View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

T.C.
SAKARYA ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

GERÇEK ZAMANLI JAVA'DA BÜYÜK NESNELERE BELLEK AYIRMAK İÇİN ANAHTARLAMALI YAKLAŞIM

DOKTORA TEZİ

Veysel Harun ŞAHİN

Enstitü Anabilim Dalı : BİLGİSAYAR VE BİLİŞİM MÜHENDİSLİĞİ

Tez Danışmanı : Prof. Dr. Ümit KOCABIÇAK

Aralık 2013

T.C.
SAKARYA ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

GERÇEK ZAMANLI JAVA'DA BÜYÜK NESNELERE BELLEK AYIRMAK İÇİN ANAHTARLAMALI YAKLAŞIM

DOKTORA TEZİ

Veysel Harun ŞAHİN

Enstitü Anabilim Dalı : BİLGİSAYAR VE BİLİŞİM MÜHENDİSLİĞİ

Bu tez …/…/…… tarihinde aşağıdaki jüri tarafından oybirliği ile kabul edilmiştir.

…………… ………………………….. Jüri Başkanı	…………… ………………………….. Üye	…………… ………………………….. Üye

…………… ………………………….. Üye	…………… ………………………….. Üye

TEŞEKKÜR

Aileme.

İÇİNDEKİLER

TEŞEKKÜR ... ii
İÇİNDEKİLER ... iii
SİMGELER VE KISALTMALAR LİSTESİ ... v
ŞEKİLLER LİSTESİ ... vi
TABLOLAR LİSTESİ ... vii
ÖZET ... viii
SUMMARY ... ix

BÖLÜM 1.
GİRİŞ ... 1

BÖLÜM 2.
BELLEK YÖNETİMİ ... 6
2.1. Manuel Bellek Yönetimi ... 7
2.2. Otomatik Bellek Yönetimi – Çöp Toplama ... 7

BÖLÜM 3.
GERÇEK ZAMANLI SİSTEMLER ... 10
3.1. Katı Gerçek Zamanlı Sistemler ... 11
3.2. Yumuşak Gerçek Zamanlı Sistemler ... 12

BÖLÜM 4.
GERÇEK ZAMANLI JAVA ... 13
4.1. Bölge Temelli Yaklaşımlar ... 16
4.1.1. Gerçek zamanlı Java belirtimi ... 17
4.1.2. Güvenlik kritik Java ... 18

4.2. Gerçek Zamanlı Çöp Toplama 19
4.2.1. Planlama stratejileri 20
4.2.2. Nesne temsilleri.. 22

BÖLÜM 5.
MINUTEMAN RTGC ÇATISI 24
5.1. Bellek Organizasyonu 26
5.2. Bellek Ayırma Stratejileri.. 28
5.2.1. Normal nesneler için bellek ayırma 30
5.2.2. Diziler için bellek ayırma... 33
5.2.3. Büyük nesneler için bellek ayırma... 40

BÖLÜM 6.
ANAHTARLAMALI YAKLAŞIM 44
6.1. Atlamalı Arama Algoritması 44
6.2. Anahtarlamalı Yaklaşım.. 49

BÖLÜM 7.
DEĞERLENDİRME.. 55
7.1. Kıyaslama 55
7.2. Sonuçlar ve Tartışma... 57

BÖLÜM 8.
SONUÇ 63

KAYNAKLAR... 64
ÖZGEÇMİŞ 69

SİMGELER VE KISALTMALAR LİSTESİ

GC : Çöp Toplayıcı (Garbage Collector)

RTGC : Gerçek Zamanlı Çöp Toplayıcı (Real-time Garbage Collector)

RTSJ : Gerçek Zamanlı Java Belirtimi (Real-time Java Specification)

RTOS : Gerçek Zamanlı İşletim Sistemi (Real-time Operating System)

SCJ : Güvenlik Kritik Java (Safety Critical Java)

ŞEKİLLER LİSTESİ

Şekil 1.1. Ovm Sanal Makinesi 5
Şekil 5.1. Minuteman Bellek Organizasyonu 28
Şekil 5.2. Çarpan işaretçi kullanımı 32
Şekil 5.3. Size class yapısı 34
Şekil 5.4. Dizi temsili 35
Şekil 5.5. Bitişik dizi bellek ayırma süreci 37
Şekil 5.6. Parçalı dizi bellek ayırma süreci 39
Şekil 5.7. Büyük nesneler için bellek ayırma süreci 43
Şekil 6.1. Atlamalı arama algoritmasının sözde kodu 45
Şekil 6.2. Atlamalı arama algoritmasının grafiksel temsili 48
Şekil 6.3. Anahtarlamalı yaklaşımın sözde kodu 50
Şekil 6.4. Atlamalı arama algoritması akış diyagramı 1 51
Şekil 6.5. Atlamalı arama algoritması akış diyagramı 2 52
Şekil 6.6. Atlamalı arama algoritması akış diyagramı 3 53
Şekil 6.7. Atlamalı arama algoritması akış diyagramı 4 54
Şekil 7.1. Bellek ayırma zamanlarının dağılımı (lineer arama) 58
Şekil 7.2. Bellek ayırma zamanlarının dağılımı (anahtarlamalı yaklaşım) 58
Şekil 7.3. Bellek ayırma zamanlarının dağılım (arraylet temsili) 59
Şekil 7.4. Bellek ayırma zamanlarının histogramı (lineer arama) 59
Şekil 7.5. Bellek ayırma zamanlarının histogramı (anahtarlamalı yaklaşım) 60
Şekil 7.6. Bellek ayırma zamanlarının histogramı (arraylet temsili) 60

TABLOLAR LİSTESİ

Tablo 7.1. Üç yaklaşımın bellek ayırma zamanları.. 61
Tablo 7.2. Ortalama dizi erişim zamanları.. 62

ÖZET

Anahtar kelimeler: Gerçek Zamanlı Sistemler, Java Sanal Makinesi, Ovm Minuteman Çöp Toplama Bellek Yönetimi

Son yirmi yılda nesne yönelimli programlama dilleri ve yönetilen çalışma zamanları, yazılım mühendisliği yönünden sağladıkları avantajlardan ötürü oldukça popüler hale geldiler. Ancak birçok uygulama alanındaki bu popülerliklerinin aksine, aynı programlama dilleri ve çalışma zamanları, gerçek zamanlı programlama için uygun görülmediler. Birçok faktörün yanı sıra, bunların gerçek zamanlı sistemlerin geliştirilmesi için kullanılmalarının önündeki bariyerlerden bir tanesi, büyük nesnelere bellek ayırma esnasında karşılaşılma olasılığı olan uzun bekleme zamanlarıdır.

Bu tez gerçek zamanlı Java için zamanımıza kadar geliştirilmiş olan farklı büyük nesnelere bellek ayırma çözümlerini inceler ve bu çözümlere alternatif olarak anahtarlamalı yeni bir yaklaşım sunar. Sunulan tekniğin performansının hali hazırda uygulanmış diğer tekniklerle karşılaştırılması amacıyla geliştirilmiş olan sentetik bir kıyaslama uygulaması da bu tezde açıklanmıştır.

A SWITCHABLE APPROACH TO LARGE OBJECT ALLOCATION IN REAL-TIME JAVA

SUMMARY

Key Words: Real-Time Systems, Java Virtual Machine, Ovm, Minuteman, Garbage Collection, Memory Management.

Over the last twenty years object oriented programming languages and managed run-times like Java have been very popular because of their software engineering benefits. Despite their popularity in many application areas, they have not been considered suitable for real-time programming. Besides many other factors, one of the barriers that prevent their acceptance in the development of real-time systems is the long pause times that may arise during large object allocation.

This thesis examines different kinds of solutions which have been developed so far and introduces a switchable approach to large object allocation in real-time Java. A synthetic benchmark application which is developed to evaluate the effectiveness of the presented technique against other currently implemented techniques is also described.

BÖLÜM 1. GİRİŞ

1970'li yıllarda mikroişlemci teknolojisinin ortaya atılması ve kullanılmaya başlanması bilişim dünyasında ve günlük hayatımızın birçok alanında ciddi değişimlere sebep olmuş, yeni bir çığır açmıştır. Mikroişlemci teknolojisi ortaya atıldığı yıllardan günümüze kadar hızlı bir şekilde gelişim göstermiştir. Çalışma hızlarından işlem kapasitelerine kadar birçok alanda ciddi iyileşmeler, gelişmeler olmuştur. Aynı zamanda üretimleri artmış, kullanımları yaygınlaşmış ve maliyetleri düşmüştür. Bu gelişmelere paralel olarak mikroişlemci destekli hesaplama sistemleri günlük hayatımızda hali hazırda kullandığımız mekanik sistemler ile bütünleştirilmeye başlanmış ve elektromekanik sistemlerin sayısı artmıştır. Günümüzde, günlük hayatımızda kullandığımız birçok yapı (otomobil, uçak vb.), hesaplama sistemleri içermekte ve bu sistemler tarafından desteklenmekte ve/veya kontrol edilmektedir. Günlük hayatımızdaki yapılar, sistemler içerisinde yer alan ve bu sistemlerin çalışmasına yardımcı olan, onları kontrol eden bu hesaplama sistemlerine **gömülü sistemler** (Marwedel, 2011; Zurawski, 2009; Henriksson, 1998) ismi verilmektedir. Evlerimizde kullandığımız beyaz eşyalar içerisindeki kontrol sistemlerinden otomobillerimizdeki elektronik denge kontrol sistemlerine, otomatik park sistemlerine kadar, uçaklardaki çarpışma önleyici sistemlerden araçlarımızdaki eğlence sistemlerine kadar birçok sistem gömülü sistemlere örnek olarak verilebilir.

Gömülü sistemler, kullanıldıkları alanlara göre farklı fonksiyonellikleri yerine getirmekle yükümlüdürler ve dolayısıyla farklı kısıtlara sahiptirler. Örneğin kimi gömülü sistemlerden yüksek performans beklenirken, diğer bir takım gömülü sistemlerden düşük güç tüketimine sahip olması beklenebilir. Aynı şekilde kimi gömülü sistemlerde ise güvenlik ve zamanlama ön plandadır. Özellikle endüstriyel otomasyon sistemleri, uzay ve havacılık sektörü, otomotiv sektörü vb. alanlarda kullanılan gömülü sistemlerden, hangi şartlar altında olursa olsun görevlerini zamanında yerine getirmesi beklenmektedir. Klasik bilgisayar sistemlerinde başarımın

ölçüldüğü en temel veri, sistemin doğru sonucu hesaplamasıdır. Ancak güvenliğin ve zamanlamanın ön planda olduğu böylesi sistemlerde, doğru sonucun hesaplanması başarı için yeterli değildir. Bunun yanı sıra, bu doğru sonucun önceden belirlenmiş olan bir zaman dilimi içerisinde hesaplanması gerekmektedir. Doğru sonuç kısıtının yanı sıra, böylesi bir zaman kısıtına da sahip olan sistemlere **gerçek zamanlı sistemler** (Marwedel, 2011; Zurawski, 2009; Henriksson, 1998; Wellings, 2004) ismi verilmektedir

Bilişim teknolojilerinin şekil değiştirdiği ve hayatımızın her alanında kullanılmaya başladığı günümüzde artık PC sonrası döneme geçilmektedir. Her yerde ve hayatımıza nüfuz eden bilişim kavramı gelişmeye başlamıştır. Bu çerçevede, giysilerimizden evlerimize, araçlarımızdan ofislerimize kadar hayatımızın her alanında bilişim teknolojileri kullanılır hale gelmiştir ve gelmektedir. Yaygınlaşan bu bilişim teknolojilerinin elbette en önemli ayağı gömülü ve gerçek zamanlı sistemlerdir. Günümüzde kullandığımız gömülü sistemlerin neredeyse tamamına yakını gerçek zamanlı sistemlerdir (Marwedel, 2011).

Hayatımızdaki yeri her geçen gün artan gömülü ve gerçek zamanlı sistemlerin sağladığı fonksiyonellikler de epeyce artmış durumdadır. Aynı zamanda her geçen gün bu fonksiyonelliklerin daha da artırılması için ihtiyaçlar belirmektedir. Bu gelişmelerin sonucu olarak gömülü ve gerçek zamanlı sistemler, gerek donanım yönünden ve gerekse yazılım yönünden olarak oldukça karmaşık bir hale gelmiştir.

Günümüzde gerçek zamanlı sistemler üzerinde, oldukça büyük ve karmaşık yazılımlar koşmaktadır ve geleneksel olarak bu yazılımlar C programlama dili kullanılarak geliştirilmektedir. Takdir edilmelidir ki, büyük ve karmaşık yazılımların geliştirme ve bakım süreçleri, özellikle C programlama dili kullanıldığında oldukça maliyetli ve aynı zamanda fazlaca hataya açık olmaktadır.

C programlama diline alternatif olan nesne yönelimli programlama dilleri, yazılım mühendisliği yönünden sağladıkları avantajlardan ötürü programlama dünyasında özellikle son 20 yılda ilgi odağı haline gelmiştir. Nesne yönelimli programlama kavramı, sunduğu nesne kavramı sayesinde gerçek dünya problemlerinin, hesaplama

sistemleri üzerinde oldukça başarılı bir şekilde modellenmesine ve temsil edilmesine olanak tanır. Yeniden kullanılabilirlik, kalıtım ve sarmalama, nesne yönelimli programlamanın üç temel ayağını oluşturur. Bu üç özellik, büyük ölçekli projelerin hayata geçirilmesi, idamesi ve bakımına oldukça yardımcı olmuştur.

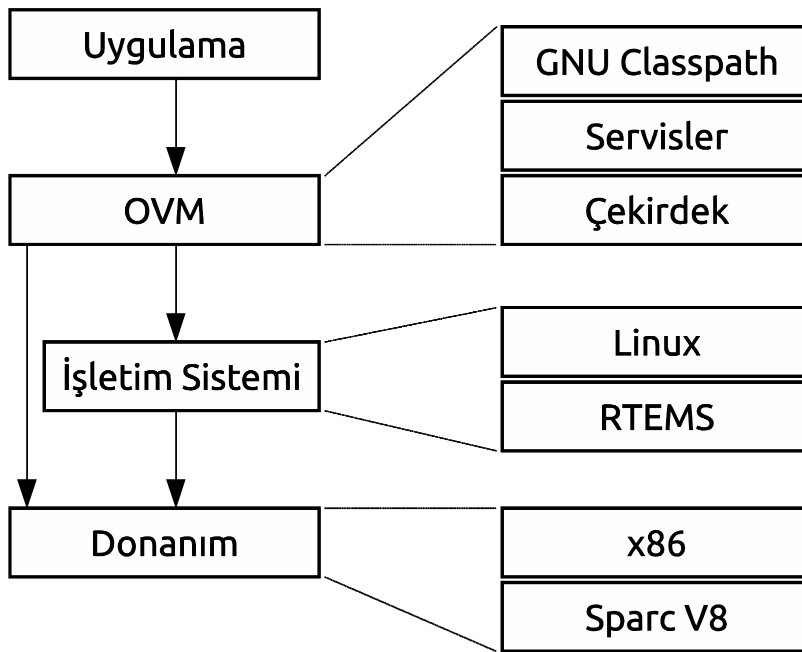
Nesne yönelimli programlama dillerinden bir tanesi olan Java, ortaya atıldığı günden bu güne oldukça popüler hale gelmiştir. Nesne yönelimli programlama kavramını uygulamasının yanı sıra, Java'nın popüler hale gelmesinin iki önemli sebebi mevcuttur. Bunlardan birincisi, geliştirici verimliliğini artıran zengin sınıf kütüphanesi ve ikincisi ise yönetilen çalışma zamanıdır. Yönetilen çalışma zamanları, kaynak yönetimi gibi hali hazırda geliştirici görevleri arasında yer alan olan birçok görevi kendi üzerlerine almışlardır. Yönetilen çalışma zamanı kullanmanın en avantajlı noktalarından bir tanesi, **çöp toplayıcı** (garbage collector, GC) (Jones ve diğerleri, 2011) adı verilen bellek yönetim sistemidir. Çöp toplayıcı, yönetilen çalışma zamanının bir alt sistemidir ve onun aracılığı ile koşturulan bir uygulamanın tüm bellek yönetim görevlerini üzerine alır. Bunu yaparak geliştiricileri, bellek yönetimi için kod yazma yükümlülüğünden kurtarır ve böylelikle geliştirici verimliliğini artırır. Aynı zamanda, çok sık rastlanılan bellek yönetim hatalarını engelleyerek bellek güvenliği sağlarlar. Bellek yönetimi kodu yazarken en sık yapılan hatalar, bellek sızması hatası, asılı işaretçi hatası ve iki kere bellek boşaltma hatasıdır.

Birçok uygulama alanındaki başarısının aksine Java, uzun zaman boyunca gerçek zamanlı uygulamalar için alternatif bir programlama dili olarak düşünülmemiştir. Java'nın en güçlü olduğu yönlerinden bir tanesi, onun gerçek zamanlı uygulamalar için kullanılmasının önündeki en büyük engellerden bir tanesi olmuştur. Bu güçlü yönü, sahip olduğu çöp toplayıcı sistemdir. Geleneksel bilgisayar sistemlerinin aksine gerçek zamanlı sistemlerin doğruluğu, yalnızca doğru sonuçları üretmeleri ile değil aynı zamanda bu doğru sonuçları sınırlı bir zaman dilimi içerisinde üretmeleri ile belirlenir. Aynı zamanda gerçek zamanlı sistemler tahmin edilebilir olmalıdır. Her bir görevin çalışma zamanı ve görevlerini bitirmek zorunda oldukları zamanlar önceden bilinmelidir. Her bir iş parçacığı, sonucunu sınırlı bir zaman dilimi içerisinde üretmek zorundadır. Öte yandan çöp toplayıcı sistemler, doğaları gereği tahmin edilebilir olmayan zamanlama davranışlarına sahiptir. Bir çöp toplayıcı sisteminin çalışma

sürecinde birkaç tane belirsizlik mevcuttur: a) Çöp toplama döngüsünün kesin başlama zamanı, b) çöp toplama döngüsü süresince harcanan zaman ve c) bir bellek ayırma talebine verilecek olan cevabın süresi önceden bilinmez. Burada (a) ve (b) şıklarındaki bilinmezlik kritiktir çünkü gerçek zamanlı sistemlerin kararlı bir şekilde çalışacağından ve her bir görevin görev bitirme süresinin aşılmayacağından emin olmak için iş parçacıklarının zamanlaması bilinmelidir. Buna ek olarak (c) şıkkında bahsi edilen cevap zamanı, bellek ayıran tüm görevlerin zamanlamasını direk olarak etkilemektedir. Dolayısıyla (c) şıkkındaki belirsizlik gerçek zamanlı sistemin zamanlaması üzerinde kesin bir etkiye sahiptir. Çöp toplayıcı sistemler tarafından bakıldığında, yukarıda bahsi edilen üç belirsizlik, Java'nın gerçek zamanlı uygulamalarda kullanımının önündeki ana engeller olmuştur.

Geçtiğimiz on yılda yapılan ciddi araştırmalar sonucunda, yukarıda verilen her bir alanda birçok çözüm üretilmiştir. Bu çalışmaların bir sonucu olarak birçok gerçek zamanlı Java sanal makinası başarılı bir şekilde geliştirilmiştir. Bu sanal makinalar halen günümüzde gerçek zamanlı sistemlerde kullanılmaktadır. Bu tez çalışması (c) şıkkında sözü edilen probleme, açık kaynak kodlu ve gerçek zamanlı bir Java sanal makinası olan Ovm (Baker ve diğerleri, 2006; Armbruster ve diğerleri, 2007) bağlamında odaklanmıştır. Ovm sanal makinesine ilişkin görsel bir temsil Şekil 1.1.'de gösterilmiştir. Bu tezde, Boyer ve Moore'un (Boyer ve Moore, 1977) karakter dizisi arama algoritmasının, bitişik nesnelere bellek ayırma için özelleştirilmiş bir varyasyonu olan ve atlamalı arama algoritması olarak adlandırılan bir teknik ve bu tekniği kullanan anahtarlamalı bir yaklaşım sunulmaktadır. Sunulan yaklaşımın amacı büyük nesnelere bellek ayırma sürecini hızlandırmaktır. Sunulan yaklaşım Minuteman çatısı (Kalibera ve diğerleri, 2009, 2011; Baker ve diğerleri, 2009) içerisinde uygulanmıştır. Minuteman çatısı, Ovm sanal makinasının çöp toplayıcı çatısıdır. Sunulan yaklaşımın testi ve değerlendirilmesi amacıyla, büyük boyutlu dizilere bellek ayırma talebinde bulunan ve bellek ayırma zamanlarını ölçen bir kıyaslama uygulaması geliştirilmiştir. Kıyaslama uygulaması, sunulan yaklaşımın yanı sıra, hali hazırda Ovm içerisinde yer alan tüm çözümler için çalıştırılmış ve devamında sonuçlar karşılaştırılmıştır. Bu çalışma, Boyer ve Moore'un algoritmasını bu bağlamda özelleştiren ilk çalışmadır.

Bu çalışma, çöp toplayıcı sistemler ve gerçek zamanlı sistemleri temel alan bir çalışma olduğundan ötürü, tezin ikinci bölümünde bellek yönetimine, üçüncü bölümünde de gerçek zamanlı sistemlere değinilmektedir. Dördüncü bölümünde ise gerçek zamanlı sistemlerin geliştirilmesinde kullanılan gerçek zamanlı Java hakkında temel bilgi verilmekte ve büyük nesne ayırma konusundaki farklı yaklaşımlardan bahsedilmektedir. Beşinci bölümde, Ovm sanal makinasının Minuteman çatısı detaylı olarak anlatılmaktadır. Önerilen teknik altıncı bölümde açıklanmakta ve yedinci bölümde kıyaslama uygulamasının sonuçları paylaşılarak yorumlanmaktadır.



Şekil 1.1. Ovm Sanal Makinesi

BÖLÜM 2. BELLEK YÖNETİMİ

Bu çalışma bellek özel olarak çöp toplayıcı sistemler ve genel olarak bellek yönetimi üzerine bir çalışma olduğundan ötürü bu bölümde bellek yönetimi ve bellek yönetim sistemleri üzerine kısaca değinilecektir.

Bellek yönetim sistemlerinin tarihi, bilgisayar dünyasının önemli zaman dilimlerinden bir tanesi olan 1957 yılına kadar dayandırılabilir. Bu yılda Fortran programlama dili (ANSI, 1957) geliştirildi. Bu programlama dilinin en büyük özelliklerinden bir tanesi günümüzde kullandığımız anlamda ilk derleyiciyi ortaya atmış olması idi. Fortran derleyicisi bellek organizasyonu görevini programcının üzerinden aldı ve kendisi gerçekleştirmeye başladı. Fortran'ın bu ilk derleyicisinde statik bellek ayırma sistemi kullanılıyordu. Statik bellek ayırma sisteminde, yazılan programın her bir üyesinin bellekteki boyutu ve konumu derleme aşamasında belirlenmek zorunda idi. Programın çalışması esnasında ek bir ayırma işlemi gerçekleşmiyordu (Henriksson, 1998).

Fortran ile birlikte gelen bellek yönetim sistemi her ne kadar programcıları rahatlatmış olsa da yeterince esnek bir sistem değildi. Çünkü programın çalışması esnasında dinamik olarak bellek ayırma işlemi gerçekleştirilemiyordu. Daha esnek bellek yönetim sistemine sahip derleyiciler üzerine yapılan çalışmaların öncülerinden bir tanesi ALGOL 60 (Naur ve diğerleri, 1960) programlama dili ile sonuçlanmış ve bu çalışma da bilgisayar tarihinin önemli noktalarından bir tanesi olmuştur. Bu programlama dili ile birlikte yığın (stack) adı verilen bir bellek yapısı, bellek organizasyonu için kullanılır hale gelmiştir (Henriksson, 1998).

Devam eden yıllarda daha da esnek bellek yönetim sistemleri geliştirebilmek adına yığıt (heap) adı verilen yeni bir bellek yapısı ortaya atılmış ve sonucunda üç temel bellek ayırma türü ortaya çıkmıştır: statik bellek ayırma, yığıt bellek ayırma ve yığın bellek ayırma (Jones ve Lins, 1996). Günümüzde, özellikle yüksek seviyeli birçok

programlama dili hem yığıt bellek ayırmayı ve hem de yığın bellek ayırmayı bir arada kullanmaktadır.

Yığıt bellek ayırma işleminin de iki çeşidi mevcuttur: Manuel bellek yönetimi, otomatik bellek yönetimi (Jones ve Lins, 1996).

2.1. Manuel Bellek Yönetimi

Manuel bellek yönetiminde yığıt, harici olarak tamamen programcı tarafından yönetilmektedir. Program içerisindeki dinamik tüm veri yapılarının ne zaman yığıta yerleştirileceği ve yığıttan ne zaman kaldırılacağı programcının sorumluluğundadır. Bu yapının kullanıldığı programlama dillerinden en popüleri C (Kernighan ve Ritchie, 1978) programlama dilidir.

Manuel bellek yönetiminin en büyük dezavantajlarından iki tanesi asılı işaretçiler ve bellek sızmasıdır. Bir veri yapısı bellekten kaldırıldığı zaman o veri yapısına işaret eden hiçbir işaretçi bulunmamalıdır. Bellekte bulunmayan bir veri yapısına işaret eden işaretçilere asılı işaretçiler denilmektedir. Böyle bir durum programda ciddi hatalara yol açabilir. Bellekte yer alan bir veri yapısına hiçbir işaretçi tarafından işaret edilmiyorsa, bu veri yapısı bellekten silinmelidir. Aksi takdirde bu veri yapısı hiç kullanılmadığı halde bellekte yer işgal edecektir. Bellekte yer alan ancak hiçbir referansı da kalmamış bellek yapılarının oluştuğu durumlara bellek sızması adı verilir. Bu da manuel bellek yönetimi esnasında en sık rastlanılan hatalardandır.

2.2. Otomatik Bellek Yönetimi – Çöp Toplama

Manuel bellek yönetiminde oluşan hataları engellemek ve programcı verimliliğini artırmak için yığıt bellek ayırması işleminin, programcının sorumluluğundan alınıp otomatik olarak gerçekleştirilmesi güzel bir çözümdür. Bu çözüme otomatik bellek yönetimi ya da çöp toplama adı verilmektedir.

Çöp toplamanın ilk örneği 1960 yılında McCarthy tarafından ortaya atılmıştır (McCarthy, 1960) ve Lisp programlama dilinde uygulanmıştır. Devam eden yıllarda

birçok çöp toplama tekniği yine Lisp programlama dili üzerinde geliştirilmiştir. Günümüzde birçok yüksek seviyeli ve özellikle nesne yönelimli programlama dili otomatik bellek yönetimini diğer bir deyişle çöp toplama sistemlerini kullanmaktadır.

Çöp toplama işlemini yerine getiren bellek yönetim sistemlerine ise çöp toplayıcı adı verilmektedir. Çöp toplayıcılar, yalnızca belleğin temizlenmesinden değil tüm yığıt alanının yönetiminden sorumludurlar. Bir çöp toplayıcının iki ana sorumluluğu mevcuttur: Kendisine referans kalmamış olan veri yapılarının bellekten temizlenmesi ve yeni veri yapıları için uygun bir bellek bölgesi bulunması.

Literatürde otomatik bellek yönetimi yerine daha çok çöp toplama terimi kullanılmaktadır ve bu tezde de bu terimin kullanılması tercih edilmiştir. Bu tezde, aynı zamanda otomatik bellek yönetim sistemleri de çöp toplayıcı olarak anılacaktır. Günümüzde birçok çöp toplama algoritması mevcuttur. (Jones ve Lins, 1996; Jones, 2007; Jones ve diğerleri, 2011) kaynaklarında genel olarak bellek yönetimi ve özel olarak çöp toplama hakkında detaylı bilgi ve incelemeler mevcuttur. Bu bölümün devamında, en yaygın çöp toplayıcıların çalışma sistemleri üzerine açıklama yapılacak ve bu konunun anlaşılmasına çalışılacaktır.

Günümüzdeki çöp toplayıcıların büyük bir bölümü işaretle ve temizle sistemi ile çalışmaktadır. Bu sistem de McCarthy (McCarthy, 1960) tarafından ortaya atılmıştır. Bu sistemde bellek yönetimi iki temel aşamada gerçekleştirilir: İşaretleme ve temizleme. İşaretleme aşamasında bellekte kendisine referans bulunan ve hali hazırda kullanılan canlı nesneler ile artık kendisine referansta bulunulmayan ölü nesneler tespit edilir. Temizleme aşamasında ise ölü nesneler bellekten temizlenir.

Günümüzde bu sistemin işaretleme aşamasında genellikle, Dijkstra ve diğerleri (Dijkstra ve diğerleri, 1978) tarafından 1978 yılında ortaya atılmış olan üç renkli işaretleme tekniği kullanılmaktadır. Bu teknikte beyaz, siyah ve gri renkler mevcuttur ve nesneler yaşam durumlarına göre renkler ile işaretlenirler.

Bir çöp toplama döngüsü başlamadan önce bellekteki tüm nesneler beyaz renktedir. İşaretleme işlemi uygulamanın kök kümesinden başlar. Kök küme, uygulamanın yığın

üzerinde bulunan iş parçacıklarının veri yapılarıdır. Öncelikle yığın üzerinde bulunan bu veri yapıları gezilerek bunların içerisinde, yığıt üzerinde bulunan veri yapılarına olan referanslar tespit edilir. Bu referansların gösterdiği nesneler gri kümeye dâhil edilir. Daha sonra gri kümedeki her bir nesne ziyaret edilir ve bu nesnenin rengi siyah yapılır. Bu nesnenin içerisinde var olan referanslar tespit edilerek bu referansların gösterdiği nesneler gri kümeye eklenir. Bu işlem gri kümede herhangi bir nesne kalmayıncaya kadar devam eder. Gri kümedeki nesnelerin tamamı ziyaret edildiğinde işaretleme aşaması bitmiş demektir. Bu aşamada, bellekte bulunan beyaz renkli nesneler, kendisine artık referans kalmamış olan nesnelerdir ve bunlara ölü nesneler denir. Bir sonraki aşamada bellekteki ölü nesneler temizlenir. Son olarak, canlı olan tüm nesnelerin siyah olan renkleri beyaz renge çevrilir ve dolayısıyla bir sonraki döngü için hazır hale gelinmiş olur. Bu noktada çöp toplama döngüsü bitirilir.

Çöp toplayıcı sistemler, programcının müdahalesine gerek bırakmadan kendileri bellekte kullanılmayan veri yapılarını tespit ederek siler ve böylelikle otomatik bellek yönetimi gerçekleştirmiş olurlar.

BÖLÜM 3. GERÇEK ZAMANLI SİSTEMLER

Bu çalışma, gerçek zamanlı sistemler üzerine gerçekleştirilen bir çalışma olduğundan ötürü, bu bölümde gerçek zamanlı sistemlere daha ayrıntılı bir şekilde değinilmiştir.

Görevlerini, önceden belirlenmiş sınırlı bir zaman dilimi içerisinde gerçekleştirmek zorunda olan sistemlere gerçek zamanlı sistemler denir (Marwedel, 2011; Zurawski, 2009; Henriksson, 1998; Wellings, 2004). Bu sistemlerin doğruluğu yalnızca doğru sonucun hesaplanması ile ölçülmez. Aynı zamanda ilgili doğru sonucun belirlenmiş, sınırlı bir zaman dilimi içerisinde hesaplanması da gerekmektedir. Eğer doğru sonuç, bu sonucun hesaplanması için önceden belirlenmiş bu zaman dilimi içerisinde hesaplanamaz ise sistem başarısız kabul edilir. Diğer bir deyişle bir gerçek zamanlı sistem tahmin edilebilir olmalıdır. Sistemde, hangi iş parçacığının ne zaman koşacağı ve ne kadar sürede görevini tamamlayacağı bilinmeli ve garanti edilebilmelidir.

Gerçek zamanlı sistemler, günümüzde otomotiv sektöründen, uzay ve havacılık sektörüne, endüstriyel otomasyon sistemlerinden, finansal uygulamalara kadar birçok farklı alanda kullanılmaktadır. Dolayısıyla farklı boyutlarda ve karmaşıklıkta gerçek zamanlı sistemler ve uygulamalar halen geliştirilmektedir. Gerçek zamanlı sistemlerin belli başlı en temel özellikleri aşağıda anlatılmıştır. Aynı zamanda (Burns ve Wellings, 2009; Marwedel, 2011; Zurawski, 2009; Henriksson, 1998; Wellings, 2004) kaynaklarından gerçek zamanlı sistemler ve özellikleri konusunda detaylı bilgi edinilebilir.

Günümüzde var olan gömülü sistemlerin büyük bir bölümü gerçek zamanlı sistemlerdir ve bu sistemlerden yine büyük bir bölümü algılayıcı/çalıştırıcı sistemleri sayesinde gerçek hayat ile direk irtibat halindedir (Marwedel, 2011). Gerçek zamanlı sistemler tek bir girdiye, tek iş parçacığına ve tek bir göreve sahip küçük sistemler olabileceği gibi; çok girdiye sahip, çok işlemcili ve çok iş parçacıklı büyük sistemler

de olabilir. Günümüzde hesaplama sistemlerinden beklenen fonksiyonellikler arttığından ötürü, karmaşık ve büyük gerçek zamanlı sistemlerin sayısı da artmakta, hayatımıza her alanda nüfuz etmeye başlamaktadır. Çok iş parçacığına sahip karmaşık gerçek zamanlı sistem uygulamalarında, her bir farklı iş parçacığının kendi zaman kısıtı olabilir. Böyle bir durumda bu iş parçacıları paralel ve/veya eşzamanlı bir şekilde koşarak görevlerini kendileri için ayrılmış olan zaman dilimlerinde gerçekleştirmeye çalışır. Yine farklı ihtiyaçlara yönelik farklı boyuttaki gerçek zamanlı sistemler, farklı işletim sistemleri kullanırlar. Gerçek zamanlı sistemler üzerinde tercih edilen bu işletim sistemlerine **gerçek zamanlı işletim sistemi** (real-time operating system, RTOS) adı verilir (Takada, 2001). Bu işletim sistemlerinin en temel özellikleri, gerçek zamanlı sistemlerin ihtiyaçları olan tahmin edilebilirliği, özel planlama stratejilerini ve zaman eşlemesini sağlamalarıdır (Marwedel, 2011).

Gerçek zamanlı sistemler, cevap zamanlarına yani zamanlama ölçütlerine, kısıtlarına göre kabaca iki gruba ayrılmaktadırlar: katı gerçek zamanlı sistemler ve yumuşak gerçek zamanlı sistemler (Wellings, 2004).

3.1. Katı Gerçek Zamanlı Sistemler

Katı gerçek zamanlı sistemler, katı zamanlama kıstaslarına sahip sistemlerdir. Bu tür sistemlerde, sistemin cevabını belirlenmiş bir zaman dilimi içerisinde gerçekleştirmesi zorunludur (Wellings, 2004). Aksi takdirde sistemde ciddi hatalar meydana gelebilir ve bu hatalar, felaketle sonuçlanabilir. Örneğin uçaklarda yer alan bir çarpışma önleyici sistem eğer işlemini zamanında yerine getirip, uçağın irtifasını değiştirmesi için pilotu zamanında uyaramaz ise uçakların çarpışmasına dolayısıyla uçakların kaybına daha da önemlisi insanların ölümüne yol açabilir. İnsan hayatının tehlikede olabileceği veya ciddi maliyet kayıplarının söz konusu olabileceği kritik sistemleri yöneten, bu tür gerçek zamanlı sistemlere katı gerçek zamanlı sistemler (Kopetz, 1997) adı verilmektedir. Katı gerçek zamanlı sistemlerde cevap zamanları genellikle milisaniyenin altında olmaktadır.

3.2. Yumuşak Gerçek Zamanlı Sistemler

Yumuşak gerçek zamanlı sistemlerde yine cevap zamanı önemli olmakla birlikte katı gerçek zamanlı sistemlere nazaran, kaçırılan bir cevap zamanı, sistem hatasıyla ve dolayısıyla felaketler ile sonuçlanmaz (Wellings, 2004; Kopetz, 1997). Kaçırılan cevap zamanını sistem, hata oluşmayacak ve sistemi durdurmayacak şekilde telafi edebilir. Ancak yine de görevlerin zamanında tamamlanması, sistemin sağlıklı bir şekilde çalışması için oldukça önemlidir. Cevap zamanları da katı gerçek zamanlı sistemlere nazaran daha uzundur. Genellikle 10-100 ms civarındadır (Henriksson, 1998).

Yumuşak gerçek zamanlı sistemlere en güzel örneklerden bir tanesi mobil iletişim sistemleridir. Kişilerin mobil iletişim gerçekleştirirken, karşılıklı bir şekilde konuşuyormuş gibi hissetmelerini temin etmek için iletişimin oldukça hızlı ve verimli olması gerekmektedir. Sistem görüşmeyi mümkün olduğunca hızlı bir şekilde gerçekleştirmeye çalışırken bir takım problemler çıkar ve cevap zamanlarının gecikmesi söz konusu olursa, ses kalitesinin düşürülmesi vb. teknikler ile bu hatalar telafi edilmeye çalışılır. Eğer hatalar telafi edilemez ise iletişimde gecikmeler meydana gelebilir. En kötü durumda ise iletişim kesilir. Eğer sistem verimli bir konuşma ortamı sunamaz ise bu bir felaketle sonuçlanmaz ancak müşteri memnuniyetsizliğiyle sonuçlanır. Böyle bir sistem yumuşak gerçek zamanlı bir sisteme örnek olarak gösterilebilir.

BÖLÜM 4. GERÇEK ZAMANLI JAVA

Gerçek zamanlı sistemler, geleneksel olarak C programlama dili kullanarak geliştirilmektedir. C programlama dili ile geliştirilen programların doğal koda dönüştürülmesi ve donanım üzerinde doğal olarak koşabilmesi bu tercihi etkileyen sebeplerden bir tanesidir. Öte yandan C programlama dilinde manuel bellek yönetimi gerçekleştirilmektedir. Bu yönüyle de çöp toplayıcı (otomatik bellek yönetimi) sistemine sahip programlama dillerine nazaran daha verimli kodlar üretilebilmektedir. C programlama dilinde manuel bellek yönetimi, malloc ve free komutları ile gerçekleştirilir. Malloc komutu ile veri yapıları için çalışma zamanında dinamik olarak bellek ayrımı gerçekleştirilir. Veri yapıları görevlerini tamamladıklarında ise sistemden alınan bellek free komutu ile sisteme geri verilir. Ne ki C programlama dilinin manuel bellek yönetimi gerçekleştiren bu yapısı gerçek zamanlı uygulamalar için problem teşkil edebilmektedir. Zira malloc komutu ile gerçekleştirilen bellek taleplerinin cevap süreleri tahmin edilemez ve dolayısıyla bu komuta işleten gerçek zamanlı iş parçacıklarının görevlerini zamanında tamamlayıp tamamlayamayacakları hakkında fikir sahibi olunamaz. Tahmin edilebilirlik, bir gerçek zamanlı sistem için oldukça kritik bir özelliktir. Sistem çalışmaya başlamadan önce, sistemin her bir iş parçacığının, özellikle de gerçek zamanlı iş parçacıklarının ne zaman çalışacakları, ne kadar sürede işlerin tamamlayacakları net bir şekilde belli olmalıdır.

Yukarıda bahsedilen sebeplerden ötürü gerçek zamanlı sistemlerin programlanmasında C programlama dilinin manuel bellek yönetimi kullanılmaz. Bunun yerine, uygulamanın kullanacağı bellek miktarı önceden belirlenir ve uygulama çalışmaya başladığı anda malloc ile tüm uygulama için gereken bellek miktarı sistemden alınır (Henriksson, 1998). Daha sonra, sistemden alınan bu alandaki bellek yönetimi tamamen programcı tarafından gerçekleştirilir. Kullanımı zaten oldukça zor ve hataya açık olan manuel bellek yönetimine bir de bu kısıt eklenince, C programlama dili gerçek zamanlı uygulama geliştirmek için oldukça zor bir ortam haline gelir. Her

ne kadar tek işlemci üzerinde koşan tek gerçek zamanlı iş parçacığına sahip uygulamalar mevcutsa ve bu şekilde programlanmalarında problem oluşmasa da, günümüzde artık bu pek mümkün değildir. Bugün, birçok gerçek zamanlı sistem çok farklı boyutlarda, karmaşıklıkta ve büyük kod boyutlarına sahip uygulamalara ihtiyaç duymaktadır. Karmaşık gerçek hayat uygulamalarını C programlama dili ile gerçekleştirmek zaten zor bir görev iken, buna bir de bellek yönetimi sorumluluğu eklenince, bu görev daha da zorlaşmaktadır. Zira yazılan uygulamalarda yalnızca problemin çözümüne uygun kod yazmak yeterli olmamaktadır. Bu kodun bellek organizasyonunu yerine getirecek bellek yönetim parçaları da uygulama geliştirme sürecine dâhil olmaktadır. Bu süreç bütünüyle düşünüldüğünde, tahmin edilebileceği oldukça zor ve aynı zamanda maliyetlidir.

Gerçek zamanlı sistemlerdeki artan ihtiyaçları karşılama konusunda her geçen gün geride kalan C programlama diline alternatif arayışlarında, dikkate değer olan en önemli diller nesne yönelimli programlama dilleridir. Zira nesne yönelimli programlama kavramı, karmaşık gerçek hayat problemlerinin bilgisayar sistemlerinde mümkün olduğunca gerçekçi bir modelinin kurulmasında oldukça verimlidir. Öte yandan yine nesne yönelimli programlama dilleri, sahip oldukları çöp toplayıcı sistemler sayesinde otomatik bellek yönetimi işlemi gerçekleştirir. Programcıyı, bellek yönetim zorluklarından arındırır ve onun daha verimli bir şekilde kod yazmasına olanak tanır. Aynı zamanda çöp toplayıcı sistemler, bellek yönetimi sürecinde yapılan hataları ortadan kaldırarak, daha güvenli programlar geliştirilmesine imkân verir.

Bütün bu avantajlarının yanı sıra nesne yönelimli programlama dillerinin gerçek zamanlı sistemlerin programlanmasında kullanılmasını engelleyen çok ciddi bir dezavantajı mevcuttur: Çöp toplayıcı. Nesne yönelimli programlama dillerine ciddi bir avantaj katan çöp toplayıcı sistemleri yine bu dillerin gerçek zamanlı sistemler çerçevesinde düşünüldüğünde, kullanılmasının önündeki en büyük engellerden bir tanesidir. Çöp toplayıcılar, doğaları itibariyle tahmin edilemez bir yapıya sahiptirler. Daha açık bir ifadeyle, bir çöp toplayıcının ne zaman çöp toplamak için devreye gireceği ve bu işlevini ne kadar sürede tamamlayacağı önceden belli değildir. Bu belirsizlik gerçek zamanlı uygulamalar için uygun değildir. Örneğin, çöp toplayıcı, bir gerçek zamanlı iş parçacığı çalışırken devreye girerek uzun bir süre çalışabilir ve bu iş

parçacığının görevini zamanında yerine getirmesine engel olabilir. Bu, özellikle katı gerçek zamanlı sistemler için kabul edilemez bir belirsizliktir.

Bu dezavantajlarından ötürü, nesne yönelimli programlama dillerinin gerçek zamanlı uygulamalar için kullanılması yönünde yapılan çalışmalar, bellek yönetimi üzerinde yoğunlaşmıştır. Bu çalışmaların büyük bir bölümü açık ve ücretsiz geliştirme ortamları ve dokümantasyonlarından ötürü Java programlama dili etrafında yürütülmüştür. Gerçek zamanlı Java uygulamalarının geliştirilebilmesine olanak tanıyan Java programlamasına genel olarak gerçek zamanlı Java adı verilmiştir. Tezin bundan sonraki kısımlarında bu terim sıkça kullanılacaktır.

Günümüze kadar gerçekleştirilen bu çalışmalar sonucunda, gerçek zamanlı Java kendisini büyük gerçek zamanlı uygulamalar geliştirilebilecek bir ortam olarak kanıtlamıştır. Açık kaynak kodlu ve ticari birçok gerçek zamanlı Java sanal makinesi geliştirilmiş ve bu makineler üzerinde gerçek zamanlı uygulamalar başarılı bir şekilde çalıştırılmıştır. Günümüze kadar geliştirilmiş olan gerçek zamanlı Java sanal makinelerinin başlıca örnekleri, Ovm (Baker ve diğerleri, 2006; Armbruster ve diğerleri, 2007), Fiji VM (Pizlo ve diğerleri, 2010a), Mackinac (Bollella ve diğerleri, 2005), IBM WebSphere Real Time (Auberbach ve diğerleri, 2007), PERC (Aonix, 2013) ve Jamaica VM (Aicas, 2005) olarak gösterilebilir. Bu Java sanal makineleri üzerinde endüstriyel otomasyon sistemlerinden, demiryolu otomasyon sistemine, uçuş kontrol sistemlerinden savaş gemisi otomasyon sistemlerine kadar birçok gerçek zamanlı sistem başarıyla uygulanmıştır (Henties ve diğerleri, 2009).

Bu tezde yapılan çalışma, Java programlama dili üzerinde gerçekleştirilmiş ve açık kaynak kodlu, Ovm gerçek zamanlı Java sanal makinesi üzerine uygulanmıştır. Bu bölümde, genel olarak nesne yönelimli programlama dillerinin özel olarak ise Java programlama dilinin gerçek zamanlı sistemlere uygulanabilmesi için, bellek yönetim sistemi üzerine günümüze kadar yapılan çalışmalara değinilecektir.

Java programlama dilinin gerçek zamanlı sistemlerde kullanılabilmesi adına yapılan çalışmalar temel olarak iki kolda devam etmiştir. Bir kolda yapılan çalışmalarda bölge temelli bellek yönetim modelleri geliştirilirken, diğer bir kolda yapılan çalışmalarda

ise hali hazırda var olan çöp toplama sistemlerinin gerçek zamanlı sistemlere uygulanabilmesi için çalışılmıştır ve buna gerçek zamanlı çöp toplama adı verilmiştir. Bundan sonraki bölümlerde bu iki yaklaşım anlatılacaktır.

4.1. Bölge Temelli Yaklaşımlar

Bölge temelli yaklaşımların (Tofte ve Talpin, 1997) amacı, çöp toplama işlemini tamamen ya da kısmen devreden çıkartmaktır. Bu yaklaşımlarda her bir nesne için ayrı ayrı bellek bölgesi ayrılmaz. Aynı zamanda nesneler de ayrı ayrı bellek bölgelerinden temizlenmezler. Uygulama geliştiriciler, program yazarken birbiri ile ilişkili olabilecek nesneleri gruplar haline getirirler. Daha sonra her bir grup nesne için çalışma zamanında bir bellek bölgesi bir bütün halinde atanır ve temizlenir. Bu bellek bölgesine **kapsam** ya da **kapsama alınmış bellek alanı** (Bollella ve diğerleri, 2002) ismi verilir. Bu sistemin çalışma şekli aşağıda detaylı bir şekilde anlatılmıştır.

İlgili nesnelerden ilki için bellek ihtiyacı doğduğunda bu gruptaki tüm nesneler için bir bellek bölgesi ayrılır. Daha sonra ayrılmış olan kapsamda ilk nesne oluşturulur ve bu nesneye geçiş yapılır. Bu işleme **"kapsama girme"** adı verilir. Bu nesne ve ilişkili tüm nesneler bu kapsam içerisinde oluşturulur ve kullanılır. Kapsam içerisindeki tüm nesnelerle olan işlemler tamamladığında kapsamın oluşturulduğu noktaya geri dönülür ve kapsam, dolayısıyla içerisindeki tüm nesneler bir bütün halinde silinir. Bu işleme **"kapsamdan çıkma"** adı verilir.

Bu yaklaşımda bellek, bölge yahut kapsam adı verilen yapılardan oluşturulur. Çöp toplayıcı sistem kapsamlara dokunmaz. Dolayısıyla kapsamdaki nesneler güvenli bir şekilde oluşturulup toplu bir şekilde silinebilir. Aynı zamanda, çöp toplama işlemi devreden çıkartıldığı için bu sistemden kaynaklanan gecikmeler ve belirsizlikler de ortadan kaldırılmış olur. Devam eden bölümlerde bu yaklaşımın uygulandığı iki temel çözümden bahsedilecektir.

4.1.1. Gerçek zamanlı Java belirtimi

Bölge temelli bellek modelini Java programlama diline uygulayan ilk yaklaşım, Gerçek Zamanlı Java Belirtimi'nde (Real-time Specification for Java, RTSJ) (Bollella ve diğerleri, 2000) önerilmiştir. RTSJ, Java programlama dilini, gerçek zamanlı uygulamaların geliştirilmesine olanak tanıyacak şekilde genişletmeyi amaçlamış bir belirtimdir. Bu belirtim, yalnızca bellek yönetimi üzerine bir belirtim değildir. Ancak burada bellek yönetimi üzerine getirdiği yeniliklerden bahsedilecektir.

RTSJ üç temel bellek bölgesi sunar: Ölümsüz bellek (immortal memory), kapsama alınmış bellek (scoped memory) ve yığıt (heap).

Ölümsüz bellek alanı, uygulama çalışmaya başladığında oluşturulup, uygulama boyunca muhafaza edilen bellek bölgesidir. Bu bellek bölgesi uygulama çalıştığı müddetçe kesinlikle temizlenmez. Dolayısıyla uygulama boyunca çalışacak olan iş parçacıklarının nesneleri bu bellek bölgesine yerleştirilir. Bu bellek bölgesi çöp toplayıcıdan korunmuş bir bellek bölgesidir ve gerçek zamanlı iş parçacıkları için kullanılır.

Kapsama alınmış bellek bölgesi, içinde kapsamlar oluşturulabilecek bir bellek bölgesidir. Geçici nesnelere ihtiyaç duyan gerçek zamanlı iş parçacıklarının nesneleri bu bölge içerisinde oluşturulan kapsamlar içerisinde oluşturulur ve tutulur. İlgili iş parçacığı devreye girmeden önce, kapsam oluşturulur. Daha sonra bu kapsama girilir ve iş parçacığı oluşturulan kapsam içerisinde çalıştırılır. Bu iş parçacığı ve ilgili iş parçacıkları bu kapsam içerisinde çalışırlar. Bu esnada oluşturulan diğer iş parçacıkları bu kapsamı paylaşabilecekleri gibi, bu kapsamın altında oluşturulan alt kapsamlarda da çalışmak üzere tasarlanabilirler. Böylelikle bir ağaç yapısı şeklinde kapsamlar kullanılmış olur. Herhangi bir kapsamdaki ve bu kapsamın tüm alt kapsamlarındaki tüm iş parçacıkları işlemlerini tamamlamış iseler bu kapsamdan çıkılır ve kapsam silinir.

Yığıt adı verilen bellek bölgesi ise gerçek zamanlı iş parçacıkları tarafından kullanılmayan bellek bölgesidir ve çöp toplayıcının müdahalesine açıktır.

Bu bellek yönetim sisteminde, dikkat edilmesi gereken en önemli noktalardan birisi referanslardır. Bir kapsamda bulunan nesneler, bulundukları kapsamdan daha kısa ömre sahip bir kapsamdaki nesnelere referans bulunduramazlar. Zira kısa ömre sahip kapsam bellekten silindiği takdirde, bulundurdukları bu referans asılı bir referans olur. Bu güvenliği sağlamak için, RTSJ, kapsamlar içerisinde bulunan nesnelerin birbirlerine olan referanslarını çalışma zamanında yaptığı kontroller yardımıyla denetler. Herhangi bir hata olması durumunda çalışma zamanında hata fırlatılır. Böylelikle RTSJ bellek güvenliğini sağlamış olur.

Bu özellikleri ile RTSJ gerçek zamanlı uygulamalar için uygun bir programlama ortamı sunar. Ancak elbette dezavantajları da mevcuttur. Bunların başında çalışma zamanı kontrollerinin maliyeti gelmektedir. Diğer bir dezavantajı ise, program tasarımı ve geliştirilmesi esnasında, kapsam kullanımının da çözüm uzayına eklenmesi ile birlikte program geliştirme karmaşıklığının ve maliyetinin artmasıdır (Pizlo ve Vitek, 2008).

Bölge temelli yaklaşımı kullanan ancak, çalışma zamanı kontrollerine gerek duymadan bellek güvenliği oluşturmayı amaçlayan birçok farklı çalışma gerçekleştirilmiştir. Bu çalışmaların en temel amacı, bu kontrolleri dilin bir parçası haline getirmektir. Bu konu üzerine birçok araştırma ve çalışma (Andreae ve diğerleri, 2007; Boyapati ve diğerleri, 2003; Christiansen ve diğerleri, 1998; Gay ve Aiken, 2001; Grossman ve diğerleri, 2002; Noble ve diğerleri, 1998; Tofte ve Talpin, 1997; Zhao ve diğerleri, 2004) gerçekleştirilmiştir. Burada bu çalışmalara değinilmeyecektir.

4.1.2. Güvenlik kritik Java

Güvenlik kritik Java (Safety Ciritcal Java, SCJ) (JSR, 2013), JSR 302 uzman ekibi tarafından hali hazırda geliştirilen yeni bir modeldir. Bu model RTSJ üzerine bina edilmektedir. Temel amacı, RTSJ'i sadeleştirerek ve kısıtlayarak, gerçek zamanlı Java uygulamaları geliştirmeyi daha güvenli ve kolay bir hale bir getirmektir. Farklı gereksinimlere sahip gerçek zamanlı uygulamalar için üç temel uyumluluk seviyesi sunar: L0, L1 ve L2. Bugün itibariyle bu uyumluluk seviyelerinin ilki olan ve tek

işlemciyi desteleyen L0 standardı hazır hale getirilmiş ve başarılı bir şekilde uygulanmıştır (Plsek ve diğerleri, 2010).

Bu yaklaşımda RTSJ'de mevcut olan, kapsama alınmış bellek bölgelerinin kullanımı oldukça sınırlandırılmış ve sadeleştirilmiştir. RTSJ'de bulunan ağaç yapısı benzeri bellek organizasyonu, bu belirtimde yığın şeklinde gerçekleştirilmiştir. Aynı zamanda yığıt bellek bölgesi de tamamen ortadan kaldırılmıştır. Temel olarak üç farklı bellek bölgesi mevcuttur: Ölümsüz bellek (immortal memory), görev belleği (mission memory) ve özel bellek (private memory). Yine bu yaklaşımda tüm program akışı görev adı verilen yapılar üzerinden sağlanır ve görevler kendileri ile ilişkili olan bellek bölgelerinde çalışırlar.

Ölümsüz bellek alanı, uygulama boyunca yaşayan bellek alanıdır ve uygulama boyunca yaşayacak olan nesneler bu bellek bölgesine yerleştirilir. Aynı zamanda uygulamanın çalıştıracağı görevleri yöneten bir görev sıralayıcısı da bu alanda mevcuttur. Uygulama çalışmaya başladığı anda görev sıralayıcısı devreye girer ve sırasıyla her bir görev için bir görev belleği atar ve bu bellek içerisinde görevi yönetecek olan görev yöneticisini oluşturur. Daha sonra görev yöneticisi devreye girer ve kendisi ile ilgili nesneleri bu alanda oluşturur. Bu görev belleği, görev sona erdiğinde silinir. Her bir görevin aynı zamanda kendisine ait zamanlanmış nesneleri mevcuttur. Ve bu zamanlanmış nesnelerin kendilerine ait özel bellek alanları vardır. Bu alanlara özel bellek adı verilir. Zamanlanmış nesne görevine başladığında oluşturulur ve görevini tamamladığında silinir.

Bu konu üzerinde şu ana kadar birçok çalışma (Plsek ve diğerleri, 2010; Henties ve diğerleri, 2009; Kalibera ve diğerleri, 2010; Vitek ve diğerleri, 2010; Zhao ve diğerleri, 2009) gerçekleştirilmiştir.

4.2. Gerçek Zamanlı Çöp Toplama

Gerçek zamanlı uygulamaların gereksinimlerini karşılayabilecek bir şekilde gerçekleştirilen çöp toplama işlemine, gerçek zamanlı çöp toplama ismi verilir. Bu işlemi yerine getiren çöp toplayıcılara ise gerçek zamanlı çöp toplayıcı (real-time

garbage collector, RTGC) ismi verilir. RTGC, gerçek zamanlı Java çalışmalarının yoğunlaştığı ve bölge temelli bellek yönetim modellerine alternatif olan bir bellek yönetim modeli getirmeyi amaçlar. Bunu sağlamak adına, var olan çöp toplayıcı sistemlerin, gerçek zaman kısıtlarına uygun olarak çalışabilmesini temin etmeye çalışır. RTGC'lerin en önemli iki bileşeni zamanlama stratejileri ve nesne temsilleridir. Devam eden bölümlerde bu iki bileşene değinilecek ve bu konuda önerilen çözümlerden bahsedilecektir.

4.2.1. Planlama stratejileri

Bir çöp toplayıcı sistemin gerçek zamanlı uygulamalarda kullanımının önünde iki büyük engel mevcuttur: Uygulama etkileşimi ve tahmin edilebilirlik.

Geleneksel çöp toplayıcılar, çöp toplama işlemini yerine getirdikleri esnada çalışan uygulama durdurulur ve bu işlem sona erene kadar uygulama çalıştırılmaz. Bu süre içerisinde, gerçek zamanlı bir iş parçacığı zaman sınırını aşabilir. Bu olay, gerçek zamanlı uygulamalar için kabul edilemez bir durumdur. Dolayısıyla çöp toplayıcının uygulamanın iş parçacıkları ile etkileşiminde daha farklı yöntemler kullanmak gerekmektedir. Bu konuda yardıma artımlı çöp toplayıcılar yetişir. Bir artımlı çöp toplayıcı, çöp toplama işlemini tek bir hamlede gerçekleştirmek yerine küçük artımlar halinde gerçekleştirir. Her bir artım arasında uygulama çalışmasına devam edebilir. Diğer bir ifade ile çöp toplama işlemi uygulama ile dönüşümlü bir şekilde gerçekleştirilir. Böylelikle uygulamanın çalışması esnasında uzun duraklamalar söz konusu olmaz. Öte yandan bu çöp toplayıcıların tasarımı ve uygulaması oldukça zordur. Ancak bu zorluk, gerçek zamanlı uygulamaları desteklemesi düşünüldüğünde çok da önemli olmamaktadır ve bu çözüm RTGC'ler üzerinde uygulanmıştır. Bugün var olan RTGC'ler aslında artımlı çöp toplayıcılarıdır.

Aynı zamanda, geleneksel çöp toplayıcıların, çalışma zamanlamaları tahmin edilemez bir yapıdadır. Ne zaman devreye girecekleri ve ne kadar süre çalışacakları belli değildir. Bu konuda, artımlı çöp toplayıcılar kullanılarak biraz fayda sağlanmıştır. Ancak bir artımlı çöp toplamada, artımların başlama bitiş zamanları ve her bir artımda toplanacak çöp miktarı yine de kesin değildir. Bu sistemlerde her ne kadar uygulama

tamamen durdurulmasa da, yine de bir artım süresinde herhangi bir gerçek zamanlı iş parçacığının zaman sınırını aşmayacağı garanti edilemez. Bu sebeplerden ötürü artımlı çöp toplayıcılarda, artımların zamanlaması belirlenmek zorundadır. Bu konuda üretilen çözümlere zamanlama stratejileri denilmektedir. Günümüzde gerçek zamanlı Java sanal makinelerinde dört farklı planlama stratejisi mevcuttur: Boşluk temelli planlama (slack based scheduling), zaman temelli planlama (time based scheduling), iş temelli planlama (work based scheduling) ve eşzamanlı.

Boşluk temelli planlama, Henriksson (Henriksson, 1998) tarafından ortaya atılmıştır. Gerçek zamanlı Java sanal makinelerinden Mackinac'da (Bollella ve diğerleri, 2005) uygulanmıştır. Bu zamanlama stratejisinde çöp toplayıcı yalnızca gerçek zamanlı iş parçacıklarının çalışmadıkları zamanlarda devreye girer ve çalışır. Bunu sağlamak için, çöp toplayıcı, gerçek zamanlı iş parçacıklarından daha düşük düzeyli bir iş parçacığında çalıştırılır. Dolayısıyla daha yüksek bir gerçek zamanlı iş parçacığı devreye girdiği anda çöp toplayıcı iş parçacığı durdurulur. Böylelikle gerçek zamanlı iş parçacıklarının zaman sınırını aşma problemleri ortadan kaldırılmış olur.

Bacon ve ekibi tarafından (Bacon ve diğerleri, 2003) tarafından ortaya atılmış olan zaman temelli planlama ise gerçek zamanlı Java sanal makinelerinden IBM'in Websphere Real Time (Auberbach ve diğerleri, 2007) sanal makinesinde uygulanmıştır. Bu stratejide, çöp toplayıcı, en yüksek önceliğe sahip olan gerçek zamanlı iş parçacığı ile aynı önceliktedir ve onunla dönüşümlü olarak çalışır. Aynı zamanda çöp toplayıcının her bir artımda ne kadar süre çalışacağı önceden belirlenir. Böylelikle, çöp toplayıcının çalışma süresi garanti altına alınmış olur ve aynı zamanda gerçek zamanlı iş parçacığının işlemini ne kadar kesintiye uğratacağı da bilinir.

Her ne kadar bu zamanlama stratejileri, zamanlama konusunda büyük bir yenilik getirse de gerçek zamanlı uygulama geliştirme aşamasında oldukça dikkatli bir zamanlama analizi yapılarak uygun bir zamanlama stratejisi seçilmeli ve gereken çalışma süreleri önceden belirlenmelidir. Bu iki zamanlama stratejisinin daha detaylı bir incelemesi için (Kalibera ve diğerleri 2009, 2011) kaynakları incelenebilir.

İş temelli zamanlama stratejisinin ilk örneklerinden bir tanesi Baker (Baker, 1978) tarafından ortaya atılmıştır. Gerçek zamanlı Java sanal makinelerinden Jamaica VM sanal makinesinde (Siebert 2004, 2007) uygulanmıştır. Bu stratejide çöp toplayıcının çalışmaya başlama zamanı ve süresinden ziyade çöp toplama miktarı önemlidir. Uygulamanın gerçekleştirdiği bellek ayırma miktarına karşılık temizlenmesi gereken bellek miktarı hesaplanır ve çöp toplayıcı devreye girdiğinde hesaplanan miktar kadar çöp toplama işlemi gerçekleştirir ve daha sonra devreden çıkar.

Eşzamanlı strateji, çok çekirdekli, çok işlemcili gömülü sistemlerin gelişmesi ile birlikte üzerinde çalışılmaya başlanmış bir stratejidir. Gerçek zamanlı Java sanal makinelerinden Fiji VM (Pizlo ve diğerleri 2010a, 2010b) sanal makinesinde uygulanmıştır. Bu strateji, çöp toplama işleminin tamamen ayrı bir işlemci ya da çekirdek üzerinde gerçekleştirilmesi temeline dayanır. Elbette yine zamanlama analizlerine ve uygulama etkileşimi analizlerine ihtiyaç mevcuttur. Zira bellek ayırma sürecinde, uygulama çöp toplayıcıyı beklemek zorunda kalacaktır.

4.2.2. Nesne temsilleri

Çöp toplayıcı ifadesi sözel olarak düşünüldüğü zaman yalnızca ölü nesnelerin bellekten temizlenmesi olarak anlaşılabilse de durum böyle değildir. Bir çöp toplayıcı sistem, çöp toplama işleminin yanı sıra bellek ayırma işleminden ve bellek organizasyonundan da sorumludur. Çöp toplayıcılar küçük nesneler için oldukça hızlı bir şekilde bellek ayırma işlemini yerine getirebilirken, durum büyük ya da çok büyük nesneler için farklıdır. Yığıt üzerinde büyük nesneleri yerleştirebilecek miktarda boş bellek bulma işlemi zaman zaman oldukça uzun süren bir işlem olabilmektedir. Bu uzun süreç gerçek zamanlı iş parçacıklarının zaman sınırını aşmalarına sebep olabilir. Günümüzde gerçek zamanlı Java sanal makinelerinde, iki farklı nesne temsili bulunmaktadır: Bitişik nesne temsili ve parçalı nesne temsili.

Bitişik nesne temsili, geleneksel nesne temsilidir. Tüm nesneler, kendileri için gereken miktarda bir bellek bölgesine bir bütün halinde yerleştirilirler. Bu temsilde büyük nesneler için bellek bulma işlemi özellikle parçalanmış belleklerde oldukça uzun bir süre alabilir. Bu temsili kullanan gerçek zamanlı Java sanal makinelerinde iki olguya

dikkat etmek gerekmektedir. Bellek sıkıştırma (birleştirme) yani nesnelerin bellek üzerinde taşınarak, birbiri ardına yerleştirilmesi suretiyle parçalanmış bellek yapısının ortadan kaldırılması işlemi, muhakkak mevcut olması gereken özelliklerden birisidir. Diğer yandan büyük nesnelere uygun boş bellek alanı bulmak için gerçekleştirilen arama algoritmalarının mümkün olduğunca verimli olması gerekmektedir. Bu temsil Ovm sanal makinesinde tercihli olarak kullanılabilmektedir. Bu temsili kullanan sistemler, bellek ayırma sürecinde yavaş kalırken, büyük nesnelerin alanlarına erişimde oldukça hızlıdırlar. Bu çalışmada büyük nesnelere bellek ayırmak için yeni bir arama algoritması tasarlanmış ve Ovm sanal makinesi üzerine uygulanmıştır.

Parçalı nesne temsili, yukarıda bahsedilen dezavantajları ortadan kaldırmaya yönelik bir temsil şeklidir. Bu temsilde nesneler, bir bütün halinde belleğe yerleştirilmezler. Bunun yerine, küçük parçalara ayrılırlar ve her bir parçaları kendileri için gereken miktarda bir bellek bölgesine yerleştirilir. Bu temsilde, nesnelerin parçaları belleğin farklı bölgelerinde bulunur. Neneler bağlı listeler ya da ağaç yapıları gibi yapılar halinde organize edilirler. Websphere Real-Time ve Ovm (tercihli) sanal makinesinde yalnızca diziler için ayrık nesne temsili kullanılırken, Jamaica VM sanal makinesinde tüm nesneler için ayrık nesne temsili kullanılır. Bu temsilin en büyük avantajı büyük nesneler için bellek ayırma işlemini hızlandırmasıdır. Ancak büyük nesnelerin alanlarına erişim süresi bitişik nesne temsiline oranla daha yavaştır.

BÖLÜM 5. MINUTEMAN RTGC ÇATISI

Gerçek zamanlı Java sanal makineleri üzerine yapılan çalışmalardaki en önemli noktalardan birisi, sanal makine üzerine uygulanmış olan gerçek zamanlı çöp toplayıcının testi ve diğer çöp toplayıcılarla karşılaştırılmasıdır. Java sanal makinelerinin, gerçek zamanlı sistemlerde kullanımının önündeki en büyük engellerden birisinin, çöp toplayıcıların tahmin edilemeyen çalışma yapıları olduğu düşünüldüğünde bu konuya neden oldukça fazla önem verildiği anlaşılır. Çöp toplayıcıların testlerinde ve mukayesesinde uygulanan genel yöntem, bir test uygulamasının, farklı çöp toplayıcılara sahip farklı sanal makineler üzerinde koşturulması temeline dayanır. Bu yöntemde öncelikle bir test uygulaması seçilir ve bir hedef platform belirlenir. Test uygulaması belirlenen hedef platform üzerinde her bir Java sanal makinesi ile ayrı ayrı koşturulur. Elde edilen test sonuçları mukayese edilir ve böylelikle, her bir Java sanal makinesi üzerinde koşan farklı çöp toplayıcıların performansına ilişkin çıkarımlar gerçekleştirilir.

Yukarıda bahsi edilen yöntem her ne kadar, en çok kullanılan yöntem olsa da ciddi bir dezavantajı mevcuttur. Bu yöntemle gerçekleştirilen testlerde, Java sanal makinesinin bizzat kendisinin performansı da test sonuçlarına bir parametre olarak eklenir. Bu parametreyi test sonuçları içerisinden ayıklayarak yalnızca çöp toplayıcının saf performans değerlerine ulaşmak ise mümkün değildir. Java sanal makinelerinin denkleme dâhil olduğu bu durum, optimizasyonlar ile daha da karmaşık hale gelmektedir. Üreticilerin, hedef kitlelerinin ihtiyaçlarını göz önünde bulundurarak, sanal makinelerini belli platformlar ve belli tipte uygulamalar için optimize etmiş olmaları olasılığı her zaman mümkündür. Böyle bir durumda, test platformu ve uygulaması için optimize edilmiş olan bir sanal makinenin üzerinde bulunan çöp toplayıcının performansı kendisinden kaynaklanmayan sebeplerden ötürü daha iyiymiş gibi görünebilir. Aynı şekilde bunun tam tersi bir durum da söz konusu olabilir.

Gerçek zamanlı çöp toplayıcıları sağlıklı bir şekilde test edebilmek için uygulanabilecek diğer bir çözüm, test edilecek olan tüm çöp toplayıcıları aynı sanal makine üzerine bütünleştirmek ve böylelikle sanal makine parametresini test sonuçlarından hariç tutmaktır. Minuteman, bu ihtiyacı çözüme kavuşturmak için geliştirilmiş ve Ovm sanal makinesi üzerine uygulanmış, tek işlemcili gerçek zamanlı çöp toplama çatısıdır (Kalibera ve diğerleri, 2009). Bu çalışmada tasarlanan algoritma Minuteman çatısı üzerine uygulanmış ve bu çatı yardımı ile test edilmiştir. Bu bölümde Minuteman çatısı ve bellek yönetimi anlatılacaktır.

Minuteman çatısının, yüksek seviyede yapılandırılabilir modüler bir yapısı mevcuttur. Tak çıkar özelliği ile farklı gerçek zamanlı çöp toplayıcıları oluşturulabilir ve bu çöp toplayıcıları aynı sanal makine üzerinde test edilebilir. Böylelikle sanal makine farklılıklarının test sonuçlarına etki etmesinin önüne geçilerek daha sağlıklı bir test işlemi gerçekleştirilmiş olur.

Minuteman iskeletinde, gerçek zamanlı çöp toplayıcılar, birçok farklı yapılandırma seçeneğine göre oluşturulabilseler de en temel yapılandırma seçeneklerini 4 başlık altında toplayabiliriz: Planlama yapılandırması, artım yapılandırması, sıkıştırma yapılandırması ve dizi temsili yapılandırması. Bu yapılandırması seçenekleri aşağıda ayrıntılı bir şekilde açıklanmıştır.

Planlama yapılandırması, çöp toplayıcının kullanacağı anahtarlama stratejisinin seçimi için kullanılmaktadır. Üç adet yapılandırma seçeneği mevcuttur: boşluk temelli planlama stratejisi (Henriksson, 1998), zaman temelli planlama stratejisi (Bacon ve diğerleri, 2003) ve her iki stratejinin bir arada kullanımına olanak tanıyan karma planlama stratejisi.

Artım yapılandırması, çöp toplayıcının toplama artımının seçiminde kullanılır. Başlıca seçenekleri artımsız toplama, seçimli artımlı toplama ve tam artımlı toplamadır.

Sıkıştırma yapılandırması, çöp toplayıcının sıkıştırma yapıp yapmayacağını ve eğer yapacaksa bunu nasıl yapacağını belirler. Eğer sıkıştırma aktif edilmezse çöp toplayıcı

nesneleri hareket ettirmeyen kipte çalışır. Eğer aktif edilirse nesnelerin hareket ettirildiği kipe geçilir ve belleğin parçalanma durumuna göre nesneler yer değiştirilerek sıkıştırma işlemi uygulanır.

Dizi temsili yapılandırması, çöp toplayıcının, dizileri bellekte nasıl temsil edeceğinin ayarlanmasında kullanılır. Diziler, bitişik ya da parçalı yapıda temsil edilebilir. Bitişik yapıda temsil edilen diziler, bütün elemanları ile birlikte tek bir bellek bölgesine yerleştirilir. Ayrık yapıda temsil edilen dizilerin elemanları belli boyutlarda parçalara bölünerek farklı bellek bölgelerine yerleştirilir. Bu dizilerde arraylet (Bacon ve diğerleri, 2003) yapısı kullanılır.

Minuteman çatısının bir diğer önemli özelliği, sahip olduğu bellek ayırma stratejileridir. Minuteman, üç farklı bellek ayırma stratejisine sahiptir. Talep edilen bellek boyutu miktarına ve dizi temsili yapılandırmasına göre bu üç bellek ayırma stratejisinden birisini seçer. Yapılan çalışma, bellek ayırma stratejileri üzerine olduğundan ötürü devam eden bölümlerde, öncelikle Minuteman iskeletinin bellek organizasyonunu (bellek yönetimi) nasıl gerçekleştirdiğine ve daha sonra bellek ayırma stratejilerine ayrıntılı bir şekilde değinilmiştir.

5.1. Bellek Organizasyonu

Ovm, ilk çalışma anında işletim sisteminden belli bir miktarda bellek alır. Alınacak bellek miktarı, Ovm’in derlenmesi esnasında heap-size parametresiyle kullanıcı tarafından belirlenir. Alınan bellek daha sonra Minuteman iskeletine, yönetmesi için devredilir. Bu bellek bölgesine yığıt adı verilir. Minuteman kendisine verilen yığıtın boyutunu sınıf kurucusuna gönderilen memSize adındaki parametre yardımıyla edinir ve yine kendi içinde aynı isimle tanımladığı alana bu değeri atar. memSize alanı tamsayı veri tipindedir ve bayt cinsinden yığıtın boyutunu depolar.

Minuteman, yığıtı, blok adı verilen bölgelere ayırarak kullanır. Her bir bloğun boyutu blockSize adındaki tamsayı veri tipinde bir alanda tutulur. Bu alan da memSize alanı gibi blok boyutunu bayt cinsinden depolar. Varsayılan değeri 2048'dir. Bu, her bir

bloğun boyutunun 2048 bayt yani 2KB olacağı anlamına gelir. İstenildiği takdirde bu değer değiştirilerek blok boyutu farklı değerlere ayarlanabilir.

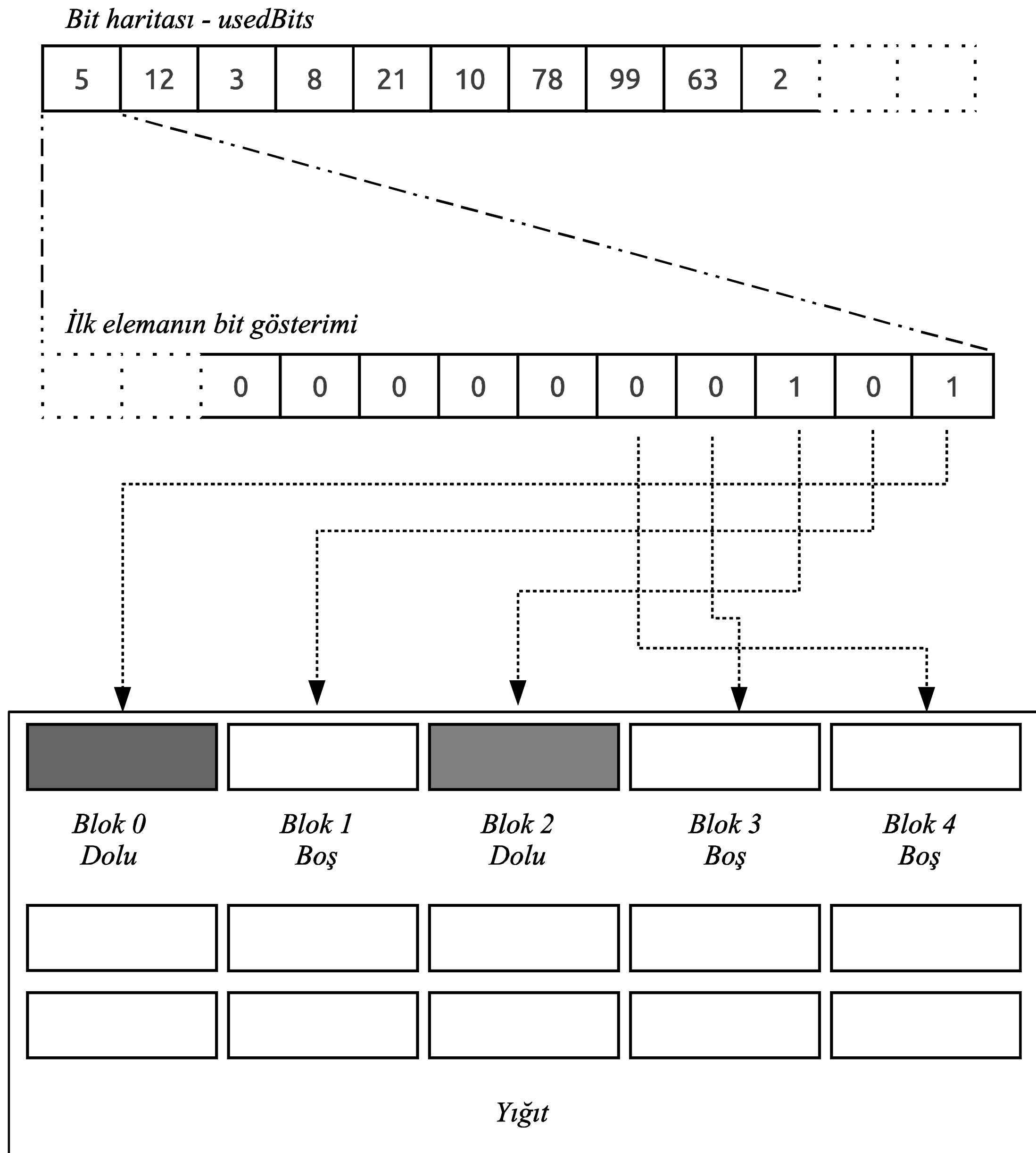
Minuteman, sınıf kurucusu içerisinde yığıt boyutunu aldıktan sonra, memSize alanını blockSize alanına bölerek, yığıt içerisinde toplam kaç tane blok bulunabileceğini hesaplar ve blok sayısını nBlocks adı verilen tamsayı veri tipinde bir alanda saklar.

Her bir bloğun kullanım durumunu takip etmek ve kaydetmek için Minuteman iskeleti, bir bit haritası (bit vektörü) kullanır. Bit haritasında bulunan her bir bit, temsil ettiği bloğun kullanılıp kullanılmadığının bilgisini tutar. Eğer bit 1 ise temsil ettiği blok kullanımdadır. Eğer 0 ise kullanımda değildir. Bu bit haritası, Minuteman içerisinde usedBits adı verilen, tamsayı veri tipinde bir dizi ile temsil edilmektedir. Dizinin elemanları 32 bit uzunluğunda tamsayılardır. Dolayısıyla dizinin her bir elamanı 32 adet bloğun durumunu depolamak için kullanılır. Dizinin elaman sayısını belirleyebilmek için ise yığıt içerisindeki blokların kaç adet 32 bitlik tamsayı ile temsil edilebileceği hesaplanmalıdır. Bu hesaplama aşağıda, Denklem 5.1'de gösterilen formül kullanılarak gerçekleştirilir ve sonucu nBitWords adı verilen bir alanda saklanır.

$$nBitWords = \frac{nBlocks + 31}{32} \qquad \textbf{(5.1)}$$

nBitWords alanında tutulan değer, daha sonra usedBits dizisi, kurucu içerisinde ilklendirilirken dizinin eleman sayısı olarak kullanılır. Böylelikle, blokların kullanım durumlarının temsili için gerekli olan bit haritası oluşturulmuş olur.

Yığıt içerisindeki blokların, bit haritası yardımıyla temsili Şekil 5.1.'de görülmektedir. Şekilde usedBits adlı bit haritasının örnek veriler içeren ilk on elemanı gösterilmektedir. Devamında usedBits dizisinin ilk elemanının bit gösterimi mevcuttur. Bu gösterimde alt seviye bitten başlamak üzere ilk on bit görülmektedir. Hemen devamında ise her bir bitin temsil ettiği blok, yığıt içerisinde gösterilmiştir. Dolu olarak belirtilen yani kullanımda olan bloklar gri renk ile boş olarak belirtilen yani kullanıma müsait olan bloklar ise arka planı boş olarak temsil edilmektedir.



Şekil 5.1. Minuteman Bellek Organizasyonu

5.2. Bellek Ayırma Stratejileri

Dilsel anlamı açısından bakıldığında, bellek ayırma işleminin bir çöp toplayıcı sisteminin görevi olmadığı kanısına varılabilir. Ancak gerçek bundan farklıdır. Çöp toplayıcıların görevleri yalnızca bellekteki ölü nesneleri temizlemek değildir. Aynı zamanda, sanal makine tarafından oluşturulacak yeni nesneler için, yığıt içerisinde

nesnelerin boyutlarına uygun boş alanı bulmak ve bu alanın adresini sanal makineye geri döndürmekle de yükümlüdürler. Bu işleme bellek ayırma adı verilmektedir.

Bellek ayırma çöp toplayıcıların gerçekleştirdiği önemli işlevlerden bir tanesidir. Gerçek zamanlı çöp toplayıcılarda bu işlev daha da önem kazanmaktadır. Zira gerçek zamanlı sistemlerde birçok iş parçacığı zaman kısıtlarına sahiptirler ve gelen bellek ayırma taleplerine, uygun bir zaman dilimi içerisinde cevap verebilmek oldukça önemlidir. Bu ihtiyaçtan ötürü Minuteman, farklı nesne türlerine göre farklı stratejiler kullanarak bellek ayırma işlemini mümkün olduğunca hızlı bir şekilde yerine getirmeye çalışır.

Çöp toplama terminolojisinde ve dolayısıyla Minuteman iskeletinde de iki farklı nesne kavrayışı mevcuttur: normal nesneler ve büyük nesneler. Çöp toplama sisteminin bellek ayırma stratejilerinin çalışma sistemine ve verimliliğine göre, belli bir eşik boyut belirlenir. Bu eşik boyuttan daha küçük boyuta sahip olan nesneler normal nesneler, büyük olanlar ise büyük ya da duruma göre çok büyük nesneler olarak adlandırılır.

Minuteman iskeletinde, bir blok içerisine yerleştirilebilecek en büyük nesne boyutu computeMaxBlockAlloc() metodu kullanılarak hesaplanır ve hesaplanan boyut maxBlockAlloc adı verilen bir alanda tutulur. Bu değer her zaman blok boyutu ile aynı olmaz. Uygun bir bellek geometrisi sağlayabilmek adına Minuteman, tüm nesnelerin adreslerini 16 değerinin katı olacak şekilde hesaplar. Aynı zamanda bellek talep edilen nesnenin boyutunu da 16 değerinin katı olacak şekilde yukarıya doğru yuvarlar. Dolayısıyla, varsayılan ayarlarında Minuteman'in blok boyutu 2048 bayt iken bir blok içerisine yerleştirilebilecek en büyük nesne boyutu 1936 bayt olarak hesaplanmaktadır.

Burada hesaplanan maxBlockAlloc değeri Minuteman'in bellek ayırma stratejisini belirlemekte oldukça önemli bir yere sahiptir. Zira bu değer Minuteman iskeletinin normal nesne, büyük nesne ayrımını gerçekleştirmesine yardımcı olmaktadır. Eğer bellek talep edilen nesnenin boyutu maxBlockAlloc değerinden küçük ise bu nesne

normal nesne olarak değerlendirilir. Eğer bu değerden daha büyük ise büyük nesne olarak değerlendirilir.

Minuteman, üç adet bellek ayırma stratejisine sahiptir: normal nesneler için bellek ayırma, diziler için bellek ayırma ve büyük nesneler için bellek ayırma. Devam eden bölümlerde bu üç bellek ayırma stratejisi açıklanmaktadır.

5.2.1. Normal nesneler için bellek ayırma

Ayrık serbest listeler (segregated free lists) bilinen bellek ayırma algoritmalarından bir tanesidir. Minuteman, normal nesnelere bellek ayırmak için bu algoritmanın kendisi için özelleştirilmiş bir uygulamasını kullanmaktadır. Bu bölümde, bu uygulamanın nasıl gerçekleştirildiği anlatılacaktır.

Minuteman, normal nesne yerleşimlerinde, yığıt içerisindeki her bir bloğu tamamen aynı boyuttaki nesneler için kullanır. Diğer bir deyişle farklı boyutlardaki nesneler farklı bloklar içerisine yerleştirilir. Minuteman nesne boyutlarını 16 değerinin katı olacak şekilde yorumlar ve blok yerleşimlerini buna göre gerçekleştirir. Örneğin 16 byte boyutunda bir nesnenin yerleştirilmiş olduğu bir bloğa daha sonra yalnızca 16 byte boyutundaki nesneler yerleştirilir. Farklı boyuttaki bir nesne geldiği zaman o nesnenin kendi boyutu için kullanılan blok tespit edilir ve ilgili nesne kendi boyutunda nesnelerin bulduğu bloğa yerleştirilir. Nesnelere, boyutlarına göre kendileri için ayrılmış bloklardan bellek tahsis etme işlemini yönetebilmek için, Minuteman, ayrık serbest listeler ve sizeClass adı verilen bir yapı kullanır.

sizeClass bir sınıftır ve bir nesne boyutunu temsil eder. 16 değerinin maxBlockAlloc değerine kadar olan tüm katlarına ilişkin nesne boyutlarını temsil edebilmek adına, her bir katı için bir sizeClass oluşturulur. Örneğin 16 bayt için bir sizeClass, 32 bayt için bir sizeClass, 48 bayt için bir sizeClass mevcuttur. sizeClass'ın taşıyacağı en büyük değer ise maxBlockAlloc alanında tutulan değerdir. Varsayılan yapılandırmada bu değer 1936'dır. sizeClass içersindeki en önemli alanlar, temsil ettiği boyutu gösteren size, halen kullandığı bloğun adresini gösteren block, kullanılan blok içerisinde atama yapılacak olan ilk adresi gösteren blockCur ve eğer parçalı bir blok içerisinden atama

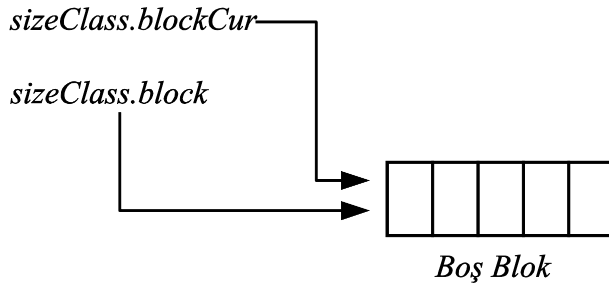
yapıyorsa, ilgili bağlı liste içerisindeki indisini tutan nonFullHead alanlarıdır. sizeClass yapısının ve nasıl kullanıldığının anlaşılması için, bellek ayırma algoritmasının kısaca anlatılması yerinde olacaktır.

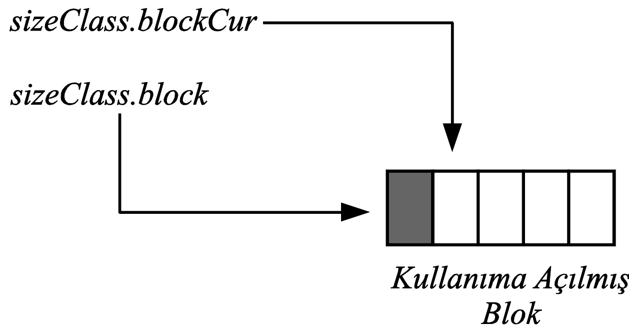
Bu bellek ayırma stratejisi devreye girdiğinde, talep edilen bellek boyutu 16 değerinin katlarına yuvarlanır ve hangi sizeClass içine yerleştirileceği hesaplanır. Daha sonra sizeClass'ı tespit edilir ve içerisindeki nonFullHead alanına bakılır. Bu alandaki değer -1 ise normal bir bloktan, değilse parçalı bloktan atama yapılacağı anlaşılır.

Eğer normal bir bloktan atama yapılacaksa, çarpan işaretçi (bump pointer) ayırması adı verilen bir yöntem kullanılır. Tespit edilmiş olan sizeClass'ın, block alanı kontrol edilir. Eğer bu alan boş ise, bu nesne için atama yapılacak hazırda bir blok yok demektir. Bu durumda bellekten boş bir blok alınır ve bloğun adresi block ve blockCur alanlarına yazılır. blockCur alanındaki değer, size alanındaki değer miktarınca artırılır. Böylelikle blockCur alanında, bir sonraki ayırma işleminde geri döndürülecek değer saklanmış olur. blockCur alanının artımdan önceki değeri ise bulunan boş belleğin adresi olarak geriye döndürülür. İstenilen alan bulunmuştur.

Eğer block alanındaki değer boş değil ise hali hazırda içinde boş alan bulunan bir bloktan atama yapılıyor demektir. Bu durumda, atama yapılabilecek sıradaki bellek bölgesinin adresini taşıyan blockCur alanındaki değer geri döndürülür ve aynı zamanda blockCur alanındaki değer, size alanındaki değer miktarınca artırılır ve bir sonraki ayırma işleminde geri döndürülecek adresi tutar. Her bir bellek isteğinde bu adımlar tekrar edilir.

Bu yöntemin görsel bir temsili Şekil 5.2.'de gösterilmiştir. Şekil 5.2.a'da yeni alınmış boş bir blok görülmektedir. Bu durumda block ismi verilen alan (sizeClass.block) ile çarpan işaretçi olarak kullanılan ve blockCur ismi verilen alan (sizeClass.blockCur) aynı adresi yani bloğun başlangıç adresini göstermektedir. Şekil 5.2.b'de ise bloğa bir nesne yerleştirildikten sonraki durum görülmektedir. Bu durumda block alanı yine bloğun başlangıç adresini gösterirken, blockCur alanı ise nesne boyutu kadar ileriye kaydırılmıştır ve bloktaki boş alanın başlangıç adresini göstermektedir.





Şekil 5.2. Çarpan işaretçi kullanımı

Eğer parçalı bir bloktan atama yapılacaksa durum biraz daha karışık hale gelir. Bu durumda bağlı listeleri kullanmak gerekecektir. Her bir parçalı blok içerisinde bellek ayırması yapılabilecek ilk adresi tutmak için freeHeadsPtrs adında bir dizi mevcuttur. Bu dizinin boyutu blok sayısı kadardır. Herhangi bir blok parçalanmış ise, bu dizide bloğun indis numarasına karşılık gelen dizi elemanında, bloğun atama yapılabilecek ilk boş alanının adresi tutulur. Blok parçalanmış değilse -1 değeri tutulur. sizeClass içerisindeki nonFullHead alanındaki -1'den farklı olan değer, freeHeadsPtrs dizisinin bir indisidir ve bu sizeClass tarafından kullanılabilecek parçalı bir blok olduğunu ifade eder. Bu değer okunur, dizinin ilgili indisine gidilir ve oradaki adres bilgisi alınır. Bu adres parçalı blok içerisinde atama yapabileceğimiz bir alanı göstermektedir. Geriye döndürülecek olan adres değeri budur. Ancak bu değeri geriye döndürmeden önce, blok içerisindeki bir sonraki boş alanın adresi dizinin, ilgili elemanına kaydedilmelidir. Bu değer, tam da bu adreste tutulmaktadır. Bellekte, alınan adrese gidilir ve oradaki değer okunarak freeHeadsPtrs dizisinin, üzerinde işlem yapılan elemanına yazılır.

Böylelikle bir sonraki elemanın adresi de kolaylıkla bir sonraki istekte edinilebilecektir.

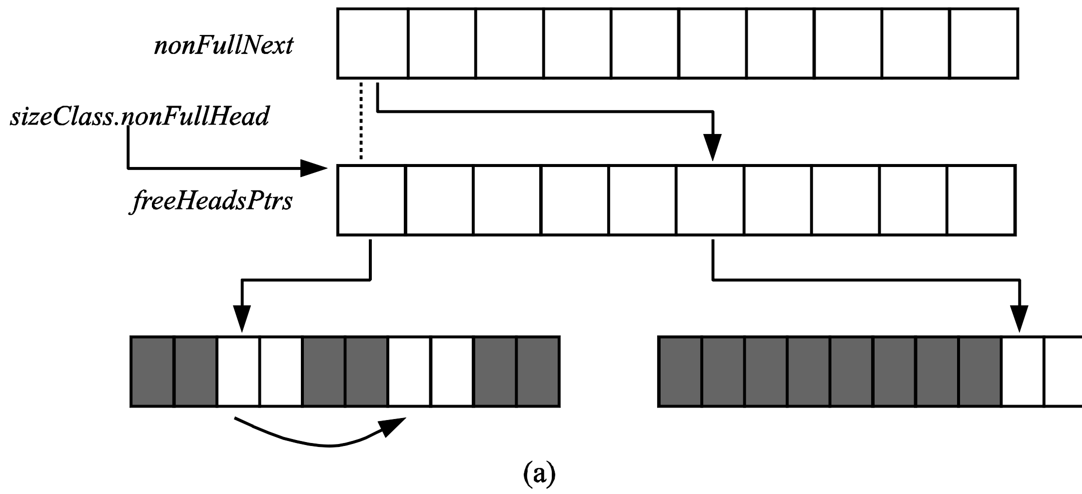
Eğer bu talep sonrasında blokta boş alan kalmaz ise, nonFullNext adı verilen bir diziden yardım alınır. Bu dizi aynı indis numaralı elemanında, aynı sınıf boyutu için bir sonraki parçalanmış bloğun adresinin freeHeadsPtrs dizisinin kaçıncı elemanında (indis numarası) olduğunu saklar. Eğer -1 ise, ilgili sizeClass için parçalanmış başka bir blok yok demektir ve sizeClass'ın nonFullHead alanı -1'e döndürülür. Ancak -1 değilse, bu alandaki değer alınarak sizeClass'ın nonFullHead alanına yazılır ve bu elemanın değeri -1 yapılır. Böylelikle sizeClass'ın bir sonraki istekte nereye bakması gerektiği belirlenmiş olur. Bu yöntemin görsel temsili adım adım Şekil 5.3.'de gösterilmiştir.

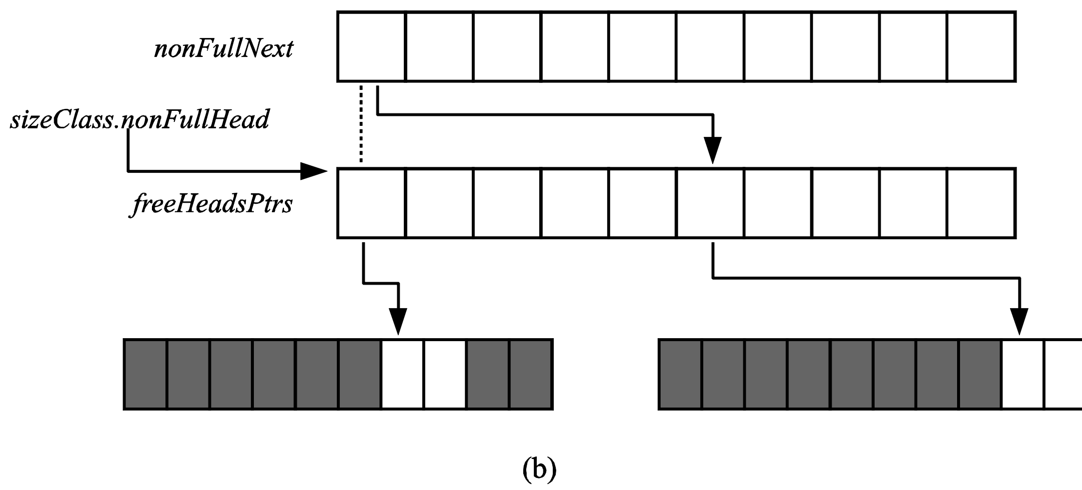
5.2.2. Diziler için bellek ayırma

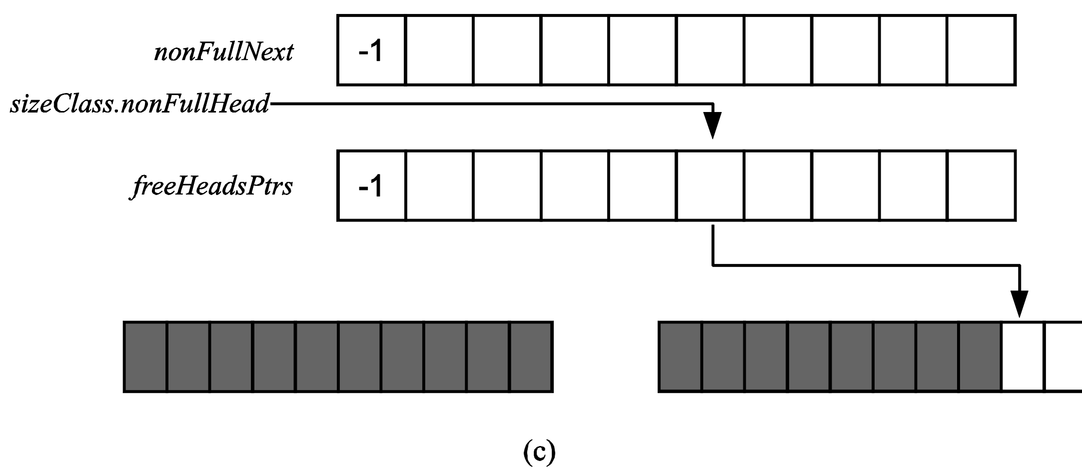
Minuteman iskeletinde iki farklı dizi temsili mevcuttur: birleşik diziler ve ayrık diziler. Her iki dizi temsili için Minuteman, farklı bellek ayırma stratejileri kullanmaktadır. Bu bölümde, bu dizi temsillerinden ve bunlara ilişkin kullanılan bellek ayırma stratejilerinden bahsedilecektir.

Bilindiği üzere diziler birden fazla eleman sayısına sahip olan yapılardır. Boyutları, dizinin eleman sayısına bağlı olarak genellikle normal bir nesneden daha büyük olmaktadır. Dizi boyutlarının, her zaman için bir blok boyutunu aşma olasılığı mevcuttur. Dolayısıyla, diziler için bellek ayırma işlemi üzerine de ayrı olarak eğilmek gerekmektedir.

Minuteman çatısında kullanılan ilk dizi temsili bitişik dizilerdir. Bu dizi yapısı, bilinen ve en çok kullanılan yapıdır. Diziler diğer tüm nesneler gibi, özelliklerini belirten bir başlık alanına sahiptirler. Başlık alanının devamında ise ardışık bir şekilde dizinin elemanları sıralanır. Dizi bir bütün halinde aynı bellek bölgesinde bulunur.



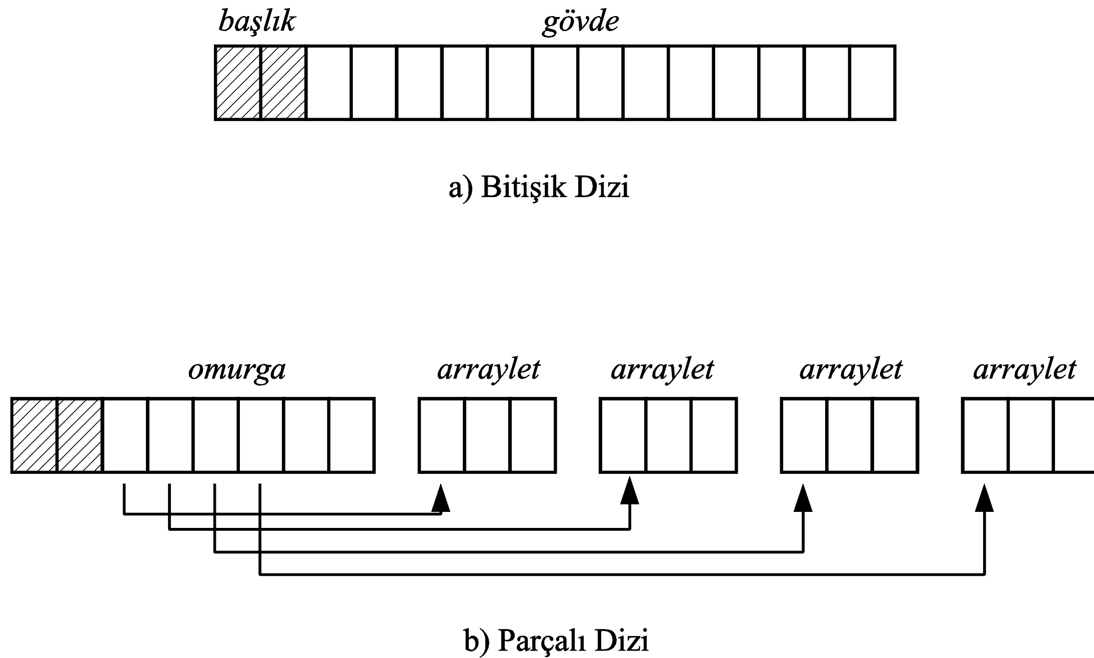




Şekil 5.3. Size class yapısı

Parçalı dizilerde ise dizi elemanlarının ardışık bir şekilde aynı bellek bölgesinde bulunma zorunluluğu yoktur. Dizi elemanları eşit boyutlarda gruplara ayrılırlar ve her bir grup bellekte farklı bir alana yerleştirilebilir. Minuteman, bu dizi temsilini, ilk olarak 2003 yılında Bacon ve ekibi tarafından geliştirilmiş olan arraylet adı verilen bir yapı kullanarak gerçekleştirir (Bacon ve diğerleri, 2003). Arraylet yapısında, dizinin başlığının ve gövdesinin yanı sıra bir de omurgası (spine) mevcuttur. Dizinin omurgası her bir grubun bellekteki başlangıç adresini tutan bir yapıdır.

Her iki dizi temsili Şekil 5.4.'de gösterilmiştir. Şekil 5.4a'da bitişik dizi örneği görülmektedir. Bu örnekte dizinin başlığı ve gövdesi yani elemanları bir arada ve tek bir bellek bölgesinde bir biri ardına bulunur. Şekil 5.4b'de ayrık dizi yapısı görülmektedir. Parçalı dizi yapısında, oluşturulan her bir gruba arraylet adı verilir. Bunun yanı sıra konu ile ilgili yapılmış çalışmalarda bu yapıda oluşturulmuş olan dizilere de arraylet ismi verilmektedir. Arraylet yapısında ise dizinin başlığı ve omurgası aynı bellek bölgesinde bir arada bulunur. Ancak her bir grup dizi elemanı bellekte farklı bölgelerde bulunabilir. Dizinin bu elemanlarına erişmek ise omurgadaki adresleri kullanılır.



Şekil 5.4. Dizi temsili

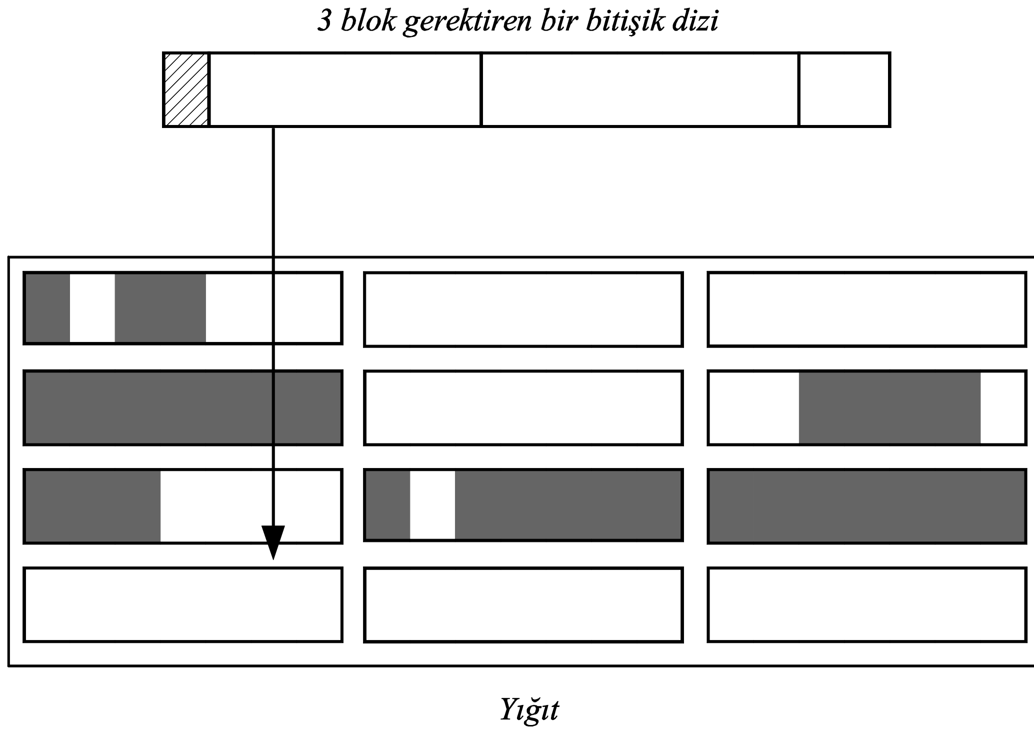
Minuteman, dizi temsillerinin kullanıcı tarafından seçilebilmesi için bir yapılandırma parametresine sahiptir. Mantıksal bir veri tipine sahip olan ARRAYLETS isimli alanın değeri doğru olarak seçilirse, dizi temsilinde arraylet yapısı kullanılır. Eğer bu alanın değeri yanlış olarak seçilirse klasik dizi temsili olan birleşik dizi temsiline geçilir.

Birleşik dizi temsili kullanıldığı durumda, gelen dizi yer ayırma taleplerine, Minuteman, dizinin boyutunu inceleyerek başlar. Eğer dizinin boyutu, maxBlockAllocSize değerinden küçük ise, normal nesneler için bellek ayırma stratejisini devreye sokar. Eğer bu dizinin boyutu ise bu değerden büyük ise büyük nesneler için bellek ayırma stratejisini kullanır.

Birleşik dizi yapısını kullanmanın hem avantajlı hem dezavantajlı yönleri mevcuttur. Yukarıda anlatılanlardan açık bir şekilde görüldüğü üzere, Minuteman'de birleşik dizilere bellek ayırma süreci oldukça basittir. Ancak bu, bellek ayırma işleminin hızlı bir şekilde tamamlanabileceği anlamına gelmez. Dizinin boyutunun blok boyutunu aştığı durumlarda, dizinin sığabileceği sayıda blok, bütün bellek içerisinde doğrusal bir arama sonucunda bulunabilir. Diğer yandan dizi, birleşik bir dizi olduğundan dolayı gereken boş blok sayısı bellekte ardışık bir şekilde bulunmak zorundadır. Bu iki etken, birleşik dizilere bellek ayırma sürecini oldukça uzatabilir. Bellek ayırma işleminin atomik olduğu düşünülürse, uzayan bu sürecin gerçek zamanlı Java sistemlerinde, nasıl bir etkide bulunacağını tahmin etmek güç değildir. Örneğin, bir diziye yer ayırma işlemi esnasında, başka bir iş parçacığının çalışma vakti gelmiş olabilir. Bu durumda, diziye yer ayırma işlemi tamamlanmadan önce bu iş parçacığı aktif hale getirilemez. Eğer bu iş parçacığı gerçek zamanlı bir iş parçacığı ise diziye bellek ayrılmasını beklerken zaman sınırını aşabilir. Böylesi bir durum, sistemde istenmeyen problemlere yol açabilir.

Birden fazla blok gerektiren birleşik dizilere bellek ayırma süreci, Şekil 5.5.'de, bir örnek üzerinden grafiksel olarak gösterilmiştir. Şekildeki örnekte, Minuteman'e, 3 adet blok gerektiren bir dizi talebi geldiği varsayılmaktadır. Minuteman, bu talebi karşılayabilmek adına, büyük nesneler için bellek ayırma rutinlerini devreye sokacak ve bellekte ardışık bir şekilde yer alan 3 adet boş blok bulmaya çalışacaktır. Belleğin başlarında bulunan 3 adet boş blok ardışık bloklar olmadıklarından ötürü dizi için

kullanılamaya müsait değillerdir. Ancak, belleğin sonunda bulunan 3 adet boş blok, ardışık olduklarından ötürü, kullanılabilirler. Böyle bir durumda, Minuteman, uygun bir boş alan bulabilmek için, belleğin sonuna kadar arama işlemi gerçekleştirmek zorundadır.



Şekil 5.5. Bitişik dizi bellek ayırma süreci

Kritik dezavantajının yanı sıra, birleşik dizi yapısının çok önemli bir avantajı da mevcuttur. Dizi yapısının birleşik olmasından ötürü, dizinin herhangi bir elemanına erişim oldukça kolaydır. Dizinin başlangıç adresi ve talep edilen elemanın indis numarası bilgileri kullanılarak basit bir işaretçi aritmetiği ile talep edilen elemanın adresi kolaylıkla hesaplanabilir. Bu hesaplama işlemi için kullanılan formüller aşağıda Denklem 5.2 ve 5.3'de verilmiştir.

$$byteOfset = başlıkBoyutu + (indisNo * bileşenBoyutu) \quad \textbf{(5.2)}$$

$$adres = başlangıçAdresi + byteOfset \quad \textbf{(5.3)}$$

Formüldeki bileşen boyutu adlı eleman, dizi elemanlarının veri tipinin bellekte ne kadar yer kapladığını bildiren bayt cinsinden bir elemandır. İlk formülde dizi elemanının indis numarasına göre bayt cinsinden ofset değeri hesaplanır. İkinci formülde ise hesaplanan ofset değeri dizinin başlangıç adresine eklenerek, ilgili elemanın bellekteki adresi bulunur.

Bu dizi temsilinin bir diğer önemli avantajı ise, dizi elemanları arasında gezinme işleminin de oldukça hızlı bir şekilde gerçekleştirilebilmesidir. Gezinilecek eleman grubunun ilk elemanının adresi hesaplandıktan sonra, dizinin bileşen boyutu bu elemanın adresine eklenerek, basit ve hızlı bir şekilde dizi elemanları taranabilir.

Ayrık dizi temsili yani arraylet kullanıldığı durumda, gelen dizi yer ayırma taleplerine, Minuteman, dizinin omurga boyutunu hesaplamayla başlar. Bu hesaplar aşağıda Denklem 5.4 ve 5.5’de gösterilen formüller kullanılarak gerçekleştirilir.

$$veriBoyutu = diziBoyutu * bileşenBoyutu \quad \textbf{(5.4)}$$

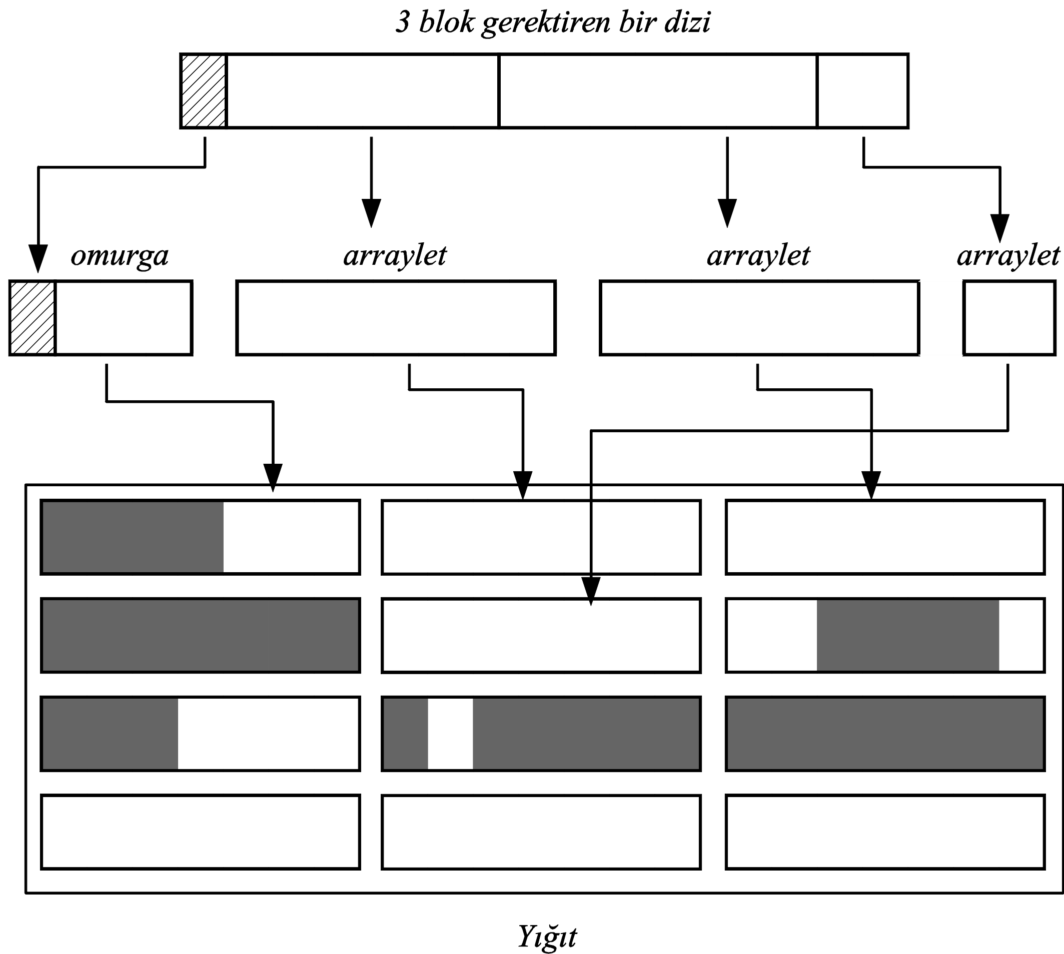
$$omurgaBoyutu = \frac{veriBoyutu}{arrayletBoyutu} \quad \textbf{(5.5)}$$

Öncelikle dizinin bileşen boyutu, dizi boyutu ile yani dizinin eleman sayısı ile çarpılarak dizi elemanlarının ne kadar yer kaplayacağı hesaplanır. Bu hesaplamanın sonucuna, veri boyutu adı verilir. Daha sonra veri boyutu, arraylet boyutuna bölünür ve omurga boyutu hesaplanmış olur. Arraylet boyutu, Minuteman yapılandırma parametrelerinden birisidir ve her bir arraylet'in boyutunun ne kadar olacağını belirtir. Minuteman içerisinde arrayletSize adındaki bir alan ile temsil edilir. Varsayılan değeri blok boyutudur (blokSize).

İkinci adımda, dizi başlığı boyutu ve omurga boyutunun toplamı kadar bellek miktarı küçük nesneler için bellek ayırma stratejisi kullanılarak edinilir. Daha sonra, dizi boyutu ve arraylet boyutu dikkate alınarak, dizi içerisinde bulunacak toplam arraylet sayısı hesaplanır. Bu hesaplamanın ardından, her bir arraylet için, küçük nesnelere bellek ayırma stratejisi kullanılarak bellek ayrılır. Ayrılan her bir belleğin adresi ise

daha sonra omurga içerisinde ilgili arraylet'i temsil edecek olan alana yazılır. Böylelikle ayrık diziler için bellek ayırma süreci tamamlanmış olur.

Birden fazla blok gerektiren parçalı dizilere bellek ayırma süreci, Şekil 5.6.'da, bir örnek üzerinden grafiksel olarak gösterilmiştir. Şekildeki örnekte, Minuteman'e, 3 adet blok gerektiren bir dizi talebi geldiği varsayılmaktadır. Minuteman, bu talebi karşılayabilmek adına, öncelikle diziyi blok boyutu büyüklüğünde parçalara ayıracak, omurga miktarını hesaplayacak ve başlık ve omurganın da dâhil olduğu her bir grup için küçük nesneler için bellek ayırma stratejini kullanarak ayrı ayrı bellek ayıracaktır. Dolayısıyla, bitişik dizi talebinin aksine ardışık blok bulma zorunluluğu ortadan kaldırılmış olacak ve belleğin sonuna kadar arama işlemi, gerçekleştirmek gerekmeyecektir.



Şekil 5.6. Parçalı dizi bellek ayırma süreci

Bitişik diziler ile kıyaslandığında, parçalı diziler için bellek ayırma algoritması daha karmaşık bir yapıdadır. Öte yandan, bellek ayırma süreci de bir o kadar hızlı gerçekleşebilmektedir. Zira bellekte, ardışık blok bulma zorunluluğu yoktur. Belleğin farklı bölgelerinde bulunan bloklar dizinin farklı elemanlarının kullanımına sunulabilir.

Parçalı dizi temsilinin en büyük dezavantajı ise dizi elemanlarına erişimin ve dizi elemanları arasında gezinmenin, birleşik dizilere oranla daha karmaşık ve yavaş olmasıdır. Bir dizi elemanına erişebilmek için öncelikle ilgili elemanın hangi arraylet içerisinde bulunduğu hesaplanır. Bu hesaplama işleminin ardından omurga içerisinde ilgili arraylet'i temsil eden adres tespit edilip, okunur. Bir sonraki adımda ise dizi elemanının arraylet içerisindeki indisi hesaplanır. Hesaplanan indis, daha sonra bileşen boyutu ile çarpılır ve bu işlemin sonucunda elde edilen değer arraylet adresi ile toplanarak dizi elemanının adresi tespit edilir.

Dizi elemanları arasında gezinme ise, aynı şekilde birleşik dizilerde olduğu gibi basit işaretçi aritmetiği ile gerçekleştirilemez. Her bir gezinme adımında, yeni elemanın aynı arraylet içerisinde bulunup bulunmadığı kontrol edilmeli ve eğer başka bir arraylet içerisinde ise yeni arraylet'in adresi omurgaya gidilerek okunmalıdır. Bu süreç de oldukça karmaşık, daha fazla hesaplama ve bellekten okuma işlemi gerektiren bir süreçtir.

Görüldüğü gibi, her biz dizi temsilinin avantajları ve dezavantajları mevcuttur. Geliştirilen gerçek zamanlı uygulamanın dizi kullanımına ve ihtiyaçlarına göre hangi dizi temsilinin kullanılacağı seçilebilir.

5.2.3. Büyük nesneler için bellek ayırma

Minuteman, kendisine gelen bellek ayırma talebi, maxBlocAlloc değerinden büyükse, büyük nesneler için bellek ayırma stratejisini devreye sokar. Bu stratejide lineer arama işlemi gerçekleştirilir. Bu bölümde bu stratejinin nasıl çalıştığı ayrıntılı bir şekilde açıklanacaktır.

Büyük nesneler için bellek ayırma stratejisinde, öncelikle ayrılacak bellek miktarının kaç blok içerisine sığabileceği hesaplanır. Bu hesaplama, talep edilen bellek boyutunun, maxBlockAlloc boyutuna bölünmesi ile gerçekleştirilir. Daha sonra gereken blok sayısı kadar 0 biti, usedBits bit haritasında ardışık bir şekilde bulunacak şekilde aranır. Bu işlem örüntü arama işlemi olarak da düşünülebilir. Bir dizi 0 ve 1 bitleri içerisinde belli sayıda yan yana 0 biti aranmaktadır. Bit haritası içerisinde gerçekleştirilen bu arama işlemi, doğrusal bir aramadır. Algoritma karmaşıklığı, diğer bir deyişle en kötü çalışma zamanı $O(n)$'dir. Devam eden paragraflarda, bit haritası içinde gerçekleştirilen, örüntü aramasından bahsedilecektir.

Arama işleminin ilk etabı, bit haritası içerisinde bir adet 0 biti bulmayı amaçlar. Arama, usedBits dizinin ilk elemanından başlatılır. Dizinin ilk elemanının değeri alınır ve val adı verilen bir alanda saklanır. Bu değer 32 bitlik tamsayı veri tipinde bir değerdir. val üzerinde bit düzeyinde işlem yapılır. val, 1 ile ve işlemine tabi tutulur. Sonuç 0 ise, val'ın en önemsiz biti 1 demektir. Bu durumda val, bir bit sağa kaydırılır ve tekrar 1 ile ve işlemine tabii tutulur ve sonuç kontrol edilir. Eğer sonuç 0 ise aynı şekilde kaydırma ve ve işlemlerine devam edilir. Bu işlem, sonuç 1 olana kadar yani val değeri içerisinde bir 0 biti bulunana kadar ya da val değerinin bütün bitleri gezilene kadar devam eder.

Eğer val değerinin bütün bitleri gezilmiş ve 0 biti bulunamamış ise usedBits dizisinin bir sonraki elemanı alınır ve değeri val alanına yazılır. Aynı şekilde bit kaydırma ve ve işlemleri, 0 biti bulunana ya da val değerinin bütün bitleri gezilene kadar devam eder. Eğer 0 biti bulunamamışsa bir sonraki elemana geçilir ve arama işlemi aynı şekilde devam eder.

Böylelikle 0 biti bulunana kadar bütün dizi elemanları ve elemanların bitleri gezilir. Dizinin bütün elemanları gezilmiş ve 0 biti bulunamamışsa "bellek yetersizliği" hatası fırlatılır. Eğer herhangi bir noktada 0 biti bulunursa bulunduğu noktanın indis numarası kayda alınır ve cur adı verilen bir alanda saklanır. Arama bir sonraki aşamaya geçer.
Arama işleminin ikinci etabı, bit haritası içerisinde, bulunan 0 bitinden sonraki bir noktada bir adet 1 biti bulmayı amaçlar. Öncelikle dizinin, 0 biti bulunmuş olan

elemanı alınır ve değeri val alanına yazılır. Bulunmuş olan 0 bitinin indeks numarasına göre eleman içerisindeki bit indeksi hesaplanır. val değeri, bit indeksi sayısının bir fazlası kadar sağa kaydırılır. Böylelikle 0 bitinin bulunduğu noktadan bir sonraki noktaya gelinmiş olur.

Bu noktada val değeri, 1 değeri ile ve işlemine tabii tutulur. Eğer sonuç 1 ise, val değerinin en önemsiz biti 0 demektir. Bu durumda val, bir bit sağa kaydırılır ve tekrar 1 ile ve işlemine tabii tutulur ve sonuç kontrol edilir. Eğer sonuç 1 ise aynı şekilde kaydırma ve ve işlemlerine devam edilir. Bu işlem, sonuç 0 olana kadar yani val değeri içerisinde bir 1 biti bulunana kadar ya da val değerinin bütün bitleri gezilene kadar devam eder.

Eğer val değerinin bütün bitleri gezilmiş ve 1 biti bulunamamışsa usedBits dizisinin bir sonraki elemanı alınır ve değeri val alanına yazılır. Aynı şekilde bit kaydırma ve "ve" işlemleri, 1 biti bulunana ya da val değerinin bütün bitleri gezilene kadar devam eder. Eğer 1 biti bulunamamışsa bir sonraki elemana geçilir ve arama işlemi aynı şekilde devam eder.

Böylelikle 1 biti bulunana kadar bütün dizi elemanları ve elemanların bitleri gezilir. Eğer herhangi bir noktada 1 biti bulunursa bulunduğu noktanın indeks numarası kayda alınır ve end adı verilen bir alanda saklanır. Arama bir sonraki aşamaya geçer. Dizinin bütün elemanları gezilmiş ve 1 biti bulunamamışsa bit haritasının en son bitinin indeks numarası kayda alınır ve arama bir sonraki aşamaya geçer.

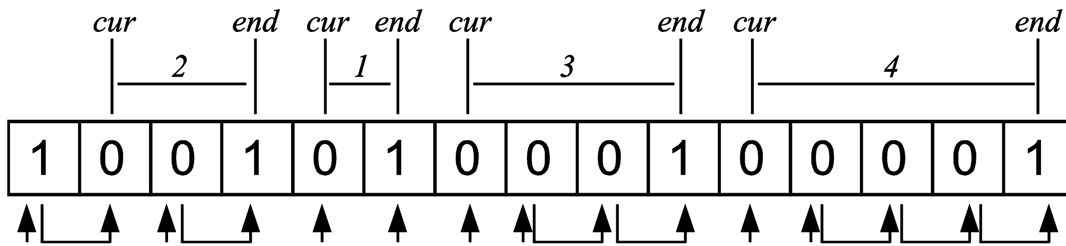
0 ve 1 bitinin indeks numaraları kayda alındıktan sonra, 1 bitinin indeks numarası (end), 0 bitinin indeks numarasından (cur) çıkartılır. Bu işlem sonucunda, ardışık bir şekilde kaç tane 0 biti bulunduğu hesaplanmış olur. Hesaplanan 0 biti sayısı, ihtiyaç duyulan blok sayısı ile karşılaştırılır.

Eğer hesaplanan 0 bit sayısı, ihtiyaç duyulan blok sayısından küçük bir değer ise aranılan miktarda alan bulunamamış demektir. Bu durumda, arama işlemi en başa döner ve birinci etaptan yeninden başlar. Ancak bu sefer, dizinin ilk elemanının ilk bitinden değil, kayda alınmış olan 1 bitinin indeks numarasının bir fazlasından (end+1)

başlar. Diğer bir deyişle, bit haritasının bulunmuş 1 bitinden sonraki konumlarında yeniden 0 biti aranmaya başlar.

Eğer hesaplanan 0 bit sayısı, ihtiyaç duyulan blok sayısından büyük bir değer ise ya da eşitse aranılan miktarda alan bulunmuş demektir. Bit haritası içerisinde bulunan 0 bitinden itibaren gereken bit sayısı kadar bitin değeri 1 yapılır. Böylelikle bu bitlerin artık boş blokları değil, kullanımdaki blokları temsil ettiği anlaşılacaktır. Daha sonra bulunan 0 bitinin indeks numarasına göre, bu bitin temsil ettiği bloğun bellek adresi hesaplanarak geriye döndürülür. Bellek ayırma işlemi başarılı bir şekilde gerçekleştirilmiştir.

Büyük nesneler için bellek ayırma süreci Şekil 5.7.'de bir grafikle temsil edilmiştir. Sürecin daha iyi anlaşılabilmesi için, bit haritasının yalnızca ilk 15 biti gösterilmiş ve bit tarama işleminin soldan sağa doğru gerçekleştiği varsayılmıştır. Bu örnekte 4 blok gereksinimi duyan bir nesne için bellek ayırma işlemi gösterilmektedir. Dolayısıyla, bit haritası içerisinde ardışık bir şekilde bulunan 4 adet 0 biti aranmaktadır. Aranan ardışık 4 bit, bit haritası içerisinde 13 numaralı indekste bulunmuştur. Arama işleminde ise ilk 15 bitin tamamı ziyaret edilmiştir.



Şekil 5.7. Büyük nesneler için bellek ayırma süreci

BÖLÜM 6. ANAHTARLAMALI YAKLAŞIM

Bu çalışmada, büyük nesneler için yeni bir bellek ayırma yaklaşımı geliştirilmiş ve Minuteman çatısı üzerine uygulanmıştır. Bu bölümde, öncelikle Boyer ve Moore'un karakter dizisi arama algoritmasının, bitişik bellek ayırma için özelleştirdiğimiz bir varyasyonu olan atlamalı arama algoritması sunulacaktır. Devamında ise daha verimli bir bellek ayırma işlemi için lineer arama algoritması ve atlamalı arama algoritmasını birleştiren anahtarlamalı bir yaklaşım sunulacaktır.

6.1. Atlamalı Arama Algoritması

Boyer ve Moore'un karakter dizisi arama algoritması temeline dayanan, onun özelleştirilmiş bir hali olan atlamalı arama algoritması, lineer arama algoritmasının ortalama çalışma zamanını iyileştirmeyi amaçlar. Bu iyileştirmeyi, arama işlemi sırasında, var olan algoritmaya kıyasla daha az biti ziyaret ederek gerçekleştirmeye çalışır. Bu bölümde bu algoritmanın detaylarından bahsedilecektir.

Atlamalı arama algoritmasının sözde kodu Şekil 6.1.'de, bir örnek üzerinden grafiksel temsili ise Şekil 6.2.'de verilmiştir.

Atlamalı arama algoritmasında, bit haritası içerisindeki tüm bitler sırasıyla gezilmez. Bunun yerine talep edilen blok sayısı kadar ileriye doğru atlama ve daha sonra geriye doğru bit tarama işlemi gerçekleştirilir. Talep edilen blok sayısı count adı verilen bir alanda tutulur.

Aramanın ilk etabı, atlama işleminin başlayacağı nokta ve atlanılacak noktanın hesaplanması ile başlar ve atlama işleminin gerçekleştirilmesi ile son bulur. Atlama işleminin başlayacağı nokta startIndex ve atlanılacak nokta endIndex adı verilen alanlarda tutulur. Bu değerler atlama işleminin başladığı noktadaki bitin ve atlama

işleminin gerçekleşeceği bitin, bit haritası içerisindeki indeks numaralarıdır. Doğal olarak endIndex, startIndex ve count alanlarındaki değerlerin toplamı alınarak hesaplanır. Arama işleminin başında startIndex 0 ve endIndex, startIndex değerinin count değerinin 1 eksiği kadar artırılmış halidir.

```
//Toplam block sayısını al
elementCount = usedBits element count;

//Nesne için gereken blok sayısını al
blockCount = required block count;

//Atlama operasyonunun başlangıç noktasını ayarla
startIndex = 0;

//Atlama operasyonunun bitiş noktasını ayarla
endIndex = startIndex + blockCount - 1;

while (startIndex != (endIndex + 1)) {

    //Yeterli boş blok bulunamdı: Terket
    if (endIndex > elementCount)
        return error;


    if (usedBits[endIndex] == 1) { //Set biti bulundu: Atla
        startIndex = endIndex + 1;
        endIndex = startIndex + blockCount - 1;
    } else { //Clear biti bulundu: Geriye doğru tara
        endIndex = endIndex - 1;
    }
}

//Boş blok bulundu
return startIndex;
```

Şekil 6.1. Atlamalı arama algoritmasının sözde kodu

Başlama ve bitiş indeksleri hesaplandıktan sonra, bu indekslerde temsil edilen bitlerin usedBits dizinin hangi elemanlarında bulunduğu tespit edilir. Dizinin her bir elemanının 32 bit boyutunda bir tamsayı değer olduğu düşünülürse, indeks değerlerinin 32 değerine bölünmesi istenilen sonucu verecektir. startIndex ve endIndex alanlarındaki değerler 32'ye bölünür ve sonuçları startWord ve endWord adı verilen alanlarda tutulur.

Bu hesaplamaların hemen ardından, atlamanın gerçekleştirileceği usedBits dizisinin elemanı endWord alanındaki değer kullanılarak edinilir ve değeri value adı verilen bir alanda saklanır. Aynı zamanda endIndex değeri üzerinde mod işlemi uygulanarak, atlanılacak olan bitin, dizinin elemanı içerisindeki bit indeksi hesaplanır ve bu değer de bitIndex adı verilen bir alanda saklanır.

Arama işlemi, atlama işleminin gerçekleştiği noktadan, atlama işleminin başladığı noktaya kadar geriye doğru bir süreç olduğundan, dizi elemanları içerisinde bit düzeyinde yüksek öncelikli bitlerden düşük öncelikli bitlere doğru tarama gerçekleştirmek gerekmektedir. Dolayısıyla her bir adımda, dizi elemanının yüksek öncelikli biti karşılaştırma işlemine tabii tutulmak durumundadır. Bu karşılaştırma işlemin gerçekleştirebilmek için, yalnızca yüksek öncelikli biti 1 olan 32 bitlik bir tamsayı değeri (2^{31}) oluşturulur ve bu değer pattern adı verilen bir alanda saklanır. Atlanılan noktadaki dizi elemanının değerini saklayan ve geriye doğru arama işleminin başlayacağı value değeri, atlanılan ve ilk olarak kaşılaştırılacak olan bit yüksek öncelikli bit olana kadar (31 – bitIndex) sola doğru kaydırılır.

Aramanın ikinci etabında, atlanılmış olan noktadan, atlamanın başladığı noktaya yani geriye doğru tarama işlemi gerçekleştirilir. Amaç, bu aralıkta bir 1 bitinin bulunup bulunmadığını tespit etmektir. Bunu gerçekleştirmek için value değeri, pattern değeri ile ve işlemine tabii tutulur.

Eğer sonuç 0 değerinden farklı ise ilgili alandaki bit 0 değerindedir. Bu durumda value değeri bir kere sola doğru kaydırılır ve pattern ile tekrar karşılaştırılır. Arama işlemi, 0 değeri elde edene kadar ya da atlamanın başladığı noktaya ulaşılana kadar devam eder. Arama işlemi esnasında, her iki durum da gerçekleşmemiş ve fakat value değerinin en düşük öncelikli biti de taranmış ise, dizinin bir önceki elemanı alınır ve bu elemanın değeri value alanında saklanır. Yine aynı şekilde en yüksek öncelikli bitinden en düşük öncelikli bitine doğru, sola doğru kaydırma ve pattern ile karşılaştırma işlemi uygulanır.

Eğer atlamanın başladığı noktaya kadar, geriye doğru tarama işlemi bitmiş ve karşılaştırma işlemlerinde 0 değeri elde edilmemişse, taranmış olan noktadaki bütün

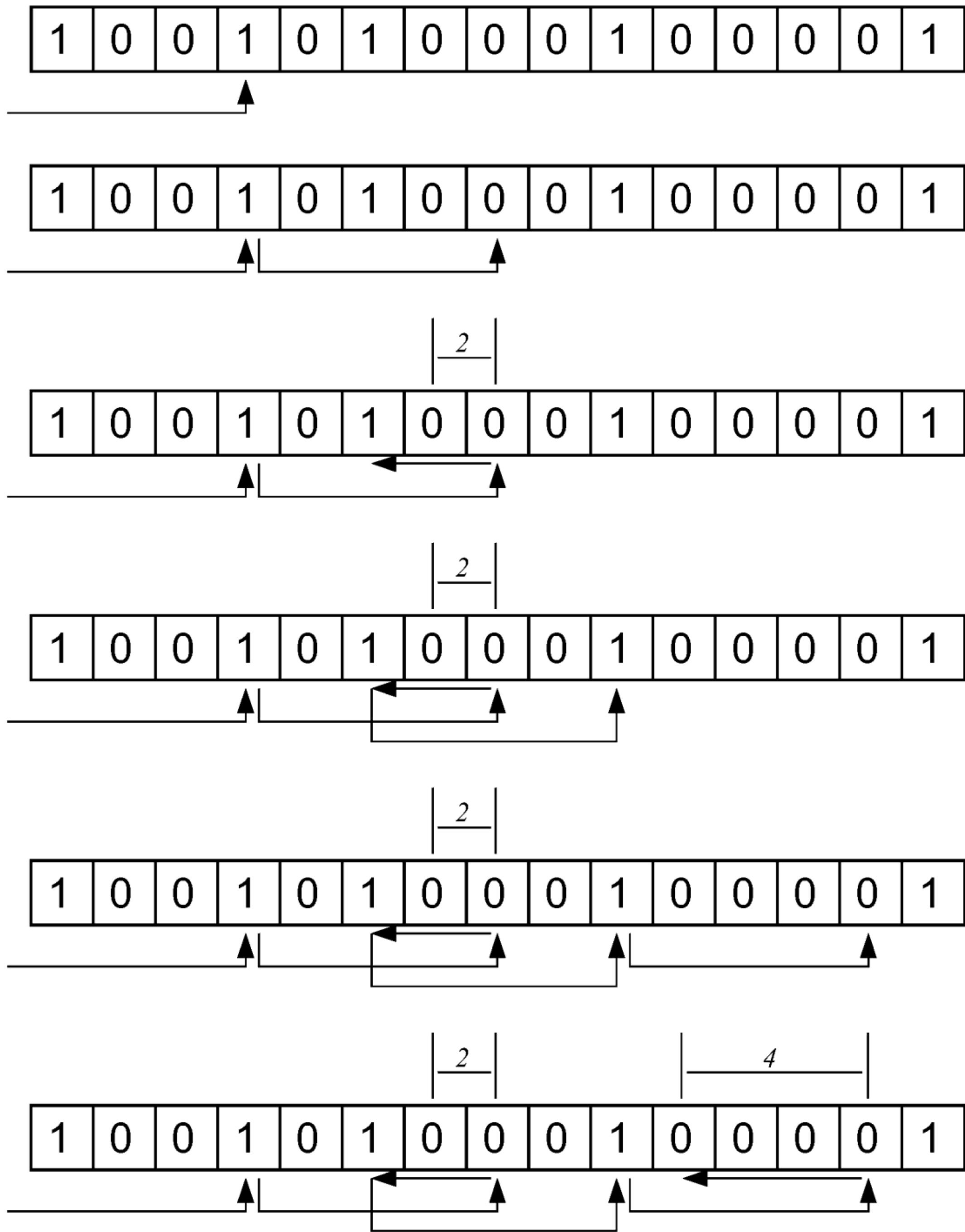
bitlerin 1 değerinde olduğu sonucuna varılır. Bu sonuç aranan bellek bölgesinin bulunduğu anlamına gelir.

Ancak, atlamanın başladığı noktaya gelinmeden önce, her hangi bir anda value değeri ve pattern değerinin ve işlemi sonucunda 0 değeri elde edilirse, o noktada var olan bit 1 değerinde demektir. Bu durum, arama yapılan aralıkta ardışık bir şekilde istenilen sayıda 0 biti olmadığı anlamına gelir. Arama sonlandırılır ve bu noktadan itibaren tekrar arama yapmak üzere ileriye doğru atlama işlemi gerçekleştirilir. Diğer bir deyişle ilk etaba geri dönülür. Yalnız bu sefer startIndex değeri 0 değil, tespit edilen 1 bitinin indeks değerinin 1 fazlasıdır. Bu şekilde atlama ve geriye doğru tarama işlemleri, gerekli sayıda ardışık 0 biti bulunana kadar devam eder.

Büyük nesneler için önerilen yeni bellek ayırma süreci Şekil 6.1.'de bir grafikle temsil edilmiştir. Sürecin daha iyi anlaşılabilmesi için, bit haritasının yalnızca ilk 15 biti gösterilmiş ve bit tarama işleminin soldan sağa doğru gerçekleştiği varsayılmıştır. Bu örnekte 4 blok gereksinimi duyan bir nesne için bellek ayırma işlemi gösterilmektedir. Dolayısıyla, bit haritası içerisinde ardışık bir şekilde bulunan 4 adet 0 biti aranmaktadır.

Arama işlemi bit haritasının başından başlar. count 4, startIndex 0 ve endIndex 3 değerindedir. Bit haritasının endIndex değerli bitine yani 3. bitine atlanır. Buradan geriye doğru tarama işlemi gerçekleştirilir. Ancak bu durumda atlanılmış olan bit zaten 1 değerinde olduğundan ötürü, bit haritası başlangıcı ve atlanılan nokta arasında 4 adet ardışıl 0 biti bulunmadığı anlaşılır.

Bu noktada 1 bitinin bulunduğu indeks numarası bir artırılarak startIndex yeniden hesaplanır ve 4 değerini alır. endIndex ise 7 olarak hesaplanır. Bit haritasının 7. bitine atlanılır. Ve aynı şekilde geriye doğru tarama işlemi gerçekleştirilir. Ancak bu tarama işleminin sondan bir önceki adımında yani bit haritasının 5. bitinde 1 biti ile karşılaşılır. Bu noktada arama durdurulur.



Şekil 6.2. Atlamalı arama algoritmasının grafiksel temsili

Başlangıç ve bitiş indeks numaraları yeniden hesaplanır. startIndex 6, endIndex 9 değeri alır. Bit haritasının 9. bitine atlama işlemi gerçekleştirilir. Atlanılan noktada, geriye doğru tarama işlemi başlamadan önce zaten atlanılan noktadaki bitin 1 olduğu tespit edilir ve arama işlemi hemen sonlandırılır. Tekrar atlama işlemi gerçekleştirmek gerekecektir.

Hesaplamalar yapılır ve startIndex 10, endIndex ise 13 olarak tespit edilir. Bit haritasının 13. bitine atlanılır. Bu bit 0 değerindedir ve geriye doğru tarama başlar. Tarama işlemi startIndex numarasına kadar devam eder ve aralıkta herhangi bir 1 bitine rastlanmaz. startIndex değerine ulaşıldığında aranılan ardışık 4 bitin bulunduğu anlaşılır.

Bu örnek, bir önceki bölümde anlatılmış olan arama algoritması ile karşılaştırıldığında en kötü çalışma zamanının aynı olduğu görülür. En kötü durumda, bit haritasının bütün bitleri gezilmek durumunda kalınacaktır. Diğer bir deyişle algoritma karmaşıklığı $O(n)$'dir.

Atlamalı arama algoritmasının akış diyagramı, Şekil 6.4., Şekil 6.5., Şekil 6.6. ve Şekil 6.7.'de gösterilmiştir. Akış diyagramındaki isimlendirmeler, Ovm uygulamasında kullanılan isimlerdir.

6.2. Anahtarlamalı Yaklaşım

Minuteman içerisinde hali hazırda yer alan lineer arama algoritmasının yerini alabilecek bir algoritma sunmak bu çalışmanın ilk amacıydı. Ne ki erken dönem kıyaslama sonuçlarımızda atlamalı arama algoritmasının ortalama bellek ayırma zamanı performansının lineer arama algoritmasının performansından iyi olduğunu gözlemlesek de atlamalı arama algoritmasının en kötü çalışma zamanı performansı için benzer sonuçları gözlemleyemedik. Aksine lineer arama algoritmasının en kötü çalışma zamanı performansı önerilen atlamalı arama algoritmasının performansından daha kötü idi.

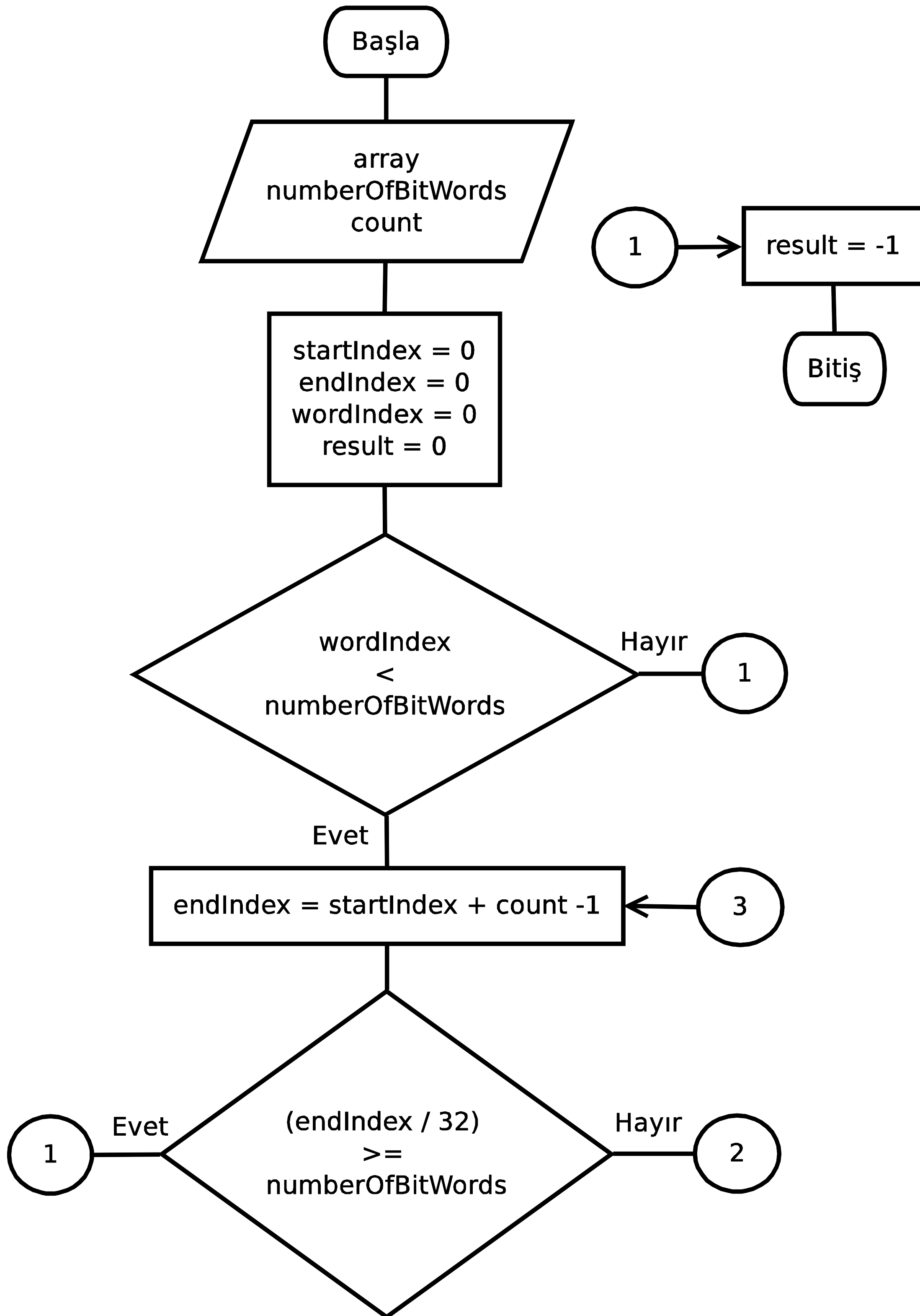
Sonuçlar üzerinde yaptığımız dikkatli inceleme sonucunda, önerdiğimiz algoritmanın en kötü çalışma zamanı performansını etkileyen birçok bellek ayırma talebinin 2 ardışık blok olduğu görüldü. Diğer bir deyişle iki ardışık blok aranırken lineer arama algoritması, önerilen algoritmadan daha iyi sonuç veriyor idi. Dolayısıyla hem ortalama bellek ayırma zamanı performansını ve hem de en kötü çalışma zamanı

performansını iyileştirmek adına her iki algoritmayı da kullanan anahtarlamalı bir yaklaşım geliştirdik.

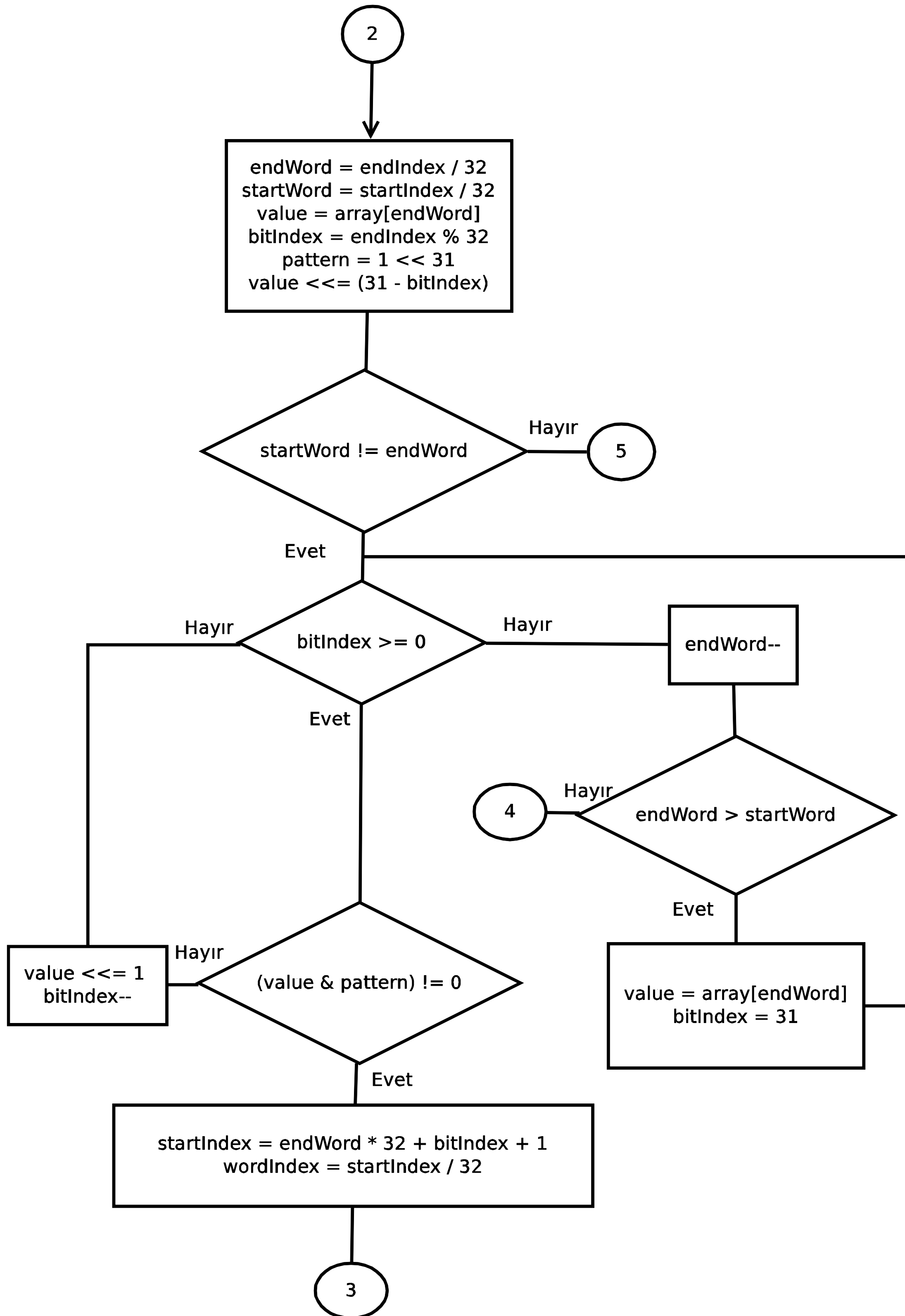
Geliştirdiğimiz anahtarlamalı yaklaşım öncelikle nesne için gereken blok sayısını hesaplar. Eğer blok sayısı 3 bloktan daha küçük ise lineer arama algoritmasına geçiş yapılır. Eğer değilse atlamalı arama algoritması kullanılır. Önerilen anahtarlamalı yaklaşımın sözde kodu Şekil 6.3.'de görülmektedir.

```
blockCount = gereken blok sayısı;
if (blockCount >= 3)
    atlamalı arama gerçekleştir;
else
    lineer arama gerçekleştir;
```

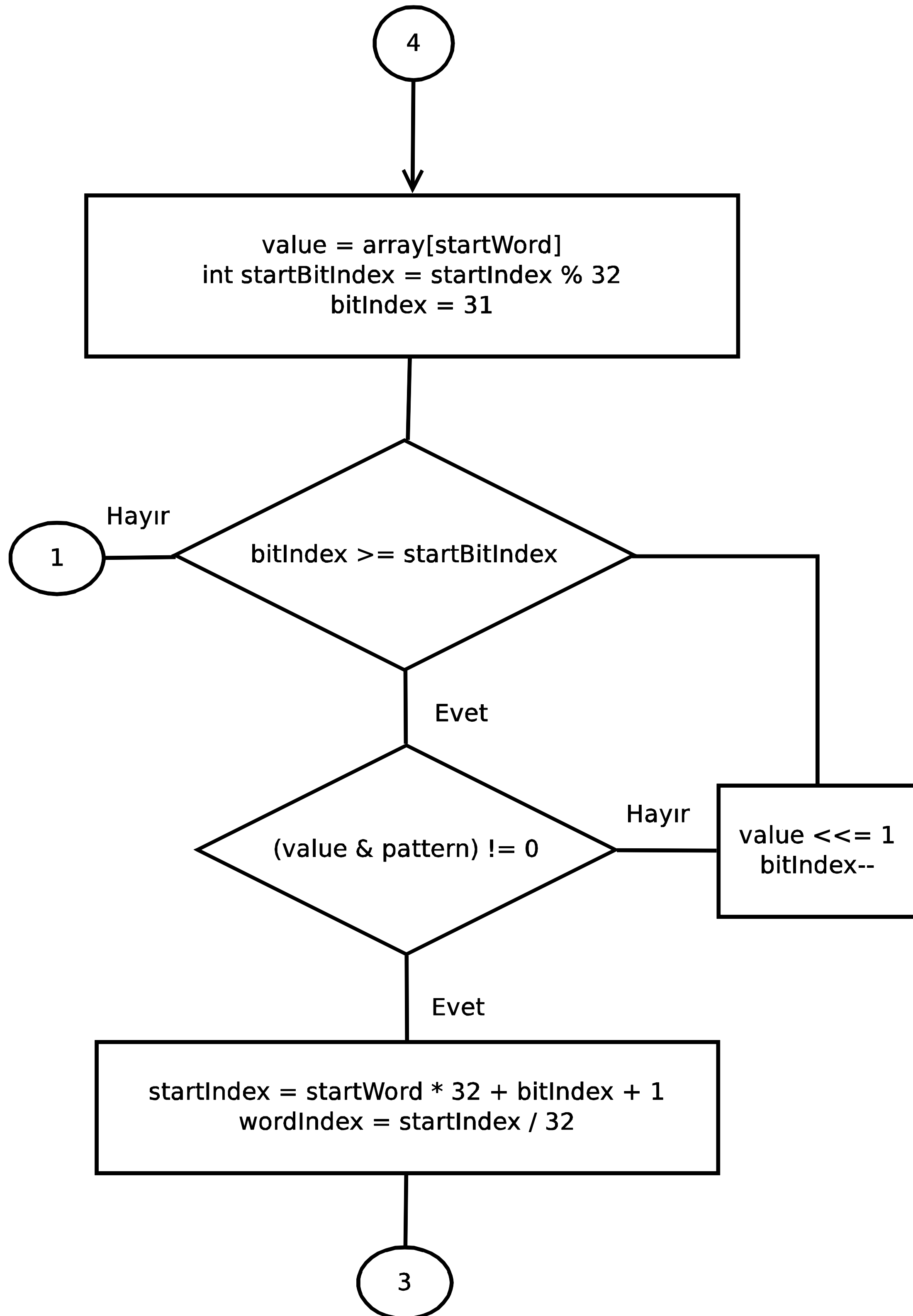
Şekil 6.3. Anahtarlamalı yaklaşımın sözde kodu



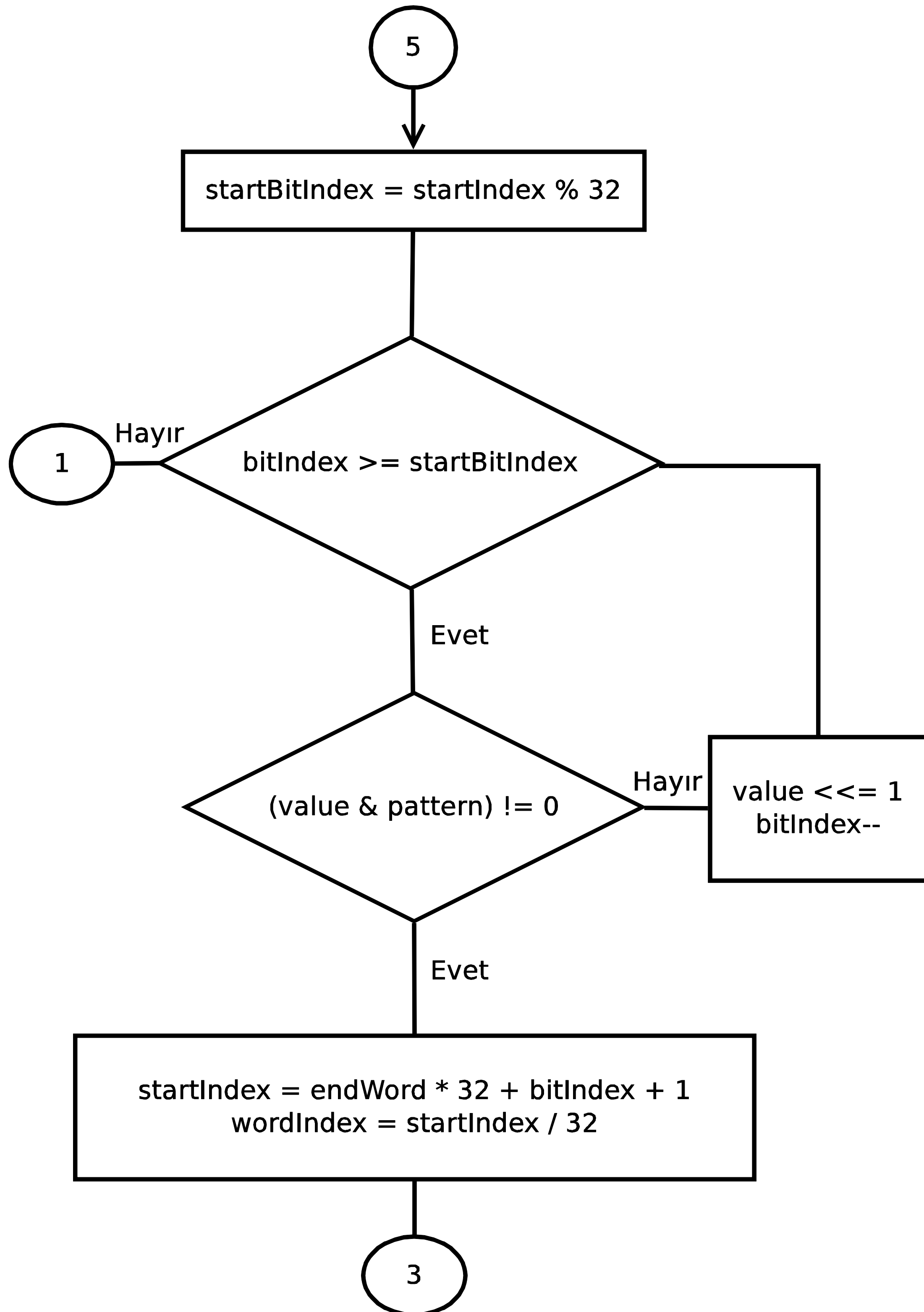
Şekil 6.4. Atlamalı arama algoritması akış diyagramı 1



Şekil 6.5. Atlamalı arama algoritması akış diyagramı 2



Şekil 6.6. Atlamalı arama algoritması akış diyagramı 3



Şekil 6.7. Atlamalı arama algoritması akış diyagramı 4

BÖLÜM 7. DEĞERLENDİRME

Anahtarlamalı yaklaşımın etkinliğini gözlemleyebilmek amacıyla, önerilen yaklaşım Minuteman çatısı altında uygulandı. Değerlendirme yapabilmek için bir kıyaslama uygulaması geliştirildi ve Minuteman çatısında var olan tüm yaklaşımlar (lineer arama, arraylet temsili ve anahtarlamalı yaklaşım), bu uygulama kullanılarak ayrı ayrı test edildi. Daha sonra anahtarlamalı yaklaşım, var olan lineer arama algoritması ve arraylet yaklaşımının bellek ayırma performansları karşılaştırıldı. Aynı zamanda her bir yaklaşımın dizi erişim zamanları test edildi.

Testler, 3.2.0-24 generic 32 bit çekirdeğe sahip bir Ubuntu Linux 12.04 dağıtımı koşan ve Intel Core i5 2.53 GHz çift çekirdekli işlemciye ve 4 GB belleğe sahip bir bilgisayarda gerçekleştirildi. Ovm sanal makinesinin tek işlemcili doğasından ötürü, testler bilgisayarın tek çekirdeği üzerinde koşturuldu. CPU benzerlik ayarı kullanılarak uygulamanın süreci tek bir çekirdek üzerine atıldı ve orada koşturuldu.

7.1. Kıyaslama

Bu testler için geliştirilmiş olan kıyaslama uygulaması Java 1.4 uyumlu bir Java uygulamasıdır. Temel amacı dizi oluşturmak ve diziler oluşturulurken geçen bellek ayırma zamanlarını hesaplamaktır. Bu amaçla, boyutları önceden belirlenmiş ve uygulama içerisine kodlanmış 1000 adet dizi için bellek ayırma talebinde bulunur ve her bir dizi için bellek ayırma zamanlarını raporlar. Test edilen tüm yaklaşımlar için benzer bir bellek kullanım deseni oluşturabilmek için boyutu önceden belirlenmiş dizilerin kullanımı tercih edilmiştir. Dizilerin boyutları, uygulamaya eklenmeden önce, rasgele sayı üreteci kullanılarak üretilmiştir. Bunun sebebi ise bellek ayırıcı için farklı yükler oluşturma ve her bir çöp toplama çevrimi sonrasında farklı parçalanma desenleri oluşturma isteğidir. Dolayısıyla daha iyi bir ölçümleme amaçlanmıştır.

Uygulamanın en az dizi boyutu 600 eleman (2,34 KB) ve en çok dizi boyutu 99229 elemandır (387,61 KB). Dizi boyutlarının değişim aralığı orta değeri 194,97 KB'dir. Uygulamaya ayrılmış olan yığıt alanı 8 MB'dır.

Bu çalışmada kullanılan, bir çöp toplayıcı çevrimi esnasında ayrılabilecek bellek miktarının üst sınırı hesaplaması, Robertz ve Henriksson (Robertz ve Henriksson, 2003) tarafından ortaya atılmıştır. İlgili hesaplamanın yapıldığı formül aşağıda Denklem 7.1'de gösterilmiştir.

$$a_{max} = \frac{H - L_{max}}{2} \quad \textbf{(7.1)}$$

Yukarıdaki denklemde a_{max}, bir çöp toplayıcı çevrimi sırasında bellek dolma problemi ile karşılaşmadan ayrılabilecek en çok bellek miktarını simgeler. H, yığıt boyutu ve L_{max} ise uygulamanın yaşam süresi boyunca yaşayan en çok canlı bellek miktarıdır. Planlama analizi hakkında daha geniş bilgi Robertz ve Henriksson'un çalışmasının yanı sıra (Schoeberl, 2010; Kalibera ve diğerleri, 2009, 2011) kaynaklarından da edinilebilir.

Kıyaslama uygulaması için L_{max} değişkeni yok sayılabilir çünkü uygulamanın yaşam süresi boyunca canlı kalan bellek miktarı oldukça küçüktür. Uygulamanın yaşamı boyunca hayatta kalan tek bir nesne vardır ve bu nesne dizilere bellek ayırma talebinde bulunan ana nesnedir. Buradan hareketle şunu söyleyebiliriz: Bir çöp toplayıcı çevrimi sırasında uygulamanın ayırabileceği en çok bellek miktarı 4 MB (a_{max} = 4 MB) boyutunda olabilir. Diziler, çöp toplama çevrimi sırasında bellek ayıracağımız nesnelerdir. Örnek kümemiz, boyutları rasgele olarak dağıtılmış dizilerden oluştuğu için, dizilerin temel boyutu olarak değişim aralığı orta değerini kabul ettik. Değişim aralığı orta değeri ve hesaplanan a_{max} değerine göre bir çöp toplayıcı çevrimi sırasında en fazla 20 adet dizi için bellek ayrılabilir.

Hesaplamaların sonucu olarak, kıyaslama uygulaması her 20 iterasyonda bir uyuyacak şekilde yapılandırıldı. Böylelikle çöp toplayıcıya çevrimi tamamlayabilecek zaman tanınmış oldu. Bunu yaparken iki temel amacımız mevcuttu: Bellek dolma

hatalarından kaçınmak ve parçalı bir bellek oluşturmak. Bu değeri seçerek testlerimiz için hedefimize ulaşmış olduk. Öte yandan, farklı örnek kümeler için yahut görev ve/veya güvenlik kritik gerçek hayat uygulamaları için, değişim aralığı orta değeri yerine en büyük dizi boyutu tercih edilebilir.

Bellek ayırma zamanlarının yanı sıra, dizi erişim zamanları da kritik değerlerdir. Bu değerler, bitişik dizi ve parçalı dizi temsillerinin karşılaştırılmasında kullanılabilir. Bu amaç doğrultusunda, kıyaslama uygulaması, her bir dizi için erişim zamanını da hesaplar. Diziye bellek ayırdıktan ve bellek ayırma zamanını hesapladıktan sonra, uygulama, dizinin tüm elemanlarını ziyaret eder ve her bir elemana yazma işlemi gerçekleştirir. Bu işlem süresince harcanan zamanı kaydeder ve daha sonra bu zamanı dizinin eleman sayısına bölerek, dizinin bir elemanına ortalama erişim zamanını hesaplar.

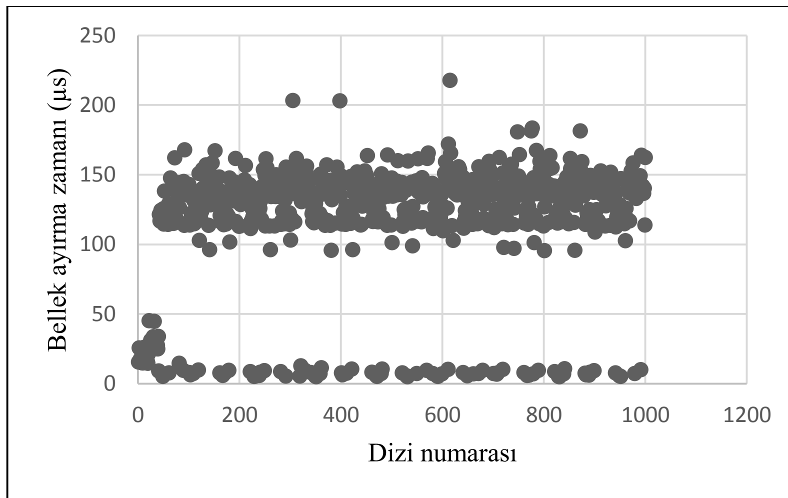
Kıyaslama uygulaması sonlandığında elimizde 1000 adet dizi için iki tür veri mevcut olur: bellek ayırma zamanları, dizi erişim zamanları.

7.2. Sonuçlar ve Tartışma

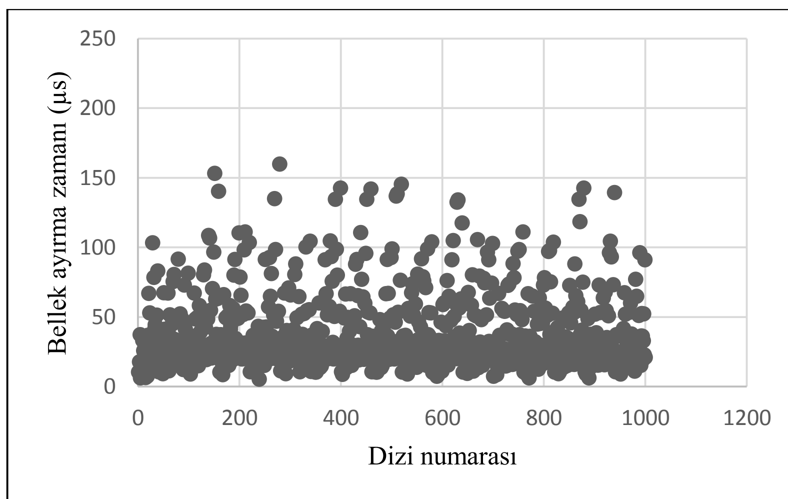
Bu çalışmada, testler her üç yaklaşım için gerçekleştirildi. İlk yaklaşım, bitişik dizi temsili üzerinde var olan lineer arama algoritması; ikinci yaklaşım, bitişik dizi temsili üzerinde önerilen anahtarlamalı yaklaşım ve üçüncü yaklaşım ise arraylet temsilidir. Her üç yaklaşım için kıyaslama uygulaması üç kere çalıştırılmış ve elde edilen verilerin aritmetik ortalaması alınmıştır. Bu işlem gerçekleştirildikten sonra 1000 adet dizinin her üç yaklaşım için bellek ayırma zamanları ve dizi erişim zamanları elde edilmiştir.

Her bir yaklaşımın dağılım zamanları Şekil 7.1., 7.2. ve 7.3.'de görülmektedir. Şekil 7.4., 7.5. ve 7.6.'da ise yaklaşımların sıklık grafikleri (histogram) gösterilmiştir. Lineer arama için ortalama bellek ayırma zamanı 119µs, anahtarlamalı yaklaşım için 36µs ve arraylet temsili için 44µs olarak hesaplanmıştır. Şekil 7.1., 7.2. ve 7.3.'de de görüldüğü gibi, anahtarlamalı yaklaşım, lineer arama algoritmasının ortalama bellek ayırma performansını artırmış ve arraylet temsilinin performansına yaklaştırmıştır. Şekil 7.4.,

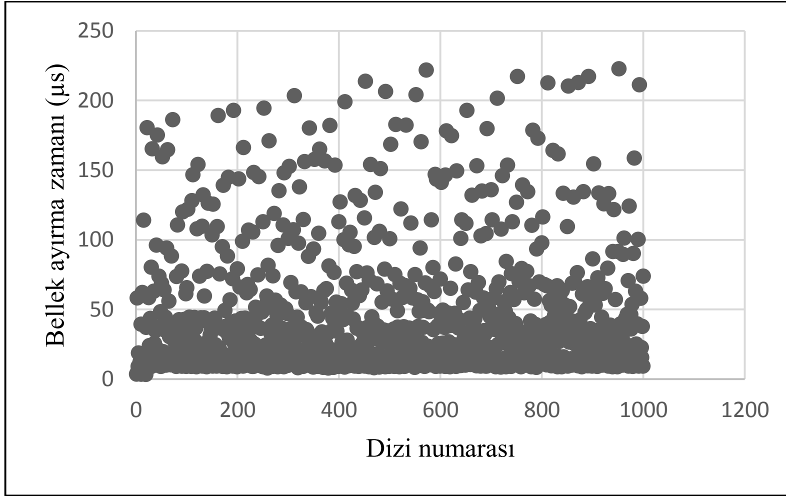
7.5. ve 7.6.'da gösterilen bellek ayırma zamanlarının sıklığı (frekans), sunulan tekniğin performansına daha net bir şekilde şahitlik eder. Anahtarlamalı yaklaşımın lineer arama algoritması üzerindeki performans kazancı, çalışma zamanlarının ortanca (medyan) değerleri ile de doğrulanabilir. Lineer arama, anahtarlamalı yaklaşım ve arraylet temsilinin medyan çalışma zamanları sırası ile 135 µs, 25 µs ve 26 µs olarak hesaplanmıştır. Anahtarlamalı yaklaşımın en kötü çalışma zamanı, lineer aramadan %37 ve arraylet temsilinden %38 daha hızlıdır. Bu sonuçlar anahtarlamalı yaklaşımın en kötü çalışma zamanı performansının lineer arama ve arraylet temsilinden daha iyi olduğunu göstermektedir. Her bir yaklaşımın bellek ayırma zamanlarına ilişkin veriler Tablo 7.1.'de gösterilmiştir.



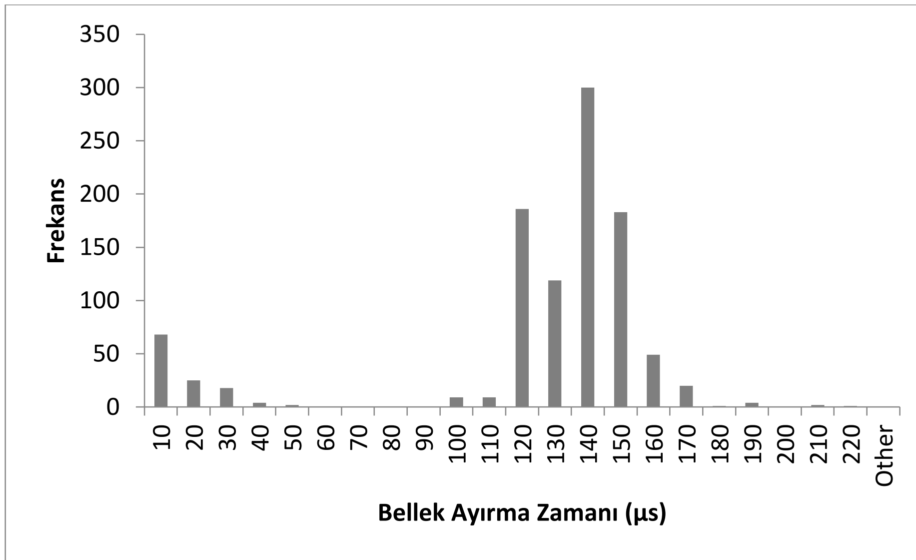
Şekil 7.1. Bellek ayırma zamanlarının dağılımı (lineer arama)



Şekil 7.2. Bellek ayırma zamanlarının dağılımı (anahtarlamalı yaklaşım)



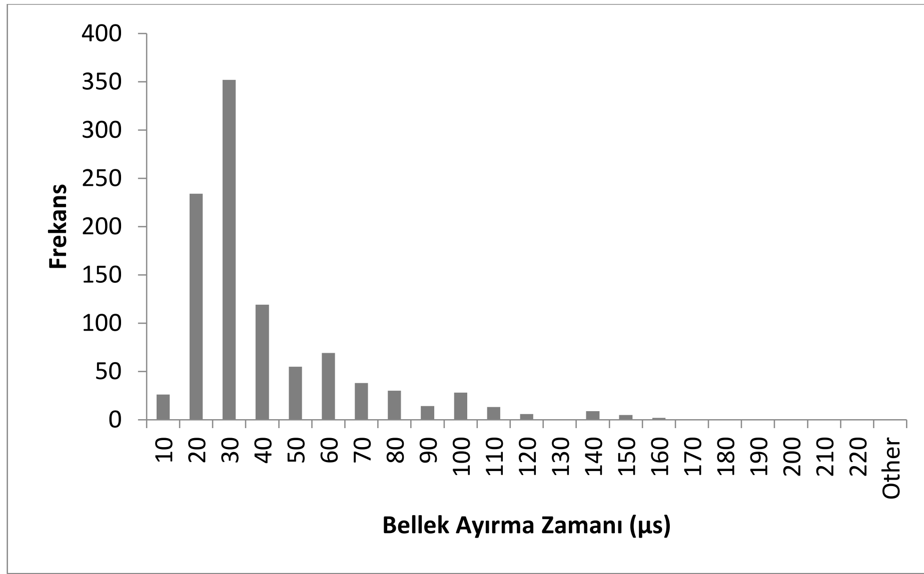
Şekil 7.3. Bellek ayırma zamanlarının dağılım (arraylet temsili)



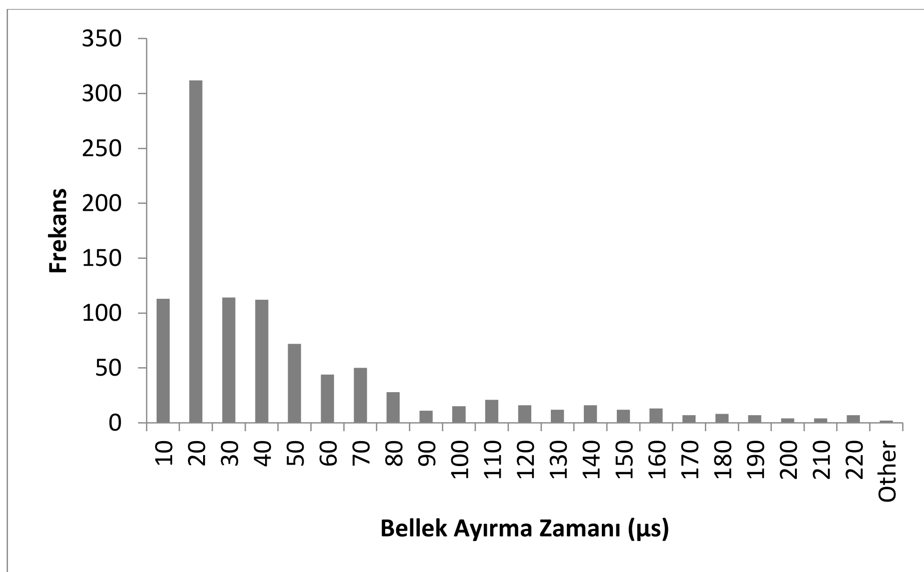
Şekil 7.4. Bellek ayırma zamanlarının histogramı (lineer arama)

Erken dönem testlerimizin sonuçlarına ve bu sonuçlara göre şunu söyleyebiliriz: Atlamalı arama algoritmasının ortalama bellek ayırma zamanı performansı lineer aramadan daha iyi iken, en kötü bellek ayırma zamanı performansı lineer aramadan daha kötüdür. Ancak her iki yaklaşımı birleştiren anahtarlamalı yaklaşım her iki durumda da lineer arama algoritmasından daha iyi performans sergiler.

Anahtarlamalı yaklaşımın zaman karmaşıklığı *O(n)*'dir. Bu değer, lineer arama algoritmasının zaman karmaşıklığı ile aynıdır. Her ne kadar zaman karmaşıklıkları aynı olsa da, bu sonuçlar önerilen yaklaşımın çalışma zamanı performansının daha yüksek olduğunu göstermektedir. Bu sonuçlardaki sürpriz, arraylet temsilinin ortalama ve en kötü bellek ayırma zamanı performanslarının önerilen yaklaşımdan daha düşük çıkmış olmasıdır. Aynı zamanda arraylet temsilinin en kötü bellek ayırma zamanı performansı, lineer aramadan da daha düşük çıkmıştır.



Şekil 7.5. Bellek ayırma zamanlarının histogramı (anahtarlamalı yaklaşım)



Şekil 7.6. Bellek ayırma zamanlarının histogramı (arraylet temsili)

Tablo 7.1. Üç yaklaşımın bellek ayırma zamanları

Yaklaşım	Ortalama Bellek Ayırma Zamanı (µs)	Ortanca Bellek Ayırma Zamanı (µs)	En Kötü Bellek Ayırma Zamanı (µs)
Lineer Arama	119	135	217
Anahtarlamalı	36	25	159
Arraylet Temsili	44	26	222

Arraylet yaklaşımının felsefesine bakarak, bu yaklaşımın daha iyi sonuç vermesi gerektiği düşünülmektedir. Kıyaslama verisini daha detaylı bir şekilde analiz ettiğimizde, arraylet temsilindeki en kötü çalışma zamanlarının birçoğunun, büyük nesneler için bellek ayırma stratejisine (anahtarlamalı yaklaşım) geçiş yapılması sonucunda oluştuğu görüldü.

Minuteman çatısı, arraylet temsilinin karma formunu kullanmaktadır. Bu formda dizinin başlığı, omurgası ve son arraylet (eğer son arraylet blok boyutundan küçük ise) birlikte, bitişik bir bellek alanına yerleştirilir. Eğer bu üçlünün değeri bir blok boyutunu aşarsa, arraylet bellek ayırıcısı, büyük nesneler için bellek ayırma stratejisine geçiş yapar. Gerçekleştirilen testlerde 164 adet dizi için blok boyutu aşılmıştır ve her ne kadar arraylet temsili kullanılsa da büyük nesneler için bellek ayırma stratejisine yani anahtarlamalı yaklaşıma geçiş yapılmıştır. Buna ek olarak bu 164 dizinin her bir arraylet'i için de normal bellek ayırma stratejisi çalıştırılmıştır. Dolayısıyla arraylet temsilinin en kötü çalışma zamanı performansları, anahtarlamalı yaklaşım ve normal nesnelere bellek ayırma stratejilerinin performanslarının toplamıdır. Bu 164 diziye ait veriyi sonuçlardan çıkardığımız zaman, arraylet temsilinin en kötü bellek ayırma zamanı 142 µs değerine düşmüştür ki bu değer diğer iki yaklaşımdan da daha iyidir. Ortalama bellek ayırma zamanı için yeni bir kıyaslama gerçekleştirmedik çünkü sonuç kümemiz değişti. Ancak yine de bu sonuçlardan şu çıkarımı yapabiliriz: Arraylet temsilinin verimli bir uygulaması bitişik bellek ayırma performanslarından daha iyi sonuçlar verecektir.

Tablo 7.2.'de, her bir yaklaşımın ortalama dizi erişim zamanları verilmiştir. Bu tabloya bakarak, anahtarlamalı yaklaşım ve lineer aramanın ortalama dizi erişim zamanlarının

arraylet temsiline göre daha iyi sonuçlar verdiğini görebiliriz. Bunun sebebi, anahtarlamalı yaklaşım ve lineer aramanın, bitişik bellek ayırması gerçekleştirmesi ve arrraylet temsilinin parçalı bellek ayırması gerçekleştirmesidir. Bitişik bellek ayırma durumunda, bir dizinin bir elemanına erişim basit bir işaretçi aritmetiği ile hızlıca halledilebilir. Öte yandan parçalı bellek ayırma durumunda, dizinin elemanına erişim daha karmaşık bir yapıda gerçekleştirilir ve daha çok zaman alır.

Tablo 7.2. Ortalama dizi erişim zamanları

Yaklaşım	Zaman (ns)
Lineer Arma	27
Anahtarlamalı Yaklaşım	26
Arraylet Temsili	124

BÖLÜM 8. SONUÇ

Bu tezde, gerçek zamanlı Java sanal makinelerinin büyük nesnelere bellek ayırma sürecinde kullanılmak üzere anahtarlamalı bir yaklaşım sunulmuştur. Bu yaklaşım kısmi olarak Boyer ve Moore'un karakter dizisi arama algoritmasına dayanmaktadır. Sunulan yaklaşım, gerçek zamanlı bir Java sanal makinesi olan Ovm sanal makinesinin Minuteman çatısında uygulanarak gerçekleştirilmiştir. Bu çalışmada ayrıca, sentetik bir kıyaslama uygulaması da sunulmuştur. Sentetik kıyaslama uygulaması, büyük nesnelere bellek ayırma ve dizi erişim performanslarının testlerinde kullanılmıştır. İlgili kıyaslama uygulamasının yardımı ile Ovm sanal makinesi üzerindeki üç yaklaşım başarılı bir şekilde kıyaslanmıştır. Var olan büyük nesnelere bellek ayırma algoritmasının performansının, sunulan anahtarlamalı yaklaşımla arttırıldığı gösterilmiştir. Ovm üzerindeki bitişik bellek ayırma yaklaşımlarının bellek ayırma performanslarının, parçalı bellek ayırma yaklaşımlarının bellek ayırma performanslarından daha düşük olduğu da gösterilmiştir. Öte yandan yine Ovm üzerinde bitişik bellek kullanan dizilerin ortalama erişim zamanlarının, parçalı bellek kullanan dizilere göre daha iyi sonuçlar verdiği gözlemlenmiştir.

Her ne kadar sunulan yaklaşım diğer gerçek zamanlı Java sanal makinelerine uygulanabilse de bizim çalışmamız iki sebepten ötürü Ovm sanal makinesi üzerinde gerçekleştirilmiştir. Birinci sebebi, Ovm sanal makinesinin BSD lisanslı açık kaynak kodudur. Bu sayede, ilgili sanal makine kaynak kodları ile birlikte ücretsiz bir şekilde edinilmiş ve düzenlemiştir. İkinci sebebi ise sahip olduğu Minuteman gerçek zamanlı çöp toplayıcı çatısıdır. Minuteman çatısı, farklı nesne temsillerinin yapılandırılmasına ve farklı yaklaşımların aynı sistem üzerinde test edilmesine olanak tanımıştır.

KAYNAKLAR

[1] American National Standart Programming Language Fortran, X3.9-1978, American National Standards Institute (ANSI), 1978.

[2] AICAS, The Jamaica Virtual Machine, http://www.aicas.com, Erişim Tarihi: 01 Kasım 2013.

[3] ANDREAE C., COADY Y., GİBBS C., NOBLE J., VITEK J., ZHAO T., Scoped types and aspects for real-time Java memory management. Realtime Systems Journal, 37, 2007.

[4] AONIX, Perc Pico 1.1 User Manual. http://research.aonix.com/jsc/pico-manual.4-19-08.pdf, Erişim Tarihi: 1 Kasım 2013.

[5] ARMBRUSTER A, BAKER J, CUNEI A, FLACK C, HOLMES D, PIZLI F, PLA E, PROCHAZKA M, VITEK J. A real-time Java virtual machine with applications in avionics. ACM Trans Embed Comput Syst, 7: 5:1-5:49, 2007.

[6] AUBERBACH J., BACON D.F., BLAINEY B., CHENG P., DAWSON M., FULTON M., GROVE D., HART D., STOODLEY M., Design and implementation of a comprehensive real-time Java virtual machine, in: Proceedings of the 7th ACM & IEEE international conference on Embedded software (EMSOFT), 2007.

[7] BACON, D. F., CHENG P., RAJAN, V. T., A real-time garbage collector with low overhead and consistent utilization. In Proceedings of the ACM Symposium on Principles of Programming Languages (POPL), 2003.

[8] BAKER H.G., List Processing in Real Time on a Serial Computer, Communications of the ACM, Vol. 21, No. 4, Nisan, 1978.

[9] BAKER J., ANTONIO C., FLACK C., PIZLI F., PROCHAZKA M., VITEK J., ARMBUSTER A., PLA E., HOLMES D., A real-time Java virtual machine for avionics. In Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2006.

[10] BAKER J, CUNEI A, KABLIBERA T, PIZLI F, VITEK J. Accurate garbage collection in uncooperative environments revisited. Concurr Comp-Pract E; 21: 182-196, 2009.

[11] BOLLELLA G., DELSART B., GUIDER R., LIZZI C., PARAIN F., Mackinac: Making hotspot real-time, in: Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05), 2005.

[12] BOLLELLA G., GOSLING J., BROSGOL B., BIBBLE P., FURR S., TURNBULL M., The Real-Time Specification for Java. Addison-Wesley, 2000.

[13] BOLLELLA G., REINHOLTZ K., Scoped memory. In International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), 2002.

[14] BOYAPATI C., SALCIANU A., BEEBEE W., RINARD M., Ownership types for safe region-based memory management in real-time Java. In Conference on Programming Language Design and Implementation (PLDI), 2003.

[15] BOYER R.S., MOORE, J.S., A fast string searching algorithm. Commun ACM, 20: 762-772, 1977.

[16] BROOKS, R.A., Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In Proceedings of the ACM Symposium on Lisp and Functional Programming (LFP), 1984.

[17] BURNS A., WELLINGS A.J., Real-Time Systems and Programming Language: Ada, Real-Time Java and C/Real-Time POSIX (Fourth Edition). Addison Wesley, 2009.

[18] DIJKSTRA E.W., LAMPORT L., MARTIN A.J., SCHOLTEN C.S, STEFFENS E.F.M., On-the-Fly Garbage Collection: An Exercise in Cooperation. Communications of the ACM, Vol. 21, No. 11, Kasım 1978.

[19] CHRISTIANSEN M.V., HENGLEIN F., NISS H., VELSCHOW P., Safe region-based memory management for objects. Teknik Rapor, University of Copenhagen, 1998.

[20] GAY D., AIKEN A., Language support for regions. In Conference on Programming Language Design and Implementation (PLDI), 2001.

[21] GROSSMAN D., MORRISET G., JIM T., HICKS M., WANG Y., and CHENEY J., Region-based memory management in cyclone. In Conference on Programming Language Design and Implementation (PLDI), 2002.

[22] HENRIKSSON, R., Scheduling garbage collection in embedded systems. Doktora tezi, Lund Üniversitesi, 1998.

[23] HENTIES T., HUNT J.J., LOCKE D., NILSEN K., SCHOEBERL M., VITEK J., Java for Safety-Critical Applications. In 2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert), 2009.

[24] JONES R., Dynamic memory management: Challenges for today and tomorrow. In International Lisp Conference, 2007.

[25] JONES R., HOSKING A., MOSS E., The Garbage Collection Handbook: The Art of Automatic Memory Management. CRC Press, Eylül 2011.

[26] JONES R., LINS R., Garbage Collection Algorithms for Automatic Dynamic Memory Management. John Wiley & Sons, West Sussex, England, 1996.

[27] JSR 302, Safety Critical Java Technology, http://jcp.org/en/jsr/detail?id=302, Erişim Tarihi: 01 Kasım 2013.

[28] KALIBERA, T., PIZLO, F., HOSKING, A.L., VITEK, J., Scheduling Hard Real-time Garbage Collection. In Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS), Aralık 2009.

[29] KALIBERA T., PARIZEK P., HADDAD G., LEAVENS G., VITEK J., Challenge Benchmarks for Verification of Real-time Programs. The Fourth ACM SIGPLAN Workshop on Programming Languages meets Program Verification (PLPV), 2010.

[30] KALIBERA T, PIZLO F, HOSKING A.L., VITEK J. Scheduling real-time garbage collection on uniprocessors. ACM Trans Comput Syst; 29: 8:1-8:29, 2011.

[31] KERNIGHAN B.W., RITCHIE D.M., The C Programming Language, Prentice-Hall, 1978.

[32] KOPETZ H., Real-Time Systems – Design Principles for Distributed Embedded Applications. Kluwer Academic Publishers, 1997.

[33] MARWEDEL P., Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems 2nd. Ed. Springer, Dordrecht, Netherlands, 2011.

[34] MCCARTHY J., Recursive functions of symbolic expressions and their computation by machine, Communications of the ACM, 3:184–195, 1960.

[35] NAUR P., BACKUS J.W., KATZ C., ve diğerleri, Report on the Algorithmic Language ALGOL 60. Regnecentralen, Copenhagen, 1960.

[36] NETTLES, S., O'TOOLE, J., Real-time replication garbage collection. In Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI), 1993.

[37] NOBLE J., POTTER J., VITEK J., Flexible alias protection. In European Conference on Object-Oriented Programming (ECOOP), 1998.

[38] PIZLO F., VITEK J. An empirical evalutation of memory management alternatives for Real-time Java. In Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS), Aralık, 2006.

[39] PIZLO, F., VITEK, J. Memory management for realtime Java: State of the art. In Proceedings of the IEEE International Symposium on Object-oriented Real-Time Distributed Computing (ISORC). Orlando, FL, Mayıs 2008.

[40] PIZLO F., ZIAREK L., BLANTON E., MAJ P., VITEK J., High-level programming of embedded hard real-time devices. In EuroSys Conference, Nisan 2010.

[41] PIZLO F., ZIRAREK L., MAJ P., HOSKING A.L., BLANTON E., VITEK J., Schism: fragmentation-tolerant real-time garbage collection. In Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation (PLDI), Haziran 2010.

[42] PLSEK A., ZHAO L., SAHIN V.H., TANG D., KALIBERA T., VITEK J., Developing Safety-Critical Applications with Java. In International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES), Ağustos 2010.

[43] ROBERTZ, S.G, HENRIKSSON R. Time-triggered garbage collection: Robust and adaptive real-time GC scheduling for embedded systems. In Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03);San Diego, California, USA. New York, NY, USA: ACM. pp. 93–102, Temmuz 2003.

[44] SCHOEBERL M. Scheduling of hard real-time garbage collection. Real-Time Syst; 45: 176-213, 2010.

[45] SIEBERT F., The impact of realtime garbage collection on realtime Java programming., in International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), 2004.

[46] SIEBERT F., Realtime Garbage Collection in the JamaicaVM 3.0. In Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES), Eylül 2007.

[47] TAKADA H., Real-time operating system for embedded systems. In: M. Imai and N. Yoshida (eds.): Tutorial 2 – Software Development Methods for Embedded Systems, Asia South-Pacific Design Automation Conference (ASP-DAC), 2001.

[48] TOFTE M., TALPIN J.P., Region-Based Memory Management. Information and Computation, 132, 1997.

[49] WELLINGS A.J., Concurrent and real-time programming in Java. John Wiley & Sons, West Sussex, England, 2004.

[50] VITEK J., PLSEK A., ZHAO L., TANG D., KALIBERA T., SAHIN V.H., LEAVENS G., HADDAD G., Open Safety-Critical Java. Teknik Rapor, Purdue University, Ocak 2010.

[51] YUASA, T., Real-time garbage collection on general-purpose machines, Journal of Systems and Software, cilt. 11, sayı. 3, 1990.

[52] ZHAO L., TANG D., VITEK J., A Technology Compatibility Kit for Safety Critical Java. In The 7th Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), 2009.

[53] ZHAO T., NOBLE J., VITEK J., Scoped types for real-time Java. In IEEE International Real-Time Systems Symposium (RTSS), 2004.

[54] ZURAWSKI R. (Derleyen), Embedded systems handbook : embedded systems design and verification 2nd ed. CRC Press, Boca Raton, Florida, USA, 2009.

ÖZGEÇMİŞ

Veysel Harun ŞAHİN, 29 Ekim 1976 tarihinde Ankara'da doğdu. İlk eğitimine Konya'da başladı. İlk, orta ve lise eğitimini Merzifon'da tamamladı. 1994 yılında Merzifon Anadolu Lisesinden mezun oldu ve Ankara Üniversitesi Elektronik Mühendisliği bölümünde lisans eğitimine başladı. 2002 yılında Sakarya Üniversitesi Elektrik Elektronik Mühendisliği Bölümünde yüksek lisans eğitimine başladı ve 2005 yılında aynı bölümden mezun oldu.