Gerçek Zamanlı Java

Serkan Ceylan[1] Yusuf İbrahim Özkök[2]

[1,2] Yazılım Mühendisliği Müdürlüğü – Komuta Kontrol ve Haberleşme, MST-ASELSAN A.Ş., Ankara

[1]e-posta: ceylan@aselsan.com.tr [2] e-posta: yozkok@aselsan.com.tr

Özetçe

Java'nın, taşınabilirlik, platformdan bağımsızlık ve nesne tabanlı olması gibi özelliklerinin, gerek hatanın en aza indirilmesine gerekse proje teslim tarihlerinin kısaltılmasına büyük etkisi olması sebebi ile Java gerçek zamanlı yazılımlarda da yaygınlaşmaya başlamıştır. Gerçek zamanlı yazılımların söz konusu olduğu yerde gerçek zamanlılık gereksinimleri ortaya çıkmaktadır. Standart Java'nın gerçek zamanlılık konusundaki eksiklikleri, yeni standartlar (RTSJ, RTJ v.b.) ile tanımlanmış olsa da uygulamada bu standartları temel alan çözümler tam olarak olgunlaşmamıştır. Gerçek zamanlı yazılımlarda yoğun olarak kullanılan C/C++, Ada dillerinin yerini almaya aday olan Java'nın, piyasada bulunan gerçek zamanlı çözümlerinin başarım olarak karşılaştırılması, gerçek zamanlı sistemlerde kullanılabilirliği açısından önem taşımaktadır.

Bu bildiride farklı platformlar üzerinde, C ve Java dilleri ile koşturulan test programlarına ait sonuçlar üzerinden karşılaştırma yapılarak gerçek zamanlı sistemlerde Java'nın, gereksinimleri ne kadar karşılayabileceği üzerinde durulacaktır.

1. Giriş

Java, günümüzde yaygın olarak kullanılan nesne tabanlı bir programlama dilidir. Java'nın yaygın olarak kullanılması, tekrar kullanılabilirlik, güvenirlik, güvenlik, üretkenlik, hata oluşumuna engel olma, yüksek üretkenlik imkanı sağlama, kolay test edilebilme ve geniş uygulama alanına sahip olması ile açıklanabilir. Bu faydalar aşağıda sıralanan yapısal özelliklerin bir sonucudur;

- platformdan bağımsız çalışabilme,
- bellek yönetiminin otomatik yapılması - atık toplama (*İng.* Garbage Collection - GC),
- geniş sınıf kütüphaneleri sayesinde birçok konuda hazır ve iyi tanımlanmış tekrar kullanılabilir kod bulunabilmesi,
- sıkı kodlama kurallarına sahip olması,
- otomatik çözümleme araçlarının işini kolaylaştıran, çok iyi tanımlanmış sözdizimi (*İng.* syntax) ve anlambilim (*İng.* semantics) özelliklerine sahip olması,
- çoklu görev yürütümüne (*İng.* Multitasking) ve ağ uygulamalarına desteğin verilmesi.

Bu özellikler proje geliştirme, test ve bakım aşamalarında Java kullanıcılarının zaman ve maliyet açısından büyük tasarruf etmelerini sağlamaktadır. Şimdiye kadar hep üst seviye uygulama yazılımlarının yararlandığı bu faydaların gerçek zamanlı yazılım geliştiricilerinin de dikkatini çekmesi ile Java diline yoğun bir talep oluşmuştur.

Fakat Java denilince ilk akla gelenler, GC' den dolayı Java'nın belirlenimci (*İng.* deterministic) olamayacağı ve ayrıca donanıma doğrudan erişime de izin vermediği için gerçek zamanlı gömülü yazılımlar için bir alternatif olamayacağıdır. Bu görüşün aksine, yakın zamanda yapılan çalışmalar ile Java'nın da gerçek zamanlı olabileceği iddia edilmektedir [1].

Son zamanlarda Java'nın Bölüm 2.1'de de anlatılacak olan kullanım alanının dışına çıkılarak, yazılımdan kaynaklı hataların maliyetinin çok yüksek olduğu, otomotiv, havacılık elektroniği, endüstriyel otomasyon, medikal alanlarında da kullanıldığı görülmektedir. Hafıza koruma özelliği olmayan ve karmaşıklık düzeyi yüksek olan C/C++ gibi dillerin hala ellerinde bulundurdukları performans avantajı karşısında Java'da bulunan hafıza koruma özelliği, düşük karmaşıklık düzeyi ve son gelişmelerle birlikte Java'nın performansının artması Java'yı, C/C++ dillerine tercih edilebilir hale getirmiştir. Bununla birlikte gömülü ve güvenlik öncelikli sistemlerin karmaşıklığının giderek artması, Java'nın sağladığı yüksek üretkenlik imkanı, Java'nın çekiciliğini arttırmaktadır [2].

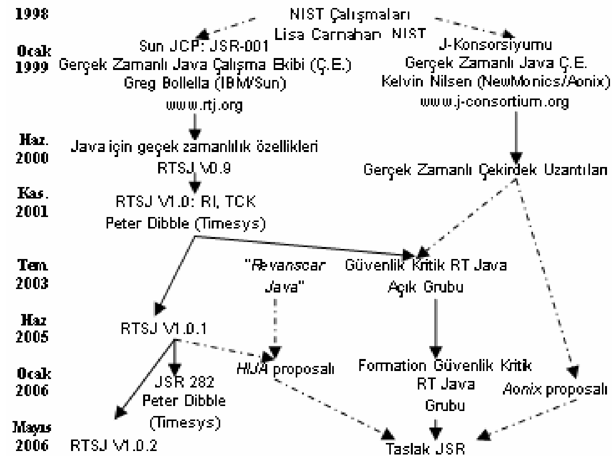
Bu bildirinin amacı, gerçek zamanlı Java gerçeklemelerinin olgunluğunun değerlendirilmesi ve gerçek zamanlılık gereksinimlerini karşılayabilme seviyesinin testlerle ortaya konmasıdır. Bunun için öncelikle Bölüm 2'de çalışma boyunca temel alınacak olan ve bu çalışmaya konu olan gerçek zamanlılık kavramlarından kısaca bahsedilecektir. Daha sonra gerçek zamanlı Java'nın gelişim süreci ve kurumsal destekler ortaya konulacaktır. Bölüm 3'de bu süreçte hangi gerçek zamanlılık özelliklerinin ele alındığı ve bu özelliklerin ne ölçüde karşılanmasının hedeflendiği, bunun için nasıl bir yol izlendiği incelenecektir. Gerçek zamanlı GC algoritmalarındaki gelişmeler, Java çalışma zamanında yapılan değişiklikler ve donanıma erişim için yapılan eklentiler anlatılacaktır. Son bölümde de bu özelliklerden önemli gördüğümüz ve kendi ortamımızda tekrarladığımız başarım testleri ve bu testlerden elde ettiğimiz sonuçlar verilerek yapılan çalışmaların başarım değerlendirilmesi yapılacaktır.

2. Genel Kavramlar

2.1. Java'nın gelişimi, tarihçe

Java, 1990'lı yıllarda, sadece masaüstü bilgisayarlar hedef alınarak geliştirilmiş yüksek seviyeli, nesne yönelimli bir programlama dilidir. Bu hedefe yönelik, Sun Microsystems Inc. tarafından JSDK kütüphanesi ve araçları üretilmiştir. Daha sonraları Java'nın gömülü sistemlerde de kullanılabilirliği değerlendirilmiştir. Sonuçta çok büyük ve hantal olan JSDK, gerçek zamanlılık kaygısı güdülmeden sadece uygun bir boyuta indirgenerek cep telefonu, "palm" gibi gömülü sistemlerde Java kullanımına olanak veren JME sürümü üretilmiştir. Java dilinin çok hızlı bir şekilde yaygınlaşması, kullanıcı görünüşünün çok genişlemesi ve

gömülü sistemlerde de kullanılmaya başlanması ile beraber Java için de gerçek zamanlılık imkânları sorgulanmaya başlanmıştır. Bu gelişmeler sonucunda ilk olarak 1998'de NIST tarafından "Requirements for Real-Time Extension for the Java Platform" [3] başlığıyla bir rapor yayınlanmıştır. Sonrasında bu raporu temel alan gerçek zamanlı Java isterlerinin (*İng.* specification) belirlenmesi için iki paralel çalışma başlamıştır. Bunlardan birisi Sun himayesinde "Sun Java Community Process" (JSR001) altında yapılan çalışmadır. Bu çalışma sonucunda RTSJ üretilmiş ve 2001'de bu ister Java'nın resmi bir uzantısı olarak kabul edilmiştir. Diğer çalışma HP önderliğinde ve Aonix, Ericsson, Microsoft gibi firmaların katılımından oluşan "J Consortium" [4] tarafından yapılan çalışmadır. Bu çalışmanın çıktısı olarak da "Real Time Core Extensions for Java" (RT Core) adı altında ayrıntılı bir tanımlama dokümanı üretilmiştir.



Şekil 1 Gerçek Zamanlı Java'nın tarihi gelişimi[10]

Yakın zamana kadar daha çok teorik ve kavramsal olarak yürütülen bu çalışmalar son zamanlarda gerçek uygulamalarda da boy göstermeye başlamıştır. Örneğin Sun, RTSJ'yi Sparc işlemcileri için gerçeklemiştir. Bazı üçüncü taraf firmalar da hem RT Core hem de RTSJ gerçeklemeleri geliştirmiştir.

2.2. Gerçek Zamanlılık

Gerçek zamanlı bir sistemde yapılan bir işlemin doğruluğu, hesaplamanın mantıksal sonucu ve bu sonucun üretildiği zaman ile ifade edilir. Bu durumda mantıksal olarak doğru olan bir sonuç eğer belirtilen zamanda üretilememişse geçersiz kabul edilir. Zannedilenin aksine hız, gerçek zamanlı sistemlerin bir gereği değildir. Bu tip sistemlerde periyodik hizmetlerin yanında eş zamanlı olmayan dürtüler de ortaya çıkabilir. Gerçek zamanlı sistemler, uç yükleme koşullarında dahi periyodik olarak yapılan işleri aksatmadan sisteme dışarıdan gelen kestirilemeyen dürtülere de kestirilebilir cevaplar üreten sistemler olarak tanımlanabilirler.

2.2.1. Gerçek Zamanlılık Nitelikleri

Gerçek zamanlı sistemlerin karakteristik özellikleri şu şekilde sıralanabilir;

Vaktindelik: Görevler (*İng.* task), kendileri için belirlenen zaman aralığında tamamlanabilmelidir.

Eş zamanlılık veya eş zamanlı işleme: Bir işlemci üzerinde aynı anda sadece bir işlem yapılabilir. Gerçek zamanlı sistemlerde, birden fazla olay eş zamanlı olarak işlenebilmeli ve her görev için vaktindeliğe uyulmalıdır. Eş zamanlılık, işlemci çalışma zamanı görevler arasında, öncelik ve çalışma sırasına göre, paylaştırılarak gerçeklenmektedir.

Determinizm: Gerçek zamanlı sistemler ihtimal dahilindeki tüm olaylara önceden belirlenebilir zaman aralıklarında, bu zaman aralıkları sistemin durumuna göre değişmemeli ve belli bir aralık dışına çıkmamalıdır, cevap verebilmelidirler.

Güvenilirlik: Sistemin bir gerekliliği olmamakla beraber gerçek zamanlı sistemlerde beklenen bir özelliktir. Sistemin tüm görevlerini her zaman doğru olarak yerine getirmesi, gereksinimlerinin dışında davranış sergilememesi ve sağlıklı bir şekilde çalışması anlamındadır.

Bu sistemler sıkı gerçek-zamanlı (*İng.* hard real-time) ve gevşek gerçek-zamanlı (*İng.* soft real-time) olarak sınıflandırılırlar. Sıkı gerçek-zamanlı sistemler hiçbir zaman bitiş zamanını kaçırmaması gereken sistemler olarak tanımlanırken, gevşek gerçek-zamanlı sistemler ise gecikme ve bitiş zamanlamalarının kaçırılmaları durumunda sistemin düşük performanslı olarak çalışmaya devam edebildiği sistemlerdir.

2.2.2. Gerçekleme Yöntemleri ve İşleyiş

Gerçek zamanlı bir sistemde gerçek zamanlılık niteliklerini karşılayabilmek için bazı mekanizmalar geliştirilmiştir. Tüm bu mekanizmalar gerçek zamanlı işletim sistemleri (*İng.* Real Time Operating System - RTOS) tarafından gerçeklenirler. Böyle bir işletim sisteminin en azından Şekil 2'de resmedilen servisleri sağlaması gerekir.

Görev Zaman Planlaması (*İng.* Task-scheduling) RTOS çekirdeğinin merkezinde bulunmaktadır. Gömülü uygulamalar için kullanılan çoğu işletim sistemi uygulamanın çalışmasını öncelik tabanlı iş-kesmeli zaman planlaması (*İng.* priority-based pre-emptive scheduling) ile kontrol etmektedir. Bu yapıda RTOS, işlemci üzerinde her zaman çalışmaya hazır halde bulunan en yüksek öncelikli görevin (*İng.* task) çalışmasını sağlar. Bu kavram ile birlikte görevlerin önceliklendirilmesi, görevlerin birbirini kesebilmesi (*İng.* preemption) gibi kavramlar gelişmiştir. Beraberinde öncelik tersleme (*İng.* priority inversion) gibi sorunlar ortaya

Şekil 2 Gerçek zamanlı işletim sistemi tarafından sağlanan temel servisler

çıkmış ve bu sorunlara öncelik kalıntılama (*İng.* priority inheritence) gibi çözümler üretilmiştir. RTOS'un ikinci en önemli parçası görevler arası iletişim altyapısını sağlayan görevler-arası iletişim ve eşzamanlılık (*İng.* Inter-task Communication and Synchronization) dır. Mesaj kuyruğu (*İng.* message queue), iletişim tüneli (*İng.* pipe), semafor (*İng.*

semaphore), posta kutusu (*İng.* mailbox) , olay grupları (*İng.* event groups) ve eşzamanlı olmayan işaretler (*İng.* asynchronous signals) gibi nesneler bu amaca hizmet eden mekanizmalardır. Çalışma zamanında ortaya çıkan bellek gereksinimleri için geçici olarak RAM bellek tamponu ayırmayı sağlayan RTOS'un bir diğer ana bölümü de dinamik bellek ayırma (*İng.* Dynamic Memory Allocation) yapısıdır. Zamanlayıcılar (*İng.* Timers) ise yazılım gecikmeleri, zaman aşımı ve zaman ölçümleri için altyapı sunan, RTOS'un vazgeçilmez bir parçasıdır. Bir diğer önemli birim ise, aygıt sürücüleri ile olan arayüzün arkasındaki motor olarak tanımlayabileceğimiz, aygıt sürücüsü yöneticisi (*İng.* Device Driver Supervisor) dir. Bu birimin görevi bir yandan uygulama yazılımlarına sürücü yazılımlarından standart bir erişim sağlamak diğer yandan uygulama yazılımından yazılım-donanım ara yüzünün alt seviye uygulamalarını gizlemektir. Gerçek zamanlılık çerçevesinde yukarıda bahsi geçen tüm bu temel RTOS birimleri kestirilebilir gerçek-zamanlı cevap sürelerine sahip olmalıdırlar.

C/C++ benzeri dillerin aksine Java dili, nesneleri kendi Java sanal makineleri (*İng.* Java Virtual Machine - JVM) üzerinde çalıştırmak üzere geliştirilmiştir. JVM Java uygulamaları için bir nevi işletim sistemi işlevi görür. Gerçek zamanlılık konusu da aslında Java dili ile değil, JVM ile ilgilidir. Bu yaklaşım esasında Java'nın en önemli özelliklerinden biri olan çalıştırılabilir kod anlamında platformdan bağımsızlığın sağlanması için geliştirilmiştir. Fakat gerçek zamanlı sistemler için şimdilik çok uzak bir hedef olacağı için biz bu özelliği konumuzun dışında tutmayı yeğledik. JVM ile ilgili ayrıntılı bilgiler Bölüm 3'te verilmiştir.

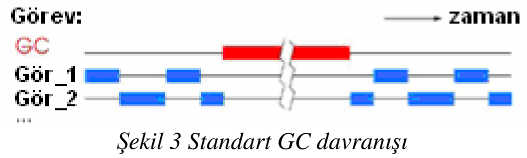
3. Gerçek Zamanlı Java

Java dili masaüstü uygulamalar için ve donanımdan bağımsız yazılım geliştirme, yazılımların güvenli çalışması, geliştirme sürecinde standart geliştirme araçlarının kullanılabilmesi gibi amaçlar hedeflenerek tasarlanmıştır.

Ancak Java'nın özellikle platformdan bağımsız olma hedefinin sonuçları olan determinizmden yoksunluk (GC'den kaynaklanmaktadır) ve donanıma erişimin kısıtlanması Java'nın gerçek zamanlı ve gömülü uygulamalar için kullanılmasına engel olan iki temel unsur olarak ortaya çıkmıştır. İşte gerçek zamanlı Java çalışmaları bu iki unsur üzerine yoğunlaşmıştır.

Java dilinin merkezinde bulunan ve görevi kullanılmayan belleğin belirlenerek kullanılabilir bellek havuzuna aktarılması ve belleğin parçalı (*İng.* fragment) hale gelmesini engellemek olan GC, geleneksel yöntemlerde görevini yerine getirmek için belirlenimci olmayan çağrı üstünlüğü (*İng.* preemption) özelliğini kullanır. Bu durum sistemde kestirilemeyen bir zamanda kestirilemeyen bir süre duraklamaya sebep olur ve bu duraklamalar sistemin zamanlama davranışının kestirimini imkânsızlaştırır.

GC, Java için bir gereksinim olmamakla beraber kullanılmayan objelerin bellekte kapladığı alanın tekrar kullanılabilir hale getirilmesinin bir başka yolu tanımlanmamıştır. Eğer bir sistemde sınırsız bellek alanımız olsa ya da Java uygulamamız kontrollü olarak nesne yaratsa ve hep aynı nesnelerle çalışsa JVM, GC sınıfı olmadan da çalışabilir. Fakat kimse bu kısıtları öne süren bir JVM'e talip olmayacağı için de gerçek zamanlı sistemler için de GC sınıfının bulunması bir zorunluluk halini almıştır.



Şekil 3 Standart GC davranışı

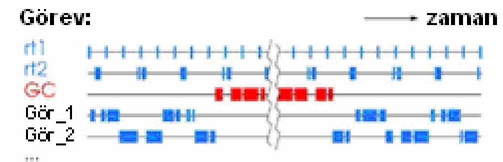
Bu nedenle gerçek zamanlılık özellikleri göz önünde bulundurularak çeşitli GC algoritmaları ("Reference Counting", "Mark and Sweep",v.b.) geliştirilmiştir [5]. En basit anlatımıyla GC süreci aşağıdaki üç adımdan [6] oluşur ve tüm algoritmalar bu temel adımları çeşitli şekillerde gerçekleştirirler;

1. Tüm nesnelerin bir haritası (*İng.* root set) çıkartılır. Sistemdeki tüm erişilebilir nesneler bu haritaya işlenir. Bu nesneler izlek (*İng.* thread) yığınları üzerindeki referanslar üzerinden erişilebilir olmalıdır.
2. Bir algoritma ile erişilebilir olan tüm nesneler bulunur.
3. Bu çıkartılan harita üzerindeki referanslarla erişilemeyen nesneler atık olarak sınıflandırılır ve atılır.

Gerçek zamanlı uygulamalar çalışma ortamları ile yoğun bir etkileşimdedirler. Standard Java çalışma ortamına erişimi sağlayan bir altyapı sunmamakla beraber gerçek zamanlı / gömülü Java'da donanıma erişim desteği JNI üzerinden sağlanabilmektedir. Ancak bu durum Java'nın taşınabilirlik felsefesine ters düşmektedir.

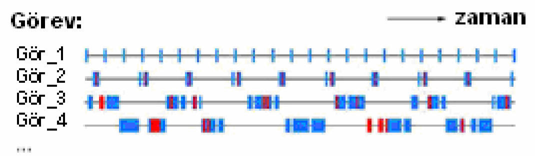
3.1. GC'nin Gerçek Zamanlılığa Uyarlanması

Gerçek zamanlı programlama yapabilmek için gerçek zamanlı görevlerin GC duraklamalarından etkilenmeyeceği bir yapı oluşturulmalıdır. Gerçek zamanlı Java isterleri GC işlemi ile ilgili olarak gerçek zamanlı görevlerin GC işlemini kesebilmesinin ötesinde bir gereksinim belirlememiştir. Bunun için en temel yaklaşım GC etkinliğinin, güvenle kesilebileceği küçük aralıklara bölünmesidir.



Şekil 4 Standart GC'nin RTSJ görevleri ile ilişkisi

Gerçek zamanlı bir GC algoritmasının çalışmasının diğer görevler ile ilişkisi Şekil 5' de görülebilir.



Şekil 5 Gerçek zamanlıGC' nin diğer görevlerle ilişkisi

Gerçek zamanlılık özellikleri göz önünde bulundurularak geliştirilen GC algoritmalarından [5] "Reference Counting" gerçek zamanlı sistemler için en uygun olanıdır. Bu algoritmada, GC işleminin kendine ait döngüsünün olması yerine GC' ana uygulama içine gömülmüştür ve GC işi için harcanan zaman çok küçüktür.

3.2. Geçek Zamanlı Java Eklentileri

Donanıma erişim ve GC mekanizmalarının ötesinde Java'nın gerçek zamanlı sistemlerde kullanılmasında ihtiyaç duyulan bazı düzenlemelere gerek duyulmuştur. Gerçek zaman davranışını sağlamak için çok iyi tanımlanmış görev, öncelik ve eşzamanlama modeline ihtiyaç vardır. Standart Java görev etkileşimi gereksinimlerinin asgari olarak tanımlamış olması itibari ile eksiktir. Önceliklendirme yeteneği tam olarak tanımlanmamıştır; çağrı üstünlüğü (*İng.* preemption) kapsanmamış ve öncelik terslenmesi çok kaba tanımlanmıştır. Gerçek zamanlı Java için ihtiyaç duyulan düzenlemeler Tablo 1' de verilmiştir.

Standart Java gerçek zamanlılık açısından bir dizi kısıtlar içermektedir. Bu dokümanın önceki bölümlerinde ara ara bu kısıtlardan ve kaynaklarından bahsedilmiştir. Bu tabloda gerçek zamanlılık ile ilgili konular "Durum" kolonu altında listelenmiştir. İkinci kolonda ise "problem" adı altında standart Java'nın bu konulardaki yeterliliği değerlendirilmiştir [7].

Tablo 1 Standart Java'nın gerçek zamanlılık kısıtları

Durum	*Problem*
İzlek Modeli	*Yetersiz: hazır izleklerin çalıştırılma politikalarının bir garantisi yoktur.*
İzlekler arası senkronizasyon	*Eksik Özellik: Bir izleği belirli bir zamana kadar geçici olarak durdurabilmenin güvenli bir yolu yoktur.*
Paylaşımlı Kaynaklara Erişim	*Yetersiz: öncelik tersleme durumuna karşılık bir önlem alınmamıştır.*
Bellek Atama	*Yetersiz: Bellek yığından atanır, fakat atama politikası tanımlanmamıştır ve öngörülemez ve belirsiz olabilir.*
GC	*Yetersiz: GC algoritması ve tetikleyici koşulları belirli değildir, kestirilemez ve belirsiz olabilir.*
Donanımsal Kesme İşleme	*Eksik Özellik: izlek ya da olay işleyicilerini donanım kesmelerine bağlamanın bir yolu tanımlı değildir.*
Asenkron Olay İşleme	*Yetersiz: Eğer alttaki izlek modeli uygunsa standart Java olay modeli de uygundur.*
Dinamik Sınıf Yükleme	*Uygun değil: Dinamik sınıf yükleme zamanlama davranışını bozar.*
Sınıf İlkleme	*Yetersiz: Sınıfların ne zaman ilkleneceğinin garantisi yoktur.*

3.3. Java Uygulamaları Çalıştırma Yöntemleri

Java uygulamaları C'den farklı olarak Java derleyicileri ile Java bayt (*İng.* byte) kodları şekline dönüştürülür. Bu bayt kodlar ancak bir Java işlemcisi üzerinde çalışabilir. JVM Java işlemcisini öykünülmeyerek (*İng.* emulation) bayt kodlarının herhangi bir işlemci üzerinde çalışmasını sağlar. Bu sayede tek bir platforma uygun olarak yazılan programlar aslında birçok platform üzerinde çalıştırılabilir hale gelmiş olur.

Gerçek zamanlı sistemler çoğunlukla zaman kritik sistemlerdir. Seçilen Java derleyicisinin çalışma zamanına doğrudan etkisi olması sebebiyle gerçek zamanlı sistemlerde hangi derleyicinin kullanılacağı önem taşımaktadır. Java derleyicileri üç grupta toplanmıştır;

Yorumlama (Interpreted) : Bir ara uygulama (JVM) bayt kodlarını sistemdeki işlemciye özel makine kodlarına çevirerek çalıştırır. Her satır bayt kod çalıştırılırken çevirme işleminin yapılması gereklidir.

İlk işletme anında bir kere yorumlanacak şekilde derleme (Just in Time Compiled - JIT) : Dinamik olarak yüklenen bir Java metodunun bayt kodları sadece ilk çalıştırıldığı anda bir kere JVM tarafından ana işlemcinin makine kodlarına dönüştürülür.

Önceden yorumlama (Ahead of Time Compiled - AOT) : Java sınıf dosyaları ana işlemciye özel makine kodlarına dönüştürülür. Üretilen kod işlemcinin makine kodu olduğu için yorumlamaya göre çok daha hızlı çalışır.

4. Başarım Testleri

Java'nın gerçek zamanlılık özelliklerini nesnel olarak değerlendirebilmek için oluşturulan bir ortamda kontrollü deneyler gerçekleştirilmiştir. Testler iki ana sınıfa ayrılmıştır.. Birinci sınıfta Bölüm 2.2'de verilen gerçek zamanlılık özelliklerinden tecrübelerimize göre en önemli gördüğümüz ve diğerlerini de kapsadığını düşündüğümüz iki özelliğe yönelik geliştirilen testler ile gerçek zamanlı olarak nitelendirilen bir Java versiyonu üzerinde ölçümler gerçekleştirilmiştir. İkinci sınıfta ise Java çalışma zamanı başarım testleri yapılmıştır.

Genel test ortamı Şekil 6'da verilmiştir. Sunucu bilgisayar üzerinde kurulu olan geliştirme ortamında derlenerek oluşturulan çalıştırılabilir kod, ethernet arayüzü ile üzerinde vxWorks koşan hedef işlemci kartlarında koşturulmuştur. Kullandığımız Java test kodları AOT yöntemi ile derlenerek çalıştırılabilir JVM uygulaması içine gömülmüştür. Gerek geliştirme ortamının sunduğu araçlar gerekse yazılımda bulunan ölçüm noktaları ile test verileri toplanmıştır. Geliştirme ortamının sunduğu araçlar işletim sisteminin görevler arası etkileşimini de görmek açısından faydalı olmuştur.

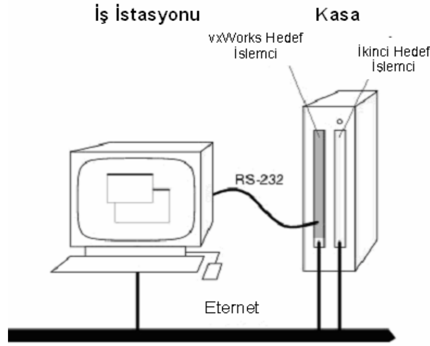
Testler ASELSAN tarafından tasarlanmış olan PPC7457 ve PPC8245 tabanlı işlemci kartları üzerinde ayrı ayrı yapılmıştır. JVM olarak, Sun'ın JRE 1.5'i ile RT-JVM çözümleri sunan bir firmaya ait bir ürün testlerde kullanılmıştır.

4.1. Determinizm Testi

Bölüm 3'te anlatılan çalışmalar ile Java'nın GC kaynaklı determinizm eksikliğinin giderildiği iddia edilmektedir.
Bu test ile JVM'in, GC algoritmasının sistem üzerindeki etkisi belirlenmeye çalışılacaktır. Gerçek zamanlı bir sistemde GC determinizmi bozmadan ve çok hızlı bir şekilde kesilebilir ve anahtarlanabilir olmalıdır.

Bu testte gerçek zamanlı bir görev, yüksek öncelikli bir izlek yaratılarak benzeştirilir (*İng.* simulate). Bu izlek her çalıştığında sadece belirli bir süre uyumak üzere programlanarak belirli bir periyotta çalışan gerçek zamanlı bir görev benzeştirilmiş olur. Sistemde bu izleğin gerçekten ne kadar uyuduğu standart Java time sınıfı ile ya da özgül (*İng.* native) metotların kullanıldığı yüksek çözünürlüklü saat mekanizmaları ile ölçülebilir. Standart Java zamanlayıcı mekanizması işletim sisteminin işaret (*İng.* Tick) periyodu ya da 1 ms çözünürlüğündedir (küçük olan). Beklenen uyuma süresi ile ölçülen uyuma süresi arasındaki fark bize JVM'in yüksek öncelikli izleğe geçiş anındaki gecikmesini verecektir.

Statik çalışma durumunda bu gecikmenin hep sabit olması beklenir. Fakat dinamik bellek



Şekil 6 Test Ortamı

atamalarının olduğu, birçok farklı görevin bellek talebinde bulunup tekrar bu bellekleri geri verdiği bir durumda bu gecikme süresi değişebilecektir. Gerçek zamanlı bir sistemde ise böyle durumlarda bile bu gecikme hep sabit kalmalıdır. Bu durumu benzeştirmek için test kodu içinde ikinci bir izlek oluşturulmuştur. "Hammer" ismi verilen bu izlek rasgele nesneler yaratır. Böylece rasgele aralıklarda GC'nin çalışması sağlanır. Eğer GC algoritması gerçek zamanlılık gereklerine uygunsa, GC izleği gerçek zamanlı izlek tarafından kesilebilmeli ve hızlı ve belirli bir sürede gerçek zamanlı izleğe geçiş yapılabilmelidir.

Bu test için Bölüm 4'te anlatılan test ortamı kullanılmıştır. Standart ve gerçek zamanlı Java ortamlarında donanımsal farklılıklardan dolayı oluşabilecek sapmaları minimize etmek üzerede JVM'lerin kullanabilecekleri en fazla yığın alanı iki ortamda da 8 MB olacak şekilde kısıtlanmıştır. Ayrıca gerçek zamanlı bir işletim sistemi olmayan Windows XP işletim sisteminin etkilerini ortadan kaldırmak için bu ortamda test süresince sadece JVM' in çalıştığı kontrol edilmiştir.

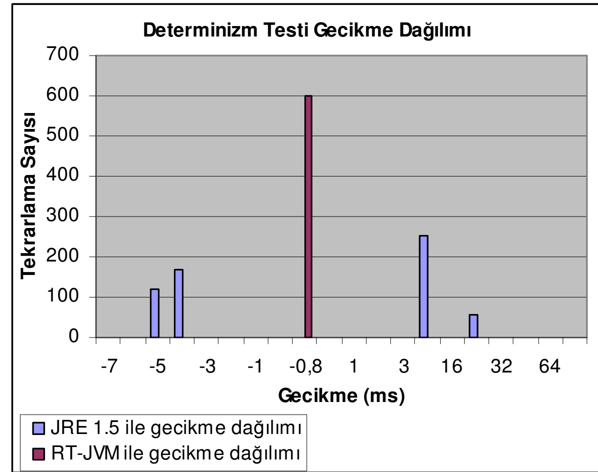
Aşağıda Şekil 7'de iki test ortamı için de test sonuçları görülmektedir. Test sonuçlarının değerlendirmesi için gecikme süresinin dağılımına bakılmalıdır. Gecikme süresinin büyüklüğü elbette ki tercih edilmeyen bir özellik olmasına rağmen gerçek zamanlılık açısından asıl önemli olan bu sürenin dağılımıdır. Gecikme süresinin büyüklüğü sistem saatinin (işaret periyodu) çözünürlüğü ile doğrudan ilintilidir. Gerçek zamanlı bir sistemde sistem saati periyodik görevin periyoduna göre çok kaba ise, mesela 20 ms periyotlu bir görev için 16 ms işaret periyodu varsa, görevin zamanında çalışması beklenmemelidir. Fakat belirlenimci bir sistemde böyle bir durumda bile standart sapmanın düşük olması beklenir.

JRE 1.5 test ortamında elde edilen çıktılar incelendiğinde JVM' in davranışının gerçek zamanlılıktan uzak olduğu cevap süresi dağılımından (10.3 ms standart sapmadan) anlaşılmaktadır. Test süresince gerçek zamanlı izleğe geçiş için harcanan süre çok çeşitli aralıklarda olmuştur. Ayrıca ortalama cevap süresi de 9.4ms olarak oldukça uzun bir süre olarak gerçekleşmiştir. (Bu aslında değerlendirilecek bir kriter değil. Windows'un sistem saat periyodu ile ilgilidir.) RT-JVM üzerinde elde edilen sonuçlar incelendiğinde ise, sonuçların gerçek zamanlılık kriterlerine daha yakın olduğu açık bir şekilde görülmektedir. RT-JVM üzerinde gerçek zamanlı izleğe her seferinde aynı gecikme süresinde geçiş yapılmıştır. Gecikme süresi de 0.85 ms gibi kabul edilebilir bir değer olarak ölçülmüştür. Aslında bu değer ideal şartlarda sistem saat frekansı ve periyodik görevin frekans bilgileri ile bulunabilir. Bu bilgiler ile Tablo 2'de hesaplanan ideal gecikme süresi ile bu değer karşılaştırıldığında sistemin ne kadar belirlenimci olduğu daha açık görülebilir.

Tablo 2 İdeal gecikme süresi hesaplaması

"Task" periyodu : 20 ms ve Sistem Saat hızı: 470 Hz. Saat periyodu (tick period) = 1/470 = 2,127 ms Görev değişimi ancak sistem tick'inin tam katlarında yapılabilir. 20 ms'ye ye en yakın değer : 2.127*9=19.143 Hesaplanan ideal gecikme süresi : 19.143-20 = -0.857

Test sonuçlarında gözlenen negatif değerler JVM' in uyuyan izlekleri zamanından önce uyandırabilmesinden kaynaklanmaktadır. Uyuyan izleklerin uyanması ancak işletim sistemi işaretlerinde gerçekleşebilir. JVM'lerin gerçeklenmesine bağlı olarak da bazı JVM'ler uyanma zamanını en yakın izleğe dönüştürerek belirlemektedirler. Bu durumda örneğin 2.2 işaretlik bir süre uyuması için programlanan izlek 2. işarette uyandırılmış olur. Tablo 2'de yapılan hesaplamada da bu negatif değerin nasıl oluştuğu gösterilmiştir.



Şekil 7 JVM gecikme testi test sonuçları – 1

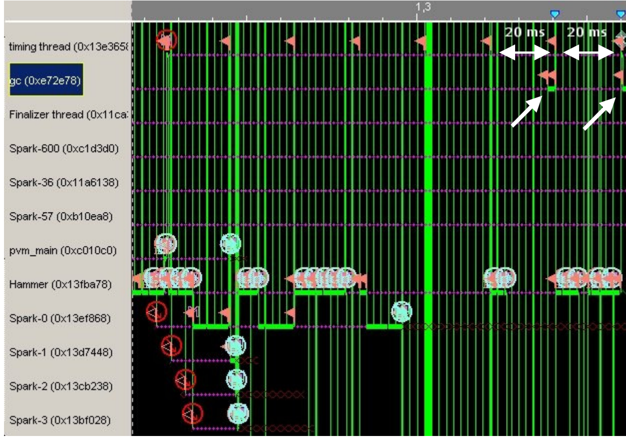
Gerçek zamanlı Java ortamı üzerinde yapılan testler için ayrıca çalışma zamanı analizi de yapılmıştır. Bu analiz sonucunda GC' nin devreye girdiği anlarda bile periyodik izleklerin periyodunun bozulmadığı ve GC' nin kesilebildiği görülebilmektedir. Şekil 8'deki analiz görüntüsünde bu durum görülebilir. Burada periyodik izlek "timing thread" ismi ile gösterilmiştir. GC izleği ise "gc" ismiyle görülmektedir.

Tablo 3 JVM gecikme testi sonuçları

	JRE 1.5	RT JVM
OS	Windows XP	VxWorks 5.5
CPU	Intel 3.0 Ghz	Mpc8245 250 Mhz
ClkRate	-	470(1/470=2.127ms)
Görev Periyodu	20 ms	20 ms
Tekrarlanma	600 döngü	600 döngü
Gecikme Uç Değerleri (ms)		
En yüksek	43.0	-0.851069
En düşük	-5	-0.851069
Ortalama	5.225	-0.851069
Mutlak Ortalama	9.491666	0.851069
Toplam	3135.0	-510.6414
Standart sapma	10.370968	0.0

"Spark" ile başlayan görevler "Hammer" izleğinin rasgele oluşturduğu ve yok ettiği izleklerdir. Eşit aralıklı boyuna çizgiler ise 2 ms'lik JVM sistem işaretleridir.

Şekil 8' da bir süre sonra ("timing thread" inin 6. periyodunda) GC' nin devreye girdiği görülebilir. GC "timing thread" tarafından kesilmiş ve "timing thread" inin periyodu değişmemiştir.



Şekil 8 Determinizm testi analiz görüntüsü

4.2. Öncelik Kalıntılama Testi

RTSJ'nin getirdiği gerçek zamanlılık özelliklerinden biri de standart JVM'de olmayan öncelik kalıntılama mekanizmasıdır. Bu test durumunda gerçek zamanlı Java'nın öncelik kalıntılama özelliği test edilecektir.

Bu test'te farklı önceliklerde üç izlek yaratılır. Bu izleklerden en yüksek ve en düşük öncelikli olanları eşzamanlıdır (senkronize). Diğeri ise bağımsız çalışır. Bu üç izlek en düşük öncelikli olan çalışmaya ilk başlayacak şekilde sırasıyla çalıştırılırlar. İlk çalışan "ResourceGrabbing" izleği ekrana çalışmaya başladığını yazdırır ve yüksek öncelikli izlekle ortaklaşa kullanacakları kaynak üzerinde kaynağa erişimi kendi üzerine kilitleyerek çalışmaya başlar. Bu izlek bir süre sonra kaynağını serbest bırakarak ekrana işini bitirdiğini yazdıracaktır. İkinci sırada "FinishingThread" olarak isimlendirilen en yüksek öncelikli izlek çalıştırılır. Bu izlek ilk izleğin kilitlediği kaynak üzerinde çalışacağı için bekleme durumunda kalır. Üçüncü olarak "WaitingThread" olarak isimlendirilen bağımsız çalışacak orta öncelikli izlek çalıştırılır. Bu izlek bir sonsuz döngü içinde yüksek öncelikli izleğin bir bayrağı kaldırmasını beklemektedir. Dolayısıyla yüksek öncelikli izlek işini bitirmeden bu izlek de işini bitiremeyecektir.

Bu testte üç ayrı durum gerçekleşebilir;
Birinci durum, öncelik kalıntılamanın hiç gerçeklenmediği bir JVM'de çalışması durumu: Bu durumda orta öncelikli izlek yüksek öncelikli izleğin beklediği bir kaynak üzerinde çalışan düşük öncelikli izleği keserek çalışmaya başlayacaktır. Bu orta öncelikli bir sonsuz döngü içinde yüksek öncelikli izleğin bir bayrağı set etmesini beklediği için bu döngü içinde kalacaktır. Sistemde bu izleği kesecek çalışabilir durumda daha yüksek öncelikli bir izleği de olmadığı için sistem bu izlek üzerinde kilitlenmiş olacaktır. Bu duruma sınırsız öncelik tersleme (*İng.* unbounded priority inversion) denilmektedir.

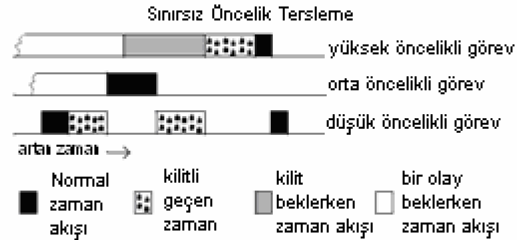
İkinci durum, öncelik kalıntılamanın tam olarak gerçeklenmediği bir JVM üzerinde çalışması durumu: Bu durum, testimizi standart Java üzerinde çalıştırdığımızda karşılaştığımız bir durumdur. Düşük öncelikli izlek bir paylaşımlı kaynak üzerinde çalışırken yüksek öncelikli izlek bu kaynağa erişmek istediğinde düşük öncelikli izleğin önceliği yükseltilmiştir. Fakat düşük öncelikli izlek bu kaynak üzerinde çalışmasını bitirmesine rağmen önceliği tekrar eski haline dönmemiştir. Test sonucunda ilk işini bitiren izleğin en düşük öncelikli izlek olduğu gözlenmiştir.

Üçüncü durum, öncelik kalıntılamanın tam ve doğru olarak gerçeklendiği bir JVM üzerinde çalışması durumu: Bu durum gerçek zamanlı JVM üzerinde izlenmiş ve öncelik kalıntılama gereksiniminin başarıyla sağlandığı görülmüştür.

Öncelik kalıntılama algoritması uygulanmış gerçek zamanlı bir sistemde "ResourceGrabberThread" en düşük öncelikli olmasına rağmen yüksek öncelikli bir izlek onun kullandığı bir kaynağı beklediği için "ResourceGrabberThread"'in önceliği bu yüksek öncelikli izleğin önceliğine yükseltilecek ve orta öncelikli "WaitingThread", "ResourceGrabberThread"'i kesemeyecektir. Düşük öncelikli izleğin paylaşımlı kaynak üzerinde çalışması bittiğinde bu izlek tekrar eski öncelik seviyesine indirilecektir.

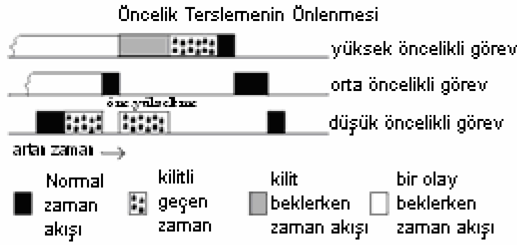
Böyle bir JVM'de beklenen durum "ResourceGrabberThread"i çalışırken "FinishingThread" aynı kaynağı beklemeye başladığında "ResourceGrabberThread"in önceliği yükseltilecektir ve "WaitingThread" beklemede kalacaktır. "ResourceGrabberThread"'i kaynağı serbest bıraktığı anda (ekrana işinin bittiğini yazamadan önce) "FinishingThread" araya girecek ve ilk olarak işini tamamlayan izlek olacaktır. Bu izlek işini bitirdikten sonra "WaitingThread", "ResourceGrabberThread"den önce çalışacak ve ikinci sırada işini bitirecektir. "ResourceGrabberThread"'i ise en son sırada işini bitirebilecektir.

Aşağıda Şekil 9'de öncelik kalıntılama mekanizması gerçeklenmediği zaman oluşan öncelik tersleme durumu resmedilmiştir. Bu şekilde de görüldüğü gibi orta öncelikli izlek yüksek öncelikli izleğin beklediği bir kaynağı kullanan düşük öncelikli izleği keserek yüksek öncelikli izleğin gecikmesine sebep olmaktadır.



Şekil 9 Öncelik Tersleme

Şekil 10'da ise bu durumun öncelik kalıntılama mekanizması ile nasıl önlendiği gösterilmiştir. Düşük öncelikli izlek bir kaynak üzerinde çalışırken daha yüksek öncelikli bir izlek aynı kaynağa erişmek istediği zaman düşük öncelikli izleğin önceliği bu kaynağı kullanmak isteyen yüksek öncelikli izleğin önceliğine yükseltilir. Böylece orta öncelikli bir izlek bu düşük öncelikli izleği kesemez.



Şekil 10 Öncelik Kalıtlama mekanizması ile öncelik terslemenin önlenmesi

Test ortamı ve testin gerçekleştirilmesi:
Bu test için geliştirilen Java uygulaması hem standart Java (JRE 1.5) üzerinde hem de bir RT-JVM üzerinde çalıştırılmıştır. Her iki durumdaki ekran görüntüleri aşağıda Tablo 4'te verilmiştir.

Tablo 4 Priority inversion testi test sonuçları

JRE 1.5 üzerinde	RT-JVM üzerinde
This demonstration illustrates priority inheritance. If priority inheritance is properly implemented on the test vm, then the threads will exit in the proper order: Thread (prio 3): started Thread (prio 8): started Thread (prio 5): started Thread (prio 3): finished! - - I should finish third. Thread (prio 8): finished! - - I should finish first. Thread (prio 5): finished! - - I should finish second.	This demonstration illustrates priority inheritance. If priority inheritance is properly implemented on the test vm, then the threads will exit in the proper order: Thread (prio 3): started Thread (prio 8): started Thread (prio 5): started Thread (prio 8): finished! - - I should finish first. Thread (prio 5): finished! - - I should finish second. Thread (prio 3): finished! - - I should finish third.

Bu ekran görüntülerinden de görüldüğü gibi standart Java kullanıldığında öncelik kalıntılamanın gerçekleştiği fakat hatalı çalıştığı görülmektedir. Gerçek zamanlı Java kullanıldığında ise öncelik kalıntılamanın doğru bir şekilde çalıştığı ve beklenen durumun gerçekleştiği görülmüştür.

4.3. Java Çalışma Zamanı Başarım Testleri

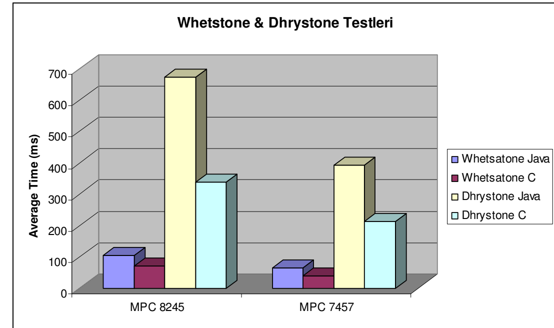
Daha önce çalışma hızının bir gerçek zamanlılık gereksinimi olmadığını vurgulamıştık. Fakat pratik uygulamalarda yazılımların donanım kaynaklarını sonuna kadar kullanma isteği ve özellikle gömülü uygulamalarda sorun teşkil eden ısı atımı gibi kısıtlardan dolayı özgürce donanım kaynaklarının arttırılamaması nedenleri ile kullanılan dile bağlı komut işleme hızının da önemi çok büyüktür. Bu nedenle Java'nın bu açıdan C/C++ dili ile karşılaştırmasının da önemli olduğunu düşünmekteyiz. Bu kapsamda yapılan başarım testleri Whetstone ve Dhrystone testleri [8] ile "CaffeineMark 3.0 Test Suite" [9] olmak üzere iki grupta gerçeklenmiştir. Her test adımı farklı optimizasyon seviyeleri ile hem C hem de Java kodları kullanılarak farklı çekirdek ve hızdaki cpu1 ve cpu2 kartları üzerinde gerçekleştirilmiştir. Java optimizasyonu için Ahead of Time ile "inline" anahtarı açık olarak C için ise tüm optimizasyon seviyelerinde (-O3, -O2, -O1, -O0) derleme yapılmıştır. Java için yapılan ayarlamalar çalışma hızını arttırırken obje kodunun bellekte kapladığı alanı arttırmaktadır. Hız değerlerinin değerlendirmenin esasını oluşturması sebebi ile obje kod boyutu dikkate alınmamıştır.

4.3.1. Whetstone ve Dhrystone Testleri

Whetstone testi ilk büyük yapay başarım ölçme testidir. Bu test ile kayan noktalı (*İng.* Floating-point) programlama ölçümleri hedeflenmiştir. Test farklı adımlarda çalıştırılan Dizi elemanları (kayar-nokta), Parametre dizisi (kayar-nokta), Koşullu dallanmalar (if then else), Tamsayı aritmetiği (sabit nokta), Trigonometrik fonksiyonlar (sin, cos, etc.), Yordam çağırma (kayar-nokta), Dizi referansları (atama), Standart fonksiyonlar (exp, sqrt, etc.) testlerinden oluşmakta ve bu testler döngü içinde değişken döngü sayıları ile çalıştırılmaktadır. Tüm test döngü içinde, alt testler kendi içinde 100 defa olmak şartıyla, 100 defa çalıştırılır. Bu tüm test bir döngü içinde döngü sayısı her seferinde 10 arttırılacak şekilde çalıştırılır. Testin sonucu bir çevrim için ortalama çalışma süresi ve on test üzerinden ortalama çalışma süresidir. Dhrystone testi tamsayı programlama ölçümlerini hedeflemektedir. Kullanılan yöntemler diğer fonksiyon ve yordamların çağırılması, bazı hesaplamaların yapılması ve sonucun çağırılan yordama döndürülmesi üzerine dayalıdır. Başarım testinin bir yürütümü ana döngünün 100.000 döngü ile çalıştırılmasını içerir. Testin sonucu bir döngünün çalışması için gereken ortalama zamandır.

4.3.2. CaffeineMark Java Başarım Testleri

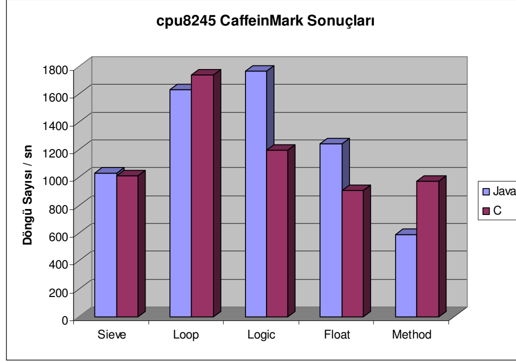
CaffeinMark, farklı yazılım ve donanım yapılandırmalarında, Java programlarının çalışma hızlarının ölçülmesini sağlayan bir dizi testten oluşmaktadır. CaffeineMark sonuçları kabaca, kullanılan bellek boyutu, disk sürücü hızı ve internet bağlantısına belirleyici bir şekilde bağlı olmadan, saniyede çalıştırılan komut sayısı ile alakalıdır. CaffeineMark JVM başarımının farklı yönlerini ölçmek için 9 farklı test kullanmaktadır. Bu testlerden aşağıda verilen 5 tanesi gömülü platformlarda da kullanılabilmektedir. Test sonuçları birim zamanda çalıştırılma sayısı ile ifade edilmektedir. **Sieve Testi,** "Sieve of Eratosthenes" metodu ile asal sayıları bulan test programıdır. **Loop Testi,** derleyici döngü eniyilemesini test etmek için, sıralama ve dizi oluşturma yöntemlerini kullanır. **Logic Testi,** JVM'in karar verme komutlarını hangi hızda çalıştırdığını ölçmek için kullanılmıştır. **Method Testi,** JVM' in yordam çağırma işini hangi hızda yaptığını ölçmek için özyineli (*İng.* recursive) fonksiyon çağırımını kullanır. **Float Testi,** nesnelerin bir nokta etrafında 3 boyutlu dönüşünün benzetimini yapmaktadır. Dhrystone ve Whetstone testleri ile elde edilen sonuçlarda C kodlarının çok daha hızlı olduğunu



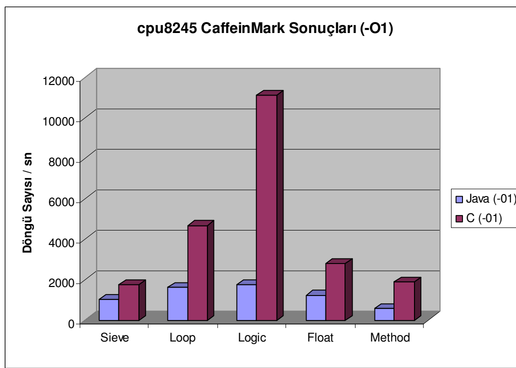
Şekil 11 Whetstone & Dhrystone Test Sonuçları

görmekteyiz (Şekil 11). Şekil 12 de verilen sonuçlar incelendiğinde C için optimizasyon kullanıldığında başarımın

Java' dan çok daha üstün olduğu görülmektedir. Bu testlerde kullanılan Java objesinin C için optimizasyon kullanılmadığı durumları ile karşılaştırıldığında Logic ve Float testlerinde daha iyi sonuç verdiği görülmektedir (Şekil 13).



Şekil 12 cpu8245 CaffeineMark test sonuçları - 1



Şekil 13 cpu8245 CaffeineMark test sonuçları – 2

Standart Java'nın çalışma zamanı komut işleme hızı C dili ile kıyaslanamayacak kadar yavaş olabilmektedir. Fakat yaptığımız testlerde aldığımız sonuçlara göre gerçek zamanlı Java ile bu fark kıyas yapılabilecek boyutlara gelmiştir. Hala Java'nın bazı durumlarda geri kaldığı görülmektedir. Fakat bu gerilik çok küçük boyutlara inmiştir. Ayrıca test için kullandığımız RT-JVM gevşek gerçek zamanlı sistemler için önerilen bir üründür. Daha yüksek performanslı ürünler ise henüz ticari olarak piyasaya çıkmadığı ve deneme sürümleri de olmadığı için temin edilememiş ve bu nedenle değerlendirilememiştir.

Gerçek zamanlılık kriterleri açısından ise RTSJ'nin kavramsal anlamda tüm gerçek zamanlılık gereksinimlerini kabaca karşıladığı görülmüştür. Ticari firmalar tarafından yapılan RTSJ gerçeklemelerinde ise RTSJ'nin geriye dönük desteğinden ödün verilerek çok daha sağlam gerçek zamanlı sistemler oluşturulabilindiği görülmektedir. Örneğin bazı ürünlerde RT-JVM tam RTSJ desteği ile çalışabileceği gibi bazı parametrik ayarlamalarla RTSJ standardı gevşetilerek sıkı gerçek zamanlı sistemler için eniyileştirilebilmektedir.

5. Sonuçlar

Standart Java'nın çalışma zamanı komut işleme hızı C dili ile kıyaslanamayacak kadar yavaş olabilmektedir. Fakat yaptığımız testlerde aldığımız sonuçlara göre gerçek zamanlı Java ile bu fark kıyas yapılabilecek boyutlara gelmiştir. Bu durumda bile elde edilen performans Java'nın getirdiği faydalar ile beraber değerlendirildiğinde C'ye göre tercih sebebi olabilir. Fakat bu performansı yakalamak için Java'nın binary taşınabilirlik özelliğinden ödün verildiği de bilinmelidir.

Gerçek zamanlılık ölçütü açısından ise RTSJ'nin kavramsal anlamda tüm gerçek zamanlılık gereksinimlerini kabaca karşıladığı görülmüştür. Ticari firmalar tarafından yapılan RTSJ gerçeklemelerinde ise RTSJ'nin geriye dönük desteğinden ödün verilerek çok daha sağlam gerçek zamanlı sistemler oluşturulabildiği görülmektedir.

Sonuç olarak standart Java gerçek zamanlı uygulamalar için uygun değildir. Gömülü uygulamalarda ise basitleştirilmiş ve küçültülmüş Java kullanılabilir. RTSJ ve RTJ çalışmaları ile Java'nın gerçek zamanlı uygulamalarda kullanımını engelleyen kısıtlarının tümü karşılanmıştır. Fakat henüz bu gereksinim özelliklerini tam olarak karşılayan yüksek başarım değerlerine sahip uygulamalar olgunlaşmamıştır. Gevşek gerçek zamanlılık gereklerini ise karşılayabilecek Java gerçeklemeleri hazır olarak bulunabilir.

6. Kısaltmalar

GC	Garbage Collection
JME	Java Micro Edition
JRE	Java Runtime Environment
JSDK	Java Standart Development Kit
NIST	National Institute of Science and Technology
PC	Personal Computer
RAM	Random Access Memory
RT	Real Time
RTJ	Real Time Java
RT-JVM	Real Time Java Virtual Machine
RTOS	Real Time Operating System
RTSJ	Real Time Specification for Java
SDK	Standart Development Kit

7. Kaynakça

[1] Sun Microsystems, "The Real Time Java Platform", Haziran 2004.

[2] James J.Hunt, Fridtjot B.Siebert, "Realtime Programming in Java", Aicas GmbH, 2006.

[3] NIST Special Publication, Requirements for Real-time Extensions for the Java Platform: Report from the Requirement Group for Real-time, http://www.nist.gov/itl/div897/ctg/real-time/

[4] J Consortium Home Page, http://www.j-consortium.org/

[5] Ritzau, T., "Memory Efficient Hard Real-Time Garbage Collection", Phd. Tezi, Linkoping Studies in Science and Technology, 2003.

[6] Dibble,P., "Ramblings on Garbage Collection – An RTSJ Perspective", Nisan 2005.

[7] Cechticky,V.,P.A., S.W., "A new approach to software development for embedded control systems"

[8] Marcus Weiskirchner, "Comparison of the Execution Times of Ada, C and Java", EADS Deutschland GmbH Military Aircraft, Eylül 2003

[9] Pendragon Software Corporation, http://www.benchmarkhq.ru/cm30/

[10] Brosgol,B, A.W, "A Comparision of Ada and Real Time Java for Safety Critical Applications", Ada Europe 2006, Haziran 2006.