli öğrenmeye dayalı öngörücü ortalama olarak %80.0 ve %80.3 doğruluk oranına sahiptir. İkinci olarak, bellek istek yoğunluğu yüksek olan ve daha az uzun DRAM atıl periyodu üreten iş yükleri için basit DRAM atıllık öngörücüsünün doğruluk oranı pekiştirmeli öğrenmeye dayalı öngörücüden daha yüksektir. Daha düşük bellek istek yoğunluğuna sahip iş yüklerinde ise, basit DRAM atıllık öngörücüsü uzun atıl periyotları kısa olarak öngörmektedir. Bu nedenle doğru tahmin oranı daha düşüktür. Son olarak, 4, 8 ve 16 çekirdekli iş yükleri için iki öngörücü de daha düşük doğruluğa sahiptir. Çünkü bu iş yüklerinde daha az atıl periyot vardır ve daha yüksek çatışma örüntüleri gözlemlenmektedir.



Şekil 8.9: DRAM atıllık öngörücünün çift çekirdekli (sol) ve çok çekirdekli iş yüklerinde (sağ) doğruluk oranları.

8.7 Düşük DRAM Kullanımı Ölçütünün Etkisi

Düşük DRAM kullanımı ölçütünün etkisini gözlemlemek için temel sisteme ek olarak DR-STRaNGe'i düşük DRAM kullanımına izin vermeyen bir tasarım ile ve DRAM istek kuyruklarında 4 bellek isteğinin bulunmasına izin veren bir tasarım ile gerçeklemekteyiz. Şekil 8.10 düşük DRAM kullanımı ölçütünün performans üzerine etkisini GRSÜ olmayan (üst) ve GRSÜ uygulamalarında (alt) göstermektedir. Deneylerimiz sonucunda, düşük DRAM kullanımı ölçütünün GRSÜ ve GRSÜ olmayan uygulamalar için ortalama performansı düşük DRAM kullanımına izin vermeyen sistem tasarımına kıyasla %11.7 ve %5.5 artırdığını gözlemlemekteyiz.



Şekil 8.10: Düşük DRAM kullanımı ölçütünün GRSÜ olmayan (üst) ve GRSÜ uygulamalarının (alt) performansına etkisi.

8.8 QUAC-TRNG ile Deneyler

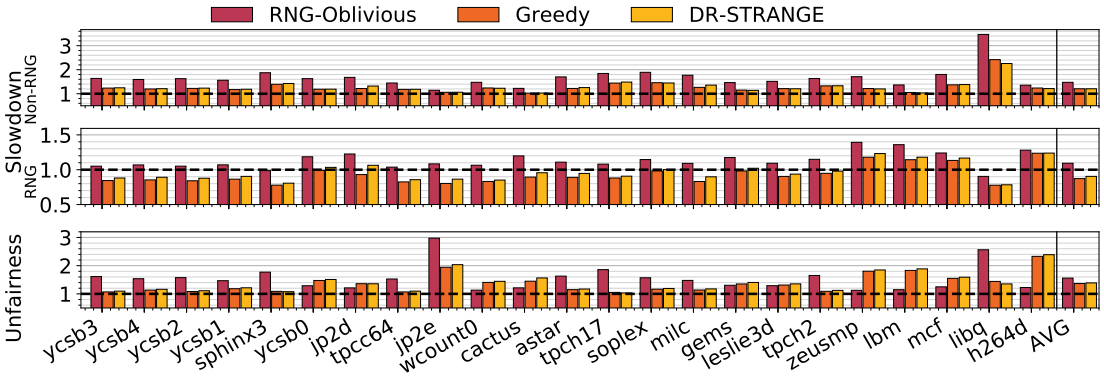
DR-STRaNGe'in farklı DRAM tabanlı GRSÜ mekanizmaları ile uyumlu olduğunu göstermek için DR-STRaNGe, QUAC-TRNG [13] ile test etmekteyiz. QUAC-TRNG, D-RaNGe [12]'e kıyasla daha yüksek hızla GRSÜ gerçekleştirebilmektedir. Ancak, aynı zamanda 64 bit rastgele sayı üretme gecikmesi de daha yüksektir. DR-STRaNGe'in performans ve sistem adilliği üzerine etkisini *RNG-Oblivious* sistem tasarımı üzerinde iki sistemde de QUAC-TRNG kullanarak göstermekteyiz.

Şekil 8.11 GRSÜ olmayan ve GRSÜ uygulamalarının performansını (üst ve orta) ve sistem adilliğini (alt) göstermektedir. Üç gözlemde bulunmaktayız. İlk olarak, DR-STRaNGe GRSÜ ve GRSÜ olmayan uygulamaların ortalama performanslarını %17.2 ve %18.2 artırmaktadır. İkinci olarak, DR-STRaNGe sistem adilliğini ortalamada %10.9 artırmaktadır. Yüksek bellek istek yoğunluğuna sahip uygulamaların bulunduğu iş yükleri (*zeusmp, lbm, mcf, h264d*) daha yüksek sistem adaletsizlik metriğine sahiptir. Bunun sebebi ise DR-STRaNGe'in GRSÜ olmayan uygulamaların performansını diğerlerine kıyasla daha fazla artırmasıdır.

Sonuç olarak, DR-STRaNGe hem performans hem de sistem adilliğini gerçeklenen DRAM tabanlı GRSÜ mekanizmasına bağlı olmadan yükseltmektedir. DR-STRaNGe aynı zamanda farklı DRAM tabanlı GRSÜ mekanizmalarının bir arada kullanımasından yararlanabilir. Daha düşük gecikmeye sahip DRAM tabanlı GRSÜ mekanizmasını kullanarak rastgele sayı arabelleğini doldurabilir ve daha yüksek hıza sahip mekanizmayı kullanarak arabellek boş olduğunda istekleri karşılayabilir. Bu mekanizmaların araştırılmasını gelecek çalışmalara bırakmaktayız.

8.9 Düşük Rastgele Sayı İstek Yoğunluğuna Sahip Uygulamalar ile Sonuçlar

DR-STRaNGe'i bir düşük GRSÜ istek yoğunluğuna sahip bir GRSÜ uygulaması (640 Mb/s) ve bir GRSÜ olmayan uygulamadan oluşan iş yükleri ile test etmekteyiz. DR-STRaNGe'in GRSÜ ve GRSÜ olmayan uygulamaların ortalama performansını %4.6 ve %3.2 düşürdüğünü gözlemlemekteyiz. Düşük RSÜ çatışması sebebi ile, DR-STRaNGe sistem adilliğini önemli bir oranda artırmamaktadır.



Şekil 8.11: Çift çekirdekli ve QUAC-TRNG [13] kullanan bir sistemde GRSÜ ve GRSÜ olmayan uygulamaların performans ve sistem adilliği.

8.10 Alan ve Enerji Tüketimi Analizi

DR-STRaNGe'in enerji tüketimini değerlendirmek için DRAMPower [189]'ı Ramulator benzetim aracının çıktıları ile çalıştırmaktayız. DR-STRaNGe'in enerji tüketimini ve toplam bellek çevrimini, *RNG-Oblivious* sistem tasarımına göre %21 ve %15.8 düşürdüğünü gözlemlemekteyiz.

Alan değerlendirmesi için ise CACTI [190] aracını $22nm$ üretim teknoloji düğümü ile kullanmaktayız. DR-STRaNGe'i rastgele sayı arabelleği, GRSÜ bellek kuyruğu ve basit DRAM atıllık öngörücüsü ile modellemekteyiz. Sonuç olarak DR-STRaNGe'in düşük alanda gerçeklenebildiğini göstermekteyiz: $0.0022mm^2$ (bir Intel Cascade Lake işlemci çekirdeğinin [61] 0.00048%'ine denk gelmektedir). Değerlendirmemiz sonucunda, basit DRAM atıllık öngörücünün 256 satır olduğu ve her satırda 2 bit sayaç bulunduğu durumda toplamda 0.0625 KB alan kapladığını göstermekteyiz. Bu öngörücü yerine pekiştirmeli öğrenmeye dayalı öngörücü kullanıldığı takdirde, DR-STRaNGe'in kapladığı alan $0.012mm^2$ (bir Intel Cascade Lake işlemci çekirdeğinin [61] 0.0033%'ine denk gelmektedir) olmaktadır ve pekiştirmeli öğrenmeye dayalı öngörücü 4 bayt ve 10 bit durum değerleri kullanıldığında $8KB$ alan kaplamaktadır.

9. GEÇMİŞ ÇALIŞMALAR

Bilindiği kadarıyla, DR-STRaNGe DRAM tabanlı GRSÜ mekanizmaları için tasarlanan ve (1) GRSÜ ve GRSÜ olmayan uygulamalar arasında bellek denetleyicisinde oluşan çatışmayı azaltan, (2) sistem adilliğini artıran ve (3) GRSÜ gecikmesini azaltan ilk uçtan uca sistem tasarımıdır. Önceki bölümlerde DR-STRaNGe'a yakın olan bellek istek planlayıcısı (Bölüm 8.4) ve DRAM tabanlı GRSÜ mekanizmaları üzerine önceki çalışmaları tartışmaktayız. DR-STRaNGe'i geçmiş çalışmalarla Bölüm 8.2, Bölüm 8.1 ve Bölüm 8.8'de karşılaştırmaktayız. Bu bölümde geri kalan geçmiş çalışmalardan bahsetmekteyiz.

9.1 Bellek İstek Planlayıcı

Bellek istek planlayıcıları üzerine yapılan geçmiş çalışmalar [98, 99, 100, 101, 104, 97, 102, 107, 106, 191, 103, 108, 97, 192, 193] uygulamalar arası çatışmayı en aza indirmeyi ve sistem performansı ve adilliğini artırmayı amaçlar. Bu öneriler bellekte rastgele sayı üretimini yoksaymaktadır. DR-STRaNGe'i geçmiş çalışmalar olan BLISS [103, 108] ve FR-FCFS [94, 95] +Cap [97] ile Bölüm 8.4'te karşılaştırmaktayız.

9.2 Düşük Hızlı DRAM tabanlı GRSÜ Mekanizmaları

Geçmiş çalışmalar, farklı entropi kaynaklarına sahip DRAM tabanlı GRSÜ mekanizmaları önermektedir. Bunlara örnek olarak, DRAM başlangıç değerleri [59] ve DRAM'de yenileme (*-ing. refresh*) işleminin zamanında yapılmamasından kaynaklanan değer kaybı hataları [23, 20, 21, 22] verilebilir. Bu mekanizmaların sağlayabildiği GRSÜ hızı sınırlıdır ve sürekli olarak kullanılmaları mümkün değildir. Bu nedenle, hızlı DRAM tabanlı GRSÜ mekanizmalarına [12, 13] kıyasla daha az kullanışlı olarak görülmektedirler.

9.3 DRAM Atıllık Öngörücüler

Önceki çalışmalar DRAM'de atıl periyot uzunluklarını tahmin ederek DRAM için tüketilen enerjiyi azaltmayı [194] ve son seviye önbellekten geri yazılacak verileri zamanlamayı [195] amaçlamaktadır. Bu mekanizmalara kıyasla DR-STRaNGe'in DRAM atıllık öngörücüsü düşük masrafla basit bir şekilde donanımda gerçeklenebilmektedir (Bölüm 8.10) ve işlemci ile bellek denetleyicisi arasındaki arayüzde herhangi bir değişiklik gerektirmemektedir.

10. SONUÇ

DRAM tabanlı GRSÜ mekanizmaları için bellek denetleyicisindeki RSÜ çatışmasını azaltan, farklı uygulama tipleri arasında sistem adilliğini sağlayan ve DRAM tabanlı GRSÜlerin gecikmelerini başarıyla saklayan ilk uçtan uca sistem tasarımı olan DR-STRaNGe'i önermekteyiz. Sistem tasarımımız üç ana bileşenden oluşmaktadır: (1) DRAM atıllık öngörücüsü ile beraber çalışan bir rastgele sayı depolama mekanizması, (2) bir GRSÜ bellek istek planlayıcısı ve (3) bir uygulama arayüzü. Deneysel değerlendirmemiz sonucunda DR-STRaNGe'in GRSÜ ve GRSÜ olmayan uygulamaların ortalama performansını %25.1 ve %17.9 ve ortalama sistem adilliğini %32.1 artırdığını göstermekteyiz. Aynı zamanda DR-STRaNGe ortalama enerji tüketimini %21 azaltmaktadır. Sonuç olarak uçtan uca bir sistem tasarımı ile DRAM tabanlı GRSÜlerin günümüz sistemlerinde kullanılabileceğini ve bu sayede düşük gecikme ve yüksek hızla gerçek rastgele sayı üretilebileceğini göstermekteyiz.

KAYNAKLAR

[1] **Bagini**, **V.** and **Bucci**, **M.** (1999). A Design of Reliable True Random Number Generator for Cryptographic Applications. In: *CHES*.

[2] **Barangi**, **M.**, **Chang**, **J. S.**, and **Mazumder**, **P.** (2016). Straintronics-Based True Random Number Generator for High-Speed and Energy-Limited Applications. In: *IEEE Trans. Magn*.

[3] **Cherkaoui**, **A.**, **Fischer**, **V.**, **Fesquet**, **L.**, and **Aubert**, **A.** (2013). A Very High Speed True Random Number Generator with Entropy Assessment. In: *CHES*.

[4] **Pareschi**, **F.**, **Setti**, **G.**, and **Rovatti**, **R.** (2006). A Fast Chaos-based True Random Number Generator for Cryptographic Applications. In: *ESSCIRC*.

[5] **Gutterman**, **Z.**, **Pinkas**, **B.**, and **Reinman**, **T.** (2006). Analysis of the Linux Random Number Generator. In: *SP*.

[6] **Kaenel**, **V. von** and **Takayanagi**, **T.** (2007). Dual True Random Number Generators for Cryptographic Applications Embedded on a 200 Million Device Dual CPU SOC. In: *CICC*.

[7] **Kim**, **J.**, **Ahmed**, **T.**, **Nili**, **H.**, **Truong**, **N. D.**, **Yang**, **J.**, **Jeong**, **D. S.**, **Sriram**, **S.**, **Ranasinghe**, **D. C.**, and **Kavehei**, **O.** (2017). Nano-Intrinsic True Random Number Generation. arXiv:1701.06020.

[8] **Drutarovsky**, **M.** and **Galajda**, **P.** (2007). A Robust Chaos-based True Random Number Generator Embedded in Reconfigurable Switched-Capacitor Hardware. In: *Radioelektronika*.

[9] **Kwok**, **S. H.** and **Lam**, **E. Y.** (2006). FPGA-based High-speed True Random Number Generator for Cryptographic Applications. In: *TENCON*.

[10] **Zhang**, **T.**, **Yin**, **M.**, **Xu**, **C.**, **Lu**, **X.**, **Sun**, **X.**, **Yang**, **Y.**, and **Huang**, **R.** (2017). High-speed True Random Number Generation Based on Paired Memristors for Security Electronics. In: *Nanotechnology*.

[11] **Labs**, **Q.** (2015). Random Number Generators. White Paper.

[12] **Kim**, **J. S.**, **Patel**, **M.**, **Hassan**, **H.**, **Orosa**, **L.**, and **Mutlu**, **O.** (2019). D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers with Low Latency and High Throughput. In: *HPCA*.

[13] **Olgun**, **A.**, **Patel**, **M.**, **Yağlıkçı**, **A. G.**, **Luo**, **H.**, **Kim**, **J. S.**, **Bostancı**, **N.**, **Vijaykumar**, **N.**, **Ergin**, **O.**, and **Mutlu**, **O.** (2021). QUAC-TRNG: High-Throughput True Random Number Generation Using Quadruple Row Activation in Commodity DRAM Chips. In: *ISCA*.

[14] **Wang**, **Y.**, **Hui**, **C.**, **Liu**, **C.**, and **Xu**, **C.** (2016). Theory and Implementation of a Very High Throughput True Random Number Generator in Field Programmable Gate Array. In: *Review of Scientific Instruments*.

[15] **Tilborg**, **H. C. v.** and **Jajodia**, **S.** (2011). Encyclopedia of Cryptography and Security.

[16] **Chevalier**, **P.**, **Menard**, **C.**, and **Dorval**, **B.** (1974). Random Number Generator. US Patent 3,790,768.

[17] **Knuth**, **D. E.** (1998). The Art of Computer Programming, 2: Seminumerical Agorithms, Addision Wesley. In.

[18] **Tsoi**, **K. H.**, **Leung**, **K.**, and **Leong**, **P. H. W.** (2003). Compact FPGA-based True and Pseudo Random Number Generators. In: *FCCM*.

[19] **Pyo**, **C.**, **Pae**, **S.**, and **Lee**, **G.** (2009). DRAM as Source of Randomness. In: *IET*.

[20] **Keller**, **C.**, **Gurkaynak**, **F.**, **Kaeslin**, **H.**, and **Felber**, **N.** (2014). Dynamic Memory-based Physically Unclonable Function for the Generation of Unique Identifiers and True Random Numbers. In: *ISCAS*.

[21] **Sutar**, **S.**, **Raha**, **A.**, **Kulkarni**, **D.**, **Shorey**, **R.**, **Tew**, **J.**, and **Raghunathan**, **V.** (2018). D-PUF: An Intrinsically Reconfigurable DRAM PUF for Device Authentication and Random Number Generation. In: *TECS*.

[22] **Hashemian**, **M. S.**, **Singh**, **B.**, **Wolff**, **F.**, **Weyer**, **D.**, **Clay**, **S.**, and **Papachristou**, **C.** (2015). A Robust Authentication Methodology Using Physically Unclonable Functions in DRAM Arrays. In: *DATE*.

[23] **Tehranipoor**, **F.**, **Yan**, **W.**, and **Chandy**, **J. A.** (2016). Robust Hardware True Random Number Generators using DRAM Remanence Effects. In: *HOST*.

[24] **Wang**, **Y.**, **Yu**, **W.**, **Wu**, **S.**, **Malysa**, **G.**, **Suh**, **G. E.**, and **Kan**, **E. C.** (2012). Flash Memory for Ubiquitous Hardware Security Functions: True Random Number Generation and Device Fingerprints. In: *SP*.

[25] **Ray**, **B.** and **Milenković**, **A.** (2018). True Random Number Generation Using Read Noise of Flash Memory Cells. In: *IEEE Transactions on Electron Devices*.

[26] **Holcomb**, **D. E.**, **Burleson**, **W. P.**, and **Fu**, **K.** (2007). Initial SRAM State as a Fingerprint and Source of True Random Numbers for RFID Tags. In: *RFID*.

[27] **Holcomb**, **D. E.**, **Burleson**, **W. P.**, and **Fu**, **K.** (2009). Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers. In: *TC*.

[28] **Leest**, **V. van der**, **Sluis**, **E. van der**, **Schrijen**, **G.-J.**, **Tuyls**, **P.**, and **Handschuh**, **H.** (2012). Efficient Implementation of True Random Number Generator Based on SRAM PUFs. In: *Cryptography and Security: From Theory to Applications*.

[29] **Chan**, **J. J. M.**, **Sharma**, **B.**, **Lv**, **J.**, **Thomas**, **G.**, **Thulasiram**, **R.**, and **Thulasiraman**, **P.** (2011). True Random Number Generator using GPUs and Histogram Equalization Techniques. In: *HPCC*.

[30] **Tzeng**, **S.** and **Wei**, **L.** (2008). Parallel White Noise Generation on a GPU via Cryptographic Hash. In: *I3D*.

[31] **Teh**, **J. S.**, **Samsudin**, **A.**, **Al-Mazrooie**, **M.**, and **Akhavan**, **A.** (2015). GPUs and Chaos: A New True Random Number Generator. In: *Nonlinear Dynamics*.

[32] **Majzoobi**, **M.**, **Koushanfar**, **F.**, and **Devadas**, **S.** (2011). FPGA-based True Random Number Generation using Circuit Metastability with Adaptive Feedback Control. In: *CHES*.

[33] **Wieczorek**, **P. Z.** (2014). An FPGA Implementation of the Resolve Time-Based True Random Number Generator With Quality Control. In: *IEEE Transactions on Circuits and Systems*.

[34] **Chu**, **P. P.** and **Jones**, **R. E.** (1999). Design Techniques of FPGA Based Random Number Generator. In: *MAPLD*.

[35] **Amaki**, **T.**, **Hashimoto**, **M.**, and **Onoye**, **T.** (2015). An Oscillator-based True Random Number Generator with Process and Temperature Tolerance. In: *DAC*.

[36] **Mathew**, **S. K.**, **Srinivasan**, **S.**, **Anders**, **M. A.**, **Kaul**, **H.**, **Hsu**, **S. K.**, **Sheikh**, **F.**, **Agarwal**, **A.**, **Satpathy**, **S.**, and **Krishnamurthy**, **R. K.** (2012). 2.4 Gbps, 7 mW All-digital PVT-variation Tolerant True Random Number Generator for 45 nm CMOS High-performance Microprocessors. In: *JSSC*.

[37] **Brederlow**, **R.**, **Prakash**, **R.**, **Paulus**, **C.**, and **Thewes**, **R.** (2006). A Low-power True Random Number Generator using Random Telegraph Noise of Single Oxide-traps. In: *ISSCC*.

[38] **Tokunaga**, **C.**, **Blaauw**, **D.**, and **Mudge**, **T.** (2008). True Random Number Generator with a Metastability-based Quality Control. In: *JSSC*.

[39] **Bucci**, **M.**, **Germani**, **L.**, **Luzzi**, **R.**, **Trifiletti**, **A.**, and **Varanonuovo**, **M.** (2003). A High-speed Oscillator-based Truly Random Number Source for Cryptographic Applications on a Smart Card IC. In: *TC*.

[40] **Bhargava**, **M.**, **Sheikh**, **K.**, and **Mai**, **K.** (2015). Robust True Random Number Generator using Hot-carrier Injection Balanced Metastable Sense Amplifiers. In: *HOST*.

[41] **Kinniment**, **D.** and **Chester**, **E.** (2002). Design of an On-chip Random Number Generator using Metastability. In: *ESSCIRC*.

[42] **Holleman**, **J.**, **Bridges**, **S.**, **Otis**, **B. P.**, and **Diorio**, **C.** (2008). A 3mu W CMOS True Random Number Generator with Adaptive Floating-Gate Offset Cancellation. In: *JSSC*.

[43] **Dorrendorf**, **L.**, **Gutterman**, **Z.**, and **Pinkas**, **B.** (2007). Cryptanalysis of the Windows Random Number Generator. In: *CCS*.

[44] **Lacharme**, **P.**, **Rock**, **A.**, **Strubel**, **V.**, and **Videau**, **M.** (2012). The Linux Pseudorandom Number Generator Revisited. Cryptology ePrint Archive, Report 2012/251.

[45] **Yang**, **K.**, **Blaauw**, **D.**, and **Sylvester**, **D.** (2016). An All-digital Edge Racing True Random Number Generator Robust Against PVT Variations. In: *JSSC*.

[46] **Matsumoto**, **M.** and **Nishimura**, **T.** (1998). Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator. In: *TOMACS*.

[47] **Blum**, **L.**, **Blum**, **M.**, and **Shub**, **M.** (1986). A Simple Unpredictable pseudo-random Number Generator. In: *SIAM Journal on Computing*.

[48] **Mascagni**, **M.** and **Srinivasan**, **A.** (2000). Algorithm 806: SPRNG: A Scalable Library for Pseudorandom Number Generation. In: *TOMS*.

[49] **Steele Jr**, **G. L.**, **Lea**, **D.**, and **Flood**, **C. H.** (2014). Fast Splittable Pseudorandom Number Generators. In: *OOPSLA*.

[50] **Marsaglia**, **G.** (2003). Xorshift RNGs. In: *Journal of Statistical Software*.

[51] **Koç**, **Ç. K.** (2009). About Cryptographic Engineering. In: *Cryptographic Engineering*.

[52] **Clarke**, **P. J.**, **Collins**, **R. J.**, **Hiskett**, **P. A.**, **Townsend**, **P. D.**, and **Buller**, **G. S.** (2011). Robust Gigahertz Fiber Quantum Key Distribution. In: *Applied Physics Letters*.

[53] **Lu**, **X.**, **Zhang**, **L.**, **Wang**, **Y.**, **Chen**, **W.**, **Huang**, **D.**, **Li**, **D.**, **Wang**, **S.**, **He**, **D.**, **Yin**, **Z.**, **Zhou**, **Y.**, **Hui**, **C.**, and **Han**, **Z.** (2015). FPGA Based Digital Phase-coding Quantum Key Distribution System. In: *Science China Physics, Mechanics & Astronomy*.

[54] **Hull**, **T. E.** and **Dobell**, **A. R.** (1962). Random Number Generators. In: *SIAM Review*.

[55] **Ma**, **X.**, **Yuan**, **X.**, **Cao**, **Z.**, **Qi**, **B.**, and **Zhang**, **Z.** (2016). Quantum Random Number Generation. In: *Quantum Inf.*

[56] **Botha**, **R.** (2005). The Development of a Hardware Random Number Generator for Gamma-ray Astronomy. PhD thesis. North-West University.

[57] **Davis**, **P.** and **Rabinowitz**, **P.** (1956). Some Monte Carlo Experiments in Computing Multiple Integrals. In: *Mathematical Tables and Other Aids to Computation*.

[58] **Corrigan-Gibbs**, **H.**, **Mu**, **W.**, **Boneh**, **D.**, and **Ford**, **B.** (2013). Ensuring High-quality Randomness in Cryptographic Key Generation. In: *CCS*.

[59] **Eckert**, **C.**, **Tehranipoor**, **F.**, and **Chandy**, **J. A.** (2017). DRNG: DRAM-based Random Number Generation Using its Startup Value Behavior. In: *MWSCAS*.

[60] **Bahar Talukder**, **B. M. S.**, **Kerns**, **J.**, **Ray**, **B.**, **Morris**, **T.**, and **Rahman**, **M. T.** (2019). Exploiting DRAM Latency Variations for Generating True Random Numbers. In: *ICCE*.

[61] **WikiChip** (n.d.). Cascade Lake SP - Intel. `https://en.wikichip.org/wiki/intel/cores/cascade_lake_sp`.

[62] **Keeth**, **B.** and **Baker**, **R.** (2001). DRAM Circuit Design: A Tutorial.

[63] **Kim**, **Y.**, **Seshadri**, **V.**, **Lee**, **D.**, **Liu**, **J.**, and **Mutlu**, **O.** (2012). A Case for Exploiting Subarray-level Parallelism (SALP) in DRAM. In: *ISCA*.

[64] **Lee**, **D.**, **Kim**, **Y.**, **Seshadri**, **V.**, **Liu**, **J.**, **Subramanian**, **L.**, and **Mutlu**, **O.** (2013). Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture. In: *HPCA*.

[65] **Lee**, **D.**, **Kim**, **Y.**, **Pekhimenko**, **G.**, **Khan**, **S.**, **Seshadri**, **V.**, **Chang**, **K.**, and **Mutlu**, **O.** (2015). Adaptive-latency DRAM: Optimizing DRAM Timing for the Common-case. In: *HPCA*.

[66] **Seshadri**, **V.**, **Kim**, **Y.**, **Fallin**, **C.**, **Lee**, **D.**, **Ausavarungnirun**, **R.**, **Pekhimenko**, **G.**, **Luo**, **Y.**, **Mutlu**, **O.**, **Gibbons**, **P. B.**, **Kozuch**, **M. A.**, and **Mowry**, **T.** (2013). RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization. In: *MICRO*.

[67] **Bhati**, **I.**, **Chishti**, **Z.**, **Lu**, **S.-L.**, and **Jacob**, **B.** (2015). Flexible Auto-Refresh: Enabling Scalable and Energy-Efficient DRAM Refresh Reductions. In: *ISCA*.

[68] **Chang**, **K. K.**, **Yaglikci**, **A.**, **Ghose**, **S.**, **Agrawal**, **A.**, **Chatterjee**, **N.**, **Kashyap**, **A.**, **Lee**, **D.**, **O'Connor**, **M.**, **Hassan**, **H.**, and **Mutlu**, **O.** (2017). Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms. In: *SIGMETRICS*.

[69] **Chang**, **K. K.**, **Kashyap**, **A.**, **Hassan**, **H.**, **Ghose**, **S.**, **Hsieh**, **K.**, **Lee**, **D.**, **Li**, **T.**, **Pekhimenko**, **G.**, **Khan**, **S.**, and **Mutlu**, **O.** (2016). Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization. In: *SIGMETRICS*.

[70] **Chang**, **K. K.**, **Lee**, **D.**, **Chishti**, **Z.**, **Alameldeen**, **A. R.**, **Wilkerson**, **C.**, **Kim**, **Y.**, and **Mutlu**, **O.** (2014). Improving DRAM Performance by Parallelizing Refreshes with Accesses. In: *HPCA*.

[71] **Chang**, **K. K.**, **Nair**, **P. J.**, **Lee**, **D.**, **Ghose**, **S.**, **Qureshi**, **M. K.**, and **Mutlu**, **O.** (2016). Low-cost Inter-linked Subarrays (LISA): Enabling Fast Inter-subarray Data Movement in DRAM. In: *HPCA*.

[72] **Chang**, **K. K.**, **Yağlikçi**, **A. G.**, **Ghose**, **S.**, **Agrawal**, **A.**, **Chatterjee**, **N.**, **Kashyap**, **A.**, **Lee**, **D.**, **O'Connor**, **M.**, **Hassan**, **H.**, and **Mutlu**, **O.** (2017). Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms. In: *SIGMETRICS*.

[73] **Ghose**, **S.**, **Yaglikci**, **G.**, **Gupta**, **R.**, **Lee**, **D.**, **Kudrolli**, **K.**, **Liu**, **W.**, **Hassan**, **H.**, **Chang**, **K.**, **Chatterjee**, **N.**, **Agrawal**, **A.**, **O'Connor**, **M.**, and **Mutlu**, **O.** (2018). What Your DRAM Power Models Are Not Telling You: Lessons from a Detailed Experimental Study. In: *SIGMETRICS*.

[74] **Hassan**, **H.**, **Pekhimenko**, **G.**, **Vijaykumar**, **N.**, **Seshadri**, **V.**, **Lee**, **D.**, **Ergin**, **O.**, and **Mutlu**, **O.** (2016). ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality. In: *HPCA*.

[75] **Hassan**, **H.**, **Vijaykumar**, **N.**, **Khan**, **S.**, **Ghose**, **S.**, **Chang**, **K.**, **Pekhimenko**, **G.**, **Lee**, **D.**, **Ergin**, **O.**, and **Mutlu**, **O.** (2017). SoftMC: A Flexible and Practical Open-source Infrastructure for Enabling Experimental DRAM Studies. In: *HPCA*.

[76] **Khan**, **S.**, **Lee**, **D.**, **Kim**, **Y.**, **Alameldeen**, **A. R.**, **Wilkerson**, **C.**, and **Mutlu**, **O.** (2014). The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study. In: *SIGMETRICS*.

[77] **Khan**, **S.**, **Lee**, **D.**, and **Mutlu**, **O.** (2016). PARBOR: An Efficient System-Level Technique to Detect Data-Dependent Failures in DRAM. In: *DSN*.

[78] **Khan**, **S.**, **Wilkerson**, **C.**, **Wang**, **Z.**, **Alameldeen**, **A. R.**, **Lee**, **D.**, and **Mutlu**, **O.** (2017). Detecting and Mitigating Data-Dependent DRAM Failures by Exploiting Current Memory Content. In: *MICRO*.

[79] **Kim**, **J. S.**, **Patel**, **M.**, **Hassan**, **H.**, and **Mutlu**, **O.** (2018a). Solar-DRAM: Reducing DRAM Access Latency by Exploiting the Variation in Local Bitlines. In: *ICCD*.

[80] **Kim**, **J. S.**, **Patel**, **M.**, **Hassan**, **H.**, and **Mutlu**, **O.** (2018b). The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency-Reliability Tradeoff in Modern Commodity DRAM Devices. In: *HPCA*.

[81] **Kim**, **Y.**, **Daly**, **R.**, **Kim**, **J.**, **Fallin**, **C.**, **Lee**, **J. H.**, **Lee**, **D.**, **Wilkerson**, **C.**, **Lai**, **K.**, and **Mutlu**, **O.** (2014). Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In: *ISCA*.

[82] **Kim**, **Y.**, **Yang**, **W.**, and **Mutlu**, **O.** (2016). Ramulator: A Fast and Extensible DRAM Simulator. In: *CAL*.

[83] **Lee**, **D.** (2016). Reducing DRAM Latency at Low Cost by Exploiting Heterogeneity. PhD thesis. Carnegie Mellon University.

[84] **Lee**, **D.**, **Khan**, **S.**, **Subramanian**, **L.**, **Ghose**, **S.**, **Ausavarungnirun**, **R.**, **Pekhimenko**, **G.**, **Seshadri**, **V.**, and **Mutlu**, **O.** (2017). Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms. In: *SIGMETRICS*.

[85] **Lee**, **D.**, **Subramanian**, **L.**, **Ausavarungnirun**, **R.**, **Choi**, **J.**, and **Mutlu**, **O.** (2015). Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM. In: *PACT*.

[86] **Liu**, **J.**, **Jaiyen**, **B.**, **Kim**, **Y.**, **Wilkerson**, **C.**, and **Mutlu**, **O.** (2013). An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms. In: *ISCA*.

[87] **Liu**, **J.**, **Jaiyen**, **B.**, **Veras**, **R.**, and **Mutlu**, **O.** (2012). RAIDR: Retention-Aware Intelligent DRAM Refresh. In: *ISCA*.

[88] **Mukundan**, **J.**, **Hunter**, **H.**, **Kim**, **K.-h.**, **Stuecheli**, **J.**, and **Martínez**, **J. F.** (2013). Understanding and Mitigating Refresh Overheads in High-density DDR4 DRAM Systems. In: *ISCA*.

[89] **Patel**, **M.**, **Kim**, **J. S.**, and **Mutlu**, **O.** (2017). The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions. In: *ISCA*.

[90] **Qureshi**, **M. K.**, **Kim**, **D.**, **Khan**, **S.**, **Nair**, **P. J.**, and **Mutlu**, **O.** (2015). AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems. In: *DSN*.

[91] **Seshadri**, **V.** (2016). Simple DRAM and Virtual Memory Abstractions to Enable Highly Efficient Memory Systems. PhD thesis. Carnegie Mellon University.

[92] **Seshadri**, **V.**, **Lee**, **D.**, **Mullins**, **T.**, **Hassan**, **H.**, **Boroumand**, **A.**, **Kim**, **J.**, **Kozuch**, **M. A.**, **Mutlu**, **O.**, **Gibbons**, **P. B.**, and **Mowry**, **T. C.** (2017). Ambit: In-memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology. In: *MICRO*.

[93] **Zhang**, **T.**, **Chen**, **K.**, **Xu**, **C.**, **Sun**, **G.**, **Wang**, **T.**, and **Xie**, **Y.** (2014). Half-DRAM: A High-bandwidth and Low-power DRAM Architecture from the Rethinking of Fine-grained Activation. In: *ISCA*.

[94] **Rixner**, **S.**, **Dally**, **W. J.**, **Kapasi**, **U. J.**, **Mattson**, **P.**, and **Owens**, **J. D.** (2000). Memory Access Scheduling. In: *ISCA*.

[95] **Zuravleff**, **W. K.** and **Robinson**, **T.** (1997). Controller for a Synchronous DRAM that Maximizes Throughput by Allowing Memory Requests and Commands to be Issued Out of Order. US Patent 5,630,096.

[96] **Moscibroda**, **T.** and **Mutlu**, **O.** (2007). Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems. In: *USENIX Security*.

[97] **Mutlu**, **O.** and **Moscibroda**, **T.** (2007). Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In: *MICRO*.

[98] **Mutlu**, **O.** and **Moscibroda**, **T.** (2008). Parallelism-Aware Batch Scheduling: Enabling High-performance And Fair Shared Memory Controllers. In: *ISCA*.

[99] **Kim**, **Y.**, **Han**, **D.**, **Mutlu**, **O.**, and **Harchol-Balter**, **M.** (2010). ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers. In: *HPCA*.

[100] **Kim**, **Y.**, **Papamichael**, **M.**, **Mutlu**, **O.**, and **Harchol-Balter**, **M.** (2010). Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In: *MICRO*.

[101] **Ghose**, **S.**, **Lee**, **H.**, and **Martínez**, **J. F.** (2013). Improving Memory Scheduling via Processor-Side Load Criticality Information. In: *ISCA*.

[102] **Ebrahimi**, **E.**, **Miftakhutdinov**, **R.**, **Fallin**, **C.**, **Lee**, **C. J.**, **Joao**, **J. A.**, **Mutlu**, **O.**, and **Patt**, **Y. N.** (2011). Parallel Application Memory Scheduling. In: *MICRO*, pp. 362–373.

[103] **Subramanian**, **L.**, **Lee**, **D.**, **Seshadri**, **V.**, **Rastogi**, **H.**, and **Mutlu**, **O.** (2014). The Blacklisting Memory Scheduler: Achieving High Performance And Fairness At Low Cost. In: *ICCD*.

[104] **Nesbit**, **K.**, **Aggarwal**, **N.**, **Laudon**, **J.**, and **Smith**, **J.** (2006). Fair Queuing Memory Systems. In: *MICRO*.

[105] **Rafique**, **N.**, **Lim**, **W.-T.**, and **Thottethodi**, **M.** (2007). Effective Management of DRAM Bandwidth in Multicore Processors. In: *PACT*.

[106] **Usui**, **H.**, **Subramanian**, **L.**, **Chang**, **K. K.**, and **Mutlu**, **O.** (2016). DASH: Deadline-Aware High-performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators. In: *TACO*.

[107] **Ausavarungnirun**, **R.**, **Chang**, **K. K.-W.**, **Subramanian**, **L.**, **Loh**, **G. H.**, and **Mutlu**, **O.** (2012). Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems. In: *ISCA*.

[108] **Subramanian**, **L.**, **Lee**, **D.**, **Seshadri**, **V.**, **Rastogi**, **H.**, and **Mutlu**, **O.** (2016). BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling. In: *TPDS*.

[109] **Halderman**, **J. A.**, **Schoen**, **S. D.**, **Heninger**, **N.**, **Clarkson**, **W.**, **Paul**, **W.**, **Calandrino**, **J. A.**, **Feldman**, **A. J.**, **Appelbaum**, **J.**, and

Felten, **E. W.** (2009). Lest We Remember: Cold-Boot Attacks on Encryption Keys. In: *Commun. ACM*.

[110] **Venkatesan**, **R. K.**, **Herr**, **S.**, and **Rotenberg**, **E.** (2006). Retention-aware Placement in DRAM (RAPID): Software Methods for Quasi-non-volatile DRAM. In: *HPCA*.

[111] **JEDEC** (2012). *Double Data Rate 3 (DDR3) SDRAM Specification*.

[112] **Mutlu**, **O.**, **Ghose**, **S.**, **Gómez-Luna**, **J.**, and **Ausavarungnirun**, **R.** (2020). A Modern Primer on Processing in Memory. arXiv: 2012.03112 [cs.AR].

[113] **Ghose**, **S.**, **Boroumand**, **A.**, **Kim**, **J. S.**, **Gómez-Luna**, **J.**, and **Mutlu**, **O.** (2019). Processing-in-memory: A workload-driven perspective. In: *IBM JRD*.

[114] **Mutlu**, **O.**, **Ghose**, **S.**, **Gómez-Luna**, **J.**, and **Ausavarungnirun**, **R.** (2019). Processing Data Where it Makes Sense: Enabling In-Memory Computation. In: *Microprocessors and Microsystems*.

[115] **Singh**, **G.**, **Alser**, **M.**, **Cali**, **D. S.**, **Diamantopoulos**, **D.**, **Gomez-Luna**, **J.**, **Corporaal**, **H.**, and **Mutlu**, **O.** (2021). FPGA-Based Near-Memory Acceleration of Modern Data-Intensive Applications. In: *IEEE Micro*.

[116] **Ghose**, **S.**, **Hsieh**, **K.**, **Boroumand**, **A.**, **Ausavarungnirun**, **R.**, and **Mutlu**, **O.** (2018). Enabling the Adoption of Processing-in-Memory: Challenges, Mechanisms, Future Research Directions. In: *arXiv preprint arXiv:1802.00320*.

[117] **Wang**, **Y.**, **Orosa**, **L.**, **Peng**, **X.**, **Guo**, **Y.**, **Ghose**, **S.**, **Patel**, **M.**, **Kim**, **J. S.**, **Luna**, **J. G.**, **Sadrosadati**, **M.**, **Ghiasi**, **N. M.**, and **Mutlu**, **O.** (2020). FIGARO: Improving System Performance via Fine-Grained In-DRAM Data Relocation and Caching. In: *MICRO*.

[118] **Giannoula**, **C.**, **Vijaykumar**, **N.**, **Papadopoulou**, **N.**, **Karakostas**, **V.**, **Fernandez**, **I.**, **Gómez-Luna**, **J.**, **Orosa**, **L.**, **Koziris**, **N.**, **Goumas**, **G.**, and **Mutlu**, **O.** (2021). SynCron: Efficient Synchronization Support for Near-Data-Processing Architectures. In: *HPCA*.

[119] **Akin**, **B.**, **Franchetti**, **F.**, and **Hoe**, **J. C.** (2015). Data Reorganization in Memory Using 3D-Stacked DRAM. In: *ISCA*.

[120] **Aga**, **S.**, **Jeloka**, **S.**, **Subramaniyan**, **A.**, **Narayanasamy**, **S.**, **Blaauw**, **D.**, and **Das**, **R.** (2017). Compute Caches. In: *HPCA*.

[121] **Ahn**, **J.**, **Hong**, **S.**, **Yoo**, **S.**, **Mutlu**, **O.**, and **Choi**, **K.** (2015). A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. In: *ISCA*.

[122] **Ahn**, **J.**, **Yoo**, **S.**, **Mutlu**, **O.**, and **Choi**, **K.** (2015). PIM-Enabled Instructions: a Low-overhead, Locality-aware Processing-in-Memory Architecture. In: *ISCA*.

[123] **Lee**, **D.**, **Ghose**, **S.**, **Pekhimenko**, **G.**, **Khan**, **S.**, and **Mutlu**, **O.** (2016). Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost. In: *TACO*.

[124] **Seshadri**, **V.**, **Hsieh**, **K.**, **Boroum**, **A.**, **Lee**, **D.**, **Kozuch**, **M. A.**, **Mutlu**, **O.**, **Gibbons**, **P. B.**, and **Mowry**, **T. C.** (2015). Fast Bulk Bitwise AND and OR in DRAM. In: *IEEE CAL*.

[125] **Seshadri**, **V.**, **Mullins**, **T.**, **Boroumand**, **A.**, **Mutlu**, **O.**, **Gibbons**, **P. B.**, **Kozuch**, **M. A.**, and **Mowry**, **T. C.** (2015). Gather-scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-unit Strided Accesses. In: *MICRO*.

[126] **Liu**, **Z.**, **Calciu**, **I.**, **Herlihy**, **M.**, and **Mutlu**, **O.** (2017). Concurrent Data Structures for Near-Memory Computing. In: *SPAA*.

[127] **Seshadri**, **V.** and **Mutlu**, **O.** (2017). Simple Operations in Memory to Reduce Data Movement. In: *Advances in Computers*.

[128] **Pattnaik**, **A.**, **Tang**, **X.**, **Jog**, **A.**, **Kayiran**, **O.**, **Mishra**, **A. K.**, **Kandemir**, **M. T.**, **Mutlu**, **O.**, and **Das**, **C. R.** (2016). Scheduling Techniques for GPU Architectures with Processing-in-Memory Capabilities. In: *PACT*.

[129] **Babarinsa**, **O. O.** and **Idreos**, **S.** (2015). JAFAR: Near-Data Processing for Databases. In: *SIGMOD*.

[130] **Farmahini-Farahani**, **A.**, **Ahn**, **J. H.**, **Morrow**, **K.**, and **Kim**, **N. S.** (2015). NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules. In: *HPCA*.

[131] **Gao**, **M.**, **Ayers**, **G.**, and **Kozyrakis**, **C.** (2015). Practical Near-Data Processing for In-Memory Analytics Frameworks. In: *PACT*.

[132] **Gao**, **M.** and **Kozyrakis**, **C.** (2016). HRL: Efficient and Flexible Reconfigurable Logic for Near-Data Processing. In: *HPCA*.

[133] **Hassan**, **S. M.**, **Yalamanchili**, **S.**, and **Mukhopadhyay**, **S.** (2015). Near Data Processing: Impact and Optimization of 3D Memory System Architecture on the Uncore. In: *MEMSYS*.

[134] **Hsieh**, **K.**, **Ebrahimi**, **E.**, **Kim**, **G.**, **Chatterjee**, **N.**, **O'Connor**, **M.**, **Vijaykumar**, **N.**, **Mutlu**, **O.**, and **Keckler**, **S. W.** (2016). Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems. In: *ISCA*.

[135] **Morad**, **A.**, **Yavits**, **L.**, and **Ginosar**, **R.** (2015). GP-SIMD Processing-in-Memory. In: *TACO*.

[136] **Sura**, **Z.**, **Jacob**, **A.**, **Chen**, **T.**, **Rosenburg**, **B.**, **Sallenave**, **O.**, **Bertolli**, **C.**, **Antao**, **S.**, **Brunheroto**, **J.**, **Park**, **Y.**, **O'Brien**, **K.**, et al. (2015). Data Access Optimization in a Processing-in-Memory System. In: *CF*.

[137] **Zhang**, **D.**, **Jayasena**, **N.**, **Lyashevsky**, **A.**, **Greathouse**, **J. L.**, **Xu**, **L.**, and **Ignatowski**, **M.** (2014). TOP-PIM: Throughput-Oriented Programmable Processing in Memory. In: *HPDC*.

[138] **Hsieh**, **K.**, **Khan**, **S.**, **Vijaykumar**, **N.**, **Chang**, **K. K.**, **Boroumand**, **A.**, **Ghose**, **S.**, and **Mutlu**, **O.** (2016). Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation. In: *ICCD*.

[139] **Boroumand**, **A.**, **Ghose**, **S.**, **Lucia**, **B.**, **Hsieh**, **K.**, **Malladi**, **K.**, **Zheng**, **H.**, and **Mutlu**, **O.** (2017). LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory. In: *CAL*.

[140] **Kim**, **J. S.**, **Cali**, **D. S.**, **Xin**, **H.**, **Lee**, **D.**, **Ghose**, **S.**, **Alser**, **M.**, **Hassan**, **H.**, **Ergin**, **O.**, **Alkan**, **C.**, and **Mutlu**, **O.** (2018). GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-memory Technologies. In: *BMC Genomics*.

[141] **Boroumand**, **A.**, **Ghose**, **S.**, **Akin**, **B.**, **Narayanaswami**, **R.**, **Oliveira**, **G. F.**, **Ma**, **X.**, **Shiu**, **E.**, and **Mutlu**, **O.** (2021). Mitigating Edge Machine Learning Inference Bottlenecks: An Empirical Study on Accelerating Google Edge Models. arXiv:2103.00768.

[142] **Boroumand**, **A.**, **Ghose**, **S.**, **Kim**, **Y.**, **Ausavarungnirun**, **R.**, **Shiu**, **E.**, **Thakur**, **R.**, **Kim**, **D.**, **Kuusela**, **A.**, **Knies**, **A.**, **Ranganathan**, **P.**, and **Mutlu**, **O.** (2018). Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks. In: *ASPLOS*.

[143] **Hassan**, **H.**, **Patel**, **M.**, **Kim**, **J. S.**, **Yaglikci**, **A. G.**, **Vijaykumar**, **N.**, **Ghiasi**, **N. M.**, **Ghose**, **S.**, and **Mutlu**, **O.** (2019). CROW: A Low-Cost Substrate for Improving DRAM Performance, Energy Efficiency, and Reliability. In: *ISCA*.

[144] **Singh**, **G.**, **Diamantopoulos**, **D.**, **Hagleitner**, **C.**, **Gomez-Luna**, **J.**, **Stuijk**, **S.**, **Mutlu**, **O.**, and **Corporaal**, **H.** (2020). NERO: A Near High-Bandwidth Memory Stencil Accelerator for Weather Prediction Modeling. In: *FPL*.

[145] **Fernandez**, **I.**, **Quislant**, **R.**, **Giannoula**, **C.**, **Alser**, **M.**, **Gómez-Luna**, **J.**, **Gutiérrez**, **E.**, **Plata**, **O.**, and **Mutlu**, **O.** (2020). NATSA: A Near-Data Processing Accelerator for Time Series Analysis.

[146] **Kwon**, **Y.-C.**, **Lee**, **S. H.**, **Lee**, **J.**, **Kwon**, **S.-H.**, **Ryu**, **J. M.**, **Son**, **J.-P.**, **Seongil**, **O**, **Yu**, **H.-S.**, **Lee**, **H.**, **Kim**, **S. Y.**, et al. (2021). 25.4 A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 with a 1.2 TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications. In: *ISSCC*.

[147] **Devaux**, **F.** (2019). The True Processing in Memory Accelerator. In: *HCS*.

[148] **Li**, **S.**, **Xu**, **C.**, **Zou**, **Q.**, **Zhao**, **J.**, **Lu**, **Y.**, and **Xie**, **Y.** (2016). Pinatubo: A Processing-in-Memory Architecture for Bulk Bitwise Operations in Emerging Non-Volatile Memories. In: *DAC*.

[149] **Chi**, **P.**, **Li**, **S.**, **Xu**, **C.**, **Zhang**, **T.**, **Zhao**, **J.**, **Liu**, **Y.**, **Wang**, **Y.**, and **Xie**, **Y.** (2016). PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. In: *ISCA*.

[150] **Orosa**, **L.**, **Wang**, **Y.**, **Puddu**, **I.**, **Sadrosadati**, **M.**, **Razavi**, **K.**, **Gómez-Luna**, **J.**, **Hassan**, **H.**, **Mansouri-Ghiasi**, **N.**, **Tavakkol**, **A.**, **Patel**, **M.**, **Kim**, **J.**, **Seshadri**, **V.**, **Kang**, **U.**, **Ghose**, **S.**, **Azevedo**, **R.**, and **Mutlu**, **O.** (2019). Dataplant: Enhancing System Security with Low-Cost In-DRAM Value Generation Primitives. arXiv:1902.07344.

[151] **Orosa**, **L.**, **Wang**, **Y.**, **Puddu**, **I.**, **Sadrosadati**, **M.**, **Razavi**, **K.**, **Gómez-Luna**, **J.**, **Hassan**, **H.**, **Mansouri-Ghiasi**, **N.**, **Tavakkol**, **A.**, **Patel**, **M.**, **Kim**, **J.**, **Seshadri**, **V.**, **Kang**, **U.**, **Ghose**, **S.**, **Azevedo**, **R.**, and

Mutlu, **O.** (2021). CODIC: A Low-cost Substrate for Enabling Custom In-DRAM Functionalities and Optimizations. In: *ISCA*.

[152] "Standard Performance Evaluation Corporation" (n.d.). http://www.spec.org/cpu2006.

[153] **Transaction Processing Performance Council** (n.d.). TPC Benchmarks. `http://tpc.org/`.

[154] **Fritts**, **J. E.**, **Steiling**, **F. W.**, **Tucek**, **J. A.**, and **Wolf**, **W.** (2009). MediaBench II Video: Expediting the next Generation of Video Systems Research. In: *Microprocess. Microsyst.*

[155] **Cooper**, **B.**, **Silberstein**, **A.**, **Tam**, **E.**, **Ramakrishnan**, **R.**, and **Sears**, **R.** (2010). Benchmarking Cloud Serving Systems with YCSB. In: *SoCC*.

[156] **Gabor**, **R.**, **Weiss**, **S.**, and **Mendelson**, **A.** (2006). Fairness and Throughput in Switch on Event Multithreading. In: *MICRO*.

[157] **Wolrich**, **G.**, **Bernstein**, **D.**, **Cutter**, **D.**, **Dolan**, **C.**, and **Adiletta**, **M. J.** (2004). Mapping Requests from a Processing Unit That Uses Memory-Mapped Input-Output Space. US Patent 6,694,380.

[158] **Lee**, **H.**, **Tyson**, **G.**, and **Farrens**, **M.** (2000). Eager Writeback - A Technique for Improving Bandwidth Utilization. In: *MICRO*.

[159] **David**, **H.**, **Fallin**, **C.**, **Gorbatov**, **E.**, **Hanebutte**, **U. R.**, and **Mutlu**, **O.** (2011). Memory Power Management via Dynamic Voltage/Frequency Scaling. In: *ICAC*.

[160] **Lee**, **C. J.**, **Narasiman**, **V.**, **Ebrahimi**, **E.**, **Mutlu**, **O.**, and **Patt**, **Y. N.** (2010). DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems. In: *HPS Technical Report*.

[161] **Stuecheli**, **J.**, **Kaseridis**, **D.**, **Daly**, **D.**, **Hunter**, **H. C.**, and **John**, **L. K.** (2010). The Virtual Write Queue: Coordinating DRAM and Last-Level Cache Policies. In: *SIGARCH Comput. Archit. News*.

[162] **Zouzias**, **A.**, **Kalaitzidis**, **K.**, and **Grot**, **B.** (2021). Branch Prediction as a Reinforcement Learning Problem: Why, How and Case Studies. arXiv:2106.13429.

[163] **Ipek**, **E.**, **Mutlu**, **O.**, **Martínez**, **J. F.**, and **Caruana**, **R.** (2008). Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In: *ISCA*.

[164] **Mukundan**, **J.** and **Martinez**, **J. F.** (2012). MORSE: Multi-objective reconfigurable self-optimizing memory scheduler. In: *HPCA*.

[165] **Peled**, **L.**, **Mannor**, **S.**, **Weiser**, **U.**, and **Etsion**, **Y.** (2015). Semantic Locality and Context-Based Prefetching Using Reinforcement Learning. In: *ISCA*.

[166] **Bera**, **R.**, **Kanellopoulos**, **K.**, **Nori**, **A.**, **Shahroodi**, **T.**, **Subramoney**, **S.**, and **Mutlu**, **O.** (2021). Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning. In: *MICRO*.

[167] **Dinakarrao**, **S. M. P.**, **Yu**, **H.**, **Huang**, **H.**, and **Xu**, **D.** (2016). A Q-Learning Based Self-Adaptive I/O Communication for 2.5D Integrated Many-Core Microprocessor and Memory. In: *IEEE Transactions on Computers* 65, pp. 1185–1196.

[168] **Watkins**, **C. J. C. H.** and **Dayan**, **P.** (1992). Q-learning. In: *Machine Learning*.

[169] **Kocher**, **P. C.** (1996). Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: *Annual International Cryptology Conference*.

[170] **Kim**, **T.**, **Peinado**, **M.**, and **Mainar-Ruiz**, **G.** (2012). STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In: *USENIX Security*.

[171] **Wang**, **Z.** and **Lee**, **R. B.** (2007). New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In: *ISCA*.

[172] **Lampson**, **B. W.** (1973). A Note on the Confinement Problem. In: *CACM*.

[173] **Ristenpart**, **T.**, **Tromer**, **E.**, **Shacham**, **H.**, and **Savage**, **S.** (2009). Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In: *CCS*.

[174] **Percival**, **C.** (2005). Cache missing for fun and profit. In: *BSDCan Ottawa*.

[175] **Maurice**, **C.**, **Neumann**, **C.**, **Heen**, **O.**, and **Francillon**, **A.** (2015). C5: Cross-Cores Cache Covert Channel. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*.

[176] **Wu**, **Z.**, **Xu**, **Z.**, and **Wang**, **H.** (2014). Whispers in the Hyper-Space: High-Bandwidth and Reliable Covert Channel Attacks Inside the Cloud. In: *Transactions on Networking*.

[177] **Xu**, **Y.**, **Bailey**, **M.**, **Jahanian**, **F.**, **Joshi**, **K.**, **Hiltunen**, **M.**, and **Schlichting**, **R.** (2011). An Exploration of L2 Cache Covert Channels in Virtualized Environments. In: *CCSW*.

[178] **Liu**, **F.**, **Yarom**, **Y.**, **Ge**, **Q.**, **Heiser**, **G.**, and **Lee**, **R. B.** (2015). Last-Level Cache Side-Channel Attacks are Practical. In: *SP*.

[179] **Hunger**, **C.**, **Kazdagli**, **M.**, **Rawat**, **A.**, **Dimakis**, **A.**, **Vishwanath**, **S.**, and **Tiwari**, **M.** (2015). Understanding Contention-based Channels and Using Them for Defense. In: *HPCA*.

[180] **Martin**, **R.**, **Demme**, **J.**, and **Sethumadhavan**, **S.** (2012). Timewarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-Channel Attacks. In: *ISCA*.

[181] **Hu**, **W.-M.** (1992). Reducing timing channels with fuzzy time. In: *Journal of Computer Security*.

[182] **Liu**, **F.**, **Ge**, **Q.**, **Yarom**, **Y.**, **Mckeen**, **F.**, **Rozas**, **C.**, **Heiser**, **G.**, and **Lee**, **R. B.** (2016). Catalyst: Defeating Last-level Cache Side Channel Attacks in Cloud Computing. In: *HPCA*.

[183] **Wang**, **Y.**, **Ferraiuolo**, **A.**, **Zhang**, **D.**, **Myers**, **A. C.**, and **Suh**, **G. E.** (2016). SecDCP: Secure Dynamic Cache Partitioning for Efficient Timing Channel Protection. In: *DAC*.

[184] **Molnar**, **I.** (2007). Modular Scheduler Core and Completely Fair Scheduler. `https://lwn.net/Articles/230501`.

[185] Ramulator Source Code (n.d.). https://github.com/CMU-SAFARI/ramulator.

[186] **Hamerly**, **G.**, **Perelman**, **E.**, **Lau**, **J.**, and **Calder**, **B.** (2005). SimPoint 3.0: Faster and More Flexible Program Phase Analysis. In: *Journal of Instruction-Level Parallelism*.

[187] **Eyerman**, **S.** and **Eeckhout**, **L.** (2008). System-level Performance Metrics for Multiprogram Workloads. In: *IEEE Micro*.

[188] **Snavely**, **A.** and **Tullsen**, **D. M.** (2000). Symbiotic Jobscheduling for a Simultaneous Mutlithreading Processor. In: *ASPLOS*.

[189] **Chandrasekar**, **K.**, **Weis**, **C.**, **Li**, **Y.**, **Goossens**, **S.**, **Jung**, **M.**, **Naji**, **O.**, **Akesson**, **B.**, **Wehn**, **N.**, and **Goossens**, **K.** (n.d.). DRAMPower: Open-Source DRAM Power & Energy Estimation Tool. `http://www.drampower.info/`.

[190] **Muralimanohar**, **N.**, **Balasubramonian**, **R.**, and **Jouppi**, **N. P.** (2009). *CACTI 6.0: A Tool to Model Large Caches*. Tech. rep. HPL-2009-85. HP Laboratories.

[191] **Zhao**, **J.**, **Mutlu**, **O.**, and **Xie**, **Y.** (2014). FIRM: Fair and High-Performance Memory Control for Persistent Memory Systems. In: *MICRO*.

[192] **Moscibroda**, **T.** and **Mutlu**, **O.** (2008). Distributed Order Scheduling and Its Application to Multi-Core DRAM Controllers. In: *PODC*. PODC '08. Toronto, Canada: Association for Computing Machinery.

[193] **Hur**, **I.** and **Lin**, **C.** (2004). Adaptive History-Based Memory Schedulers. In: *MICRO*.

[194] **Thomas**, **G.**, **Chandrasekar**, **K.**, **Åkesson**, **B.**, **Juurlink**, **B.**, and **Goossens**, **K.** (2012). A Predictor-Based Power-Saving Policy for DRAM Memories. In: *2012 15th Euromicro Conference on Digital System Design*.

[195] **Wang**, **Z.**, **Khan**, **S. M.**, and **Jiménez**, **D. A.** (2012). Rank Idle Time Prediction Driven Last-Level Cache Writeback. In: *MSPC*. MSPC '12. Beijing, China: Association for Computing Machinery.