Saydam Artıklı Çalıştırma İçin Vekil Tasarım Örüntüsü Kullanımı

Dindar Öz[1], Sinan Öz[2], and Işıl Öz[3]

[1] Yazılım Mühendisliği Bölümü, Yaşar Üniversitesi, İzmir, Türkiye
dindar.oz@yasar.edu.tr
[2] MovieStarPlanet ApS, Kopenhag, Danimarka sinanoz1980@gmail.com
[3] Bilgisayar Mühendisliği Bölümü, İzmir Yüksek Teknoloji Enstitüsü, İzmir, Türkiye
isiloz@iyte.edu.tr

Abstract. Bu çalışmada, nesneye yönelik programların güvenilir bir şekilde çalıştırılması için saydam bir model önermekteyiz. Geçici donanım hatalarına karşı istenen seviyede güvenilirliği sağlayabilmek amacıyla artıklı (redundant) program çalıştırması için genel bir nesneye yönelik programlama aracı tasarladık. Bunun için yazılım sistemlerini esnek ve kolay sürdürülebilir yapabilmek için oluşturulmuş ve yaygınca kullanılan GoF tasarım örüntülerinden biri olan vekil tasarım örüntüsünü kullandık. Vekil tasarım örüntüsü, var olan bir nesneye erişirken ona yeni fonksiyonellikler eklemeye yarayan saydam bir düzenek ve kontrollü bir erişim sağlamaktadır. Java programlama dilindeki dinamik vekil ve *annotation* araçlarını birleştirerek, artıklı çalıştırma ve çoğunluk oylaması için genel, saydam ve yapılandırılabilir bir araç olan *RedundantCaller*'ı sunmaktayız. Aracımız, herhangi bir nesneyi alır ve özgün kullanıcı koduna en az miktarda değişiklik gerektirerek nesnenin metotlarını farklı iş parçacıklarında çoklu miktarda çalıştıran ve arka planda çoğunluk oylaması yapan bir dinamik vekil yaratır. *annotation*lar sayesinde, kullanıcılar artıklı çalıştırmayı metot seviyesinde yapılandırabilirler. Deneylerimiz göstermektedir ki; aracımız herhangi bir nesneye yönelik program için çok iş parçacıklı çalıştırma sayesinde makul bir performans düşüşüyle kayda değer bir güvenilirlik seviyesi sağlamaktadır.

Keywords: Nesneye yönelik programlama · Vekil tasarım örüntüsü · Artıklı çalıştırma.

Using Proxy Design Pattern for Transparent Redundant Execution

Dindar Öz[1], Sinan Öz[2], and Işıl Öz[3]

[1] Software Engineering Department, Yaşar University, İzmir, Turkey
dindar.oz@yasar.edu.tr
[2] MovieStarPlanet ApS, Copenhagen, Denmark sinanoz1980@gmail.com
[3] Computer Engineering Department, İzmir Institute of Technology, İzmir, Turkey
isiloz@iyte.edu.tr

Abstract. In this study, we propose a transparent model for reliable execution of object-oriented software. We design a generic object-oriented programming tool for redundant software execution to provide the desired level of reliability against transient hardware faults. To achieve this, we utilize the Proxy design pattern which is one of the well-known GoF design patterns that are formed to make software systems flexible and easy to maintain. Proxy design pattern provides a controlled access and a transparent mechanism for adding new functionalities to an existing object when accessing it. Combining the instruments of dynamic proxy and annotations in Java programming language, we present, *RedundantCaller*, a generic, transparent, and configurable tool for redundant execution and majority voting. Our tool takes any object and creates a dynamic proxy for it which executes the methods of the object multiple times in separate threads, and performs majority voting on the background, requiring minimum amount of change in the original user code. Thanks to annotations, users can configure the redundant execution scheme methodwise. Our experiments demonstrate that our tool provides a significant level of reliability to any object-oriented software with a reasonable amount of performance degradation through multithreaded execution.

Keywords: Object-oriented programming · Proxy design pattern · Redundant execution.

1 Giriş

Bilgisayarlar ve üzerinde çalışan yazılımlar gün geçtikçe hayatın her alanında yer alırken, bilgisayarların güvenilir çalışması önem kazanmaktadır. Askeri operasyonlar, sağlık, uzay araştırmaları, finansal işlemler gibi bazı uygulama alanlarında bu sistemlerin işleyişlerinde meydana gelebilecek en küçük bir hatanın bile kabul edilemez sonuçları olabilmektedir. Bu anlamda bilgisayar sistemlerini daha güvenilir kılabilmek için uygulanabilecek yöntemler bilgisayar mühendisliğinin önemli araştırma alanlarından biri olmuştur [1]. Artıklı sistemler bu yöntemlerin başlıcaları arasındadır [2, 3]. Sistemin ya da sistemin işleyişinin gerekenden fazla

şekilde çoklanması temeline dayanan bu yöntem, donanımsal ya da yazılımsal olarak uygulanabilmektedir. Donanımsal artıklı sistemler, çoklanan donanım bileşeninin arıza yapması ve hatalı çalışması ihtimaline karşı uygulanmaktadır. Bu sistemlerde önlem alınmaya çalışılan hata kalıcı hatalardır. Bir takım donanım hataları ise geçici olarak adlandırılmakta ve sadece belirli bir zamanda oluşup daha sonra kaybolmaktadır. Bu hatalara karşı sıklıkla maliyeti daha az olan artıklı yürütme yöntemi kullanılmaktadır. Bu yöntem temelde, yapılmak istenen işlemin birden fazla defa yapılıp elde edilen sonuçların oy çokluğu prensibine göre oylanarak işlemin beklenen sonucu olarak kabul edilmesidir. Yöntemin ana fikri, aynı geçici hatanın arka arkaya ve aynı hatalı sonucu vererek oluşması ihtimalinin ihmal edilebilir düzeyde olmasıdır. Özellikle artıklı ve çoklu iş parçacıklı (redundant multithreading) yöntemler, hataların farklı çekirdeklerde çalışan kopyalarda eş zamanlı gerçekleşme ihtimalinin daha da düşük olması sebebiyle yüksek hata toleransı sağlamakta, paralel çalıştırma sayesinde artıklı çalıştırmanın performans üzerindeki olumsuz etkisinin önüne geçilmektedir [5, 6].

Artıklı yürütme yönteminin uygulanması yazılım seviyesinde yer alabilmekte ve yazılım geliştiricilere önemli bir miktarda gerçekleme yükü doğurmaktadır. Artıklı çalıştırılacak bütün işlemlerin bu şekilde gerçeklenmesi ve gerekli oylama işlemlerinin kodlanması gerekmektedir. Bu çalışmada, bu gerçekleme yükünü ortadan kaldıracak genel, yeniden kullanılır bir programlama aracı geliştirmeyi hedefledik. Bu amaçla nesneye yönelik bir programlama ortamında GoF tasarım örüntülerinden [10] biri olan vekil tasarım örüntüsünü kullanarak söz konusu gerçekleme yükünün herhangi bir nesne için otomatik ve saydam olarak yapıldığı, *RedundantCaller* (artıklı çalıştırıcı) adını verdiğimiz bir programlama aracı (sınıf) tasarladık.

Bölüm 2'de bu alandaki ilgili çalışmalar üzerinden kısaca geçilmekte ve bu bölümü vekil tasarım örüntüsünün özetlendiği Bölüm 3 izlemektedir. *RedundantCaller* aracımızın işleyişinin yeraldığı Bölüm 4'ün ardından aracın neden olduğu performans düşüşü ve güvenilirlik artışının incelendiği deneysel çalışmaların bulunduğu Bölüm 5 gelmektedir. Bölüm 6 ile makale sonlanmaktadır.

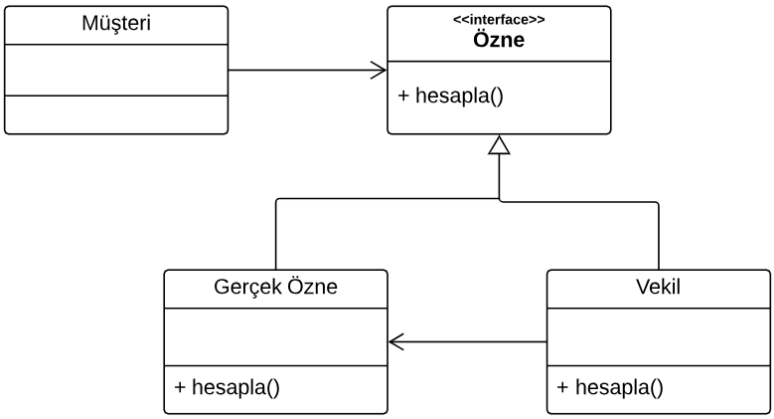
2 İlgili Çalışmalar

Literatürde yazılım tabanlı artıklı çalıştırma için çok sayıda çalışma yapılmıştır. Bu bölümde bu konulardaki çalışmalara yer verilmiştir.

SRMT [7], verilen program kodunu iki iş parçacığına çoklayarak hata tespiti sağlamaktadır. Paralel çalışan iş parçacıkları, komutların sonuçlarını karşılaştırarak uyuşmazlık durumunda hata durumu olduğunu tespit etmektedir. Komut seviyesinde sonuç kontrolü sağlandığından performansa etkisi oldukça fazladır, o yüzden bunu iyileştirmek için farklı iş parçacıklarının iletişimini sağlayan optimizasyonlar da önerilmiştir. Detaylı bir derleyici ve yürütme sistemi gerçeklemesi içeren bu sistem, sadece hata tespiti sağlamakta olup hata toleransı seviyesinin yeniden yapılandırılmasına imkan vermemektedir.

RedThreads [8], C/C++ programları için artıklı çok iş parçacıklı çalıştırma sağlayan bir arayüz sağlamaktadır. Yazılım geliştirici, arayüz tarafından tanımlanan direktiflerle (directives) programın hangi parçasının, hangi seviyede (kaç iş parçacığıyla) artıklı çalışacağını belirtebilmektedir. Çalışmada hem derleyici, hem de yürütme sistemi gerçeklemesi yapılmış, kullanıcının belirlediği özelliklere göre artıklı çalıştırma sağlanmıştır. Kullanım kolaylığı açısından *RedundantCaller* aracımıza benzemekle birlikte C/C++ programlarına yönelik olması ve çok fazla gerçekleme içermesi yönleriyle bizim çalışmamızdan farklılık göstermektedir.

Chen ve Chen [9] yazılım tabanlı artıklı çalıştırma için bir programlama modeli önermişlerdir. Çalışmalarındaki programlama modeli, hata toleransını gerçeklemek için çok iş parçacıklı teknikleri kullanmaktadır. Hata düzeltmeyi de çoğunluk oylamasıyla gerçekleştirmektedir. Ayrıca sistemin cevap vermediği hata durumları için watchdog zamanlayıcısı kullanılmaktadır. Yazılım geliştiricinin, yazılımın gerekli yerlerine müdahale etmesi ve programlama modelinin sunduğu fonksiyonları eklemesi gerekmektedir. Yazılım geliştiriciye saydam bir araç olarak sunulan *RedundantCaller* aracını kullanmak için, hata toleransı eklenmek istenen yazılımda neredeyse hiçbir değişiklik yapılmasına gerek duyulmamaktadır.



Şekil 1. Vekil tasarım örüntüsünün UML sınıf şeması.

3 Vekil Tasarım Örüntüsü

Tasarım örüntüleri nesneye yönelik programlamada çok karşılaşılan tasarım problemlerine çözüm olarak sunulmuş yeniden kullanılır, esnek, ve doğrulanmış çözüm önerileridir [10]. Vekil tasarım örüntüsü Eric Gamma ve arkadaşlarının önerdiği 23 iyi bilinen tasarım örüntüsünden biridir ve şu iki temel probleme çözüm sunmayı hedeflemektedir:

– Nesneye erişimi kontrol altına almak
– Nesneye saydam bir şekilde yeni işlevler kazandırmak

Temelde hedef olarak seçilen sınıf ile aynı arayüze sahip bir vekil sınıfının tanımlanması ve vekil sınıfın hedef sınıfı içinde bulundurması (composition) fikrine dayanır. Hedef sınıf kullanıcıları bu sınıfa vekil sınıf üzerinden erişim sağlarlar. Şekil 1'de vekil tasarım örüntüsünün UML sınıf şeması görüntülenmektedir [10]. Vekil tasarım örüntüsü farklı gerçekleme senaryoları ile yazılım geliştirme alanlarında uygulanmaktadır [11, 12].

4 Vekil Tasarım Örüntüsü Temelli Hata Toleransı

Bu çalışmamızda vekil tasarım örüntüsü temelli *RedundantCaller* ismini verdiğimiz saydam bir artıklı çalıştırma aracı geliştirdik (https://github.com/isil-oz/RedundantCaller). Bunu yaparken Java programlama dilindeki dinamik vekil (dynamic proxy) mekanizmasını kullandık. Bu mekanizma bir arayüz (interface) ile erişilen herhangi bir sınıfa çalıştırma zamanında otomatik olarak bir vekil sınıf oluşturmak için kullanılmaktadır. *RedundantCaller*, verilen herhangi bir sınıfın nesnesi için bu sınıfın metotlarını artıklı bir şekilde çalıştıracak vekil nesneyi otomatik olarak oluşturur ve geliştirici bu vekil nesneyi kullanarak gerçeklemek istediği kodu yazar. *RedundantCaller* hedef nesnenin artıklı çalıştırılacak bir metodu çağrıldığında öncelikle o nesnenin artıklı çağırma sayısınca klonunu üretir. Bu işlem *RedundantCaller* kullanımının getirdiği bir gereksinim olmayıp çağrılan metot nesne üzerinde değişiklik yapıyor ise sonraki metotların tutarlı ve doğru çalışması için gereklidir ve artıklı çalıştırma işleminin doğası gereği yapılmaktadır. Daha sonra, artıklı çalıştırma çok iş parçacıklı (multithreaded) olarak yapılandırılmış ise oluşturulan hedef nesne klonlarını içeren birer iş parçacığı üretilir ve Java programlama dilinde eşzamanlı çalıştırma (concurrent execution) için yaygınca kullanılan Fork-Join mekanizması ile bu iş parçacıkları paralel olarak çalıştırılır. Tek iş parçacıklı yapılandırma senaryosunda ise bu işlem bir döngü içerisinde klon nesnelerin metotlarının arka arkaya çalıştırılması şeklinde gerçekleştirilir. Her iki senaryoda da artıklı çalıştırma neticesinde elde edilen sonuçlar çoğunluk oylaması yöntemi ile oylanır ve kazanan sonuç işlemin beklenen sonucu olarak döndürülür. Burada özetlenen bütün bu işlemler geliştiricinin herhangi bir gerçekleme yapmasını gerektirmeden *RedundantCaller* nesnesi tarafından saydam bir şekilde gerçekleştirilir. *RedundantCaller* sınıfında yukarıda özetlenen işleri gerçekleştiren kod parçacığı Kod 1'de listelenmektedir.

Ayrıca geliştirici, vekil nesne ile erişmek istediği nesnenin arayüz tanımında artıklı çalıştırmak istediği metotları ve bunların artıklı çalıştırma parametrelerini belirleyebilir. Metot bazında yapabildiği bu yapılandırma imkanı için yine Java programlama dilindeki annotation mekanizması kullanılmıştır. Artıklı çalıştırma yapılandırma parametreleri ihtiyaca göre kolayca arttırılabilmekle beraber bu çalışmada şu parametrelere yer verilmiştir:

– **votecount** : Tamsayı. Metodun kaç defa çalıştırılacağını belirtir.

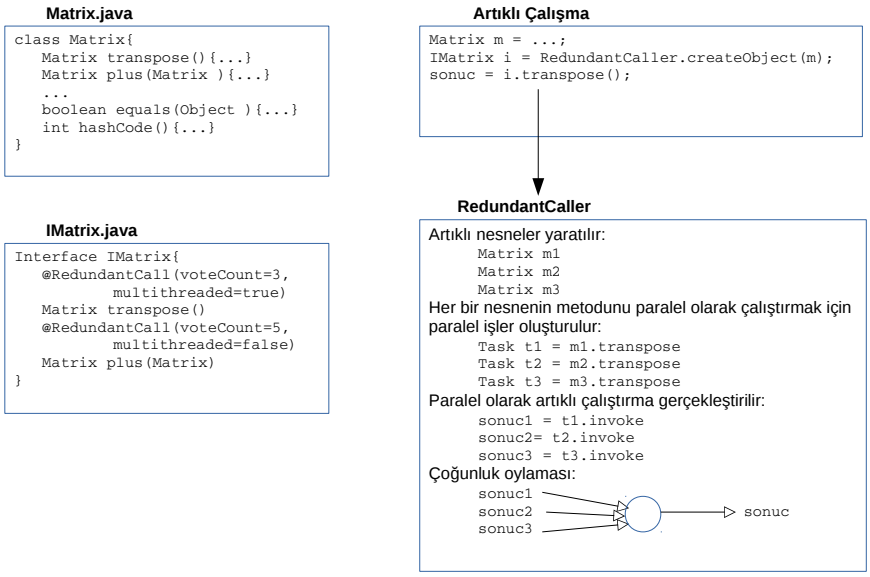
– **multithreaded** : true/false. Artıklı çalıştırma işleminin çok iş parçacıklı çalıştırılıp çalıştırılmayacağını belirtir.

Kod 1. *RedundantCaller* artıklı çalıştırma kodu

```
public Object invoke(Object proxy, Method method, Object[] args)
    throws Throwable {
    /* some implementation details*/

    if (method.isAnnotationPresent(RedundantCall.class))
    {
        int voteCount = method.getAnnotation(RedundantCall.class).
                        voteCount();
        buildTargetCopies(voteCount);

        boolean multithreaded =  method.getAnnotation(RedundantCall.class)
                                                .multithreaded();
        if (multithreaded) {
            List<Future<Object>> list =
                        redundantRun(method, args,voteCount);
            return threadedVoting(list);
        }
        else return singleThreadedVoting(method, args,voteCount);
    }
    else return method.invoke(target,args);
}
```



Şekil 2. *RedundantCaller*'ın kullanımı.

RedundantCaller aracımızın kullanım örneği Şekil 2'de bir nesne üzerinde gösterilmektedir. Örnekte kullanılmak istenen sınıf Matrix sınıfı ve bu sınıfa erişim için tanımlanan arayüz IMatrix arayüzüdür. Bu arayüz içinde tanımlanan tüm metotlar için artıklı çalıştırma imkanı sağlanacaktır. Metotlar için belirtilen annotation aracılığıyla da artıklı çalıştırma yapılandırılabilmektedir. Şekildeki örnekte *transpose* metodu için tanımlanan @RedundantCall(voteCount = 3 ,multithreaded = true) annotation'ı ile, *transpose* metodunu 3 iş parçacığıyla artıklı olarak çalıştırmak istediğimiz belirtilmiştir. *plus* metodu için tanımlanmış olan @RedundantCall(voteCount = 5 ,multithreaded = false) annotation'ıyla ise 5 artıklı çalıştırmanın tek bir iş parçacığıyla gerçekleştirilmesi istenmiştir.

5 Deneysel Çalışma

RedundantCaller aracının nesneye yönelik uygulamaların performans ve güvenilirlik üzerindeki etkilerini gözlemlemek için gerçekleştirdiğimiz deneyler bu bölümde yer almaktadır.

Deneylerimizi temel lineer cebir işlemleri içeren bir paket olan JAMA kütüphanesini kullanarak gerçekleştirdik [13]. JAMA, *Matrix* ana sınıfı altında farklı operasyonlar sunmaktadır. Bu kütüphanenin kullanımı, hata hassasiyeti ve hata toleransı çalışmalarında [14, 15] sıklıkla kullanılan matris hesaplamaları içerdiğinden çalışmamız için oldukça uygun olmuştur. Deneylerimizde 1000x1000 boyutunda rasgele değerlere sahip matrisler üreterek Tablo 1'te listelenen temel 31 operasyonu kullandık.

Artıklı çalıştırmanın çok iş parçacıklı versiyonlarını çalıştırabilmek için Intel Xeon E5-2680 temelli çok çekirdekli mimaride deneylerimizi çalıştırdık.

Öncelikle *RedundantCaller*'ın performansa olan etkisini gözlemleyebilmek için deneyler gerçekleştirdik. Bu deneylerde matris operasyonlarının artıklı çalıştırma olmadan tamamlandığı zamanı, ve artıklı çalıştırmanın tek iş parçacıklı, 3 iş parçacıklı ve 5 iş parçacıklı versiyonlarının tamamlandığı zamanları ölçtük. Çalıştırmalar arasındaki eşitsizliklerden kaynaklanabilecek olası yanıltıcı sonuçlardan etkilenmemek için her bir operasyonu 20 defa çalıştırıp en yüksek olan bir değeri dışarıda tutarak ortalamalarını hesapladık, ve her bir operasyonun her bir versiyon için çalıştırma zamanını belirledik. Operasyonlar farklı çalıştırma zamanlarına sahip olduğundan ve hepsini tek bir şekilde göstermek zor olduğundan operasyonları çalıştırma zamanlarına göre üç gruba ayırarak farklı şekillerde performans sonuçlarını raporladık. Şekil 3 farklı konfigürasyonlar için operasyonların çalıştırma zamanlarını göstermektedir. Bu şekilde performans sonuçları verilen farklı konfigürasyonlar ve açıklamaları ayrıca Tablo 2'de verilmiştir.

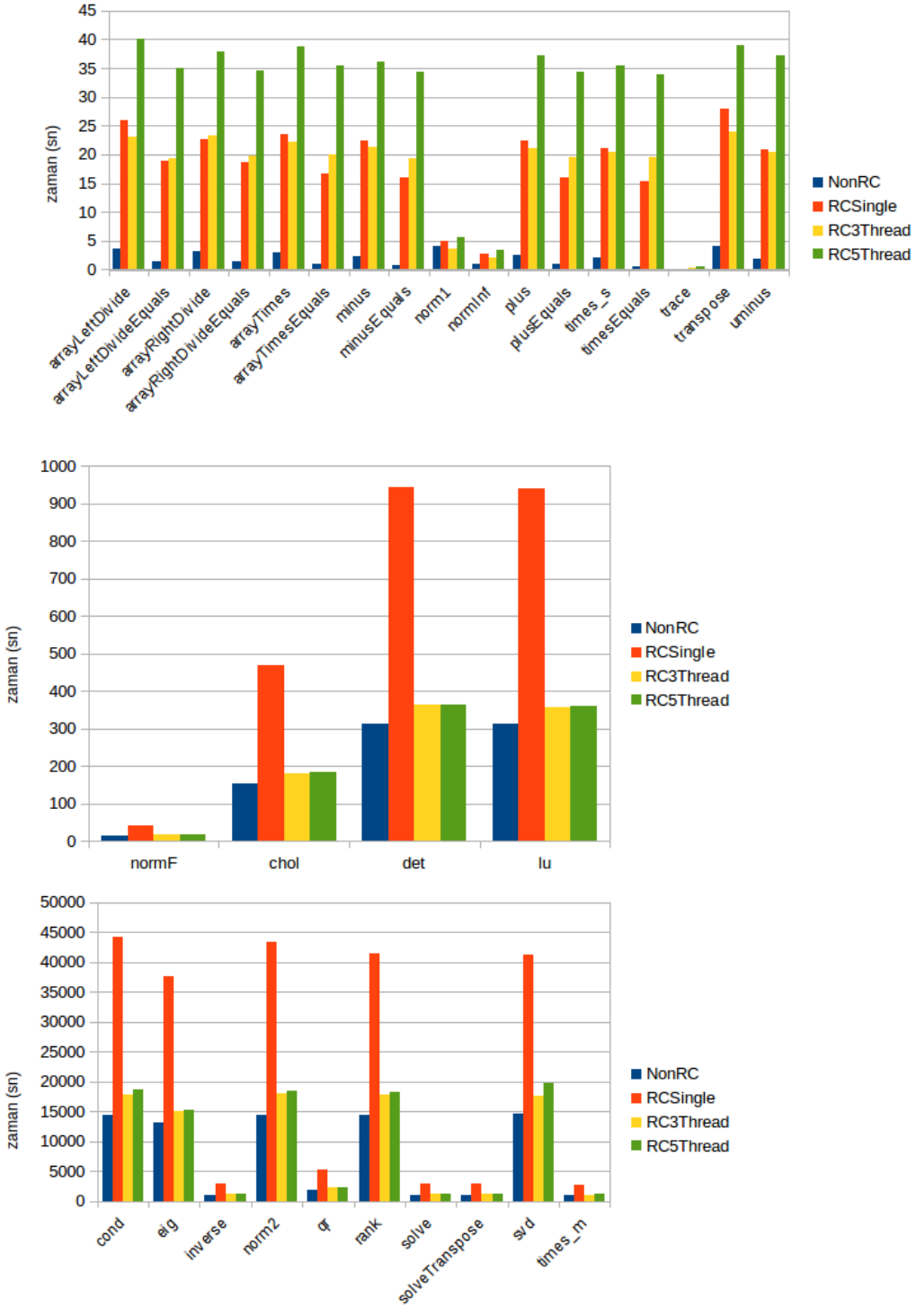
Şekil 3'de görüldüğü gibi çalıştırma zamanı küçük olan (normal çalıştırma zamanı 5 saniyeden düşük) operasyonlar için artıklı çalıştırma maliyeti oransal olarak çok yüksekken, tek iş parçacıklı (RCSingle) ve üç iş parçacıklı (RC3Thread) üç artıklı çalıştırma konfigürasyonları benzer sonuçlar vermektedir. Bu durumlarda beş iş parçacıklı, beş artıklı çalıştırma (RC5Thread) konfigürasyonunun maliyeti 10-20 kata kadar çıkmaktadır. Bu gruptaki operasyon-

Tablo 1. Deneylerimizde kullandığımız matris operasyonları.

Operasyon Adı	Operasyon
arrayLeftDivide	Soldan bölme (C = A.\B)
arrayLeftDivideEquals	Soldan bölüp kendine eşitleme (A = A.\B)
arrayRightDivide	Sağdan bölme (C = A./B)
arrayRightDivideEquals	Sağdan bölüp kendine eşitleme (A = A./B)
arrayTimes	Çarpma (C = A.*B)
arrayTimesEquals	Çarpıp kendine eşitleme (A = A.*B)
chol	Cholesky Decomposition
cond	Matrix condition
det	Matris determinant
eig	Eigenvalue Decomposition
inverse	Ters
lu	LU Decomposition
minus	Çıkartma (C = A - B)
minusEquals	Çıkartıp kendine eşitleme (A = A - B)
norm1	1 norm (En büyük sütun toplamı)
norm2	2 norm (En büyük tekil değer)
normF	Frobenius norm (Elemanların karelerinin toplamının karakökü)
normInf	Sonsuz norm (En büyük satır toplamı)
plus	Toplam (C = A + B)
plusEquals	Toplayıp kendine eşitleme (A = A + B)
qr	QR Decomposition
rank	Matris rank
solve	A*X = B çözümü
solveTranspose	X*A = B ve ayrıca A'*X' = B' çözümü
svd	Singular Value Decomposition
times_s	Skalar ile çarpım (C = s*A)
times_m	Lineer cebirsel matris çarpımı (A * B)
timesEquals	Skalar ile çarpıp kendine eşitleme (A = s*A)
trace	512, 1024, 2048
transpose	Matris trace (Diyagonal elemanların toplamı)
uminus	Negatifini alma (-A)

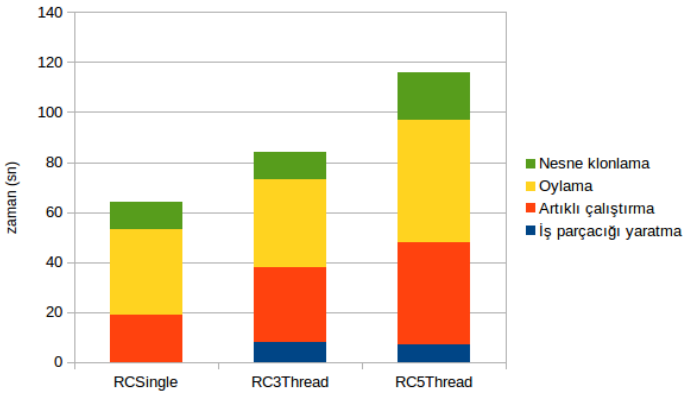
Tablo 2. Karşılaştırma yaptığımız konfigürasyonlar.

Konfigürasyon Adı	Konfigürasyon
NonRC	Normal çalıştırma (Artıklı çalıştırma yok)
RCSingle	Tek iş parçacıklı, üç artıklı çalıştırma
RC3Thread	Üç iş parçacıklı, üç artıklı çalıştırma
RC5Thread	Beş iş parçacıklı, beş artıklı çalıştırma



Şekil 3. Farklı artıklı çalıştırma konfigürasyonları için çalıştırma zamanları.

ların çalıştırma zamanlarının çok kısa olmasından dolayı; artıklı çalıştırmadaki nesnelerin klonlanması, çoğunluk oylaması gibi fazladan çalıştırma haricindeki işlemlerin zamanının normal çalıştırma zamanının çok fazla üstüne çıkmasına sebep olmaktadır. Özellikle paralel versiyonlarda iş parçacıklarının yaratılması ve yönetilmesi işlerinin de eklenmesiyle çalıştırma zamanları göreceli olarak çok artmaktadır. Örneğin, çalıştırılması 2 saniye süren bir operasyonu üç iş parçacıklı, üç artıklı çalıştırma ile çalıştırdığımızı düşünelim. Bir iş parçacığı yaratma işlemi 0.5 saniye sürüyorsa, sadece iş parçacığı yaratma işleminin eklenmesiyle operasyonun çalıştırma zamanı yaklaşık 2 katına çıkmaktadır. Bu durumu gözlemleyebilmek için artıklı çalıştırma konfigürasyonları için işlem seviyesinde zaman ölçümleri yaptık. Şekil 4, normal çalışması 3-4 saniye süren *arrayLeftDivide* operasyonunun çalıştırma zamanının hangi işlemlerde geçtiğini göstermektedir. Diğer operasyonlar için de benzer durumlar söz konusudur. Normal çalışması 1000-10000 saniye süren bir uygulama için artıklı çalıştırmanın fazladan çalıştırma haricindeki işlemleri görece çok kısa olduğundan, artıklı çalıştırmanın toplam maliyeti oldukça düşük olmaktadır. Tek iş parçacıklı konfigürasyonda (RCSingle), üç artıklı çalıştırma tek bir iş parçacığı tarafından arka arkaya yapıldığı için onun performansı en kötüyken; paralel konfigürasyonlarda (RC3Thread ve RC5Thread) üç-beş iş parçacığı, artıklı çalıştırmaları farklı çekirdeklerde eş zamanlı olarak gerçekleştirdiğinden toplam çalıştırma zamanları ciddi artış göstermemektedir (Şekil 3). Uzun hesaplamalarda fazladan çalıştırma haricindeki işlemlerin maliyeti görünmemekte ve asıl büyük hesaplama iş parçacıklarına yaptırıldığı için maliyet gizli kalmış olmaktadır.



Şekil 4. Farklı artıklı çalıştırma konfigürasyonları için *arrayLeftDivide* operasyonunun çalıştırma zamanının dağılımı.

RedundantCaller'ın performansının yanı sıra hata kapsamını gözlemleyebilmek için hata enjeksiyonu deneyleri gerçekleştirdik. Bunun için deneylerde kullandığımız sınıf (Matrix) için operasyon seviyesinde belirli bir ihtimalle hesaplama hata durumları oluşturduk, normal çalıştırmanın

ve artıklı çalıştırmanın hesapladığı değerin hesaplaması gereken değere eşit olup olmadığını kontrol ederek sessiz veri bozulumu (silent data corruption) durumlarını tespit etmeye çalıştık. Hata enjeksiyonu deneyleri uzun zaman aldığından nispeten kısa çalıştırma zamanlarına sahip *transpose*, *norm1*, *norm2*, ve *uminus* operasyonlarına hata enjeksiyonu yaptık. *RedundantCaller* aracı açısından farklılık oluşturmadığı için bu operasyon alt kümesinin yeterli olduğunu düşünmekteyiz. Her bir operasyon için 1000 farklı çalıştırma yaparak normal çalıştırmanın ve artıklı çalıştırmanın hatalı hesaplama durumlarını gözlemlemeye çalıştık. İlk olarak literatürde de olası hata oranı olarak tespit edilen 0.001 değeriyle hata enjeksiyon testleri çalıştırdık. Bu hata değeriyle normal çalıştırma için 0.002 (4000 çalıştırmada 8 hatalı veri hesaplama) veri bozulumu oranı gözlemlerken üç artıklı çalıştırmada herhangi bir veri bozulması ile karşılaşmadık. Hata oranını 0.01, 0.1, 0.2 gibi daha az gerçekçi değerlere değiştirdiğimizde normal çalıştırmanın veri bozulumu oranı artış gösterirken, üç artıklı çalıştırmada veri bozulumu oranı sıfırda kalarak %100 hata kapsamı gözlemlenmiştir. Hata analizi deneylerimiz, çoğunluk oylaması temelli hata toleransı sistemlerindeki beklenen hata kapsamı seviyesini doğrulamıştır. Deney sonuçlarımız, aracımızın tanımlanan hata modelindeki hata kapsamı performansını göstermektedir.

6 Sonuç

Bilgisayarların ve üzerinde yürütülen yazılımların hatasız ve güvenilir çalışması, çoğu zaman performans ve maliyet gibi diğer birçok kriterin önüne geçmekte ve bilgi sistemlerinin tasarım kararlarında önemli bir rol oynamaktadır. Geliştiriciler yazılımları oluştururken çözmek istedikleri orjinal problemin yanı sıra, yazılımın daha güvenilir olması için almaları gereken tedbirleri de göz önünde bulundurmak durumundadırlar. Bu çalışmada nesne yönelimli programlamada yaygınca kullanılan vekil tasarım örüntüsünü kullanarak geliştiricilere güvenilir yazılım mekanizmalarından biri olan artıklı çalıştırma tekniğini esnek, saydam ve otomatik bir şekilde sunan bir geliştirme aracı oluşturduk. Vekil tasarım örüntüsünün diğer kullanım alanlarının yanında bu alanda da etkin bir çözüm olarak kullanılabileceğini göstermiş olduk. Java programlama dilinin sunduğu dinamik vekil (dynamic proxy) ve annotation dil öğelerini kullanarak tasarladığımız aracı genel ve kolayca yapılandırılabilir bir şekilde gerçekledik. Yine Java'da yaygınca kullanılan fork-join sistemi ile artıklı çalıştırma işinin isteğe bağlı olarak çok iş parçacıklı olarak yapılmasını sağladık. Örnek bir Java kütüphanesi ile gerçekleştirdiğimiz deneylerde paralel çalıştırmanın sağladığı fayda ile aracımızın küçük bir performans kaybı ile önemli oranlarda güvenilirlik artışı sağladığını gösterdik. Yine deneyler için oluşturduğumuz test kodunda *RedundantCaller* kullanımının mevcut kodu minimum miktarda değiştirdiğini ve geliştiriciye ihmal edilebilir miktarda bir ilave iş yükü getirdiğini gözlemledik. Ayrıca önerdiğimiz aracın benzer programlama öğelerinin (dynamic proxy, annotation) desteklendiği diğer nesneye yönelik programlama dillerinde (C#, C++ vb.) de kolaylıkla gerçeklenebileceğini düşünmekteyiz.

7 Teşekkür

Bu çalışmada kullanılan hesaplama kaynakları Ulusal Yüksek Başarımlı Hesaplama Merkezi'nin (UHeM), 1005202018 numaralı desteğiyle, sağlanmıştır.

Kaynaklar

1. Israel Koren, C. Mani Krishna: Fault-Tolerant Systems, Morgan Kaufmann, (2007).
2. George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I. August: Swift: Software implemented fault tolerance, International Symposium on Code Generation and Optimization, (2005).
3. George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I. August, Shubhendu S. Mukherjee: Design and evaluation of hybrid fault-detection systems. International symposium on Computer Architecture (ISCA), (2005).
4. Alex Shye, Joseph Blomstedt, Tipp Moseley, Vijay Janapa Reddi, Daniel a. Connors: Plr: A software approach to transient fault tolerance for multicore architectures. IEEE Transactions on Dependable and Secure Computing, 6(2):135–148, (2009).
5. S. S. Mukherjee, M. Kontz, S. K. Reinhardt: Detailed design and evaluation of redundant multi-threading alternatives, International Symposium on Computer Architecture (ISCA), (2002).
6. M. Gomaa, C. Scarbrough, T. N. Vijaykumar, I. Pomeranz: Transient-fault recovery for chip multiprocessors, International Symposium on Computer Architecture (ISCA), (2003).
7. Cheng Wang, Ho seop Kim, Youfeng Wu, Victor Ying: Compiler-Managed Software-based Redundant Multi-Threading for Transient Fault Detection,International Symposium on Code Generation and Optimization (CGO), (2007).
8. Saurabh Hukerikar, Keita Teranishi, Pedro C. Diniz, Robert F. Lucas: RedThreads: An Interface for Application-Level Fault Detection/Correction Through Adaptive Redundant Multithreading, International Journal of Parallel Programming, 46:225-251, (2018).
9. Yi-Shen Chen, Peng-Sheng Chen: A Software-Based Redundant Execution Programming Model for Transient Fault Detection and Correction, 45th International Conference on Parallel Processing Workshops (ICPPW), (2016).
10. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns (the "Gang of Four" book), Addison-Wesley, (1994)
11. K. Soundararajan and R.W. Brennan: A Proxy Design Pattern to Support Real-Time Distributed Control System Benchmarking, Holonic and Multi-Agent Systems for Manufacturing (HoloMAS), (2005).
12. K. Soundararajan and R.W. Brennan: Design patterns for real-time distributed control system benchmarking, Robotics and Computer-Integrated Manufacturing, 24, 5:606-615, (2008).
13. JAMA Homepage, https://math.nist.gov/javanumerics/jama/. Last accessed 11 June 2018
14. Greg Bronevetsky and Bronis R. de Supinski: Soft Error Vulnerability of Iterative Linear Algebra Methods, International conference on Supercomputing (ICS), (2008).
15. Konrad Malkowski, Padma Raghavan, Mahmut Kandemir: Analyzing the soft error resilience of linear solvers on multicore multiprocessors, International Symposium on Parallel and Distributed Processing (IPDPS), (2010).