TOBB EKONOMİ VE TEKNOLOJİ ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

İKİLİ ARAMA AĞACINDA ARAMA İŞLEMLERİNİN DONANIM KULLANILARAK PARALEL OLARAK HIZLANDIRILMASI

YÜKSEK LİSANS TEZİ

Öykü MELİKOĞLU

Bilgisayar Mühendisliği Anabilim Dalı

Tez Danışmanı: Prof. Dr. Oğuz ERGİN

AĞUSTOS 2019

Fen Bilimleri Enstitüsü Onayı

……..
Prof. Dr. Osman EROĞUL
Müdür

Bu tezin Yüksek Lisans derecesinin tüm gereksinimlerini sağladığını onaylarım.

…….
Prof. Dr. Oğuz ERGİN
Anabilimdalı Başkanı

TOBB ETÜ, Fen Bilimleri Enstitüsü'nün 161111085 numaralı Yüksek Lisans Öğrencisi **Öykü MELİKOĞLU** 'nun ilgili yönetmeliklerin belirlediği gerekli tüm şartları yerine getirdikten sonra hazırladığı **"İKİLİ ARAMA AĞACINDA ARAMA İŞLEMLERİNİN DONANIM KULLANILARAK PARALEL OLARAK HIZLANDIRILMASI"** başlıklı tezi **01.08.2019** tarihinde aşağıda imzaları olan jüri tarafından kabul edilmiştir.

Tez Danışmanı : **Prof. Dr. Oğuz ERGİN** .....
TOBB Ekonomi ve Teknoloji Üniversitesi

Jüri Üyeleri : **Prof. Dr. Mehmet Önder EFE (Başkan)** .....
Hacettepe Üniversitesi

Doç. Dr. Fatma Betül ATALAY SATOĞLU .....
TOBB Ekonomi ve Teknoloji Üniversitesi

TEZ BİLDİRİMİ

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

Öykü MELİKOĞLU

ÖZET

Yüksek Lisans Tezi

İKİLİ ARAMA AĞACINDA ARAMA İŞLEMLERİNİN DONANIM KULLANILARAK PARALEL OLARAK HIZLANDIRILMASI

Öykü MELİKOĞLU

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

Danışman: Prof. Dr. Oğuz ERGİN

Tarih: Ağustos 2019

Alan Programlanabilir Kapı Dizileri üretimden sonra istenen uygulamaya göre birden fazla programlanabilen yarı iletken devrelerdir, performans kaybı yaşamadan aynı anda birden fazla farklı işlemi yürütebilmektedirler. İkili Arama Ağacı verileri sıralı bir şekilde ağaç düzeninde tutan bir veri yapısıdır. Alan Programlanabilir Kapı Dizileri'nin paralellik özelliğini kullanarak İkili Arama Ağacı'nda yapılan arama işlemlerini paralel olarak çalıştırarak ve boru hattı yöntemini kullanarak aramaları hızlandırmak, çevrim başına düşen verimi yükseltmek amaçlanmıştır. Ağaç farklı kesit yöntemleri ile parçalanarak, aynı anda birden fazla anahtarın aranabileceği bir ortam oluşturulmuştur. Ağacın her bir seviyesini farklı bir blok belleğin içine yerleştirilmiş ve ağacın her bir seviyesinin aynı anda aranabilmesi sağlanmış, bu yönteme yatay kesme yöntemi adı verilmiştir. Yatay kesme yönteminden elde edilen çevrim başına verimi arttırmak için çoklama yöntemi önerilmiştir. Çoklama yöntemi ile ağacın kopyaları yaratılarak aynı anda daha fazla anahtarın arama işlemine dahil edilebilmesini sağlanmıştır. Tüm seviyelerin kopyalanmadığı böylece daha az yer gereksinimine sahip olan farklı bir çoklama çeşidi de sunulmuştur. Ağacın ilk

seviyedeki düğümleri yazmaçlara konulmuş böylece blok belleklerin daha verimli kullanılması sağlanmış, çoklama işleminde kopyalanması gereken düğüm sayısı azaltılmıştır. Çoklama işleminin yarattığı alan ihtiyacına neden olmayan fakat ona benzer verim alabileceğimiz bir yöntem olarak hibrit kesme yöntemi önerilmiştir. Hibrit kesme yönteminde blok belleklerdeki port sayısının çevrim başına aranmak istenen anahtar sayısına kıyasla yetersiz olmasından ötürü duraklamalar oluşmakta, diğer yöntemlerde port başına bir anahtar arandığı için bu durum meydana gelmemektedir. Hibrit kesme yönteminde oluşabilecek duraklama sorunu için tamponlar eklenmiş , bu tamponlara anahtar eklenirken kullanılacak doğrudan ve kuyruk yolu eşlemeler açıklanmıştır. Tampona anahtar ekleme yöntemleri içinde kuyruk eşleme yönteminin duraklamaları daha etkili bir şekilde azaltmaktadır fakat daha karmaşık bir yapısı olduğundan ötürü doğrudan eklemeye kıyasla daha fazla yer tutmakta, daha fazla çevrim zamanına ihtiyaç duymaktadır. Çoklama yöntemi ile 8 kat hız yakalanmıştır, yeterli kaynak olmaması durumunda tampon kullanılan hibrit yöntemleri kullanılabilir. Hibrit kesme yönteminde alınan verim çoklama ve yatay kesme yöntemlerinin aksine sabit değildir ve verim en iyi senaryoda çoklama en kötü senaryoda ise yatay kesme yönteminin verimine yakınsamaktadır.

Anahtar Kelimeler: İkili arama ağacı, Alan programlanabilir kapı dizisi, Verim, Boru hattı, Paralel arama, Donanım hızlandırıcısı.

ABSTRACT

Master of Science

ACCELERATED HIGH THROUGHPUT PARALLEL SEARCH ON BINARY SEARCH TREES VIA FIELD PROGRAMMABLE GATE ARRAYS

Öykü MELİKOĞLU

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

Supervisor: Prof. Dr. Oğuz ERGİN

Date: August 2019

Field Programmable Gate Arrays are reprogrammable semiconductors that can be programmed according to the desired application after production, they can execute different instructions at the same time without performance loss. Binary Searh Tree is a data structure that stores the data as a sorted tree order. The aim is to increase the throughput by taking advantage of the Field Programmable Gate Array's parallel and pipeline capable architecture. The tree is split into partitions by different techniques to create an environment that lets more keys to be searched at the same time. The approach where each level of the tree is placed to a different block ram therefore increasing the number of ports that can be used, is called horizontal partitioning. To increase the throughput achieved by horizontal partitioning, duplication method is proposed. Duplication method creates duplicates of the tree to make it possible to have more than two keys searching the same level of the tree. Another duplication approach where some but not all of the tree levels are duplicated is also explained. By storing the first levels of the tree inside of registers instead of block memories, the block memories are fully utilized and the number of duplicated nodes are

decreased. The hybrid approach whose aim is to have similar throughput to duplication however by not duplicating the tree, is proposed. Although hybrid approach can reach the high throughput levels of the duplication approach, due to not having enough ports to be able to fetch the same amount of keys the search may stall. The horizontal and duplicate methods do not have stalls since they have one port for each key to be searched. As a solution to the stalls, buffers were added to hybrid implementations and two ways of mapping the keys to the buffers were suggested: direct and queue based. The queue based approach decreases the stalls more efficiently than the direct approach however since it has a more complex structure it needs more clock period to function and more resources. With the duplication method the throughput has increased 8X, if there is not enough resources the hybrid method can be used. The throughput of the hybrid approach is not constant unlike the other approachs, its throughput converges to duplication in the best case and horizontal partitioning in the worst case.

Keywords: Binary search tree, Field programmable gate array, Throughput, Pipeline, Parallel search, Hardware accelerator

TEŞEKKÜR

Çalışmalarım boyunca tecrübelerinden yararlandığım Adrian Cristal, Osman Ünsal ve Behzad Salami'ye, çalışma sürecimde araştırmamdan ötürü Severo Ochoa programı ile beni konuk araştırmacı olarak davet eden Barcelona Supercomputing Center'a çok teşekkür ederim. Ayrıca, 4 yıl boyunca çalıştığım Kasırga Mikroişlemciler Laboratuvarı'na, TOBB ETÜ'ye ve danışmanım Oğuz Ergin'e bana sağladıkları olanaklardan ötürü teşekkür ederim.

İÇİNDEKİLER

ŞEKİL LİSTESİ

Sayfa

ÇİZELGE LİSTESİ

Sayfa

KISALTMALAR

BST : İkili Arama Ağacı (Binary Search Tree)
FPGA : Alan Programlanabilir Kapı Dizisi (Field Programmable Gate Array)
GPU : Grafik İşlemci Ünitesi (Graphics Processing Unit)
BRAM : Blok Rasgele Erişilebilir Bellek (Block Random Access Memory)
HLS : Yüksek Seviyeli Sentez (High Level Synthesis)
ASIC : Uygulamaya Özgü Tümleşik Devre (Application Specific Integrated Circuit)
OTP : Bir Kez Programlanabilen (One Time Programmable)
CPU : İşlemci (Central Processing Unit)
CLB : Programlanabilir Lojik Blok (Configurable Logic Block)
I/O : Giriş/Çıkış (Input/Output)
LUT : Konfigüre Edilebilir Arama Tabloları (Lookup Table)
CPU : İşlemci (Central Processing Unit)
CBST : Eksiksiz İkili Arama Ağacı (Complete Binary Search Tree)
BSV : Bluespec System Verilog
BSC : Bluespec Derleyici (Bluespec Compiler)
HDL : Donanım Tanımlama Dili (Hardware Description Language)
APU : Hızlandırılmış İşlem Ünitesi (Accelerared Processing Unit)
SIMD : Tek Buyruk Çoklu Veri (Single Instruction Multiple Data)
BPT : İkili Ön Ek Ağacı(Binary Prefix Tree)
TCAM : İçeriği Adreslenebilir Üçlü Bellek (Ternary Content Adressable Memory)
IP : İnternet Protokolü (Internet Protocol)
DST : Dinamik Arama Ağacı (Dynamic Search Tree)

1. GİRİŞ

İkili Arama Ağacı (BST) verileri sıralı bir şekilde ağaç düzeninde tutan bir veri yapısıdır. $\Theta(\log(n))$ zaman karmaşıklığına sahip bu veri yapısı veri tabanları, makine öğrenmesi, dosya sistemleri gibi birçok alanda kullanılmaktadır. BST'lerin başlıca işlemleri ekleme, silme ve bulma işlemleridir.

Alan Programlanabilir Kapı Dizisi (FPGA) üretimden sonra istenen uygulamaya göre birden fazla programlanabilen yarı iletken devrelerdir. FPGA'ler performans kaybı yaşamadan aynı anda birden fazla farklı işlemi yürütebilmektedirler. Paralel olarak işlem yapma imkanı sağladığı ve GPU'lara göre daha az enerji harcadıkları için tercih edilen FPGA'ler güvenlik algoritmaları [1], yapay sinir ağları [2], kontrol sistemleri [3] gibi birden çok alanda kullanılmaktadır.

Ağaç veri yapısının hem oluşturulma aşamasına [4,5] hem de operasyonlarına yönelik hızlandırılma çalışmaları yapılmaktadır. Ağaç hızlandırılmalarına birden çok platformda çalışılmaktadır [6-9]. FPGA'in paralellik özelliğinden [10-17] gibi boruhattı yöntemi kullanılarak daha fazla yararlanılabileceğini düşündük. Arama işlemi paralel olarak çalışma gösterebilir, aynı anda birden fazla arama işlemini çalıştırarak birden çok arama yapabiliriz. Bu çalışmada eksiksiz (complete) BST arama işlemini yüksek throughput amacıyla hızlandırma üzerine çalıştık. Gecikme (Latency) sonuca erişmek için gereken zaman, verim (throughput) ise birim zamanda alınan sonuç sayısı olarak belirtilebilir. Sonsuz anahtar akışı (infinite key stream) sırasında ağaç türü değiştiğinde verim değişmeyecek olmasından ötürü eksiksiz BST üzerinde çalıştık.

FPGA'in on-chip belleği olan BRAM'lere paralel erişimlerden yararlanarak ve arama işlemini botu hattı yöntemi kullanarak gerçekleyerek hızlandırıcımızı oluşturduk. Her bir ağaç seviyesini farklı BRAM'lere koyarak her bir seviyeye aynı anda erişim imkanı sağladık. Trade-offlara göre aralarından uygun olanı

seçilebilecek şekilde Basit, Yatay Kesme, Çoklama, Yazmaç Ekleme, Hibrit Kesme ve Buffer Ekleme olmak üzere birden fazla yöntem ve iyileştirmeler (optimization) sunduk. Yatay kesme yöntemi her bir seviyeyi farklı bir BRAM'e koyarak, seviyelerin aynı anda aranmasına olanak sağlamakta ve boru hattı yöntemini kullanmaktadır. Çoklama yöntemi kullanılan ağacı kopyalayarak daha yüksek throughput sağlarken daha fazla belleğe gereksinim duymaktadır. Kopyalanan veri sayısını azaltmak için farklı seviyeleri farklı sayılarda çoklanarak da kullanılabilir. BST'nin bazı seviyelerini yazmaçlara (register) koyarak, BRAM'leri tam kapasite kullanılmış, daha az kopyalama yapılmıştır. Fakat daha fazla yazmaca ihtiyaç doğmuştur. Yatay kesme yöntemine ve dikey kesme de eklenerek oluşturulan hibrid yöntem, veri kopyalaması yapmadan throughputu arttırırken, çevrim zamanında artışa neden olmaktadır. Duraklamaları (stalling) azaltmak için eklenen bufferlar ise daha fazla saklama alanı ihtiyacına sebebiyet vermektedir.

Aranacak anahtarlar gruplar halinde getirilir. Bu gruplarda kaç tane anahtar olacağı hızlandırıcının tek bir çevrimde (cycle) paralel olarak aranabilecek maksimum anahtar sayısına eşit olacak şekilde belirlenir.

Önerilen yöntem ve iyileştirmeler Bluespec HLS dili ile gerçeklenmiş ve VC709 platformunda farklı anahtar listeleri aranarak incelenmiştir. En optimize yöntemimiz karşılaştırılan hızlandırıcıya [17] kıyasla 8 kat daha yüksek throughputa sahiptir. Buffer boyutu, ağaç sayısı, yazmaç sayısı ve yatay kesit sayıları sabit değildir ve istenildiği şekilde değerler verilebilir. Bu sayede throughput istenildiği şekilde değişkenlik gösterebilir ve güç, frekans ve bellek kapasitesinden feragat edilir ise daha yüksek throughput değerleri elde edilebilir.

1.1 Literatür Araştırması

Ağaç yapıları bir çok alanda kullanılmaktadır. Ağaçlar hem oluşturulma aşamasına [4,5] hem de operasyonlarına yönelik hızlandırılma çalışmaları yapılmaktadır. Ağaç hızlandırılmalarına birden çok platformda çalışılmaktadır. [6] yazılımsal , [7-8] grafik işlemci ünitesi (GPU) kullanarak ve [9] APU kullanarak ağaç hızlandırması yapmışlardır. [18-19] özellikle ikili arama ağaçlarını baz almıştır.

Ağaç operasyonları yıllardır hem yazılımsal hem de donanımsal olarak hızlandırılmaya çalışılmaktadır. Ağaçların birden çok parçaya bölünmesi fikri ise yıllardır vardır, bu fikri gerçekleştirmede kullanılan ortamlar değişiklik göstermiştir.

[20] ağaçların her bir düğümünün farklı bir işlemciye eşlenmesinin üzerinde durmaktadır. [21-23] çalışmalarında ağaçlar yatay olarak seviyelere bölünmüş ve ağacın her bir seviyesi farklı bir işlemciye eşlenmiştir. [24] ise tek bir işlemcide hafızayı farklı banklere bölmüş, her bir bank içerisinde istenen veriyi alan bir mandal (latch) olacak şekilde ayarlanmıştır ve boruhattı yöntemi ile arama fonksiyonları çalıştırılmaktadır. Bir arama işlemi istenildiğinde paralel olarak ağacın yaprak düğümlerine bakılmakta, her çevrimde doğru veri köke doğru ilerlemekte ve işlemci bu veriyi okumaktadır.

[25,26] önceki çalışmalar gibi veriyi farklı işlemcilerin hafızalarına yerleştirmek yerine, işlemcileri ağacın farklı alanlarında arama yapacak şekilde bölüştürmüştür. Bir işlemci baktığı alanda aramayı bitirdiği zaman boşta durumuna geçmektedir. Boşta olan işlemciler yeniden yapılan iş bölümleri ile, çalışmakta olan diğer işlemcilerin yükünü bölüşmektedir. İstenilen sonuç bulunduğunda tüm işlemciler durmaktadır.

[27] ağacın büyüklüğünden ötürü farklı logical sayfalarda, farklı veriler olacağına işaret etmiş ve aynı anda farklı logical sayfaları aramayı önermiştir.

[28] SIMD kullanarak hızlandırma yapmış, [29] ise CPU, GPU, SIMD birleştirerek hızlandırma işlemini yapmıştır. [30] BPT (Binary Prefix Tree) için TCAM kullanılarak hızlandırılma yapılmıştır.

FPGA alanında yapılan yatay bölme işlemlerinde, bu çalışmada olduğu gibi her bir ağaç seviyesi farklı bir BRAM'e eşlenmiştir. [10,11] paket sınıflandırmak için kullandıkları dörtlü ağaç (quadtree) yapısında, [12-14] decision tree kullanımında, [15,16] IP lookup enginelerinde ve [17] DST lerde bu eşlemeyi uygulamıştır.

Daha önce yapılan ağaç hızlandırma çalışmalarında yatay kesme yöntemi görülmüş olsa da, dikey kesme yönteminin uygulandığı bir çalışma literatürde bulunamamıştır. Tezin 2. bölümünde çalışmaya dair teknik arka plan verilmiştir. 3. bölümde arama operasyonunun FPGA'in BRAM'leri aracılığıyla paralel ve boru hattı yöntemiyle

hızlandırılarak yüksek throughput elde edilen yöntemler sunulmuştur. BRAM'lerin kapasitelerinin tam kullanılması ve kopyalamaları azaltmak amaçlı, throughputu azaltmayacak yazmaç kullanımı açıklanmıştır. BRAM'lerin bandwith kullanımını en yüksek seviyeye getirebilmek için duraklamaları azaltan tampon iyileştirmesi önerilmiştir. Bu iyileştirme için doğrudan eşleme ve kuyruk yoluyla eşleme başlıkları altında iki farklı yöntem gösterilmiştir. Elde edilen test sonuçları 4. bölümde incelenmesine ayrılmıştır. Son bölümde ise çalışmanın genel değerlendirmesi yapılmıştır.

2. TEKNİK ARKA PLAN

2.1 FPGA

FPGA'ler ASIC'ler gibi parallel işlemlere olanak sağlamalarına rağmen, ASIC'lerin aksine uygulamalara özel olarak üretilmeyen ve üretimden sonra istenen fonksiyonlara göre programlanabilen yarı iletken entegre devrelerdir. Sadece OTP türü FPGA'ler yeniden programlanamamaktadır [31].

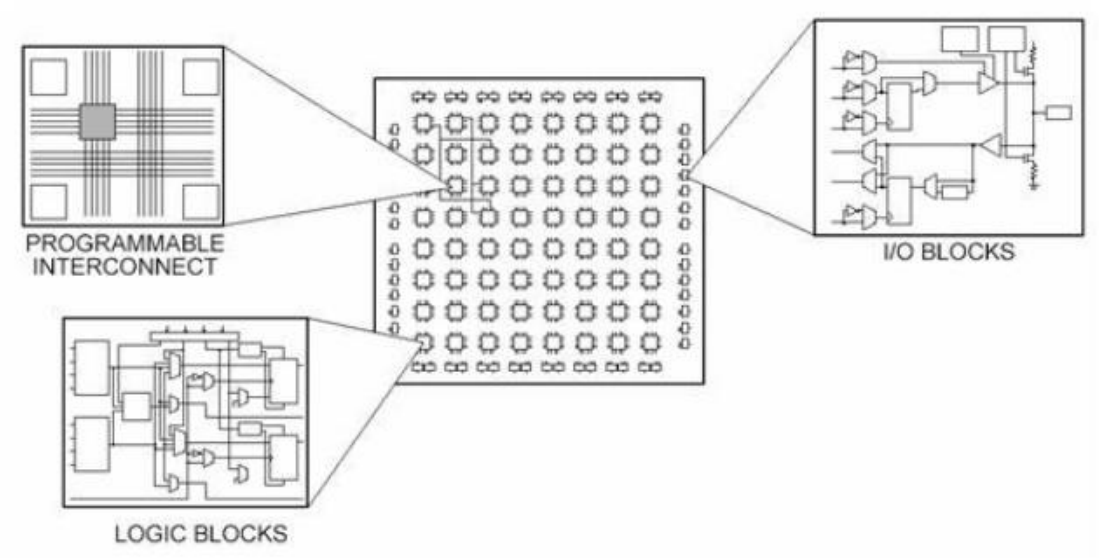
CPU'ların aksine FPGA'ler paralel olarak farklı işlemleri yürütebilirler. Her bağımsız işlem, yonganın özel bir bölümüne atandığı için diğer mantık bloklarına etki etmeden çalışabilir. Dolayısı ile farklı işlemler aynı kaynağa sahip olmaya çalışmaz ve yeni işlemler eklendiği zaman uygulamanın diğer bölümlerinin performansına bir etkide bulunulmaz [32].

CPU'lara kıyasla yüksek throughputa ve düşük enerji tüketimine sahip olmalarından dolayı donanım hızlandırıcıları olarak GPU, FPGA ve ASIC kullanımı tercih edilmektedir. Donanım hızlandırıcıları güvenlik algoritmaları[1], yapay sinir ağları[2], kontrol sistemleri[3] gibi birden çok alanda kullanılmaktadır.

FPGA'ler CLB adı verilen programlanabilir lojik bloklardan, I/O blocklarından (Giriş/Çıkış bloklarından) ve bu bloklar arasındaki bağlantıyı sağlanan programlanabilir ara bağlantılardan (routing) oluşur. Şekil 2.1'de FPGA'i oluşturan parçalar gösterilmiştir.

2.2 BRAM

CLB bloklarının içinde konfigüre edilebilir arama tabloları (LUT) bulunur. LUT'lar lojik fonksiyonlar için kullanılır fakat veri saklamak için de kullanılabilirler. Bu tabloların veri saklamak için birlikte kullanılmasında elde edilen belleğe dağınık bellek (Distributed RAM) adı verilmektedir. FPGA'lere dağınık belleklerin haricinde

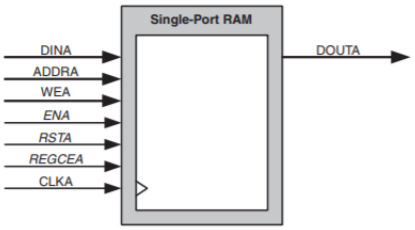


Şekil 2.1: FPGA'in bileşenleri [32]

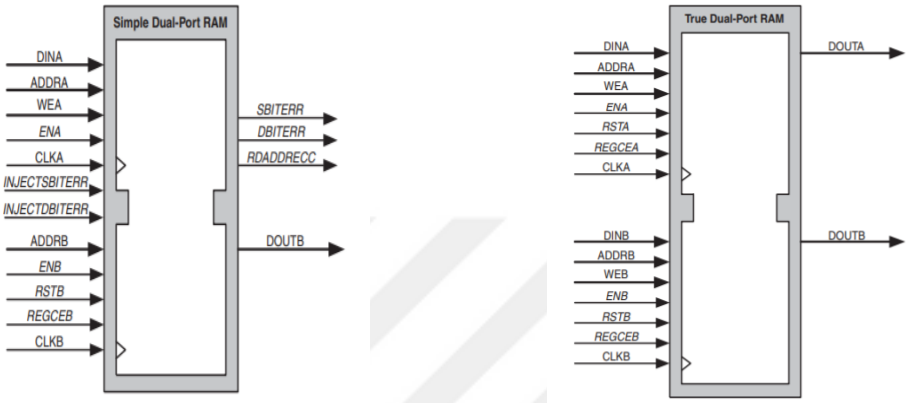
gereksinimlerden ötürü blok bellek (BRAM) adında özel fonksiyonlu bellek blokları da eklenmektedir. BRAM'ler harici belleklere göre daha az kapasiteye sahip olsalar da, dahili oldukları için BRAM'lere erişim daha hızlı sağlanmaktadır. Tek (Single) port ya da çift (dual) port olarak kullanılabilirler. Çift portlu bellekler basit (simple) çift port ya da gerçek (true) çift portlu bellekler olarak ayarlanabilirler. Tek portlu belleklerde okuma ve yazma için tek bir port bulunur. Basit çift portlu belleklerde iki adet port bulunmaktadır, yazma işlemleri için bir port ayrılmış okuma işlemleri için ise diğer port ayrılmıştır. Gerçek çift portlu belleklerde basit çift portlu belleklerin aksine port ayrımı yapılmaz ve iki port da yazma ve okuma işlemleri için kullanılabilir [33] . Şekil 2.2'de tek, basit çift ve gerçek çift portlu bellekler gösterilmiştir.

Xilinx-7 Serisi FPGA'lerin Çift Port BRAM'lerinde portların her biri kendi adres, data girişi, data çıkışı, saat, saat enable, yazma enable giriş/çıkışlarına sahiptirler [21].

Bir veri okunmak istediğinde adres girişinden BRAM'de okunulmak istenilen adresin değeri verilir ve saat vuruşu ile BRAM'deki istenilen adreste bulunan veri veri çıkışından okunur. Bir veri yazılmak istenildiğinde ise, adres girişinden verinin yazılmasının istendiği adres değeri verilir, veri girişinden bu adrese yazılmasi istenen



(a) Tek Portlu

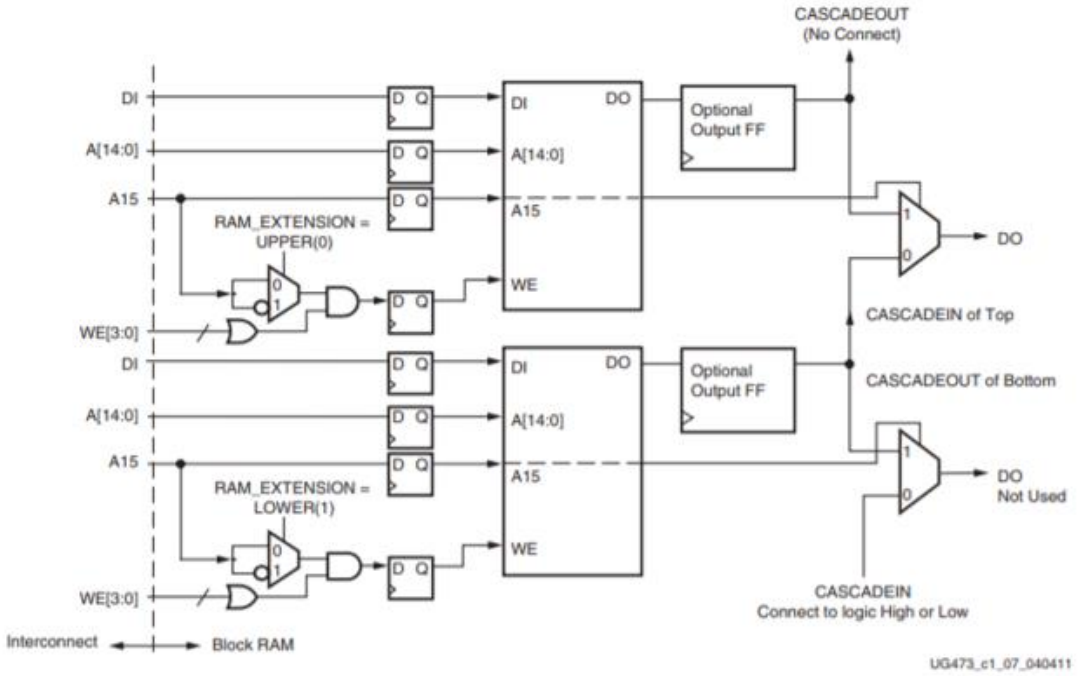


(b) Basit Çift Portlu **(c) Gerçek Çift Portlu**

Şekil 2.2: Blok bellek tipleri [33]

değer verilir ve yazma enable girişi aktif edilir. Bir adrese kaç bitlik veri denk geleceği BRAM'leri konfigüre ederken belirlenir. Örneğin tezimde 32 bit anahtar ve 32 bit değer ikililerinden oluşan bir veri seti kullanmış olduğum için BRAM'lerde data giriş ve data çıkış tellerimi 32 + 32 = 64 bit olarak konfigüre ettim.

BRAM'ler ayrı olarak kullanılabilir ya da birden fazlası birleştirilecek şekilde konfigüre edilebilir. Xilinx-7 Serisi FPGA'lerde bulunan BRAM'ler 36Kb veri kapasitesine sahiptirler ve iki ayrı 18Kb ya da bir 36Kb blok olarak kullanılabilirler. Her bir 36Kblik blok 64Kx1, 32Kx1, 16Kx2, 8Kx4, 4Kx9, 2Kx18, 1Kx36 ya da 512x72 şeklinde konfigüre edilebilir. Her bir 18Kblık blok ise 16Kx1, 8Kx2, 4Kx4, 2Kx9, 1kx18 ya da 512x36 olarcak şekilde ayarlanabilir [21]. Çalışmamızda, kullandığımız BRAM'lerin derinliğini arttırmak için iki adet 32x1'lik belleği birleştirerek 64Kx1 bellek kullanacağız. BRAM'lerin birleştirilerek derinliğinin nasıl arttırıldığını Şekil2.3'de gösterilmektedir.



Şekil 2.3: Kademelendirilebilir (Cascadable) BRAM [34]

2.3 İKİLİ ARAMA AĞACI

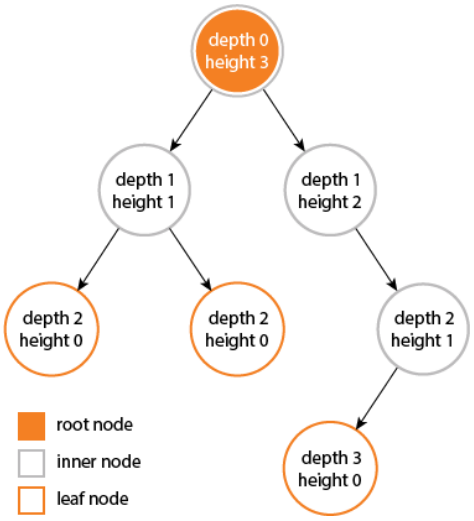
İkili arama ağacı en temel veri yapılarından birisidir. Bir BST'de her bir düğüm bir anahtara sahiptir ve en fazla çocuk (child) düğüme sahiptir. Anahtarlar genel olarak benzersiz (unique) anahtar olarak değerlendirilir fakat benzersiz olmadıkları uyarlamalar da bulunmaktadır. Bir anahtarın değerine sahip başka bir anahtar olmaması, yani her bir anahtarın tek olması durumunda bu anahtarlara benzersiz anahtar adı verilmektedir. Bir düğümün sağ çocuğunun anahtarı o düğümün anahtarından daha büyük, sol çocuğunun anahtarı ise o düğümün anahtarından daha küçük değere sahiptir. Bir düğüm iki veya tek çocuk düğüme sahip olabilir ya da çocuk düğüme sahip olmayabilir. Çocuğa sahip olmayan düğüm yaprak düğüm olarak adlandırılır.

Her bir ağaç bir kök düğüme sahiptir bu kök düğüm arama işlemlerinin başlangıç noktasıdır. Ağaçtaki her bir düğüm kendisinin de kök düğümü sayılabileceğinden, bu düğümün kök kabul edildiği ağaca alt ağaç adı verilmektedir. Bir düğümün sol çocuğunun alt ağacı sol alt ağaç olarak nitelendirilirken, sağ çocuğunun alt ağacı ise sağ alt ağaç olarak nitelendirilir [35].

Θ(n) zaman karmaşıklığına sahip lineer arama veri yapılarının aksine arama yapılırken her bir değere teker teker bakılmaz. Kök düğümden (Root) başlanarak istenilen anahtarın kök düğümün anahtarı ile karşılaştırılması yapılır, eğer istenilen anahtar daha büyük değere sahip ise kökün sağ çocuğuna, daha küçük değere sahip ise kökün sol çocuğuna geçilerek arama devam ettirilir. Bu özelliğinden ötürü zaman karmaşıklığı Θ(log(n)) olarak belirlenir. Arama algoritması sözde kod olarak aşağıda verilmiştir [36].

```
TREE-SEARCH ( x, k)
        if x == NIL or k == x.key
                return x
        if k < x.key
                return TREE-SEARCH (x.left, k)
        else return TREESEARCH (x.right, k)
```

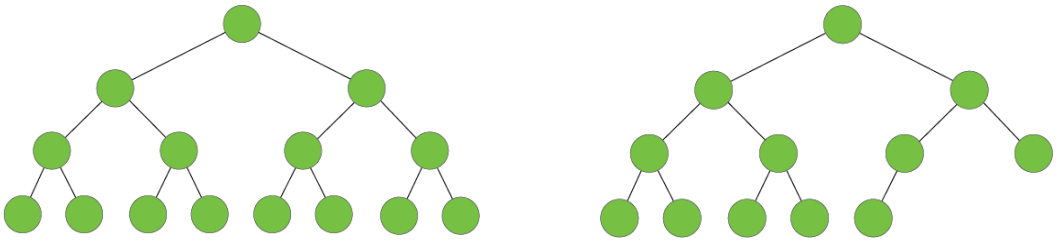
Bir düğümün yüksekliği (height) yaprak düğüme giden en uzun yol kadardır, derinliği (depth) ise köke uzaklığı kadardır. Kökün derinliği 0 kabul edilmektedir. Bir ağacın düğümlerinin yüksekliği ve derinliği Şekil 2.4 'te gösterilmektedir. Çalışmamızda ağaç seviyeleri belirtilirken düğümlerin derinlikleri incelenmektedir.



Şekil 2.4: Düğümlerin yükseklik ve derinliği [37]

Bir ikili ağacın son seviyesi hariç, tüm seviyelerindeki düğümlerinin sağ ve sol çocukları varsa ve en son seviyedeki çocuklar soldan başlayarak yerleştirilmiş ise bu

ikili ağaç eksiksiz (complete) ikili ağaç olarak adlandırılır. Şekil 2.5'te gösterilen ağaçlar eksiksiz ikili ağaçtır ve ilk figürdeki ağaç eksiksiz olmanın yanı sıra dolu (full) bir ağaçtır. Çalışmamızda sonsuz anahtar akışı (infinite key stream) sırasında ağaç türü değiştiğinde verimin sabit kalmasından ötürü eksiksiz BST üzerine çalıştık.



(a) (b)

Şekil 2.5: Eksiksiz İkili Arama Ağaçları

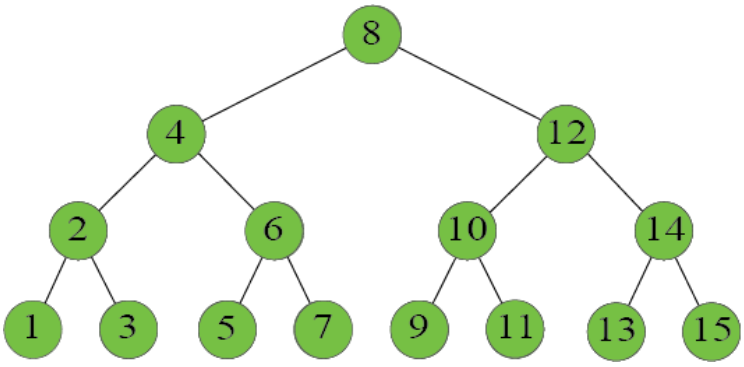
Dolu ve eksiksiz bir BST'nin d derinliğine sahip düğüm sayısı 2^d olarak, d derinliğine kadar olan düğüm sayısı ise 2^d-1 olarak hesaplanır. Şekil 2.5a'da verilmiş ağaçta 2 derinliğine sahip düğümlerin sayısı 2^2 ve derinliği 4'ten daha az olan düğüm sayısı ise 2^4-1 ile hesaplanır.

3. DONANIM BAZLI HIZLANDIRMA YÖNTEMLERİ

İkili arama ağaçlarında arama işlemini hızlandırma ve çevrim (cycle) başına düşen verimliliğini arttırmak için Basit, Yatay Kesme, Çoklama, Yazmaç Ekleme, Hibrit Kesme ve Buffer Ekleme olmak üzere altı adet yöntem ve iyileştirmeler sunduk.
Sunulan hızlandırıcıları kullanarak aranmak istenen anahtarlar, sistemlere gruplar halinde getirilir. Bu gruplarda kaç tane anahtar olacağı hızlandırıcının tek bir çevrimde paralel olarak aranabilecek maksimum anahtar sayısına eşit olacak şekilde belirlenmiştir.

3.1 Basit Yöntem

CBST veri yapısında bulunan veriler, tek bir BRAM Bölmesi'nde muhafaza edilmektedir. Çalışmamızda bir ya da daha fazla BRAM'den oluşan gruplara BRAM Bölmesi adını veriyoruz. Bir BRAM Bölmesi birden fazla blok bellekten oluşuyor olsa bile, port özellikleri tek bir bloktan farklılık göstermez sadece daha fazla kapasiteye sahip bir blok bellek olarak düşünülebilir. BRAM Bölmesi'nde bulunan



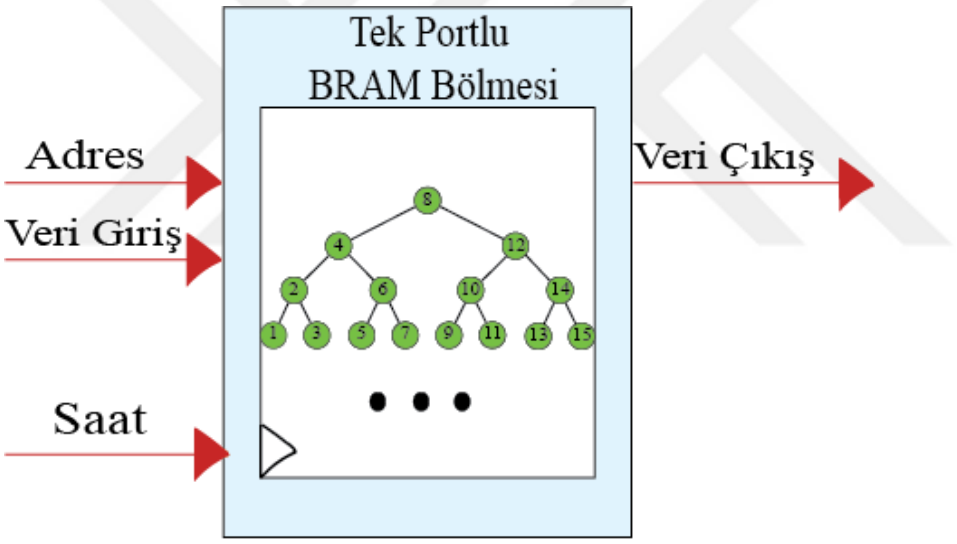
32 bit	32 bit
8	X
4	X
12	X
2	X
6	X
10	X
14	X
1	X
3	X
5	X
7	X
9	X
11	X
13	X
15	X

Şekil 3.1: Bir CBST'nin bellekte saklanması

ağacı ortada kök ziyareti (inorder) ile gezinilmiş halde liste olarak muhafaza ediyoruz. Örnek bir ağaç ve bellekte saklanma şekli Şekil 3.1'de gösterilmiştir, her bir düğümde 32 bit anahtar ve 32 bit değer saklandığı varsayılmıştır.Ağacın tutulduğu BRAM Bölmesi tek portlu ve çift portlu olmak üzere konfigüre edilmiştir. İyileştirmelerin daha anlaşılır olması için basit yöntemin üzerinde durulmuştur.

3.1.1 Tek Portlu BRAM

BRAM'ler tek port olarak ayarlandıkları durumda, yazma ve okuma için sadece bir giriş/çıkışa sahip olduklarından ötürü aynı anda sadece bir işlem yapabilmektedirler. Tek bir adres girişi ve tek bir sonuç çıkışları vardır. Şekil 3.2'de gösterilen BRAM Bölmesi'nin adres girişine okunmak istenen adresin değeri verilir ve saat vuruşu ile istenilen değer veri çıkışından okunur.



Şekil 3.2: Tek Portlu BRAM Bölmesi

Şekil 3.1'de verilen ağacın 10 numaralı anahtarının değerine erişmek istenirse:

1. 9 numaralı anahtarı aramaya kökten başlanır. Bölmenin *Adres* girişine kökün adresi olan 0 değeri verilir. Veri çıkışından 64 bitlik veri alınır. Bu veri 32 bitlik anahtar ve 32 bitlik değer ikilisinden oluşmaktadır.

2. Veri çıkışından alınmış verinin ilk 32 biti (anahtar bitleri) aranan anahtar ile karşılaştırılır. Çıkıştan alınan anahtar 8, aranan anahtar 10'a eşit değildir, aramaya devam edilmelidir. 8 < 10 olduğu için 8 anahtarının olduğu düğümün sağ çocuğuna geçiş yapılır.
3. Sağ çocuğa geçiş yapılacağı için sağ çocuğun adresine ihtiyaç duyulmaktadır. Sol çocuğun adresi hesaplanırken ebeveyn düğümün adresinin 2 katının 1 fazlası, sağ çocuğun adresi için ise ebeveyn düğümün adresinin 2 katının 2 fazlasını almak gerekmektedir. Sağ çocuğa geçişmek istendiği için 0 x 2 + 2= 2 numaralı adresteki veriye erişilmesi gerekmektedir.
4. Adres girişine 2 değeri sürülür ve veri çıkışından 2 numaralı adreste bulunan 12 anahtarı ve ona bağlı değer elde edilir.
5. Çıkıştan alınan anahtar 12, aranan anahtar 10'a eşit değildir, aramaya devam edilmelidir. 12 > 10 olduğu için 12 anahtarının bulunduğu düğümün sol çocuğuna geçiş yapılır.
6. Sol çocuk istendiği için, sol çocuğun adresi 2 x 2 +1 = 5 olarak bulunur ve adres girişine sürülür.
7. 5 numaralı adresten okunan veri 10 numaralı anahtarı içermektedir. Aranan anahtar ile eşleşme gösterildiği için arama tamamlanmıştır ve <10, 10 anahtarına bağlı veri> ikilisi döndürülür.

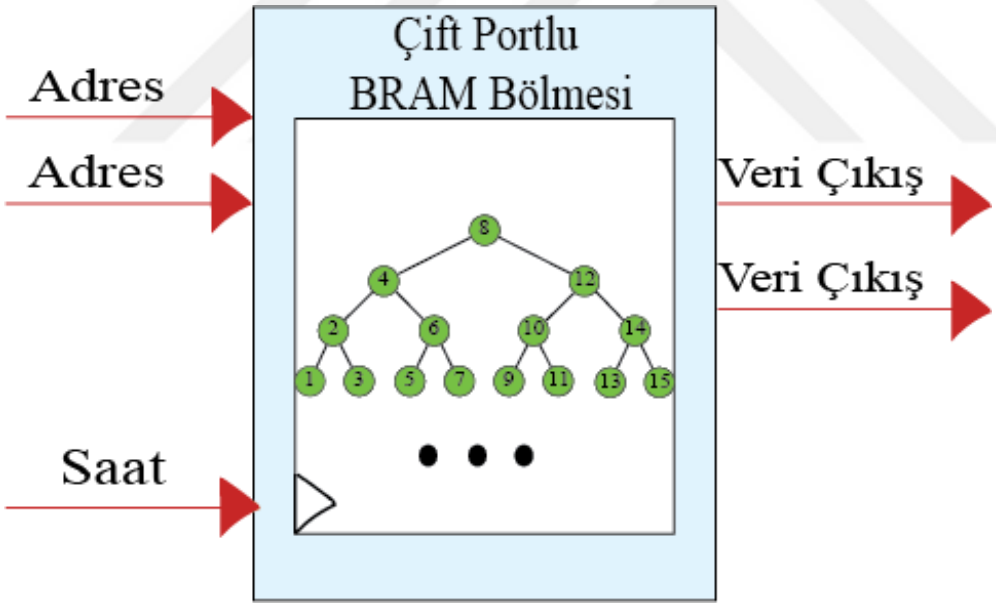
3.1.2 Çift Portlu BRAM

Arama işlemi ekleme ya da silme gibi işlemlerinin aksine veriler üzerinde değişiklik yapmaz, sadece verileri okumamıza olanak sağlar. Bu yüzden aynı anda birden çok arama fonksiyonunun birlikte çağırılması bir sorun teşkil etmeyecektir. Aynı anda birden fazla ekleme ve silme işlemleri için ise farklı tutarlılık yolları gerçeklenmelidir [38]. Aynı anda veri okumak tutarsızlık sorununa yol açmayacağı için BRAM Bölmesi gerçek çift port olarak ayarlanabilir ve aynı anda iki adet anahtar okunarak 2 kat fazla verim elde edilebilir.

Çift port konfigürasyonu yapılmış bölme, iki adet adres ve iki adet veri çıkışı gitiş/çıkışlarına sahiptir. Her bir porttan bir anahtar arayacak şekilde Şekil 3.1'de

gösterilen ağacın 10 (i) ve 5 (ii) numaralı anahtarları aranmaktadır:

1. Her iki anahtar için de aramaya kökten başlanır. Bölmenin ilk adres girişine (i) anahtarı, ikinci adres girişine ise (ii) anahtarı sürülür. İki veri çıkışından da kökün anahtarı ve değeri elde edilir.
2. İlk çıkıştan alınan verinin ilk 32 bitlik anahtarı ile (i) anahtarı 10 karşılaştırılır, ikinci alınan verinin ilk 32 bitlik anahtarı ile ise (ii) anahtarı 5 karşılaştırılır.
3. Her iki anahtar için de arama süreci tek portlu bölmede olduğu gibi gerçekleşir. Aynı anda BRAM Bölmesi verilerine iki farklı porttan ulaşıp verileri iki farklı porttan okuyabildiğimiz için bu iki arama işlemi birbirlerinin sonucuna etkide bulunmayacak ve aynı anda arama işlemi yapabilmemize imkan sağlayacaktır.
4. Her iki anahtar da bulunduktan sonra sıradaki iki yeni anahtar seçilir ve aranmaya başlanır.

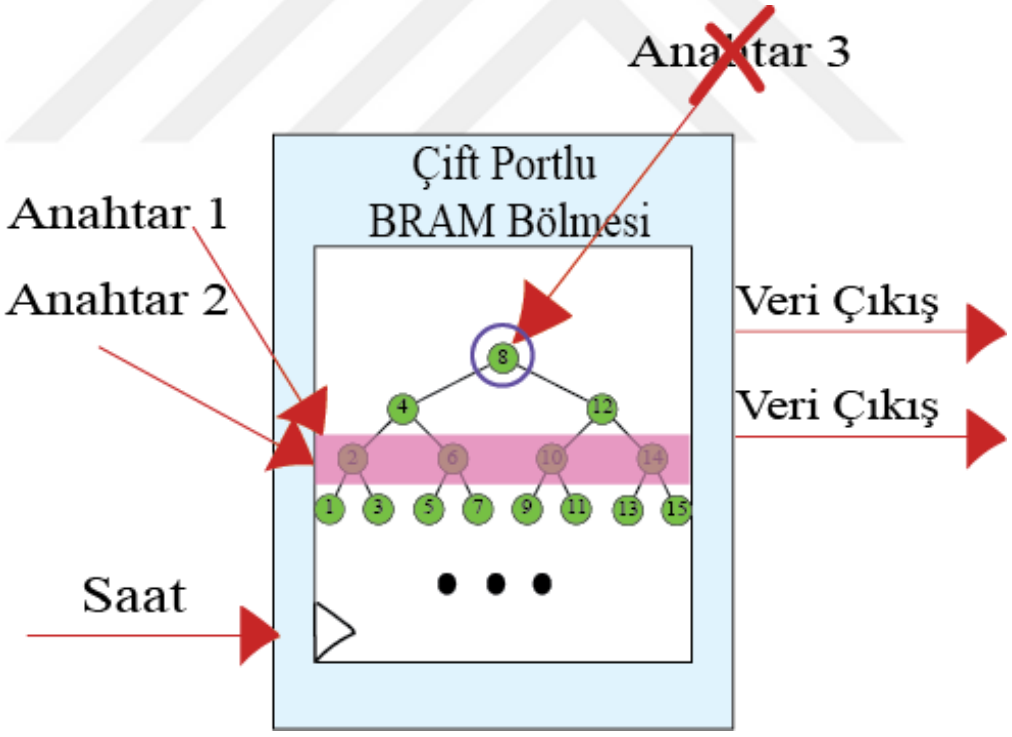


Şekil 3.3: Çift Portlu BRAM Bölmesi

3.2 Yatay Kesme Yöntemi

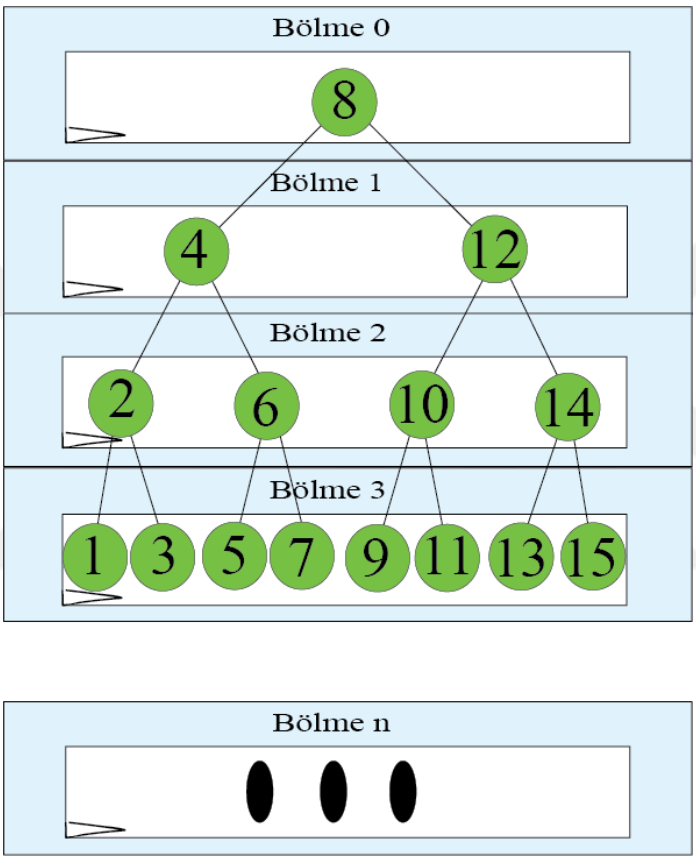
Bir önceki yöntemde bir tane BRAM bölmesi kullanmıştık ve bu bölmenin içine CBST'mizi yerleştirmiştik. Bu ağaca tek portlu bölme ile her çevrimde en fazla bir

kez, iki portlu bölme ile ise her çevrimde en fazla iki kez ulaşabiliyoruz. Fakat bir arama sürecinde bir anahtar her bir ağaç derinliğine aynı anda erişim sağlamıyor ve sadece tek bir derinliğe erişiyor. Hem tek port için hem çift port için aranacak anahtarlar ilk çevrimde derinliği 0 olan kök düğümünden başlayarak, her bir çevrimde derinliği son çevrimde aranan düğümün derinliğinden bir fazla olan düğümler arasında aranıyor. İlk çevrimde derinliği 0 olan düğümler, ikinci çevrimde 1 olan düğümler, üçte 2 olan düğümler arasında arama yapılıyor bir anahtar için. Her bir çevrimde yeni bir derinlik değerine geçildiği için, eski derinlik değerlerine sahip düğümlerde o anahtar için yeniden arama yapılmayacaktır. Anahtarlar aynı anda aranmaya başlandıkları için çift portlu bölmelerde de bu durum değişmiyor ve aynı anda sadece bir seviye aranıyor. Bu durumda ağacın sadece bir derinlik seviyesini incelemek isterken yetersiz port sayısı yüzünden ağacın geri kalanında kullanılmasını engelleniyor. Bu sebepten ötürü aranacak yeni anahtarlar ağacın aynı seviyesine erişmeyecek olsalar bile, araması başlanmış anahtarların bölmeyi kullanmayı bırakması beklenmek zorunda kalınıyor.



Şekil 3.4: Bir BRAM Bölmesinde farklı seviyede aranacak bir anahtarın aramaya başlayamaması

Şekil 3.4'te gösterilen çizimde yeni aranacak anahtar 3, sadece kök düğüme erişeceği halde BST ağacının tümü tek bir bölmede olduğu için ve 1 ve 2 numaralı anahtarlar bu bölmeyi kullandığı için 3 numaralı anahtar diğer iki anahtarın arama işleminin bitmesini beklemek zorundadır. 3 numaralı anahtar eğer 1 ya da 2 numaralı anahtarlardan herhangi birisi yaprak düğümlerde ise en kötü durumda (worst case) ağacın yüksekliği kadar çevrim beklemek (stall etmek) zorunda kalacaktır.



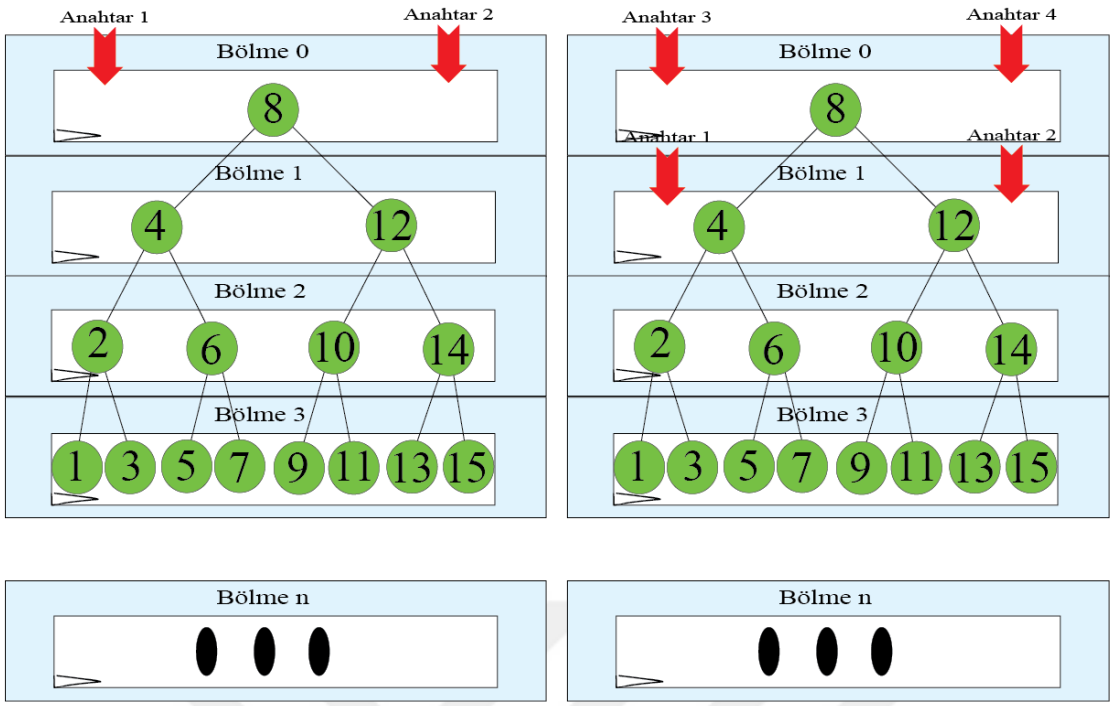
Şekil 3.5: Yatay Kesme Yöntemi ile farklı bölmelerde saklanan CBST

Ağacın farklı seviyelerini kullanamamaktan doğan bekleme probleminden kurtulmak ve farklı seviyelere aynı anda erişim sağlayabilmek için CBST'yi her bir seviye farklı bir bölme de olacak şekilde böldük. Aynı anda birden (çift port için ikiden) fazla anahtarın aranmasına olanak sağlamak ve verimi arttırmak amaçlanmıştır. Şekil 3.5'te her bir bölmenin ağacın farklı bir seviyesinin tutulduğu gösterilmektedir. Bölme 0, 0 derinliğine sahip düğümleri tutarken, 1. Bölme 1 derinliğine sahip

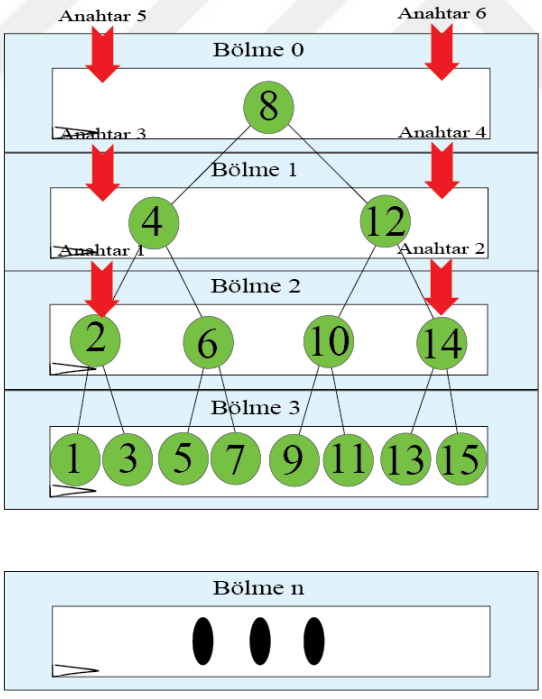
düğümleri tutar, Bölme n ise n derinliğine sahip düğümleri tutar. Bu yapı sayesinde her bölmenin iki portu olduğu için, ağacın her bir seviyesinde aynı anda 2 anahtar araması yapılabilmesini sağladık.

Şekil 3.5'te gösterilen yapıyı boruhattı yöntemi ile çalışacak şekle getirdik. Bir anahtarı aramak için önceki anahtarların aranmalarını tamamlamalarını beklemek yerine, bir çevrim sonra Bölme 0'ı kullanan anahtar kalmayacağı için kökte aranmak üzere yeni anahtarların aranması başlatılıyor. Her bir çevrimde, her bir bölmede aranan anahtarlar bulunamazlarsa bir sonraki bölmeye geçecek ve başlangıç bölmesinden yeni anahtar akışı sağlanmış olacaktır. Şekil 3.6 boruhattı yoluyla Yatay Kesme Yöntemi ile aranan anahtarların farklı çevrimlerdeki durumlarını göstermektedir. Şekilde görüldüğü gibi her bölme aynı anda iki anahtar aramakta fakat birden fazla bölme olduğu için aynı anda aranan toplam anahtar sayısı artış göstermiştir. En kötü (worst case) senaryoya göre tüm anahtarların yapraklarda bulunacağı varsayılır ise, ağacın yüksekliği kadar çevrim geçtiğinde bütün bölmeler aramalar için kullanılıyor olacaktır. Bu çevrimden sonra her bir çevrimde iki adet anahtarın değer sonucu döndürülecektir. Son bölmedeki anahtarlar sonuca eriştiği için son bölmeyi terk edecekler ama sondan bir önceki seviyede bulunan anahtarlar son bölmeye geçeceklerdir. Tüm anahtarlar bölme olarak ilerleyecekleri ve başlangıç bölmesinden yeni anahtarların girişi sağlanacağı için bütün bölmeler her zaman çalışacak ve aynı anda hep ağacın seviye sayısı x 2 adet anahtar aranıyor olacaktır. Verimimiz bu aşamadan sonra çevrim başına 2 anahtara yükselmiştir.

Ağaçta bulunan veriler bu yöntem ile parçalara ayrılmış olduğu için istenen veriye erişmek için gereken adres hesaplamasında da değişiklik meydana gelmiştir. Basit yöntemde olduğu gibi verileri saklarken 0, 1, 2, 3 şeklinde tek boyutlu düşünmek yerine, 0. Bölme 0. Adres, 1. Bölme 0. Adres, 1. Bölme 1. Adres şeklinde çok boyutlu bir adresleme sistemi ile işlemler yapılacaktır. Bir önceki yöntemde tutulan [8 , 4, 12 , 2 ,6 …] liste adreslemesi Bölme 0 [8], Bölme 1[4, 12], Bölme 2 [2, 6 …] şeklinde değişim göstermiştir. İlk yöntemde 12 anahtarına ulaşmak için gereken adres 2 iken, bu yöntemde Bölme 1, adres 1 olarak erişilmektedir.



(a) Başlangıç çevrimi **(b) Birinci çevrim**



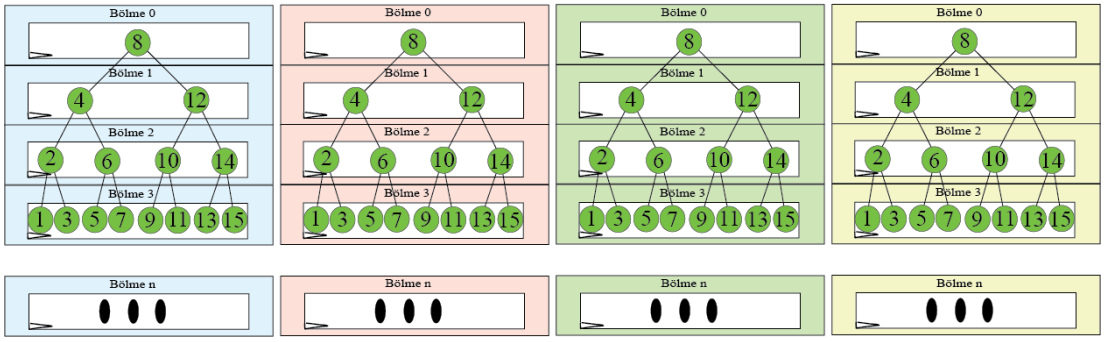
(c) İkinci çevrim

Şekil 3.6: Yatay Kesme Yönteminin boru hattı ile kullanılması ile farklı çevrimlerde anahtarların durumları

3.3 Çoklama Yöntemi

Yatay Kesme Yöntemi'nden alınan verimi daha fazla arttırmak için çoklama yöntemini öneriyoruz. Bu yöntemde kullanılan bütün bölmeleri kopyalayarak çoğalttık. Kopyalama sayısını m olarak belirtirsek, yatay kesme yönteminde olan n adet olan bölme sayısı çoklama yönteminde n x m adet bölme oluşturmaktadır. Bu durum ağacın seviyesi başına düşen port sayısını 2'den 2 x m 'ye çıkarmaktadır. Ağacın bir seviyesine erişim için port sayısı artınca, bu seviyede aynı anda aranabilecek anahtar sayısı da artmış olmaktadır. Bu durumda bu yöntem ile verim çevrim başına m x 2 anahtar olarak artış gösterecektir. m sayısı kullanılan donanımda kaç adet BRAM olduğu, bu BRAM'lere kaç adet kopyanın sığabieceği (kapasitesi) ve veri aktarımı yapılırken bant genişliğinin (bandwith) kaç bit olduğuna dikkat edilerek seçilmelidir. m değeri bant genişliğinden daha yüksek veri aktarımı oluşturacak şekilde seçilir ise, bant genişliğinden ötürü bottleneck oluşturulacak ve istenilen hızlandırılma sağlanamayacaktır. Ağacın kopya sayısı derleme anında ayarlanabilecek şekilde kodlanmıştır.

Çoklama yöntemi aynı seviye çoklama ve farklı seviye çoklama olmak üzere iki farklı şekilde gerçeklenmiştir.



Şekil 3.7: Aynı seviye çoklama yöntemi yapısı

3.3.1 Aynı Seviye Çoklama

Aynı seviye çoklama ile CBST değişiklik yapılmadan kopyalanmıştır, her bir ağaç birbirinin birebir aynısıdır. Kopyalama işleminden ötürü bu yöntem verimi arttırmış

olsa da, aynı veriden birden fazla tutarak daha fazla alan kaplamaktadır. Özellikle fazla düğüm sayısına sahip ağaçlar için saklama alanı yetersizliği doğurabilmektedir. Şekil 3.7 kopya sayısı 4 olarak seçilmiş aynı seviye çoklama yöntemi sonucu oluşan yapıyı göstermektedir.

3.3.2 Farklı Seviye Çoklama

Çoklama yönteminde oluşan daha fazla saklama alanı gereksiniminden ötürü, kopyalama işleminde daha az alan kaplanması için her ağacın her bir seviyesinin kopyalanması yerine sadece seçilen bazı seviyeleri kopyalanabilir. Çizelge 3.1'de verilerini rastgele oluşturduğumuz bir CBST ağacında, değerleri rastgele verilmiş aranan anahtarların her birinin BRAM Bölmeleri'ni kaç kez kullandıkları gösterilmektedir. Tüm anahtarlar ilk bölmeden başlamaktadırlar fakat derinlik arttıkça anahtarların bir bölümü bulunduğu için kullanım sayılarında azalış görülmüştür. Çizelgede gösterilmiş değerler için sınırlı (finite) bir anahtar listesi kullanılmıştır. Arama süresince ortalama kullanım yüzdelerinin %100 olmamasının nedeni boruhattının dolma süresi ve sınırlı bir aranacak anahtar listesi olmasından ötürü yeni anahtar girişi yapılmamasıdır.

Bir bölmenin derinliği arttıkça kullanılma oranının azaldığı göz önüne alınır ise, daha az kullanılan bölmelere çoklama işlemi yapmayarak aynı seviye çoklama yöntemine göre daha az yer gereksinimi oluşturulabilir. Her bir kopya ağaç, asıl ağacın son seviyeleri kırpılmış bir kopyası olursa daha az yere gereksinim duyulacaktır. Farklı seviye çoklama yönteminde anahtar akışı Şekil 3.8 de gösterilmiştir. n seviyesine kadar kopyalanmış bir kopya ağacın, n. seviyesinde de bulunamayan bir anahtar n+1 seviyesine kadar kopyalanmış bir kopya ağaca, ağacın n+1. Seviyesinde aranması için gönderilir, sırasıyla daha büyük kopya ağaçlar arasında geçiş yapılır ve en son asıl ağaçta aranır. n seviyesi derleme anında ayarlanabilecek şekilde kodlanmıştır.

n seviyesi ağaçta tutulan verinin ve anahtarların ortalama ne kadar ortalama kullanım yüzdesi oluşturduğu bilindiği durumda daha optimal seçilebilir. Optimal n değeri bir ağaçtan diğer ağaca geçiş yaparken, geçiş yapılan ağacın istenilen seviyelerinde daha az kullanımı olması ve bu sayede boşta port olmasına göre belirlenir. Eğer tüm aranacak anahtarlar seçilecek n değerinden daha fazla bir derinlikte ise, bu yöntem

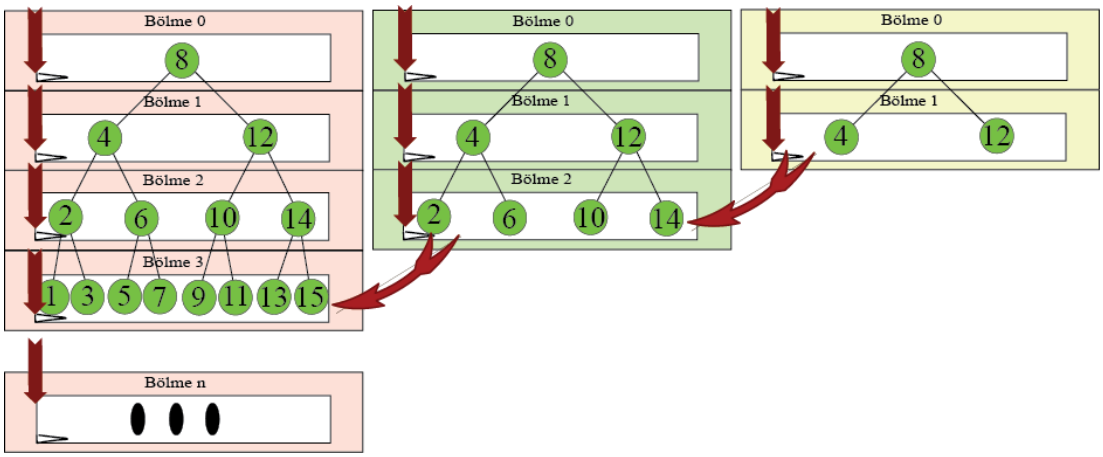
yeterli port sayısı olmadığı için beklemelere yol açacaktır. Her ağaç kendisinden başlayan anahtarlara öncelik vereceği için en az seviyeye sahip kopya ağaçlar son bölmesindeki anahtarları diğer ağaçlara gönderemeyecek, son bölmede takılı kalan anahtarlar boruhattı tıkanıklığına neden olacak ve o ağaçlar için yeni anahtar alma işlemi beklemeye alınacaktır.

Çizelge 3.1: BRAM Bölmelerinin kullanım oranları

Bölme Numarası	1. Portta Bulunan Anahtar Sayısı	2. Portta Bulunan Anahtar Sayısı	1. Port Kullanımı	2.Port Kullanımı	Ortalama Kullanım	Ortalama Kullanım Yüzdesi
0	1	0	4096	4096	4096.0	99.63512
1	1	0	4095	4096	4095.5	99.62296
2	1	1	4094	4096	4095.0	99.61080
3	1	0	4093	4095	4094.0	99.58647
4	0	0	4092	4095	4093.5	99.57431
5	1	2	4092	4095	4093.5	99.57431
6	4	2	4091	4093	4092.0	99.53782
7	9	11	4087	4091	4089.0	99.46485
8	21	22	4078	4080	4079.0	99.22160
9	34	37	4057	4058	4057.5	98.69861
10	62	81	4023	4021	4022.0	97.83507
11	146	141	3961	3940	3950.5	96.09584
12	248	259	3815	3799	3807.0	92.60520
13	490	467	3567	3540	3553.5	86.43882
14	996	1043	3077	3073	3075.0	74.79931
15	2081	2030	2081	2030	2055.5	50.00000

3.1 Yazmaç (Register) Ekleme

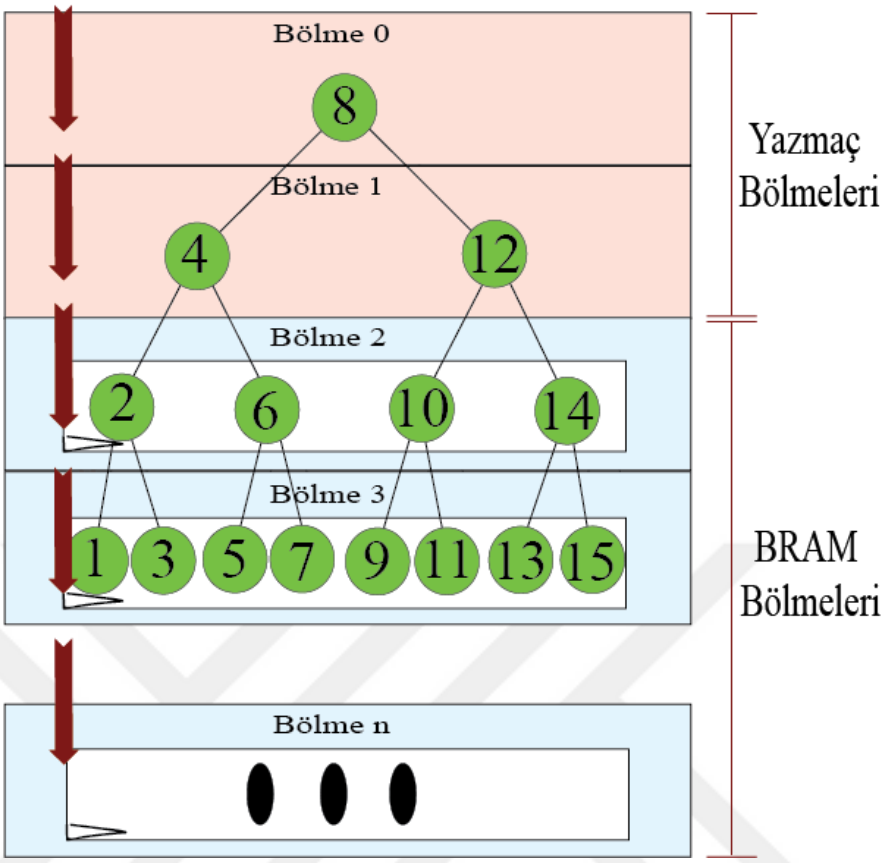
Çoklama yönteminde iki farklı sorun ortaya çıkmaktadır. Birincisi aynı seviye için tüm ağacı, farklı seviye için bazı seviyeleri kopyaladığımız için yer sıkıntısı yaşanmaktadır. İkinci sorun ise hem çoklama yönteminde hem de yatay bölme



Şekil 3.8: Farklı seviye çoklama yönteminde anahtar akışı

yönteminde bir seviyede tek bir düğüm olsa bile bir tane BRAM Bölmesi kullanılmaktadır. Örnek olarak 0. Bölme sadece kök düğümü barındırmaktadır. Hem birden fazla aynı veriye sahip olmaktan oluşan fazla kapasite gereksinimi hem de BRAM'lerin tamamen doldurulmayarak israf edilmesini engellemek için yazmaç (register) ekleme yöntemini öneriyoruz.

BRAM Bölmeleri'nde saklanan ağacımıza Yazmaç Bölmeleri ekledik. Bir ya da birden fazla yazmaç topluluğuna Yazmaç Bölmesi adı veriyoruz. Bu yazmaç bölmelerinin her biri BRAM Bölmeleri gibi ağacın bir seviyesini saklamaktadır. Ağacımızı iki parçaya böldük ve bu ağacın ilk seviyelerini yazmaç bölmelerine diğer seviyelerini ise BRAM bölmelerine yerleştirdik. Bu yapıda son yazmaç bölmesinde bulunan düğümler ilk BRAM bölmesinde bulunan düğümlerin ebeveyn düğümleri olarak belirlendi. Şekil 3.9'da Yazmaç Ekleme optimizasyonu ile oluşturulan yapı gösterilmektedir ve şekildeki örnekte ağacın ilk iki seviyesi yazmaç bölmelerinde saklanırken diğer bölmeler BRAM bölmelerinde saklanmaktadır. Anahtarların aranış yönteminde bir değişiklik yapılmamış ve boru hattı özelliği kaybedilmemiştir. Bir anahtar kök düğümün bulunduğu yazmaçtan aramaya başlanılarak her bir saat vuruşunda bir sonraki yazmaç bölmesine anahtarın geçişi yapılır, yazmaç bölmelerinin son seviyesinde de bulunamayan ve aranması devam edecek olan anahtar ilk BRAM bölmesine geçiş yapar ve aramasını sürdürür. Bu optimizasyon çoklama ve yatay kesme yöntemlerinin çevrim başına düşen verimini arttırmamakta fakat kullanılan FPGA kaynaklarının daha verimli kullanılmasını sağlamaktadır.



Şekil 3.9: Bir ağacın yazmaç ve BRAM bölmelerinde tutulması ve anahtar arama işlemi

3.1.1 Yer Sorunu

Yazmaçlar BRAM'lere göre daha esnek erişim sağlamaktadır. Bir BRAM Bölmesi'ne erişimde port başına en fazla bir anahtar arandığı için, aynı anda en fazla iki anahtar araması yapılabilmektedir, bu durum Yazmaç Bölmeleri'nde yoktur ve aynı anda istenildiği kadar anahtar aranabilir.

Çoklama yönteminde Yazmaç Ekleme optimizasyonundan sonra sadece BRAM Bölmeleri çoğaltılacaktır. Yazmaç Bölmeleri'ne erişim için bir kısıt olmadığı için ve bu bölmelerde bulunan veriye aynı anda erişmek mümkün olduğu için bu bölmeleri çoklamamıza gerek yoktur. CBST'nin her bir seviyesini çoğaltmak yerine, sadece BRAM Bölmesi'nde bulunan verileri çoğalttığımız için gerekli depolama kapasitesi azalacaktır.

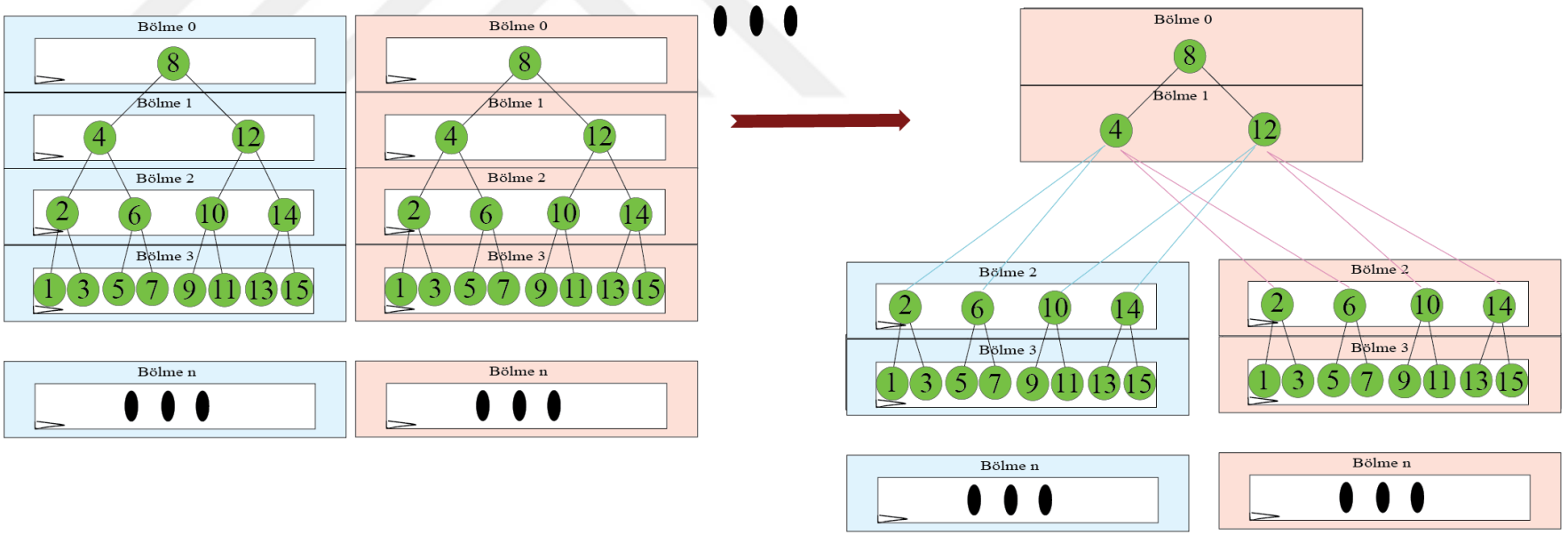
3.1.2 BRAM İsrafı

Yatay Kesme ve Çoklama yöntemlerinde BRAM Bölmeleri'nin kapasitelerinin tamamının kullanılmamasından ötürü BRAM israfı oluşmaktadır. Bir seviyede bir tane düğüm olsa bile bu seviye için bir bölme ayrılmaktadır. Bir BRAM bölmesini dolduramayan sayıda düğümlerin bulunduğu seviyeler, Yazmaç Bölmesi'ne konuşarak bu israfın engellenmesi amaçlanmıştır.

Çizelge 3.2 tek ağaç olduğu durumda Yazmaç Bölmesi sayısına karşılık, ilk BRAM Bölmesi'nin saklaması gereken düğüm sayısını göstermektedir. Çalışmamızda kullandığımız bir BRAM Bölmesi'nin kapasitesini 64Kx1 olarak ayarlamış olduğumuz ve ağacımız 32 bit anahtar, 32 bit veri ikililerinden oluşan düğümlerden oluştuğu için bir bellek bölmesinin kapasitesi 1024 düğümdür. 1024 adet düğüm bir bellek bölmesini dolduracak olduğu için Yazmaç Bölmesi sayısını 10 adetten az ayarlamak BRAM israfına, 10 adetten çok ayarlamak ise yazmaç israfına neden olacaktır. Bu sebeple Yazmaç Bölmesi sayısını 10 olarak belirledik. İlk 10 seviye yazmaçlarda, diğer seviyeleri ise bellekte sakladık.

Çizelge 3.2: Yazmaç Bölmesi sayısına karşılık, ilk BRAM Bölmesi'nin saklaması gereken düğüm sayısı

Yazmaç Bölmesi Sayısı	0. BRAM Bölmesi Düğüm Sayısı
0	$2^0 = 1$
1	$2^1=2$
2	$2^2=4$
3	$2^3=8$
4	$2^4=16$
5	$2^5=32$
6	$2^6=64$
7	$2^7= 128$
8	$2^8 = 256$
9	$2^9 = 512$
10	$2^{10} = 1024$
11	$2^{11}= 2048$
n	2^n

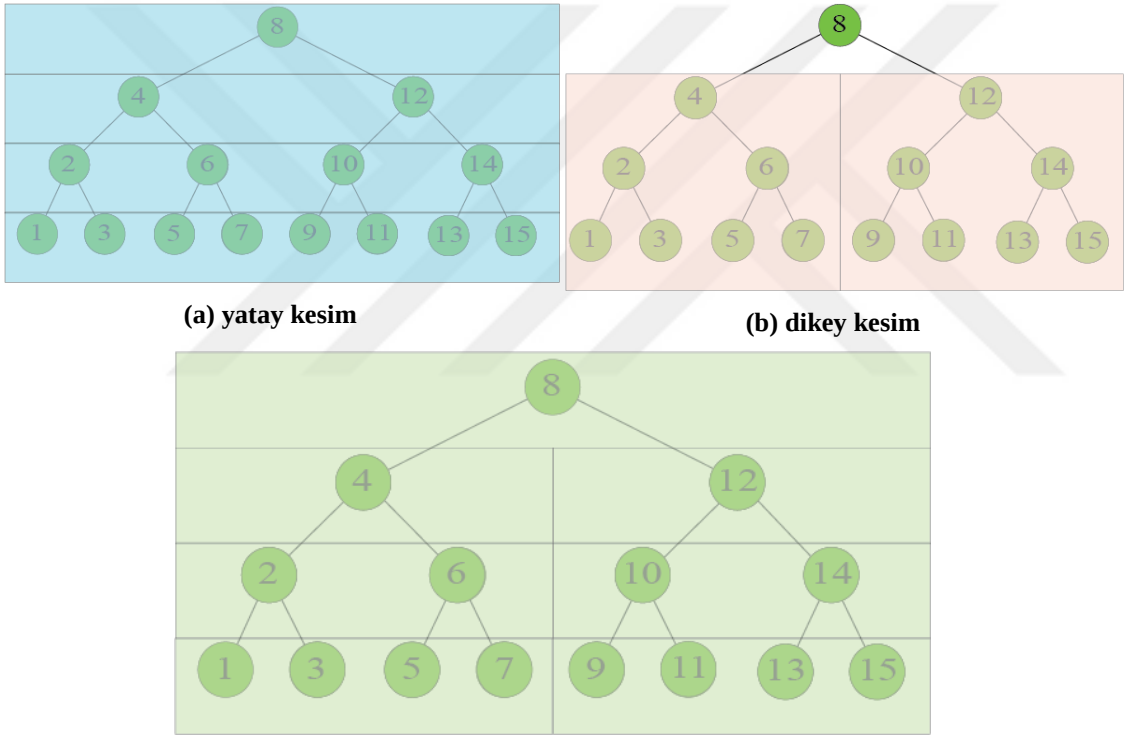


Şekil 3.10: Çoklama işleminde meydana gelen yer sorununun yazmaç kullanarak azaltılması

3.2 Hibrit Kesme Yöntemi

Yeterli bellek sayısı ya da yazmaç sayısı derleme anında ayarlanabilir olduğu için, yeterli kaynak eksikliği var ise yazmaç ve bellek bölme sayısı ideal değer dışında seçilebilir.

Yazmaç Eklemesi optimizasyonu ile belleklerin tüm kapasitesinin kullanılması sağlanmış olsa da aynı verinin birden fazla kopyalanmasından oluşan yer sorunu tamamen çözülmemiştir. Bu sorunu çözmeye yönelik Hibrit Kesme Yöntemi'ni öneriyoruz.



(a) yatay kesim **(b) dikey kesim**

(c) hibrit kesim

Şekil 3.11: CBST kesim türleri

Hibrit Kesme Yöntemi'nde ağacı sadece yatay değil aynı zamanda dikey olarak da kesitlere ayırıyoruz. Yatay kesitler ağacı seviye seviye ayırırken, dikey kesitler ise ağacı sağ ve sol çocuk olarak ayırıyor. CBST'nin yatay, dikey ve hibrit kesitleri Şekil 3.11'de gösterilmektedir.

Hibrit Kesme Yöntemi ile yazmaç bölmelerinde erişim sorunu olmadığı için sadece bellek bölmelerini hibrit olarak kesiyoruz. Bir bellek bölmesi en az o kadar düğüme sahip olduğu sürece istenildiği kadar parçaya ayrılabilir. Bir seviyede 16 adet düğüm bulunuyor ise 2, 4, 8 ya da 16 adet eş parçaya ayrılabilir. Parçalanma sayısı arttıkça 0. Bram Bölmeleri'nin toplam port sayısı da arttığı için daha fazla anahtarın aynı anda boruhattına eklenerek aranması mümkündür. 2 parçaya ayrılan bir seviyeye aynı anda 2 x 2 = 4 adet anahtar araması yapılabilecektir. 4 parça için 8, 8 parça için 16 ve 16 parça için 32 anahtar aynı anda aynı seviyede bulunabilecektir. Bir seviye kaç parçaya ayrılırsa o sayının iki katı kadar anahtar o seviyede aranabilecektir. Parça sayısı derleme sırasında ayarlanabilecek şekilde kodlanmıştır. Şekil 3.12'de 2, 4 ve 8 parçalı hibrit kesilmiş ağaçların aynı anda arayabileceği en fazla anahtar sayısı gösterilmektedir. Şekilde gösterilen anahtarlar daha sade bir görsel olması amacıyla bellek bölmelerine sıralı olarak bölünmüştür, A1 anahtarı herhangi bir bölmeye geçiş yapabilir, listenin başında olması ilk bölmeye geçirileceği anlamını taşımaz, anahtarlar sırasızdırlar. Çoklama işlemi yapılmayan 3.12(a), iki kez, 3.12(b) dört kez ve 3.12(c) sekiz kez çoklama yöntemi ile çoklanmış ağaç ile eşit sayıda maksimum çevrim başına verime sahip olmaktadır. Maksimum çevrim başına verim aranacak anahtarların ağaçta birbirlerinden farklı bellek bölmelerinde saklanması durumunda oluşan verimdir. Çoklama Yöntemi'nin aksine her bir bellek bölmesinde farklı bir veri saklandığı için aynı bölmeye erişmek isteyen anahtar sayısı ikiden fazla ise beklemeler yaşanacak ve çevrim başına verim azalacaktır.

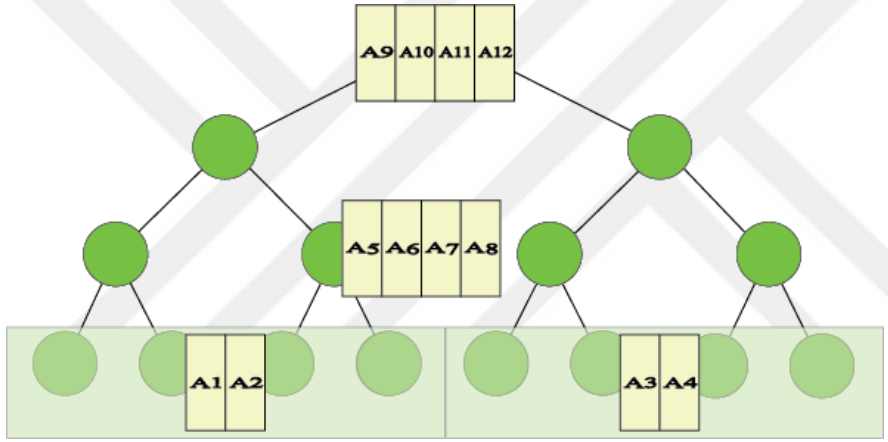
Hibrit Kesit Yöntemi ile ayrılmış bir CBST'de aranan bir anahtar boruhattı yöntemi ile yazmaç bölmelerinde aranır, son yazmaç seviyesinde de bulunamayan bir anahtar, son bakılan düğümün anahtar değerinden büyük ise o düğüme bağlı sağ bellek bölmesine, küçük ise o düğüme bağlı sol bellek bölmesine geçiş yapar. Bellek bölmelerinde boruhattı yöntemi ile aranmaya devam eder.

Parça sayısı değiştikçe bellek bölmelerini doldurmak için gerekecek Yazmaç Bölme sayısı da değişkenlik göstermektedir. Çizelge 3.3 BRAM'lerin kapasitelerinin tamamen kullanılması için gereken yazmaç bölme sayısını sırasıyla 2 ,4 ,8 ve 16 parçalı hibrit olmak üzere 11, 12, 13 ve 14 olarak göstermektedir.

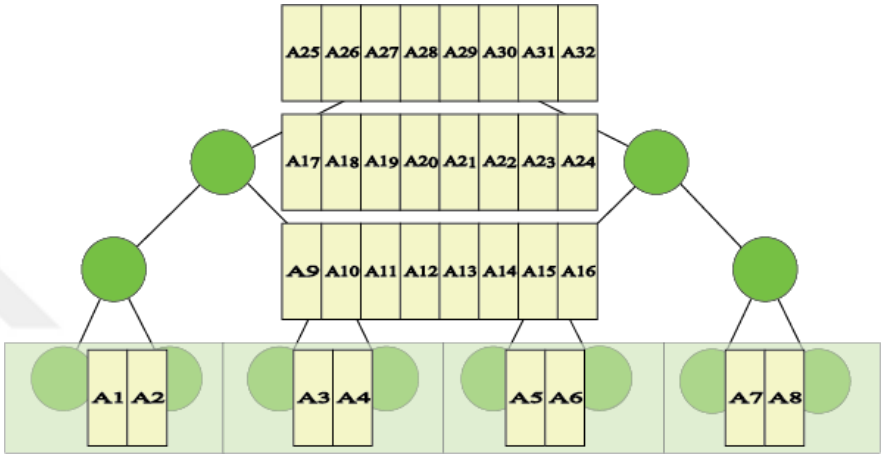
Çizelge 3.3: Yazmaç Bölmesi sayısına karşılık, ilk BRAM Bölmesi'nin saklaması gereken düğüm sayısı ve parça başına düşen düğüm sayısı

Yazmaç Bölmesi Sayısı	0. BRAM Bölmeleri Toplam Düğüm Sayısı	Parça Başına Düşen Toplam Düğüm Sayısı			
		Parça Sayısı			
		2	4	8	16
0	$2^0 = 1$	-	-	-	-
1	$2^1=2$	$2^0 = 1$	-	-	-
2	$2^2=4$	$2^1=2$	$2^0 = 1$	-	-
3	$2^3=8$	$2^2=4$	$2^1=2$	$2^0 = 1$	-
4	$2^4=16$	$2^3=8$	$2^2=4$	$2^1=2$	$2^0 = 1$
5	$2^5=32$	$2^4=16$	$2^3=8$	$2^2=4$	$2^1=2$
6	$2^6=64$	$2^5=32$	$2^4=16$	$2^3=8$	$2^2=4$
7	$2^7= 128$	$2^6=64$	$2^5=32$	$2^4=16$	$2^3=8$
8	$2^8 = 256$	$2^7= 128$	$2^6=64$	$2^5=32$	$2^4=16$
9	$2^9 = 512$	$2^8 = 256$	$2^7= 128$	$2^6=64$	$2^5=32$
10	$2^{10} = 1024$	$2^9 = 512$	$2^8 = 256$	$2^7= 128$	$2^6=64$
11	$2^{11}= 2048$	$2^{10} = 1024$	$2^9 = 512$	$2^8 = 256$	$2^7= 128$
12	$2^{12}= 4096$	$2^{11}= 2048$	$2^{10} = 1024$	$2^9 = 512$	$2^8 = 256$
13	$2^{13}= 8192$	$2^{12}= 4096$	$2^{11}= 2048$	$2^{10} = 1024$	$2^9 = 512$
14	$2^{14}= 16384$	$2^{13}= 8192$	$2^{12}= 4096$	$2^{11}= 2048$	$2^{10} = 1024$
n	2^n	2^{n-1}	2^{n-2}	2^{n-3}	2^{n-4}

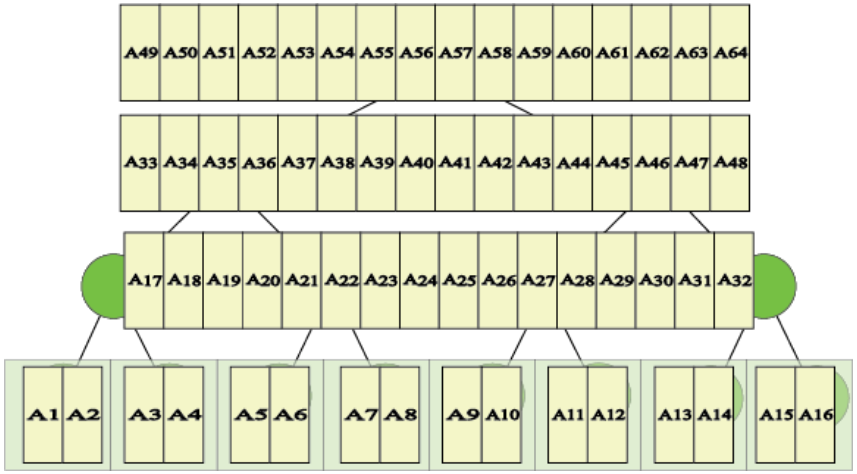
BRAM Bölmelerini tam kapasite doldurmak, daha az kaynak harcanmasına neden olsa da hibrit kesme yönteminde yaşanacak beklemelerin oranını arttıracaktır. Son yazmaç seviyesinde bir yazmacın sağ ve sol çocuklarının ayrı bir BRAM bölmesi oluşturması durumunda bu çocuklara erişecek anahtarlar farklı bölmelerin portlarına yönlendirilecek ama ilk bölmede birden fazla ağaç düğümü saklanması durumunda



(a) 2 Parçalı Hibrit Ağaç



(b) 4 Parçalı Hibrit Ağaç



(c) 8 Parçalı Hibrit Ağaç

Şekil 3.12: Hibrit ağaçların aynı anda arayabileceği en fazla anahtar sayısı

bu bölmeye daha fazla anahtarlar erişmek isteyecektir. Bu daha az kaynak kullanma ya da daha az bekleme arasında ödünleşmedir.

3.3 Tampon (Buffer) Ekleme

Hibrit Kesit Yöntemi'nde veri kopyalanması sorunu çözülmüş ve çoklama yöntemine kıyasla aynı maksimum çevrim başına verim elde edilmiştir. Fakat her bir bellek bölmesinde farklı bir veri saklandığı için aynı veriye erişmek isteyen ikiden fazla anahtar, çoklamada olduğu gibi bir kopya veri olmadığı için istenilen adrese erişemeyecektir. Bu erişim sorunu yeni anahtarların sisteme girişinde beklemeye yol açacak, boruhattında boşluklara neden olacak ve çevrim başına verimi azaltacaktır. Eğer sistemde birden fazla anahtar aynı bellek bölmesine erişmek isterken bekleme yapılmaz ise, bir önceki seviyede bulunan anahtarlar saat vuruşu ile bu seviyeye gelecek ve yerleştirilmesinde sorun yaşanmış anahtarların verisi kaybolacaktır. Veri kaybolmasını engellemek ve bekleme durumunu en aza indirmek için Tampon Ekleme optimizasyonunu öneriyoruz.

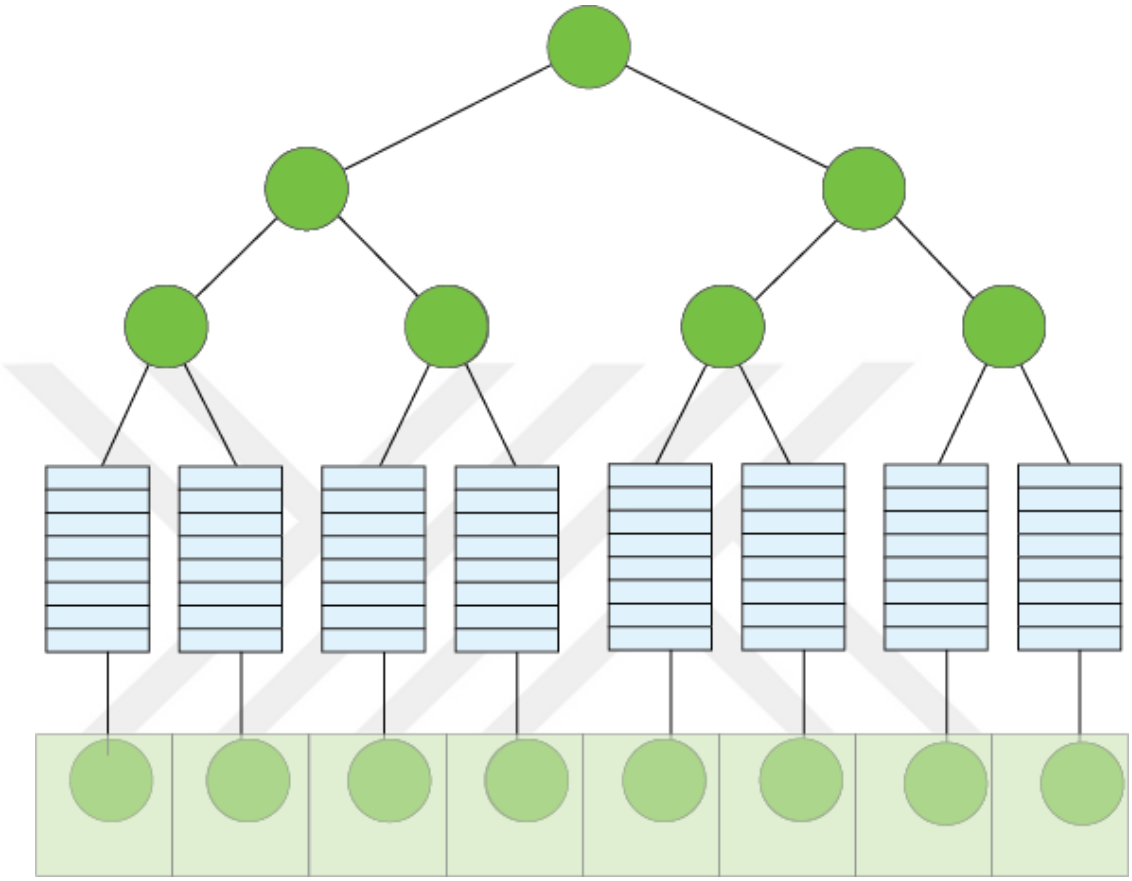
Tampon ekleme optimizasyonu ile sisteme yazmaç kademeleri ve bellek kademelerinin dışında tampon kademesi adı altında, son yazmaç bölmesi ile ilk bellek bölmesi arasında geçiş olarak kullanılacak tamponlar ekliyoruz. Her bir parçaya bir tane denk gelecek şekilde, 2 parçalı hibrit için toplam 2, 4 parçalı hibrit için 4 adet tampon oluşturuyoruz. Bu tamponların boyutları derleme esnasında ayarlanabilecek şekilde kodlanmıştır. Tamponların ağaçtaki yeri Şekil 3.13'te gösterilmiştir.

Her bir tampon istenildiği sayıda yuvadan (slot) oluşur. Bir anahtar yazmaç bölmesinden bellek bölmesine geçiş yapmak istediğinde, geçmek istediği bellek bölmesinin tamponunda uygun bir yuvaya yerleştirilir. Bu yuvaya hem anahtar, hem de erişmek istenen bellek adresi kaydedilir. Her saat vuruşu ile her bir bellek bölmesi kendisine ait tampondan < Anahtar, Adres > ikililerini alır ve aramaya devam edilir.

Tampon kullanımı sayesinde ikiden fazla anahtar aynı bölmeye erişmek istediğinde bu anahtarlar tampona kaydedilecek, bekleme olmayacak ve anahtarlar kaybedilmemiş olacaktır. Ancak, tamponların kapasitesini geçecek sayıda anahtar

tampona kaydedilmeye çalışılır ise beklemeler yaşanacaktır.

Gerçeklenen tamponlarda okuma, yazmaya göre önceliklidir. İlk olarak okuma işlemleri yapılmakta, sonra yazma işlemleri yapılmaktadır.



Şekil 3.13: Tamponların kademeler arasına yerleştirilmesi

Tamponlara anahtar kaydı işleme sürecinde bir sorunla karşı karıya kalınmaktadır. FPGA'in paralel olarak çalışma prensibinden ötürü, tüm anahtarların hangi bölmeye geçmeleri gerektikleri aynı anda bulunmaktadır, bu sebeple aynı bölmeye yerleşmek isteyen anahtarların tamponlara yerleştirilmesi sorun olmaktadır. Anahtarlar aynı anda hangi bölmeye geçmeleri gerektiğini buldukları için, kendileri dışındaki anahtarlar hakkında bir bilgiye sahip değildirler. Bu yüzden aynı anda aynı seviyede aranan anahtarlar A2, A6, A10 aynı bölmeye geçiş yapmak istemekte, hepsi de tamponun ilk boş olan yuvası Y1'e kaydedilmek isteyeceklerdir. Bu durum yanlış, eksik veri yazımına ve anahtar kaybolmasına yol açmaktadır ve bir çözüm

bulunulması gerekmektedir. Bu duruma çözmek için Doğrudan Eşleme ve Kuyruk Yoluyla Eşleme yöntemlerini öneriyoruz.

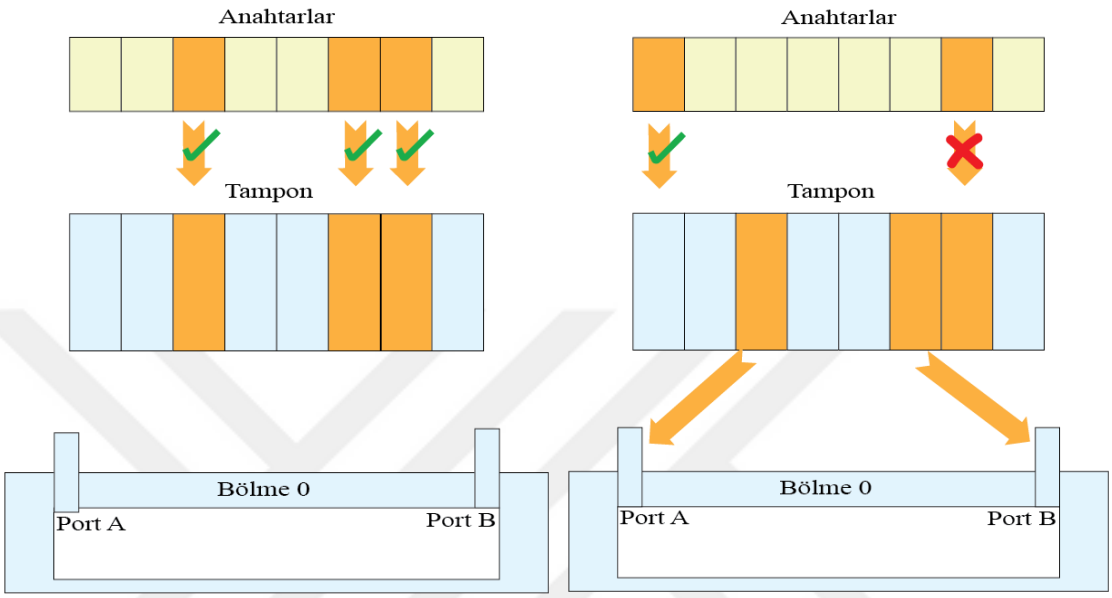
3.3.1 Doğrudan Eşleme

Doğrudan Eşleme yönteminde anahtarların hangi bölmeye geçmeleri gerektiği, o çevrimdeki anahtar listesinde kaçıncı sırada olduklarına göre belirlenir. Arama işlemine başlanırken anahtarlar gruplar halinde getirilmektedir. Bu getirilen anahtarların hepsinin grupta benzersiz bir sıra numarasına sahiptirler. Örnek olarak, her çevrimde 16 adet anahtarın bulunduğu gruplar sisteme getiriliyor ise, bu gruptaki ilk anahtar birinci sıradadır. Aynı anda birinci sırada olan birden fazla anahtar olamayacağı için sıra numaraları benzersizdir. Aynı çevrimdeki tüm diğer anahtarlar için de bu durum geçerlidir. Bu özellik kullanılarak, bir anahtar tampona yerleştirilmek istendiğinde anahtar grubundaki sıra numarasına doğrudan denk gelen tampon yuvası seçilmektedir. Her çevrimde kendi grubunun sırasında ikinci olan anahtarlar yerleşmesi gereken tamponun ikinci yuvasına yerleştirilecek, altıncı anaharlar tamponların altıncı yuvasına yerleştirilecektir. Aynı anda aynı yuvaya yerleşmek isteyen anahtar olmayacağından ötürü yerleştirme sırasında bir çakışma oluşmayacaktır.

Tasarlanan sistemde anahtarlar tamponlarda bulundukları sıraya göre bellek erişimi önceliğine sahiptirler. İlk yuvada bulunan anahtarlar en yüksek erişim önceliğine sahip iken son yuvada bulunan anahtarlar en az önceliğe sahiptir. Çift portlu belleklerde de önceliğin sıra olarak daha önde olan yuvalara verilmiş olmasının yanı sıra, tamponların ilk yarısındaki anahtarlar birinci portlarda ikinci yarısı ise ikinci portlarda aranacak şekilde gerçeklenmişlerdir. Her çevrimde en yüksek önceliğe sahip anahtarlar uygun bellek portlarına yönlendirilir. Bellek portuna yönlendirilmek için gereken önceliğe sahip olmayan anahtarlar tampon yuvalarında bir sonraki çevrimde öncelik sıralamalarına bakılmaları için bekletilmeye devam edilirler.

Bir anahtarın istediği tampon yuvasının önceki çevrimlerden ötürü dolu olması durumunda, anahtarlar istenilen yuvalara konulamayacağı için sisteme yeni anahtar girişi engellenir ve yazmaç bölmelerindeki aramalar durdurulur. Bellek bölmelerinde arama işlemi devam ettirilerek tamponlarda boş yuva sayılarının artışı sağlanır.

Durmaya sebep olan anahtarlar, uygun yuvalara erişim sağlayabildiklerinde yeni anahtar girişinin engellenmesi kaldırılır ve yazmaç bölmelerindeki aramalar devam ettirilir. Tampona anahtar eklenmesi Şekil 3.14a'da ve oluşabilecek çakışma durumu Şekil 3.14b'de gösterilmiştir.



(a) Tampona Anahtar Ekleme İşlemi **(b) Tamponda Oluşan Çakışma Durumu**

Şekil 3.14: Doğrudan Eşleme Yöntemi ile tampon işlemleri

Doğrudan Eşleme yöntemi duraklamaları azaltma konusunda başarılı olan sade bir yöntem olmasına karşın, bu yöntem ile tamponların kapasitesinden tamamen faydalanılamamaktadır. Bir tamponda kullanılmayan yuvalar olduğu halde, istenilen yuva dolu olduğu için duraklamalar oluşabilmektedir. Şekil 3.14b'de gösterilmiş olan bir grup anahtar içinde yedinci olan anahtar, tamponun yedinci yuvasına yerleşmek istediği için duraklamalar yaşanmaktadır. Bu tamponda boş olan beş adet yuva kullanılabilecek durumda olmasına rağmen bu alanlar kullanılamamış ve duraklamaya sebebiyet verilmiştir.

3.3.2 Kuyruk Yoluyla Eşleme

Doğrudan Eşleme yönteminde ortaya çıkan tamponların tam kapasite kullanılamama sorununa bir çözüm olarak Kuyruk Yoluyla Eşleme yöntemini öneriyoruz. Bu

yöntem doğrudan eşlemeye kıyasla daha karmaşık olduğu için ondan daha uzun çevrim zamanına gerek duymaktadır fakat tamponların daha etkili kullanılabilmesini sağlamaktadır.

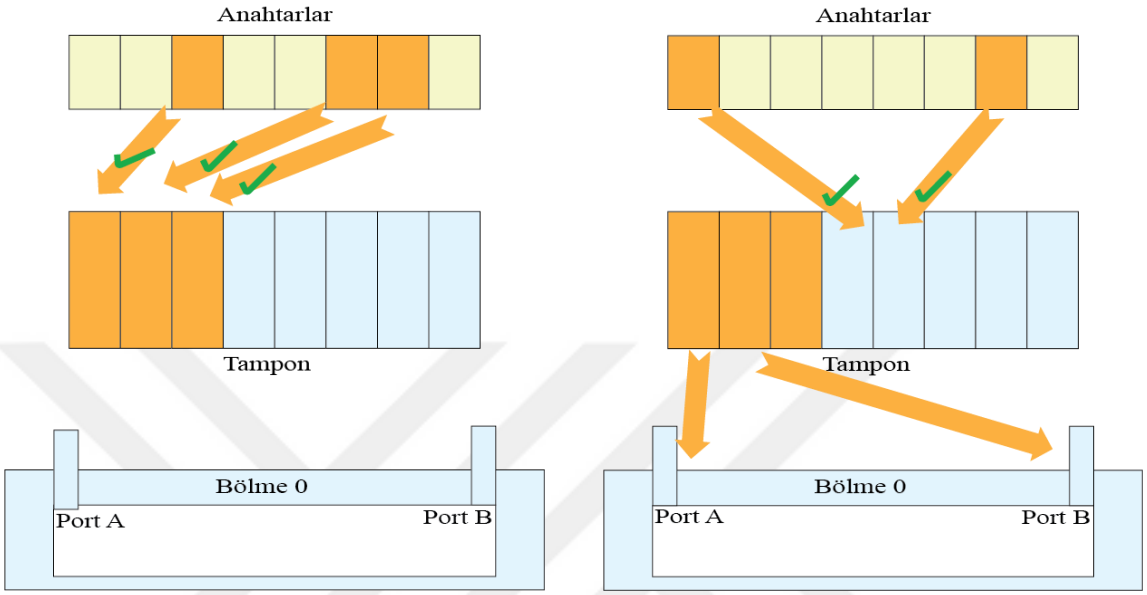
Kuyruk Yoluyla Eşleme yönteminde tamponların yanısıra her bir tampon için okuma ve yazma göstericileri (pointer) saklanmaktadır. Okuma göstericisi bellek bölmesine getirilecek sıradaki anahtarın tamponda yerini işaret eder, yazma göstericisi ise tampona anahtarların konulabileceği sıradaki boş alanı işaret eder. Bu yöntemde Doğrudan Eşleme yönteminin aksine anahtarlar arası öncelik tamponların yuva numarasına göre yüksekten düşüğe sıralı değildir, bir tampona ilk eklenen anahtar o tampondan ilk çıkarılan anahtardır. Tamponlara anahtar eklendikçe ve çıkarıldıkça okuma ve yazma göstericileri güncellenir. Okumalar, yazmalara göre önceliklidir.

Şekil 3.15: Anahtar ve etiket bilgilerinin gösterimi

Yazmaç bölmelerinde araması biten anahtarlar tamponlara yerleştirilmeden önce hangi ağacın tamponuna yerleştirilecekleri bilgisi ile etiketlenir. Bu sayede bir sonraki çevrimde, her bir tampona kaç adet anahtar yerleştirileceği bilgisi edinilir ve bu bilgi ile her bir anahtara bir numara verilir. Şekil 3.15'te anahtar ve etiketleri gösterilmiştir. Adres etiketi anahtarın hangi adreste aranması gerektiğini, tampon no hangi ağacın tamponuna yerleştirilmesi gerektiğini ve sıra no ise tampona yerleştirilirken kaçıncı sırada yerleştirileceğini belirtmektedir. Sıra no hesaplamasında bir anahtar kendi grubundaki anahtarlarla karşılaştırılır ve aynı tampona gidecek kaçıncı anahtar olduğu belirlenir. Örnek olarak, yazmaç bölmelerinde araması biten 3 anahtarın beş numaralı tampona gideceği belirlenmiştir. Bu anahtarların sıra noları grup sırasında en önce olan anahtardan başlanılarak 0,1 ve 2 olarak belirlenir.

Sıra no belirleme aşamasından sonra artık tüm anahtarların etiketlerinde kendisinden önce kaç tane daha anahtarın aynı tampona yerleştirileceğinin bilgisi saklanmaktadır, bu bilgiye bakılarak anahtarlar yazma göstericisinden gösterilen yuvadan itibaren

yerleştirilmeye başlanırlar. Anahtarlar yazma göstericisi ile gösterilen yuva numarası ile sıra nonun eklenmesi ile denk gelen yuvaya yerleştirilirler. Yerleştirilmeler paralel olarak gerçekleşmektedir. Yazma işlemlerinden sonra yazma göstericisi güncellenir.



(a) Tampona Anahtar Ekleme İşlemi **(b) Boş Olmayan Bir Tampona Anahtar Ekleme İşlemi**

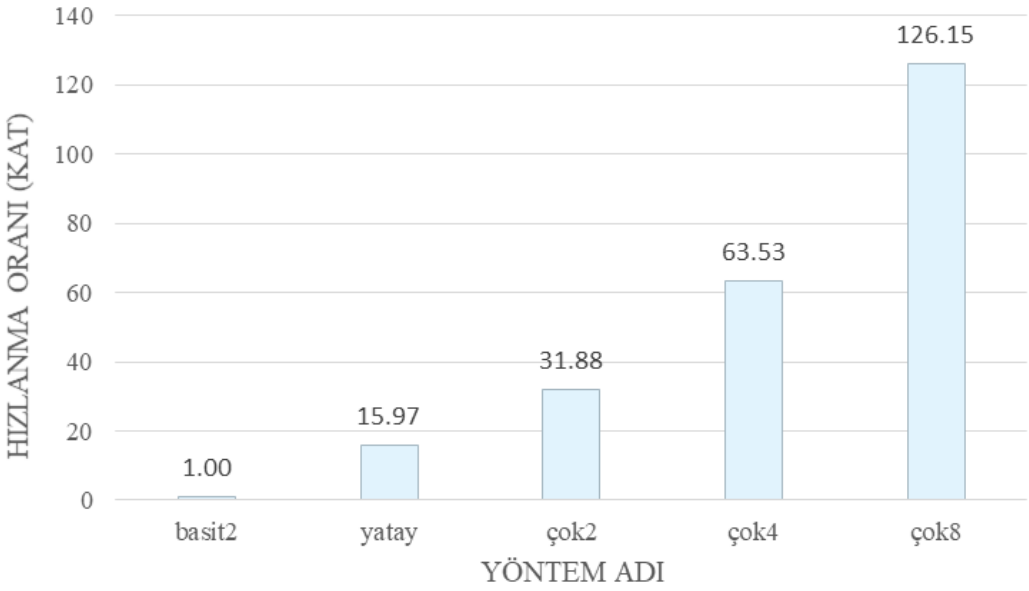
Şekil 3.16: Kuyruk Yoluyla Eşleme Yöntemi ile tampon işlemleri

Bir tampon dolu olduğu için daha fazla anahtar eklenmesine izin vermiyor ise duraklama gerçekleşir. Tamponun yazma göstericisi ile anahtarın sıra nosunun toplamı dolu olan bir yuvayı gösteriyor ise kapasitenin yetersiz olduğu belirlenmiş olur. Yeni anahtarların sisteme girişi engellenir, ağacın yazmaç bölmelerinde arama işlemleri durdurulur. Bellek bölmelerinde arama işlemi devam ettirilerek tamponlarda boş yuva sayılarının artışı sağlanır. Kapasite yeterli olduğunda yeni anahtar girişinin engellenmesi kaldırılır ve yazmaç bölmelerindeki aramalar devam ettirilir. Şekil 3.16'da, Şekil 3.15'teki durumun Kuyruk Yoluyla Eşleme yöntemi kullanıldığı zaman nasıl sonuçlandığı gösterilmiştir.

Her çevrimde her bir tamponun her bir okuma göstericisinin gösterdiği anahtarlar, çift portlu bellekler için ayrıca bir sonraki anahtarlar, tampondan alınarak belleklere getirilir, okuma göstericisi güncellenir.

4. TEST SONUÇLARI VE TARTIŞMA

Üçüncü bölümde açıklanan yöntemleri bir HLS (high level synthesis) dili olan Bluespec SystemVerilog (BSV) [39] ile gerçekledik. BSV kodu BSC (Bluespec compiler) kullanılarak Verilog HDL'e dönüştürüldü [40] ve Xilinx Virtex-7 VC709 platformunda yürütüldü. Bu platformda her biri 36 Kb büyüklüğünde olan 1470 adet BRAM bulunmaktadır [41].



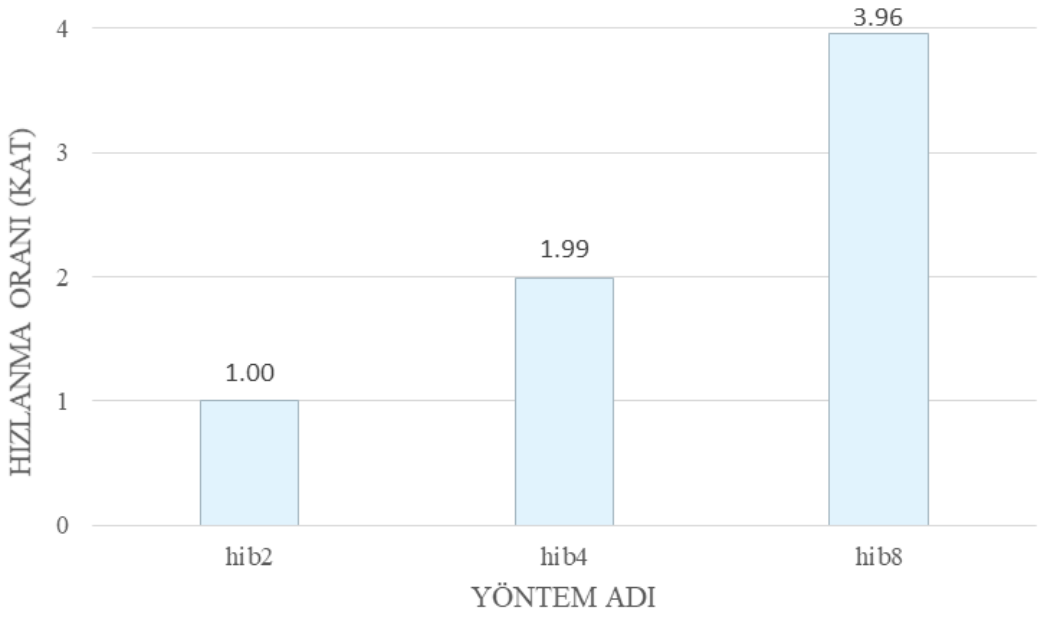
Şekil 4.1: basit2 baz alınarak yatay kesilmenin ve ağaç sayısı artışının hızlanmaya etkisi

Şekil4.1'de Çift portlu BRAM kullanılarak kodlanan basit yöntemi (basit2) baz alınarak yatay seviye bölünmesinin hızlanmaya etkisi ve ağaç sayısı artışının hızlanmaya etkisi gösterilmektedir. Bu şekilde alınan sonuçlarda 64K'lık bir anahtar listesi kullanılmış ve listedeki tüm anahtarlar ağaçların yaprak düğümlerinde bulunacak (en uzun arama sürecinden geçecek) şekilde seçilmiştir. Aramada kullanılan BST'nin yüksekliği 15 olup, 16 seviyeye sahiptir. Yatay kesit ve iki, dört

ve sekiz ağaçlı çoklama yöntemlerinde çift port kullanıldığı için basit yöntemin iki portlusu seçilmiştir. Yatay yöntemde boruhattı yöntemi kullanılmış ve tüm ağaç seviyelerine göre bölünmüş, boruhattının dolması gereken çevrim sayısından sonra ağacın her seviyesinde aynı anda arama yapılabilmiştir. Bu sebeple yeni bir anahtarın aranmaya başlanabilmesi için diğer anahtarın aramasının bitmiş olmasına gerek duyan basit2 yöntemine kıyasla yaklaşık 16 kat hızlanma yakalanmıştır. Çoklama yöntemlerinde ise çoklanan ağaç sayısı arttıkça listeden getirilen anahtar gruplarında bulunan anahtar sayısının artacağı, başka bir deyişle aynı anda arama işlemine başlayabilecek anahtar sayısı artacağı için hızlanma oranlarının da arttığını görüyoruz. Yatay kesimde tek bir kopya olduğu için, sekiz kez çoklanmış bir ağaca sahip çok8 yöntemine kıyasla daha yavaş olduğu görülmektedir. Kopya sayısı arttıkça kullanılabilecek port sayısı da artmaktadır, bu sebeple çevrim başına düşen verim de artmaktadır.

Hibrit kesme yönteminde dikey kesit ya da alt ağaç sayısını parça sayısı olarak adlandırmıştık. Dikey kesit sayısının artışının hızlanma oranına etkisi Şekil4.2'de gösterilmektedir. 64K'lık bir anahtar listesi kullanılmış olup bu anahtar listesinde bulunan anahtarların hepsi yaprak düğümlerinde bulunmaktadır. Ayrıca kullanılan listedeki anahtarlar farklı ağaçlarda olacak şekilde belirlenmiştir, duraklama olmamaktadır. Duraklama olan durumlar üzerinde yeniden durulacaktır. Duraklama olmadığı zaman Şekil 4.2'de de görüldüğü gibi parça sayısı arttıkça her çevrim sistemde aramaya başlanabilecek en fazla anahtar sayısı da arttığı için, çevrim başına verimde artış gözükmektedir. Hib2'de iki alt ağaç bulunduğu için bir seviyede aynı anda en fazla dört anahtar aranabilmektedir. Hib4'te bu sayı sekiz olmakta ve hib8'de ise on altı olmaktadır. Bu sebeple hib8 hib4'ten ve hib4 hib2'den yaklaşık iki kat daha hızlanmıştır. Bu da alt ağaç sayısı arttıkça hızlanmanın da arttığını göstermektedir.

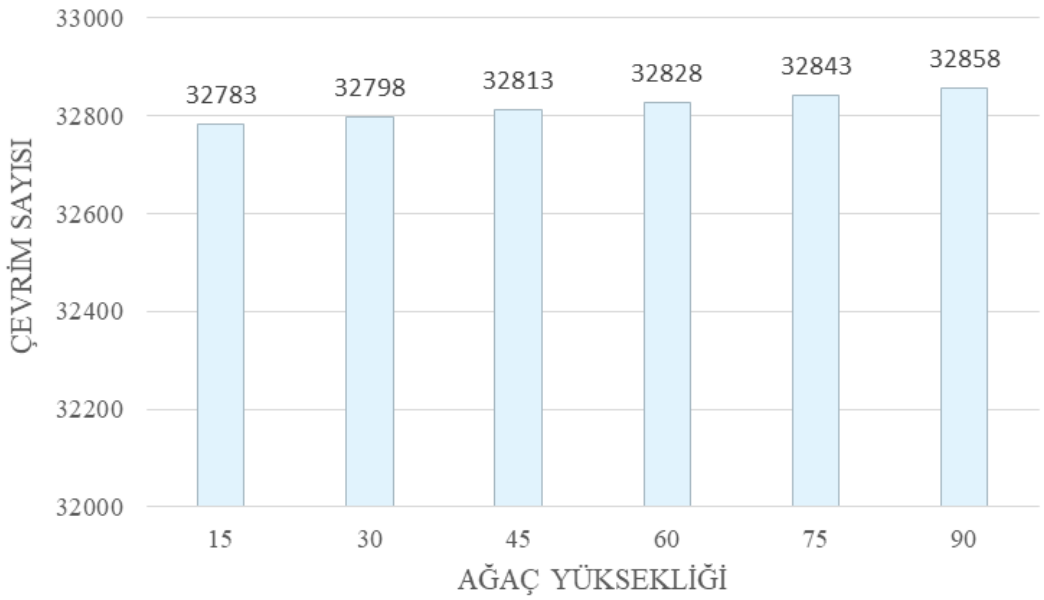
BST ağacının büyüklüğünün yöntemlerimize etkisini incelemek için farklı yüksekliklere sahip ağaçlarda Şekil4.1'de kullanılan anahtar listesini yürüttük. Her yöntemimizde ağaçların seviyelendirildiği için bu karşılaştırmada yatay kesme yöntemini kullandık. Ağaç yüksekliklerinin, 64K'lık anahtar listesinin tamamlanması için gereken çevrim sayısına etkisi Şekil 4.3'te gösterilmiştir. Boruhattı dolduktan



Şekil 4.2: hibrit yöntemde parça sayısı değişiminin hızlanmaya etkisi

sonra çevrim başına düşen verimin iki anahtar olmasına, ağaçların yüksekliği bir etki etmemektedir. Yüksekliği 15 olan bir ağaçta da boruhattı dolduktan sonra her çevrimde iki anahtar bulunacakken, yüksekliği 90 olan bir ağaçta da boruhattı dolduktan sonra her çevrimde iki anahtar bulunacaktır. Yükseklik arttığı için ağacın tüm seviyelerinin dolması için gereken çevrim sayısında artış olacaktır fakat her bir çevrimde bir seviye dolduğu için bu artış şekilde de görüldüğü gibi çok büyük olmayacaktır. Yükseklik arttığı için tek bir anahtarın bulunması için gereken çevrim sayısı yani gecikme artmış olacaktır fakat biz bu çalışmamızda gecikmeye değil verime önem veriyoruz. Ağaç yüksekliği verime bir etkide bulunmamıştır, her ağaç dolduktan sonra çevrim başına iki anahtar sonuç vermektedir.

Ağaç yüksekliği verime etki etmese de, ağaç büyüdüğü için daha fazla düğüme sahip olacak ve ağacı saklamak için ihtiyaç duyulan kapasite artacaktır. Şekil 4.4'te önerilen yöntemler için 10, 15 ve 30 yüksekliğine sahip ağaçlar kullanıldığında saklanması gereken düğüm sayıları gösterilmiştir. Çoklama yöntemlerinde ağaç kopyalandığı için hafızada tutulması gereken düğüm sayısı da büyük bir artış göstermiştir. Bellek bölmesinde tutulan seviyeler kopyalandığı için yazmaç bölmelerinin sayısı arttırılarak ve böylece bellek bölme seviyeleri azaltılarak daha az



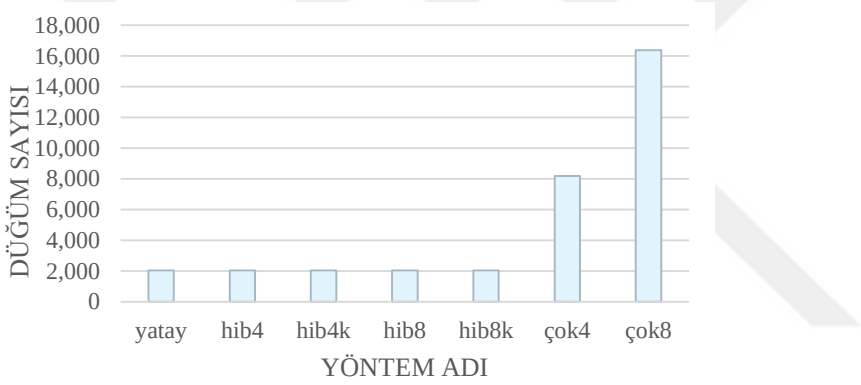
Şekil 4.3: Yatay kesme yönteminde ağaç yüksekliğinin çevrim sayısına etkisi

kapasite kullanılabilir. Lakin ağaç büyüdükçe bir seviye için gerekli olan yazmaç sayısı da üssel olarak artmaktadır. Ağacın yüksekliği 10 iken saklanması gereken ekstra düğüm sayısı yaklaşık 14.000 iken, ağacın yüksekliği 30'a çıkarıldığında saklanması gereken düğüm sayısı yaklaşık 15.000.000'u bulmaktadır.

Sunduğumuz yöntemlerin birbirleriyle hız karşılaştırmasını farklı anahtar listeleriyle yaptık. Bu karşılaştırmada;

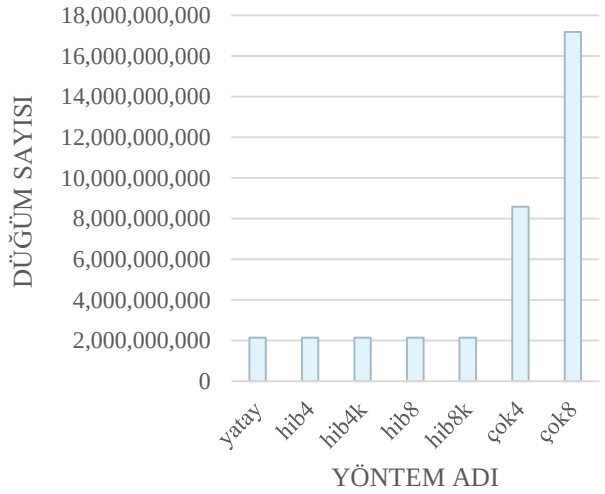
- Yatay Kesme (yatay etiketi kullanılmıştır)
- 4 Ağaçlı Çoklama (çok4)
- 8 Ağaçlı Çoklama (çok8)
- Hibrit, 4 Alt Ağaçlı, Doğrudan Eşlemeli (hib4)
- Hibrit, 4 Alt Ağaçlı, Kuyruk Yoluyla Eşlemeli (hib4k)
- Hibrit, 8 Alt Ağaçlı, Doğrudan Eşlemeli (hib8)
- Hibrit, 8 Alt Ağaçlı, Kuyruk Yoluyla Eşlemeli (hib8k)

Yöntemleri kullanılmıştır. Yatay kesme gerçeklememiz [1] 'de gösterilen gerçeklemeye benzemektedir ve sonuçlarımızda bu yöntem baz alınacaktır. Hibrit ve



(a) h= 10

(b) h= 15



(c) h=30

Şekil 4.4: Farklı yüksekliklere (h) sahip ağaçlar için yöntemlerin saklaması gereken düğüm sayısı

çoklama implementasyonlarında ağaç sayılarının 4 ve 8 olarak seçilmiştir, bu bir sorunluluk değildir, açıklamalarında da belirttiğimiz gibi derleme zamanında tamamen değiştirilebilirler. Kullandığımız ağaç 15 yüksekliğindedir. Her bir yöntemde yazmaç ekleme yöntemini kullanarak ilk seviyeleri yazmaçlara, diğer seviyeleri ise belleklere yerleştirdik. Yazmaçlara yerleştirilen düğüm sayısını adil olması açısından eşit tuttuk. Karşılaştırmada yöntemleri farklı açılardan karşılaştırmamız için 64K ve 256K çeşitleri olacak şekilde farklı anahtar listeleri oluşturduk. Bu anahtar listeleri (a) her bir anahtarı aynı olan: Eşit, (b) anahtarların hepsinin rasgele seçildiği: Karma ve (c) anahtarların hepsinin farklı ağaçlardan seçildiği: Ayrık olarak seçilmiştir. Eşit anahtar listesinde tüm anahtarlar aynı olduğu için hepsi aynı arama yolunu izlemek zorunda kalacaktır, her anahtar aynı bellek bölmesine erişmek istediği için aynı tampona yerleştirilmeye çalışacak, tampon dolacak ve duraklamalar oluşacaktır. Bu sebeple eşit anahtar listesi en kötü senaryoyu göstermektedir. Karma listedeki anahtarlar rasgele seçildikleri için bize ortalama bir senaryo durumunda ne olabileceğini belirtecektir. Ayrık listede tüm anahtarlar farklı ağaçlardan seçildikleri için, farklı yollar izleyecek aynı bellek bölmesine erişmeye çalışılmayacağı için tampon dolma sorunları oluşmayacak ve duraklama yaşanmayacaktır. Bu nedenle ayrık liste bize en iyi senaryoyu gösterecektir. Kullanıcının aratmak istediği anahtar listesi değişiklik gösterdikçe hibrit ağaçlardan alınan sonuçlar tampon kullanımına bağlı farklılık gösterecektir. Sınırlı sayıda anahtar listesi olmadığı ve bu yüzden her bir listeyi yürütmek mümkün olmayacağı için en iyi, en kötü ve ortalama senaryoları seçtik.

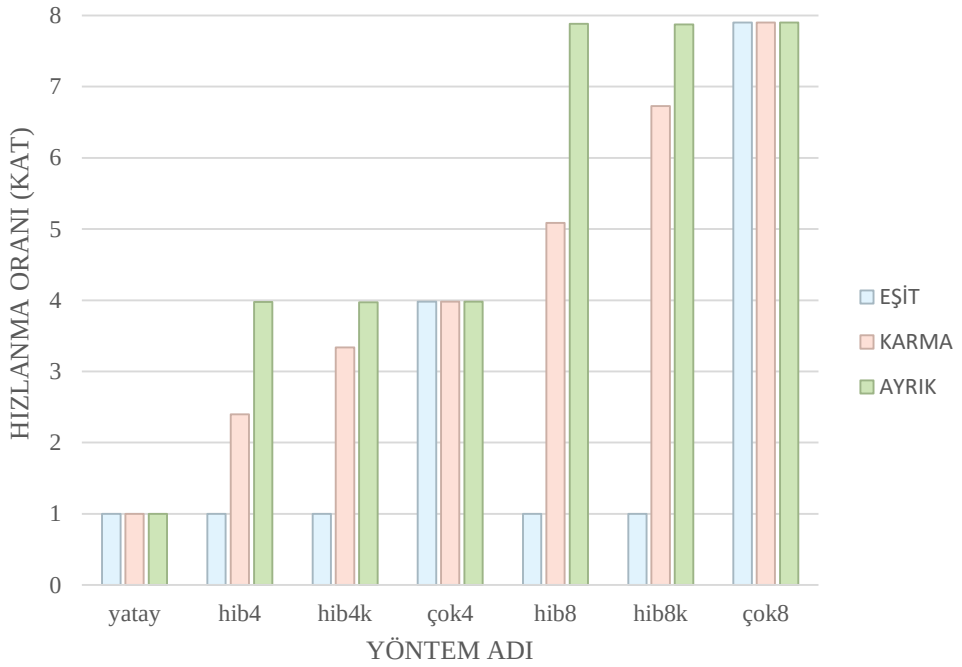
Gerçeklediğimiz yöntemlerde açıkladığımız anahtar listelerini yürüttük ve listelerdeki tüm anahtarların bulunması için gereken çevrim sayısını kaydettik. Şekil 4.5'te yatay yöntem baz alınarak, yöntemlerin hızları gösterilmiştir. 4.5a 64K boyutlu anahtar listesi için sonuçları gösterirken 4.5b ise 256K boyutlu liste için sonuçları göstermektedir. 4.5a ve 4.5b grafiklerinde farklı büyüklükte anahtar listeleri kullanılmış olsa bile, her bir yöntem yaklaşık olarak çevrim başına aynı verimi verdiği için büyük bir farklılık gözlenmemektedir. Örnek olarak çok8 her zaman yatay'dan yaklaşık 8 kat daha hızlı olacaktır çünkü 8 katı kadar daha fazla anahtar arayabilecek port sayısına sahiptir. Bu kat sayısının tam olarak 8 kat değil de

sonsuza giderken limitinin 8 olması, arama başlangıcındaki dolma süresinden ötürüdür.

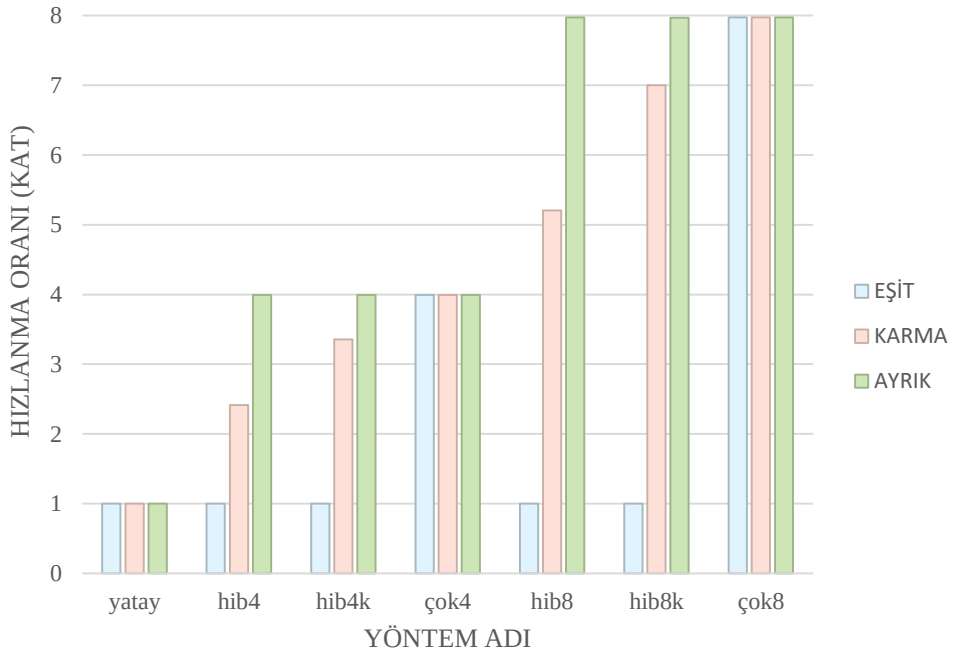
Şekil 4.5'te de görüldüğü gibi çok8 tüm anahtarları aramak için diğer yöntemlere kıyasla daha az çevrime ihtiyaç duymaktadır. Bunun nedeni çok8'in diğer tüm yöntemlere göre daha fazla porta sahip olmasıdır, ve aynı anda 8x2 = 16 adet anahtar arayabilmektedir. Hib8'de çok8 kadar porta sahip olsa da, bir bellek bölmesinden sadece bir tane olduğu için ve bu nedenle aynı bölmede aranmak istenen anahtarların aynı bölmeye gitmesi gerektiğinden, tamponları doldurarak duraklamaya neden olmasından ötürü çok8 ile farklı sonuçlar vermiştir. Çoklama yöntemleri ve yatay yöntem duraklama durumu yaşamadıklarından ötürü üç farklı listede de aynı verimi göstermiştir. Duraklama yaşama olasılığı olan hibrit yöntemlerin ise farklı listelerde farklı verimleri alabildiği gözlemlenmektedir. Eşit anahtar listesinde, her bir anahtar aynı yolu kullanmak istediği ve aynı bellek bölmesine erişmek istediği için sadece iki port üzerinden işlem yapılabilmiştir, bundan dolayı çevrim sonuçları yatay yöntemin sonuçlarına benzemiştir. Karma listede anahtarlar rasgele olduğu için, duraklamalar yine de yaşanmış fakat bu duraklama sayısı eşit listesindeki kadar olmamıştır. Çünkü aynı bellek bölmesine gitmek isteyen anahtarlar olduğu gibi farklı bölmelere gitmek isteyen anahtarlar da bu listede mevcuttur. Ayrık listesinde ise tüm anahtarlar farklı bir bellek bölmesine erişmek istediği için duraklama oluşmamıştır ve hibrit yöntemler aynı anda tüm portlarını kullanabilmişlerdir bu sebeple de alınan çevrim sonuçları çoklama yöntemlerinin sonuçlarıyla benzerlik göstermektedir.

Hibrit yöntemin en kötü çizgisi (worst line) yatay yöntem, en iyi çizgisi (best line) ise çoklama yöntemidir. Grafiklerde hibrit kuyrukla eşleme yönteminin, doğrudan eşleme yöntemine göre daha hızlı olduğu görülmektedir bunun nedeni kuyruk yönteminde tamponlar daha etkin kullanıldığı için duraklama sayısının daha az olmasıdır.

Verimi yaklaşık 16 anahtar/çevrim olan çok8 en kullanışlı yöntem olarak gözükse de Şekil4.4'te gösterilen ek düğümler göz önüne alınmalı ve çok8'in diğer yöntemlere göre daha fazla kapasite gerektirdiği unutulmamalıdır. Yöntemlerin kullanmış olduğu kaynak sayısı Şekil 4.6'da gösterilmiştir. Bu şekilde de çoklama yöntemlerinin diğer yöntemlere kıyasla daha fazla bellek harcadığı gözükmektedir. Hibrit yöntemler



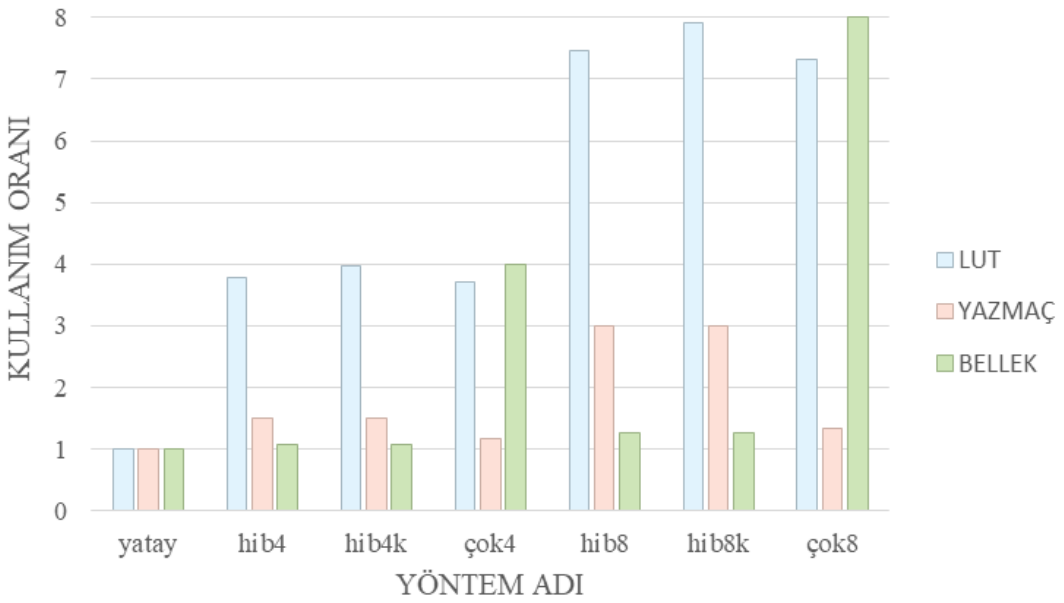
(a) 64K boyutlu anahtar listesi



(b) 256K boyutlu anahtar listesi

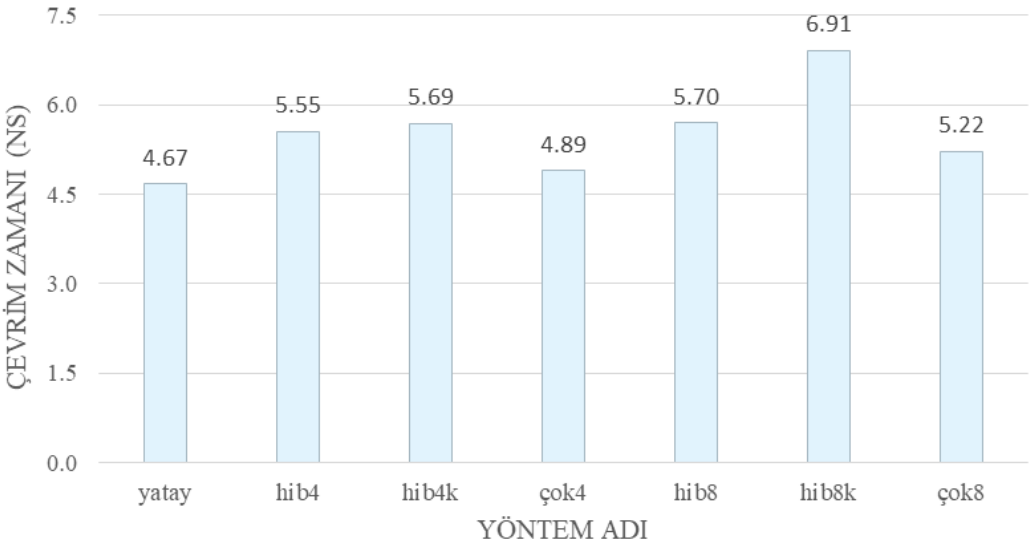
Şekil 4.5: Farklı anahtar listeleri yürütülmesi sonucu yatay yöntem baz alınarak hızlanma oranları

yatay yöntem ile benzer bellek sayısına ihtiyaç duysa da, tampon kullanmalarından ötürü daha fazla LUT'a ve yazmaca ihtiyaç duymaktadır. Kuyruk yoluyla eşleme yöntemi, doğrudan eşlemeye göre daha fazla kaynağa ihtiyaç duymuştur. Doğrudan yöntemde göstericiler tutulmadığından ve anahtarlar arası karşılaştırma yapılmadığından, doğrudan eşlemenin kuyruk eşlemesi kadar donanıma ihtiyacı olmamıştır. Daha az donanım istediğinden ötürü doğrudan eşleme, kısıtlı kaynağa ya da daha dağınık anahtar listelerine sahip olunduğunda kullanılabilir. Anahtarlar daha az aynı belleğe geçiş yapmak istiyorsa, tamponda yuva çakışması olma ihtimali de daha düşük olmaktadır.

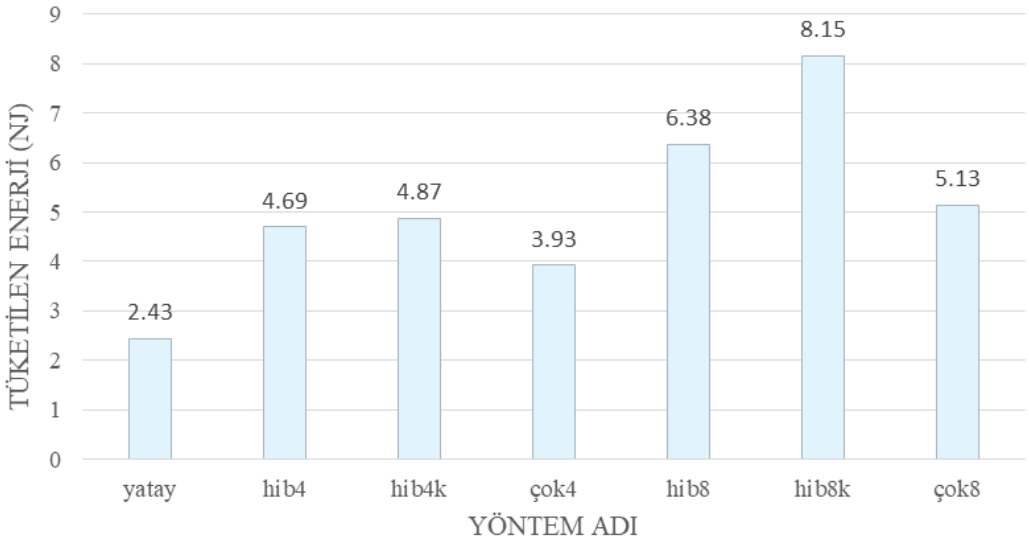


Şekil 4.6: Yatay yöntem baz alınarak kullanılan kaynaklar

Şekil 4.7'de önerilen yöntemler için gerekli olan çevrim zamanı ve yöntemlerin tükettikleri enerji gösterilmiştir. Şekil4.7a'da Hibrit yöntemlerin diğer yöntemlere kıyasla daha uzun çevrim zamanına sahip oldukları ve Şekil4.7b'de ise daha fazla enerji tükettikleri gözlemlenmektedir. Kullanılan tamponlar ve bu tamponlara anahtar konulması için gereken hesaplamalardan ötürü daha yüksek değerler görülmektedir. Özellikle kuyruk yoluyla eşleme yönteminde tamponlara anahtar yerleştirmek için daha fazla hesaplama yapılması enerji tüketimini arttırmıştır.



(a) Yöntemlerin çevrim zamanı



(b) Yöntemlerin tükettiği enerji

Şekil 4.7: Yöntemler için çevrim zamanı ve tüketilen enerji sonuçları

Tamponlara yerleştirilme yapılırken anahtarların diğer anahtarlarla karşılaştırılma işleminden sonra tamponlara yerleştirilmesi bu yöntemin en uzun yolu (critical path) olduğu için de frekansda azalma oluşmuştur. Bu yerleştirilme işlemi de boruhattı yöntemi ile kullanılır ise, en uzun yol daha kısalacağı için bir frekans hızlanması mümkündür fakat boruhattı yöntemi ile tüm anahtarların her çevrimde sadece başka

bir anahtarla karşılaştırılması durumunda yeni tamponlara ihtiyaç duyulmaktadır, çünkü bu karşılaştırılmalar yapılırken her bir çevrimde yeni anahtarlar gelmektedir. Yazmaç bölmeleri ve bellek bölmeleri olduğu gibi, karşılaştırma bölmeleri adı altında en fazla aranabilecek anahtar sayısına göre seviyeleri arttırılabilecek bir yöntem ile gerçeklenebilir fakat çok fazla kaynak gerektireceği için kuyruk ile eşlemenin gerçeklemesini bu şekilde yapmadık. Şekil4.7'de kullanılan hibrit yöntemlerde tamponların büyüklüğü bir çevrimde getirilebilecek en fazla anahtar sayısı olarak belirlenmiştir. Bu da hib4 ve hib4k yöntemlerinde 8 ve hib8 ve hib8k yöntemlerinde 16 olarak tampon büyüklüğü seçildiği anlamına gelmektedir.

5. SONUÇ VE ÖNERİLER

Bu tezde ikili Arama Ağacındaki arama işlemini hızlandırmak için FPGA'den yararlandık ve yatay kesme, çoklama, hibrit kesme yöntemlerini ve yazmaç , tampon ekleme iyileştirmelerini önerdik. Tampon eklemesi durumunda oluşan çakışan anahtarlar sorununa çözüm olarak iki farklı yöntem sunduk.

Çoklama yöntemi istikrarlı olarak en fazla verimi kazandıran yöntem olmasında rağmen diğer yöntemlere kıyasla yaptığı kopyalamalardan ötürü daha fazla alan gerektirdiğini gözlemledik. Daha az yer kaplaması için farklı seviyelerde kopyalama yapılabileceğini önerdik. Çoklama yönteminin ve yatay kesit yöntemlerinin verimlerinin sabit kaldığını ve verilen anahtarlara bağlı olarak verimlerinde bir değişiklik olmadığını belirttik. Hibrit yöntemlerde ise verilen anahtar setine bağlı olarak duraksamaların olabileceğini ama kopyalama yapmadan çoklama yönteminin performansına yakınsanabildiğini gösterdik.

Önerilen tampona anahtar ekleme yöntemleri içinde kuyruk eşleme yönteminin duraklamaları daha etkili bir şekilde azalttığını fakat daha karmaşık bir yapısı olduğundan ötürü doğrudan eklemeye kıyasla daha fazla yer tutup daha az bir frekansla çalışabileceğini gözlemledik.

Yer problemi olmayacak durumlarda çoklama yöntemi seçilerek 8 kat hızlanma yakalanabilir. Yetersiz yer sorunu ortaya çıkıyor ise hibrit yöntem seçilebilir. Eğer aranacak anahtarların hakkında bir bilgi elimizde ise, bu bilgiyi kullanarak hibrit yöntemin tampona ekleme eklentilerinden birisi seçilebilir. Anahtar listesindeki anahtarların aynı rotayı izleme oranı daha düşük ise doğrudan eşleme seçilerek daha az kaynak kullanılır ve arama yapılır, fakat bu oran daha yüksek ise kuyruk yolu seçilerek daha az bekleme oluşnası sağlanabilir.

Bu çalışma daha fazla geliştirilmek istenir ise bu ağaca ekleme ve silme işlemleri eklenebilir. Daha fazla alan sağlamak için harici bir bellek kullanılarak, gerektiğinde

dahili bellekteki veriler ile değiş tokuş yapılarak ağaç araması yapılabilir. Ayrıca aramalarımızın daha az güç gerektirmesi için voltaj düşürücü tekniklerden [42,43] yararlanılabilir.

KAYNAKLAR

[1] **Chodowiec, P., Gaj K.,** (2003). Very Compact FPGA Implementation of the AES Algorithm, *Cryptographic Hardware and Embedded Systems - CHES 2003*, 2779, 319-333.

[2] **Zhang, C., Li, G., Sun, G., Guan, Y., Xiao, B., Cong, J.,** (2015). Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks, *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 161-170.

[3] **Monmasson, E., Cirstea, M.N.,** (2007). FPGA Design Methodology for Industrial Control Systems—A Review, *IEEE Transactions on Industrial Electronics*, 54, 1824-1842.

[4] **Jin, R., Agrawal, G.,** (2003). Communication and Memory Efficient Parallel Decision Tree Construction, *Proceedings of the 2003 SIAM International Conference on Data Mining*, 119-129.

[5] **Al-Furajh, I., Aluru, S., Goil, S., Ranka, S.,** (2000). Parallel construction of multidimensional binary search trees, *IEEE Transactions on Parallel and Distributed Systems*, 11, 136-148.

[6] **Bronson, N.G., Casper, J., Chafi, H., Olukotun, K.,** (2010). A practical concurrent binary search tree, *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 45, 257-268.

[7] **Fix, J., Wilkes, A., Skadron, K.,** (2011). Accelerating Braided B+ Tree Searches on a GPU with CUDA, *2nd Workshop on Applications for Multi and Many Core Processors: Analysis, Implementation, and Performance (A4MMC), in conjunction with ISCA.*

[8] **Suda, R., Rocki, K.,** (2009). Parallel Minimax Tree Searching on GPU, *International Conference on Parallel Processing and Applied Mathematics*, 6067, 449-456.

[9] **Fix, J., Wilkes, A., Skadron, K.,** (2011). Exploiting Coarse-Grained Parallelism in B+ Tree Searches on an APU, *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, 240-247.

[10] **Jiang, W., Prasanna, V.K.,** (2009). A FPGA-based Parallel Architecture for Scalable High-Speed Packet Classification, *20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, 24-31.

[11] **Qu, Y., Prasanna, V.K.,** (2013). Scalable high-throughput architecture for large balanced tree structures on FPGA, *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays, ACM,* 278-278.

[12] **Saqib, F., Dutta, A., Plusquellic, J., Ortiz, P., Pattichis, M.S.,** (2015). Pipelined Decision Tree Classification Accelerator Implementation in FPGA (DT-CAIF), *IEEE Transactions on Computers,* 64, 280-285.

[13] **Owaida, M., Zhang, H., Zhang, C., Alonso, G.,** (2017). Scalable inference of decision tree ensembles: Flexible design for CPU-FPGA platforms, *27th International Conference on Field Programmable Logic and Applications (FPL), IEEE,* 1-8.

[14] **Owaida, M., Alonso, G.,** (2018). Application partitioning on FPGA clusters: inference over decision tree ensembles, *International Conference on Field-Programmable Logic and Applications (FPL).*

[15] **Qu, Y., Prasanna, V.K.,** (2013). High-performance pipelined architecture for tree-based IP lookup engine on FPGA, *International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, IEEE,* 114-123.

[16] **Qu, Y.R., Prasanna, V.K.,** (2014). Scalable and dynamically updatable lookup engine for decision-trees on FPGA, *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays, ACM,* 278-278.

[17] **Yang, Y.E., Prasanna, V.K.,** (2010). High throughput and large capacity pipelined dynamic search tree on fpga, *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays, ACM,* 83– 92.

[18] **Solworth, J.A., Reagan, B.B.,** (1995). Parallelizing tree algorithms: Overhead vs. parallelism, *Languages and Compilers for Parallel Computing, LCPC 1994,* 892, *438-452.*

[19] **Feng, J., Naiman, D.Q., Cooper, B.,** (2011). Parallelized Binary Search Tree, *Journal of Information Technology & Software Engineering,* 1, 1-5.

[20] **Browning, Sally A.** (1979). Computations on a Tree of Processors. *Caltech Conference on VLSI,* 453-478.

[21] **Stolfo, S.J., Miranker, *D.P.*,** (1986). Binary tree parallel processor, *United States Patent,* No: US4860201A Tarih: 02.09.1986.

[22] **Kröll, B., Widmayer, P.,** (1994). Distributing a Search Tree Among a Growing Number of Processors, *Proceedings of the 1994 ACM SIGMOD international conference on Management of data,* 23, 265-276

[23] **Ferguson, C., Korf, R.E.,** (1988). Distributed tree search and its application to alpha-beta pruning, *Proceedings of the Seventh AAAI National Conference on Artificial Intelligence,* 128-132.

[24] **Sherwood, T., Varghese, G., Calder, B.,** (2003). A pipelined memory architecture for high throughput network processors, *30th Annual International Symposium on Computer Architecture*, 31, 288-299.

[25] **Karypis, G., Kumar, V.,** (1994). Unstructured tree search on SIMD parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 10, 1057-1072.

[26] **Powley, C., Ferguson, C., Korf, R.E.** (1991). *Parallel tree search on a SIMD machine. Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, 249-256.

[27] ***Archer, C.J., Lyname, B.E., Ricard, G.R.,*** (2006). Parallel execution of operations for a partitioned binary radix tree on a parallel computer, *United States Patent*, No: US7779016B2 Tarih: 14.06.2006.

[28] **Zeuch, S., Huber, F., Freytag, J.C.** (2014). Adapting Tree Structures for Processing with SIMD Instructions, *17th International Conference on Extending Database Technology (EDBT)*, 97-108.

[29] **Kim, C., Chhugani, J., Satish, N., Sedlar, E., Nguyen, A.D., Kaldewey, T ve diğ.** (2010). FAST: fast architecture sensitive tree search on modern CPUs and GPUs. *ACM SIGMOD/PODS Conference*, 399-350.

[30] **Lim, H., Lee, B.,** (2004). A new pipelined binary search architecture for IP address lookup. *Workshop on High Performance Switching and Routing*, 2004. HPSR, 86-90.

[31] **Xilinx, Field Programmable Gate Array,** https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html010, erişilen tarih: 24.06.2019

[32] **National Instruments, FPGA Fundamentals,** http://www.ni.com/es-es/innovations/white-papers/08/fpga-fundamentals.html, erişilen tarih: 24.06.2019

[33] **Xilinx**, *Block Memory Generator v.8.4, Logicore IP Product Guide*, Vivado Design Suite, PG058, 4 Ekim 2017

[34] **Xilinx**, *7 Series FPGAs Memory Resources User Guide*, UG473 (v1.13), 5 Şubat 2019

[35] **Pfaff, B.,** *An Introduction to Binary Search Trees and Balanced Trees,* Libavl Binary Search Tree Library, Volume 1: Source Code, v.2.0.2.

[36] **Cormen, T.H., Leiserson C.E., Rivest, R.L., Stein, C.** *Introduction to Algorithms, 3rd Edition, THE MIT Press, Cambridge*, Massachusetts, Massachusetts Institute of Technology (2009)

[37] **COP 3530, Data Structures Lecture Notes, FIU**, https://users.cs.fiu.edu/~giri/teach/3530/f16/Lectures/Lec6-Trees.pdf, erişilen tarih: 24.06.2019

[38] **Walulya, I. (2018).** *On Design and Applications of Practical Concurrent Data Structures* (doktora tezi). Adres:

https://research.chalmers.se/publication/505740/file/505740_Fulltext.pdf

[39] **Bluespec**, https://bluespec.com/54621-2/ , erişilen tarih: 24.06.2019

[40] **Bluespec Documentation Page,** http://wiki.bluespec.com/Home/BSVDocumentation, erişilen tarih: 24.06.2019

[41] **Xilinx**, *VC709 Evaluation Board for the Virtex-7 FPGA*, User Guide, UG887 (v.1.6) March 11, 2019

[42] **Salami, B., Unsal, O., Cristal, A.,** (2018) . A Demo of FPGA Aggressive Voltage Downscaling: Power and Reliability Tradeoffs, *International Conference on FieldProgrammable Logic and Applications (FPL)*, 451.

[43] **Salami, B., Unsal, O., Cristal, A., (2018)** Fault Characterization Through FPGA Undervolting. *International Conference on Field Programmable Logic and Applications (FPL)*, 85

ÖZGEÇMİŞ

Ad-Soyad : Öykü MELİKOĞLU

Uyruğu : Türkiye Cumhuriyeti

Doğum Tarihi ve Yeri : 28.10.1994 - Ankara

E-posta : omelikoglu@etu.edu.tr

ÖĞRENİM DURUMU:

- **Lisans** : 2016, TOBB Ekonomi ve Teknoloji Üniversitesi, Mühendislik Fakültesi, Bilgisayar Mühendisliği

MESLEKİ DENEYİM:

Yıl	Yer	Görev
2016-2018	TOBB ETÜ	Eğitim Asistanı
2016, 2017	Barcelona Supercomputing Center	Stajyer, Konuk Araştırmacı
2015	Kasırga Mikroişlemciler Laboratuvarı	Stajyer

YABANCI DİL: İngilizce, Fransızca, İspanyolca

TEZDEN TÜRETİLEN YAYINLAR, SUNUMLAR VE PATENTLER:

- **Melikoglu O.,** Salami, B., Pavon, J., Ergin, O., Unsal, O., Cristal, A., 2019. A Novel FPGA-Based High Throughput Accelerator For Binary Search Trees, 17th International Conference on High Performance Computing & Simulation (HPCS2019), July 15-19, Dublin, Ireland