Bağımlılık Örüntüleri ile Nesneye Dayalı Yazılımların Cep Bellek Bilinçli Paralelleştirilmesi

Tolga Ovatman[1] Feza Buzluca[2]

[1,2]Bilgisayar Mühendisliği Bölümü, İstanbul Teknik Üniversitesi, İstanbul
[1]e-posta: ovatman@itu.edu.tr [2] e-posta: buzluca@itu.edu.tr

Özetçe

Çok çekirdekli işlemcilerin kişisel bilgisayar sistemlerinde kullanımlarının artması ile birlikte yazılımların paralel hale getirilmesi konusu tekrar önem kazanmıştır. Günümüzde kullanılan nesneye dayalı yazılımların etkin halde paralelleştirilebilmesi için yeni tekniklere ihtiyaç vardır. Bu yazıda ardışıl olarak geliştirilmiş nesneye dayalı yazılımların UML sınıf çizenekleri kullanılarak paralel çalışır hale getirilmesi üzerinde durulmuştur. Yazılım modeli içerisinde sıkça kendini tekrar eden bağımlılık örüntüleri keşfedilerek bunların paralelleştirme açısından önemi araştırılmıştır. Çalışmaların devamında çok çekirdekli işlemcilerde bulunan çekirdeğe özel cep bellek ve paylaşılan cep belleklerin daha etkin kullanılması amacıyla cep bellek bilinçli bir iş dağıtım yaklaşımı önerilerek yaklaşımın Linux işletim sisteminin iş sıralayıcısı(CFS) üzerine yaptığı gelişmeler sunulmuştur. Bazı tasarım kalıpları ve örnek bir görüntü filtreleme programı üzerinde yapılan denemelerde, bağımlılık kalıplarının da yardımıyla, CFS'nin adil işlemci kullanımını etkilemeden cep belleklerin etkin kullanımının arttırılarak başarımın iyileştirildiği gözlemlenmiştir. Yapılan çalışmalar sonucunda, model tabanlı teknikleri kullanmanın daha yazılım üretilmeden yazılımın paralelleştirilmesi ve iş sıralaması hakkında bilgi kaynağı olabileceğini göstermiştir.

1. Giriş

Günümüzde çok çekirdekli işlemcilerin kullanımının artmasıyla birlikte paralel yazılım geliştirme tekniklerinin kişisel bilgisayarlar için geliştirilen uygulama yazılımları düzeyinde uygulanması ilgi çekici hale gelmiştir. Daha alt düzeyli programlama dillerine yönelik olarak geliştirilmiş OpenMP [1] ve MPI [2] gibi paralel programlama modellerinin bu amaçla kullanımı mümkün olsa da eldeki sıradüzensel yazılımlara paralellik kazandırmak halen büyük bir problem olarak karşımıza çıkmaktadır. Çok çekirdekli işlemcilerde etkin bir biçimde koşabilecek yazılımlar üretmek için kaynak kodunun paralelleştirilmesi gereken bölgelerini bulmak ciddi bir iyileştirme(refactoring) çabası gerektirmektedir.

Son beş yıla kadar üretilen uygulama yazılımları kişisel bilgisayarlarda yaygın olarak bulunan tek işlemcili mimari göz önünde tutularak yapıldığı için genellikle paralellik ihtiva etmemekteydiler. Çok görevli (multitasking) çalışmaya uygunluk sağlamak için ekseriyetle kullanıcı arayüzü gibi etkileşim isteyen kısımlarda iplikler (thread) kullanılmaktaydı. Çok işlemcili sistemlerde etkin çalışmayı sağlayacak paralel yazılım geliştirme teknikleri ise daha ziyade yüksek başarımlı hesaplama işlemlerinde kullanıldığı için bu amaçla geliştirilmiş teknikler daha düşük düzey buyruksal programlama dillerine(imperative programming languages) yönelikti.

Günümüzde çok çekirdekli işlemciler kişisel bilgisayarlara dahil edilmekte ve dolayısıyla paralel işlem de uygulama yazılımlarına uyarlanmak durumunda. Öte yandan kullanıcı etkileşimi, giriş/çıkış işlemleri, veri işleme gibi farklı işlemleri bir arada bulundurabilen uygulama yazılımları için paralel çalışabilecek noktaları ortaya çıkarmak başlı başına bir sorun. Bu sorunu çözmek için yazılıma daha yukardan bakmak ve yazılımın çözdüğü problem içerisinde bulunan paralelliği ortaya çıkarmak gerekmekte. Bu amaçla yazılım modellerinin kullanılması yazılımı anlamak için kaynak kodu kullanımından daha etkin olabilir.

İlk bakışta, yapılacak model analizinin davranışsal modeller üzerinde uygulanması daha akla yatkın olarak gelebilir. Bu amaçla literatürde birçok çalışma yapılmıştır [3,4,5,6,7]. Öte yandan davranışsal modellerin kullanımı iki yönden zordur. İlkin, bu tür modellerin kaynak kodundan otomatik olarak elde edilmesi çoğu zaman yazılımı çalıştırmadan mümkün değildir. İkinci olarak her bir davranışsal model eldeki sistemin sadece tek bir senaryo için çalışmasını temsil eder ve bütün olası senaryolar için davranışsal modellerin elde edilmesi pratikte mümkün değildir. Bu yönden bakıldığında durağan sınıf modellerinin kullanılması, davranışsal modellerin bu eksikliklerini kapatabilir. Durağan sınıf modellerinin kaynak kodundan elde edilmesi çok daha kolaydır ve bu modellerde yazılım bileşenleri arasındaki olası bütün ilişkiler bulunur. Durağan modellerin kullanımının da kendine göre dezavantajları bulunur. Örneğin durağan sınıf modellerinde her ne kadar sınıflar arası tüm ilişkiler bulunsa da bu ilişkiler çalışma zamanında çok nadir olarak ortaya çıkıyor olabilir. Bu sebeple durağan modeller paralellik analizinde genelde davranışsal modelleri desteklemek amacıyla kullanılmışlardır.

Çok çekirdekli sistemlere geçişte yaşanacak tek sorun yazılımların paralelleştirilmesi değildir. Çok çekirdekli mimarilerle birlikte kişisel bilgisayarlarda bulunan bellek hiyerarşisi de karmaşıklaşmaya başlamıştır. Günümüz işletim sistemi iş sıralayıcıları genelde sözü edilen bellek hiyerarşisinin etkin kullanımı yerine işlemcinin adil kullanımı üzerine yoğunlaştıklarından[8,9,10,11], eldeki cep belleklerin etkin olarak kullanılması ikinci plana atılmıştır. Kullanıcı açısından uygulamaların dengeli olarak işlemciyi kullanması daha önemli olduğundan bu durum doğal olarak karşılanabilir fakat yine de iş sıralayıcının adaletini bozmadan cep bellek israfını önleyebilecek bir yaklaşımlar model tabanlı olarak sağlanabilir. Literatürde cep bellek israfı sorunu üzerine birçok çalışma varsa da [12,13,14,15,16] bu sorunu model tabanlı kotarmak üzerine görece çok daha az çaba gösterilmiştir.

Bu yazıda, yukarıda sözü edilen paralelleştirme ve bellek hiyerarşisine uygun iş sıralama sorunlarının model tabanlı

olarak çözümü üzerine yapılan çalışmalar ve sonuçları açıklanmaktadır. Yapılan çalışmalarda UML sınıf çizenekleri kullanılarak yazılım modellerinde tekrar eden örüntüler bulunarak bunların paralelleştirilmesi ve iş sıralaması üzerinde durulmuştur. Yazıda takdim edilen örüntüler yazılım bileşenlerinin birbirlerine olan bağımlılık ilişkileri kullanılarak tanımlanmıştır. Birbiriyle bağımlılığı az olan -bağımsız- parçaların paralelleştirilmesi ve yazılımın geri kalanına çok bağımlı sınıfların senkronizasyonu üzerine tasarımcıyı/geliştiriciyi paralelleştirme çabaları esnasında yönlendirecek esaslar sunulmuştur. Bu yöntem kullanılarak örnek bir yazılım üzerinde paralelleştirme işlemi gerçekleştirilmiştir.

Bir sonraki aşamada bu örüntülerin iş sıralaması sırasında ortak veri kullanımına bağlı olarak cep bellek kullanımını daha verimli hale getirecek bir iş dağıtma yaklaşımı sunulmuştur. Sunulan iş dağıtma yaklaşımı Linux'ün CFS iş sıralayıcısına yardımcı olarak adapte edilmiş ve tasarım kalıpları üzerinde yapılan denemelerde başarım artışı sağlamıştır.

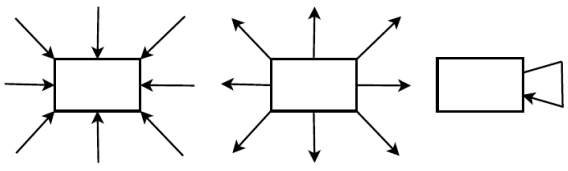
Yapılan çalışmaların sonucununda gerek yazılımlardaki gizli paralelliğin açığa çıkarılmasında gerek cep bellek etkinliğini arttıran iş sıralama algoritmlarının geliştirilmesinde model tabanlı tekniklerin kullanılabilirliğine ilişkin destekleyici yönde bulgularla ortaya çıkmıştır.

2. Bağımlılık Örüntüleri

Bağımlılık örüntüleri sınıf çizeneklerinde bulunan bağımlılık ilişkilerinin incelenmesiyle elde edilebilir. Benzer kavramlar Chatzigeorgiou ve ar.[17] çalışmalarında da tanımlanmıştır. Bu yazıda bağımlılık kavramı sınıflar arası doğrudan etkiye sahip ilişkiler olarak kullanılmaktadır. Kalıtım gibi hiyerarşiye dayalı ilişkiler bağımlılık ilişkisi olarak kullanılmamıştır.

Bağımlılık örüntüleri sınıf çizeneklerinden elde edilen bağımlılık çizgeleri üzerinde sıklıkla tekrar eden yapılardır. Bu örüntüler tek sınıf ve bağımlılık ilişkilerini içerebildikleri gibi bir grup sınıf ve bu sınıfların grup içi/grup dışı bağımlılık ilişkilerini içerebilir.

2.1. Tek Sınıflı Bağımlılık Örüntüleri



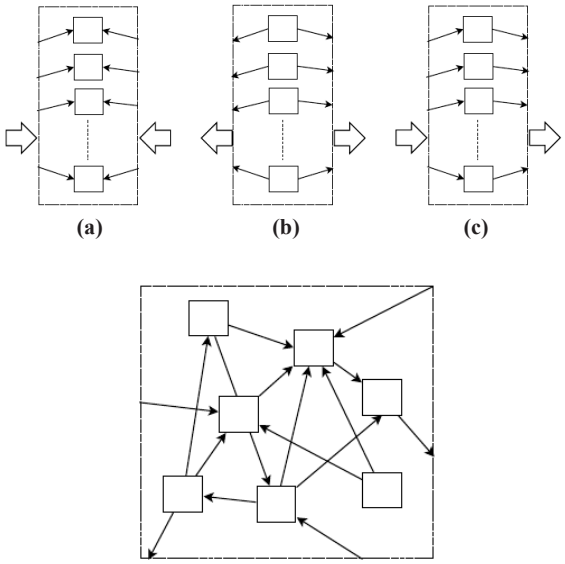
Şekil 1. Tek sınıflı bağımlılık örüntüleri

Tek sınıflı bağımlılık örüntüleri sahip oldukları bağımlılık ilişkilerine göre üç kategoride incelenebilir.

- Bir otorite diğer sınıfların yüksek miktarda bağımlı olduğu bir sınıftır (Şekil1.a). Bir sınıfı otorite olarak tanımlayabilmek için gerekli olan bağımlılık miktarı incelenen sınıf çizeneğinin büyüklüğüne göre değişebilir. Eldeki bağımlılık miktarına göre olarak belirli bir oranın üzerinde bağımlılığa sahip olan sınıflar otorite olarak adlandırılabilirler.
- Bir dağıtım sınıfı diğer sınıflara yüksek miktarda bağımlı olan sınıftır(Şekil1.b). Otoriteye benzer biçimde belirlenmesi sübjektiftir ve eldeki sınıf çizeneğindeki bağımlılık sayısına göre değişebilir.
- Bir döngü sınıfı kendine bağımlılık içeren sınıftır(Şekil1.c). Döngü sınıflarının bulunması otorite ve dağıtım sınıflarına oranla daha kolaydır.

Otoriteler ve dağıtım sınıfları çalışma zamanında sıklıkla erişilme potansiyeline sahip olduklarından paralelleştirme açısından önemli sınıflardır. Döngü sınıfları da pratikte sıradüzensel olarak gerçekleştirilmeye yatkın oldukları için paralelleştirmeye engel teşkil edebilirler.

2.2. Çok Sınıflı Bağımlılık Örüntüleri



Şekil 2. Çok sınıflı bağımlılık örüntüleri

Çok sınıflı bağımlılık örüntüleri "köprü" ve "adacık" olmak üzere iki kategoride incelenebilir. Bu örüntüler, içerdikleri sınıfların kendi aralarındaki ve örüntü dışına sahip oldukları bağımlılık ilişkilerine göre şekillenmiştir.

- Bir köprüde birbiriyle görece çok az bağımlılığı olan ve tüm bağımlılıkları grup dışında iki veya daha fazla sınıfa odaklanmış birçok sınıf bulunur(Şekil 2.a-b-c). Köprüler üç farklı biçimde karşımıza çıkabilirler. Bunlardan otorite köprüde(Şekil 2.a) örüntü dışından sınıflar örüntü içindeki sınıfların her birine bağımlıdırlar. Dağıtım köprüsünde(Şekil 2.b) örüntüdeki sınıfların her birinin örüntü dışındaki birkaç sınıfa bağımlılığı bulunur. Akış köprüsünde(Şekil 2.c) ise örüntüdeki her sınıfa bağımlılığı olan ortak bir sınıf ve örüntüdeki her sınıfın bağımlı olduğu ortak bir sınıf bulunur. Köprüler üst üste binebilir ve ortak sınıflar içerebilirler.
- Bir adacıkta bulunan sınıflar büyük oranda birbirleri arasında bağımlılık ilişkileri içerirler. Grup içi bağımlılık sayısı grup içi bağımlılıklara oranla oldukça azdır. Bu tür örüntüler bağımlılıklardan oluşan çizgeler üzerinde öbekleme yapılarak bulunabilirler.

Köprü sınıflar kendi aralarında bağımlılığa sahip olmadıkları için çalışma zamanında paralel olarak çalışmaya daha

yatkındırlar. Adacıklarda ise grup içi sınıflar yoğun miktarda birbirlerine bağımlı oldukları için ortak bellek kullanan çekirdeklere yerleştirilmeleri durumunda daha etkin bir cep bellek kullanımı sağlayabilirler.

3. Bağımlılık Örüntülerinin Paralelleştirilmesi

Bağımlılık örüntüleri sadece sınıflar arası bağımlılık ilişkilerini göz önüne almaktadır. Yazılım bileşenleri arasındaki tek çeşit ilişkiyi göz önünde bulundurarak paralellik hakkında yorum yapmanın yanlış sonuçlar doğuracağı muhtemeldir. Bu nedenle bağımlılık örüntülerini taşıdıkları kalıtım, içerme gibi farklı ilişkileri de göz önüne alarak değerlendirmek gerekir.

3.1. Bağımlılık örüntülerini paralelleştirirken uygulanacak esaslar

Bağımlılık örüntülerini sınıf çizenekleri içerisinde oynadıkları rollere göre farklı biçimlerde değerlendirmek mümkündür. Bu rollerden bazıları ve bu biçimde ortaya çıkan bağımlılık kalıplarını paralelleştirmeye ilişkin esaslar aşağıdaki gibi sıralanabilir:

- Üst sınıf bir otorite: Bir otorite sınıfın içinde bulunduğu sınıf hiyerarşisinde üst sınıf olarak görev alması durumudur. Bu durumda paralelleştirmede aşağıdaki esaslar uygulanabilir.
  - o Otorite üst sınıflar, alt sınıflar tarafından kullanılacak bilgileri ihtiva eder.
  - o Otorite üst sınıfın alt sınıflara kalıtılmış soyut olmayan kısımları paralel erişim için korunmalıdır.
  - o Alt sınıflar içerisine gömülecek senkronizasyon yapıları otorite üst sınıfa soyut bir biçimde gömülebilir

- Bir otoriteye birçok alt sınıf: Bir otorite sınıfın farklı bir sınıf hiyerarşisinde türemiş sınıfların birçoğuyla bağıntı halinde olması. Bu durumda paralelleştirmede aşağıdaki esaslar uygulanabilir.
  - o Otorite yerel bir kritik bölge haline gelmiştir.
  - o Otoriteye yapılan bağımsız erişimler paralelleştirilebilir
  - o Otoritede erişilen değişkenlerin tutarlılığı gözetilmelidir.

- Hakim sınıf: Bir dağıtım sınıfının aynı zamanda birçok sınıfın yaratıcısı olması. Bu durumda paralelleştirmede aşağıdaki esaslar uygulanabilir.
  - o Dağıtım sınıfı sistem geneline hakimdir ve sistem çalışmasını düzenlemek amacıyla sıklıkla farklı sınıfları kullanır.
  - o Bu kullanımları arasından bağımsız olanlar ayıklanarak doğrudan paralelleştirilebilir.
  - o Döngü paralelleştirme gibi daha bilindik teknikleri uygulamak için bu sınıflar daha fazla fırsat yakalanabilir.

- Kendine bağımlı sınıflar: Döngü sınıfların yanı sıra içinde bulundukları sınıf hiyerarşisinde bir üst sınıfa bağımlı olan sınıflar bu gruba girer. Bu durumda paralelleştirmede aşağıdaki esaslar uygulanabilir.
  - o Bu tür sınıflar sıradüzensel bir yapıya sahip olduğundan mümkün olduğunca paralelleştirilmelidir.
  - o Bu tür sınıfların paralelleştirilmesi mümkündür. (Ör. Bağlı listeden tablo erişimine dönüşüm)
  - o Genellikle bu tür sınıflarda global değişkenler veya sınıf değişkenleri(instance variable) bulunabilir. Bu tür değişkenlerden paralelliğin önüne geçebilecek olanlar ayıklanmalıdır.

- Köprü kardeşler: Bir köprü içerisindeki sınıfların aynı zamanda içinde bulundukları sınıf hiyerarşisinde aynı ataya sahip olması. Bu durumda paralelleştirmede aşağıdaki esaslar uygulanabilir.
  - o Bu tür sınıflara erişim genellikle birbirinden bağımsız ve çok biçimli biçimde gerçekleşir. Bu nedenle bu sınıflar daha rahat paralelleştirilebilir.
  - o Eldeki bir dağıtım köprüsüyse köprü sınıfları paralelleştirilip serbestçe işlemciler arasında dağıtılabilir. Bu durumda köprü uçları paralel erişime korunmalıdır.
  - o Eldeki bir otorite köprü veya akış köprüsüyse paralelleştirilen sınıflar işlemcilere dağıtılmalı ve fazla göç ettirilmemelidir. Bunun nedeni paralel erişime uğrayan köprü elemanlarının birbirinin senkronizasyonunu beklemesini önlemektir.

3.2. Gerçek bir yazılımda bağımlılık örüntüsü paralelleştirme

Şimdiye kadar söz edilen bağımlılık örüntülerine birkaç yıl önce geliştirilmesine son verilen açık kaynak kodlu Jikes Java derleyicisinde rastlanabilir. Paralelleştirme için bir derleyici yazılımının seçilmesinin nedeni çalışma şekli nedeniyle paralelleştirmenin görece daha zor olacağı bir yazılımda geliştirilen yöntemin ortaya çıkarabileceği paralelleştirme olanaklarını inceleyecerek başarımını sınamaktır. Örneğin Ek A Şekil 12'de Jikes'ın soyut sözdizim ağaçlarını(Abstract Syntax Tree(ks. AST)) ele almakta kullandığı bir modülüne ilişkin bağımlılık çizeneği görülebilir.

Şekil 12'de görülen bağımlılık kalıpları arasında Java kodlarında bulunan sözdizim elemanlarını sırasıyla ele almakta kullanılan AstExpression ata sınıfından türemiş alt sınıfların bir köprü oluşturduğu görülebilir. Bunun yanında yine StoragePool gibi ortak kullanıma sıklıkla konu olan bir sınıfın da otorite sınıf olarak ortaya çıktığı görülebilir.

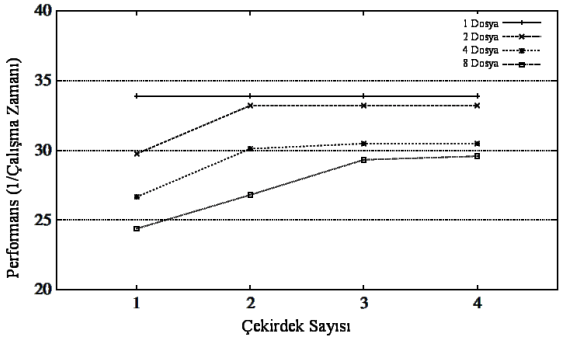
Bu çizenekte de görülebilen AstExpression sınıfı bir otorite üst sınıfa örnek olarak gösterilebilir. Bir önceki bölümde bahsedilen

esaslara uygunluk gösteren bu sınıf yazılımın çalışması esnasında yoğun bir erişim trafiğine sahne olur. Bu sınıftan alt sınıflara kalıtan IsConstant() metodu 400 satırlık bir Java kodunun derlenmesi esnasında bile 700 farklı nesneden ortalama beşer kez çağırılmaktadır ki bu metodun çağırıldığı sınıflar da paralelleştirilmeye müsait köprü sınıflarıdır. Olası bir paralelleştirilme işleminde bu sınıf muhtemel bir kritik bölge olacaktır.

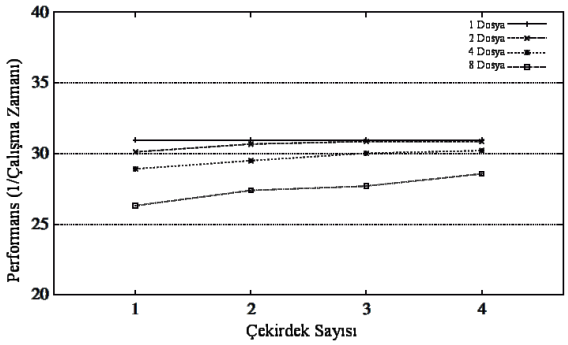
Yazılımın diğer modüllerinde de bağımlılık örüntülerine sıklıkla rastlanmaktadır. Bunların aralarında en etkileyicisi bir hakim sınıf olarak bulunan Control sınıfıdır. Adının da hissettirdiği üzere bu sınıf derleyicinin derleme işlemine soktuğu sınıfları sırasıyla ardı ardına farklı derleme modüllerine gönderdiği sınıftır. Kısa bir göz atma sonucu bu sınıf içerisinde birbirinden bağımsız çalışan birçok işlem keşfedilebilir.
Bunlarda biri farklı derleme modüllerine gönderilen dosyaların her bir ayrı modüle paralel olarak gönderilmesidir. Bu yolla örneğin bir dosya programlama dili elemanlarının bulunması için ayrıştırma işlemine gönderilirken diğer dosya sözdizimsel analiz işlemine sokulabilir.

Bir başka paralelleştirme fırsatı ise yazının girişinde bahsedilen problem doğasında bulunan paralelleştirmeye bir örnektir. O anki haliyle her bir dosyayı sırasıyla arka arkaya derleme

işlemine sokan derleyicisi basit bir paralelleştirme işlemi ile birbirinden bağımsız dosyaları aynı anda derleme işlemine alabilir. Sözü edilen iyileştirmelerden sonra elde edilen başarım artışları Şekil 3 ve Şekil 4'de görülebilir.



Şekil 3. Eş uzunluktaki dosyaları derleme başarımı



Şekil 4. Değişen uzunluktaki dosyaları derleme başarımı

Grafiklerde görülen başarım artışları paralelleştirme yapılmadığı takdirde kullanılan işlemci sayısından bağımsız olarak tek işlemcili derleme başarımına sahip olmaktadır. Oysa ki yapılan paralelleştirmelerin ardından artan çekirdek sayısı başarımı da arttırmaktadır. Her ne kadar elde edilen başarım zaman zaman %5-%10 seviyesinde olsa da unutulmamalıdır ki yapılan paralelleştirme işlemleri eldeki yazılım hakkında geniş çaplı bir analiz yapılmadan ve detaylı bilgi edinilmeden sadece model tabanlı analize bağlı olarak yapılmıştır. Yazılıma geliştiren ve ona daha hakim olan kişilerce performans artışı çok daha yükse seviyelere çıkartılabilir.

Bağımlılık kalıplarına ilişkin daha detaylı çalışmalar yazarların konuyla ilgili diğer yazılarında mevcuttur [18,19].

4. Bağımlılık Örüntülerinin Cep Bellek Bilinçli İş Sıralaması

4.1. Cep Bellek Bilinçli İş Sıralama

Cep bellek bilinçli iş sıralama yaklaşımı, yazılımın çalışması esnasında işlemcide ortak cep bellek kullanılan çekirdeklerin ortak cep belleklerden yararlanma miktarını arttıracak biçimde yapılan iş sıralama olarak özetlenebilir. Bu tür bir yaklaşımın geliştirilebilmesi için yazılım bileşenlerinin ortak veri kullanım miktarlarının önceden belirlenmesi gereklidir. Bu işlem çalışma zamanı sırasında yapılacak analizlerle gerçeklenebilir fakat yazılım çalıştırılmadan hatta daha gerçeklenmeden bu bilgiye ulaşmak faydalı olabilir. Bu nedenle, aşağıda takdim edilen çalışmalarda yazılımın sınıf çizeneklerinden faydalanılarak cep bellek bilinçli olarak iş sıralamasının yapılması amaçlanmıştır.

Öte yandan bu tür bir analizi yapmak için paralelleştirme esnasında ortaya çıkan farklı ihtiyaçları da göz önünde bulundurmak gerekir. Bu ihtiyaçlar:

- Paralelleşme: Birbirinden bağımsız koşabilecek parçalar. Bunlar mümkün olduğunca çekirdeklere dağıtılmalıdır.
- Veri paylaşımı: Büyük miktarda veri paylaşan yazılım bileşenleri mümkün olduğunca ortak cep bellek kullanan çekirdeklere atanmalıdır.
- Kaynak kullanımı: Eldeki çekirdekler mümkün olduğunca kullanılmalı, boş bırakılmamalıdır.

Bu üç madde birbirini ortagonal olarak etkilemekte, paralelleşme yazılımın paralel koşan parçalarının serpiştirilmesini öngörürken veri paylaşımı paralel parçaların öbeklenmesini gerektirmekte, kaynak kullanımı ise eldeki çekirdeklerin mümkün olduğunca dolu kalmasına ihtiyaç duymaktadır.

Yapılan çalışmalarda bu üç etkenin cep bellek bilinçli zamanlama üzerindeki etkileri gösterilmeye çalışılmış ve cep bellek kullanımını arttırmanın paralelleştirme başarımı üzerine etkileri incelenmiştir.

Bu amaçla GoF tasarım kalıpları[20] gerçeklenerek, cep bellek bilinçli iş sıralamada kullanılmıştır. Tasarım kalıpları günümüz yazılımlarında sıklıkla kullanılmakta ve bu sayede geliştirilen yaklaşımın tasarım kalıplarının yeniden kullanımı ile birlikte yazılımlarda tekrar tekrar kullanılmasına olanak vermektedir.

Çalışmanın son safhasında geliştirilen yaklaşım kullanılarak bir görüntü filtreleme yazılımı içerisinde bulunan bağımlılık örüntüleri yardımıyla cep bellek bilinçli iş dağıtımı yapılmış ve elde edilen sonuçlarla Linux'ün CFS isimli iş sıralayıcısının başarımı arttırılmıştır.

4.2. Tasarım Kalıplarının Cep Bellek Bilinçli İş Sıralanması

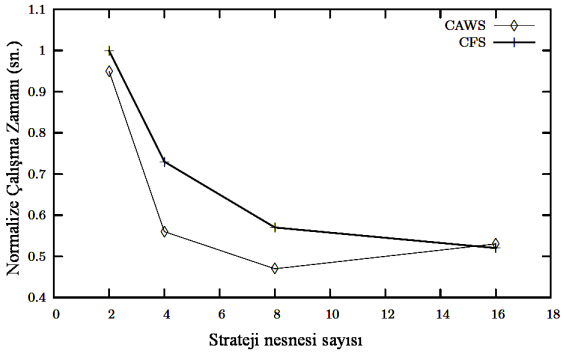
Bu kısımda strateji(strategy), ziyaretçi(Visitor) ve gözlemci(observer) isimli üç farklı tasarım kalıbı farklı yaklaşımlarla cep bellek bilinçli iş sıralamaya tabi tutulmuş ve elde edilen başarım sonuçları Linux CFS iş sıralayıcısı ile karşılaştırılmıştır[21].

Yapılan deneylerde yazılım kalıpları bileşenleri veri paylaşım miktarları göz önünde bulundurularak mümkün olduğunca ortak cep bellek kullanan çekirdeklere dağıtılmaya çalışılmışlardır.

Deneylerin yapıldığı sistem her biri çift çekirdekli 4 adet Xeon işlemci barındıran bir sunucudur. Bu da iş sıralama işlemleri esnasında nesnelerin dağıtılabileceği 8 farklı çekirdek sunar. Deneyler sırasında Java programlama dili ve JNI yardımıyla Linux çekirdeğinin iplik ilişkilendirme(thread affinity) sistem çağrıları kullanılmıştır. Deneylerde her bir nesne ayrı bir iplik olarak kullanıldığından yapılan açıklamalarda nesne ve iplik kavramları birbirinin yerine kullanılabilmektedir.

- Strateji. Strateji kalıbı için, belirli sayıdaki istemci nesne için her biri farklı bir stratejiyi temsil eden sabit sayıda strateji nesnesi yaratılmıştır. Her bir strateji nesnesi kendisine çalışma zamanında atanan bir istemci ile birlikte

ortak bir miktar veri üzerinde çalışarak işlemlerini tamamlamıştır.

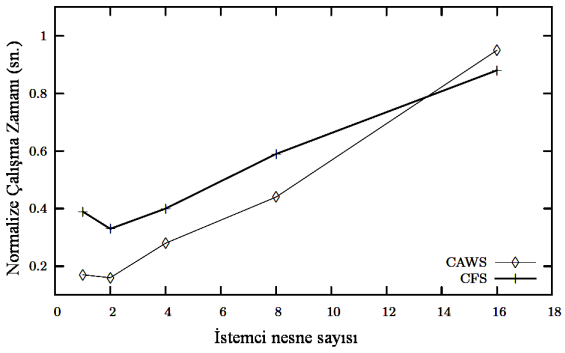


Şekil 5. Strateji kalıbının cep bellek bilinçli iş sıralaması

Şekil 5'de 32 adet istemci için farklı sayılarda strateji nesnesinin cep bellek bilinçli iş sıralaması sonucu elde edilen başarım artışı görülebilir. Şekilde CFS Linux'ün "Completely Fair Scheduler" isimli iş sıralayıcısını CAWS ise "Cache Aware Scheduling" olarak adlandırılan cep bellek bilinçli zamanlama yaklaşımını temsil etmektedir. Programların çalışma süreleri en kötü çalışma süresine göreli olarak normalize edilmiştir. Böylece çizimlerde grafik 1'e yaklaştıkça çalışma zamanı en kötü çalışma zamanına yaklaşmakta dolayısıyla da başarım düşmektedir. Grafikte de görüldüğü üzere, cep bellek bilinçli iş sıralama kullanıldığı zaman paralelleşen bileşen sayısı işlemci sayısına ulaşana kadar başarım artmıştır. Bu durumun nedeni iş sıralama üzerine yapılan geliştirmenin nesne göçü yapmamasıdır.

- Ziyaretçi. Ziyaretçi kalıbı için ziyaretçi ve istemci nesnelerin birbirinden bağımsız olarak yaratılmış ve paralel olarak çalışacak şekilde ziyaretçiler rastgele istemci nesnelere atanmışlardır. Her bir ziyaretçi ve istemci bir sonraki sefer birlikte çalışacağı nesneleri kendi içlerinde birer kuyruk yapısında tutmuş, böylece bir ziyaretçi çalışırken başka bir istemciye yönelik ziyaretini sırada tutabilecektir.

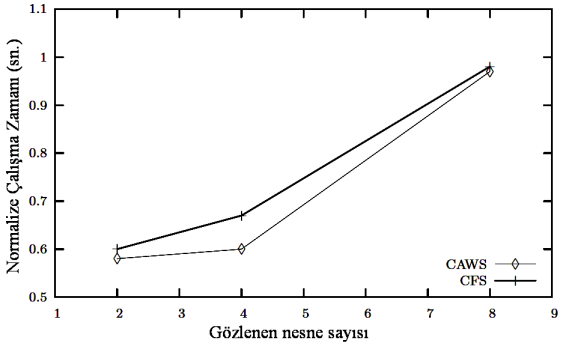
Şekil 6'da 8 adet ziyaretçi farklı sayıda istemci nesneye hizmet vermek üzere çalıştırılmıştır. Deneylerde, stratejiye benzer bir sonuç ziyaretçi kalıbı için de elde edilmiştir. Bu sefer stratejiye oranla daha rastgele bir çalışma mekanizmasının kullanılması nedeniyle başarım artışı stratejiye oranla daha azdır.



Şekil 6. Ziyaretçi kalıbının cep bellek bilinçli iş sıralaması

- Gözlemci. Gözlemci kalıbı için diğer iki kalıba oranla daha farklı bir paralelleştirme anlayışı uygulanmış ve her bir gözlemci kendi istemcisi tarafından oluşturularak çalıştırılmaya başlanmıştır. Bu tür bir senaryoda çok daha fazla sayıda nesne yaratılacak ve istemci-gözlemci grupları birbirinden daha izole halde çalışacaklardır.

Şekil 7'de iki adet gözlemci nesnenin farklı sayıdaki istemciler için gösterdiği başarımlar görülebilir. Bu deneyde iki adet gözlemci kullanılmasının nedeni ortak cep bellek kullanan en fazla iki adet çekirdeğin sistemde bulunması, bu nedenle de farklı sayılarda istemci üzerinde deneme yapmak için dört işlemciye dört istemci ve her iki çekirdeğe iki ortak gözlemcinin denk gelmesinin uygun olmasıdır.

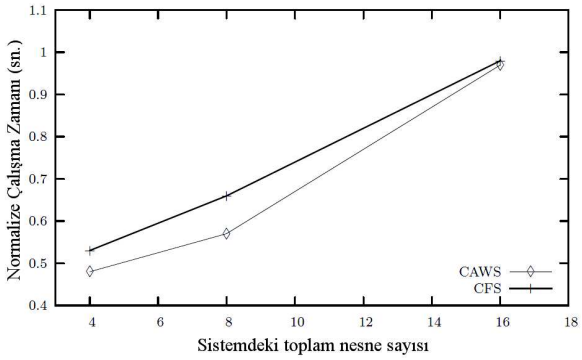


Şekil 7. İki gözlemci cep bellek bilinçli iş sıralaması

Birçok farklı sayıda istemci gözlemci çifti için de başarımın ölçülmesi daha gerçekçi deneyler için elzemdir. Bu amaçla sistemdeki toplam nesne sayısı çeşitlendirilmiş ve cep bellek bilinçli iş sıralama uygulanmıştır. Şekil 8'de görülen sonuçlar önceki deneylerle paralellik göstermektedir.

4.3. Bir görüntü filtreleme yazılımı üzerinde cep bellek bilinçli iş sıralama

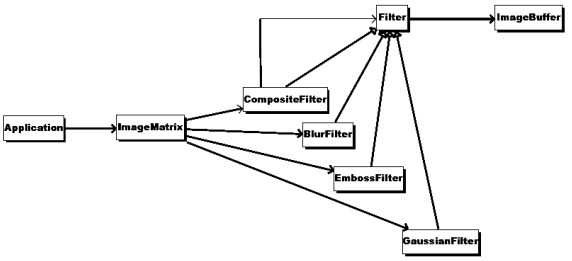
Yukarıda sıralanan deneyler farklı tasarım kalıpları için cep bellek bilinçli iş sıralamanın uygulanabilirliği konusunda olumlu sonuçlar vermiştir. Doğal olarak bu tür bir iş sıralamanın gerçek dünya yazılımlarına daha yakınsayan bir örnek üzerinde denenmesi uygulanabilirliği konusunda daha elle tutulur fikirler verecektir.



Şekil 8. Farklı sayıda nesne için gözlemci kalıbının cep bellek bilinçli iş sıralaması

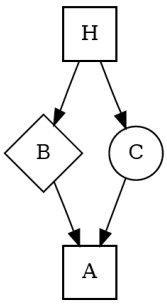
Son olarak, cep bellek bilinçli iş sıralama yaklaşımıyla gerçeklenen bir iş dağıtıcısı aracılığıyla Şekil 9'da modeli

sunulan bir görüntü filtreleme yazılımının iş sıralama işlemi gerçekleştirilecek ve elde edilen başarım Linux'ün CFS iş sıralayıcısı ile karşılaştırılacaktır.



Şekil 9. Örnek bir görüntü filtreleme yazılımı modeli

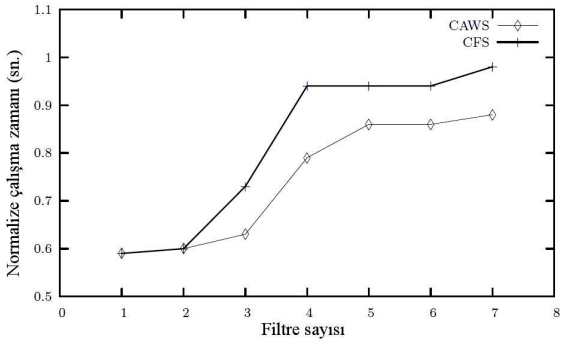
Paralelleştirme işleminde ve cep bellek bilinçli iş sıralamada yazılımda bulunan bağımlılık kalıplarından faydalanılacaktır. Bu amaçla Şekil 9'da sunulan yazılım modelinden Şekil 10'da sunulan bağımlılık çizeneği elde edilmiştir.



Şekil 10. Şekil 10'daki yazılım modelinin bağımlılık çizeneği

Şekil 10'daki çizeneğe göre paralelleştirme işlemi H ile gösterilen ve bir dağıtım sınıfı olan ImageMatrix sınıfından başlatılacak, B ile temsil edilen köprü oluşturan Filter sınıfları çekirdeklere dağıtılacak, bu esnada C ile temsil edilen döngü sınıfı CompositeFilter'da bulunabilecek sıradüzensel yaklaşım paralelleştirilerek A ile temsil edilen otorite sınıfı ImageBuffer üzerinde gerekli senkronizasyon işlemleri gerçeklenecektir. Yukarıdaki yönergeler doğrultusunda paralelleştirilen yazılımın yüksek miktarda ortak veri kullanması muhtemel, köprü-otorite sınıflarının cep bellek bilinçli zamanlanması ile başarımı arttırılacaktır.

Yapılan deneyler sonucu Şekil 11'de sunulan başarım artışı elde edilmiştir. Deneylerde yaklaşık 1.6MB büyüklüğünde bir resim dosyası bir ImageMatrix nesnesi tarafından belleğe alınmış, akabinde ImageBuffer nesnelerine bölünerek farklı Filter nesneleri tarafından paralel olarak filtrelenmiştir. Filter ve ImageBuffer nesneleri mümkün olduğunca ortak bellek kullanan çekirdeklere atanmış, böylece ardı ardına aynı bölge üzerinde filtreleme yapacak olan filtrelerin ortak cep bellek üzerinde veri paylaşması sağlanmıştır.



Şekil 11. Görüntü filtreleme yazılımının cep bellek bilinçli iş sıralaması

5. Sonuçlar

Elde edilen sonuçlar aşağıdaki gibi özetlenebilir:

- Yazılımların durağan modelleri içerisinde bulunabilecek örüntüler yardımıyla paralelleştirme işlemi esnasında önem kazanacak yapılar keşfedilebilir.
- Keşfedilen bu yapılar problemin doğasına ilişkin paralelleştirme fırsatlarının yakalanmasına neden olabilmektedir.
- Cep bellek bilinçli iş sıralama ile uygulama yazılımlarının çok çekirdekli işlemcilerdeki başarımı arttırılabilir.
- Model tabanlı yaklaşımlar kullanılarak cep bellek bilinçli iş sıralamada kullanılabilecek ön bilgiler henüz yazılım çalıştırılmadan ya da daha yazılım üretilmeden elde edilebilir.

6. Bilgilendirme

Bu yazı İTÜDergisi/d için hazırlanan benzerinden kaynaklanılarak hazırlanmıştır.

7. Kaynakça

[1] Dagum, L., Menon, R., OpenMP: An industry-standard API for shared memory programming. *Computing in Science and Engineering,* **5**, 46–55.

[2] Message passing interface., http://www.mpi-forum.org

[3] Kaveh, N., Using model checking to detect deadlocks in distributed object systems. In: *Revised Papers* from the Second International Workshop on Engineering Distributed Objects. EDO'00. Springer-Verlag, UK, sf. 116–128.

[4] Li, X., Lilius, J., Timing analysis of UML sequence diagrams. In: *Proceedings* of the Second International Conference on The Unified Modeling Language. Beyond the Standard. Springer, Fort Collins, CO, USA, sf. 661– 674.

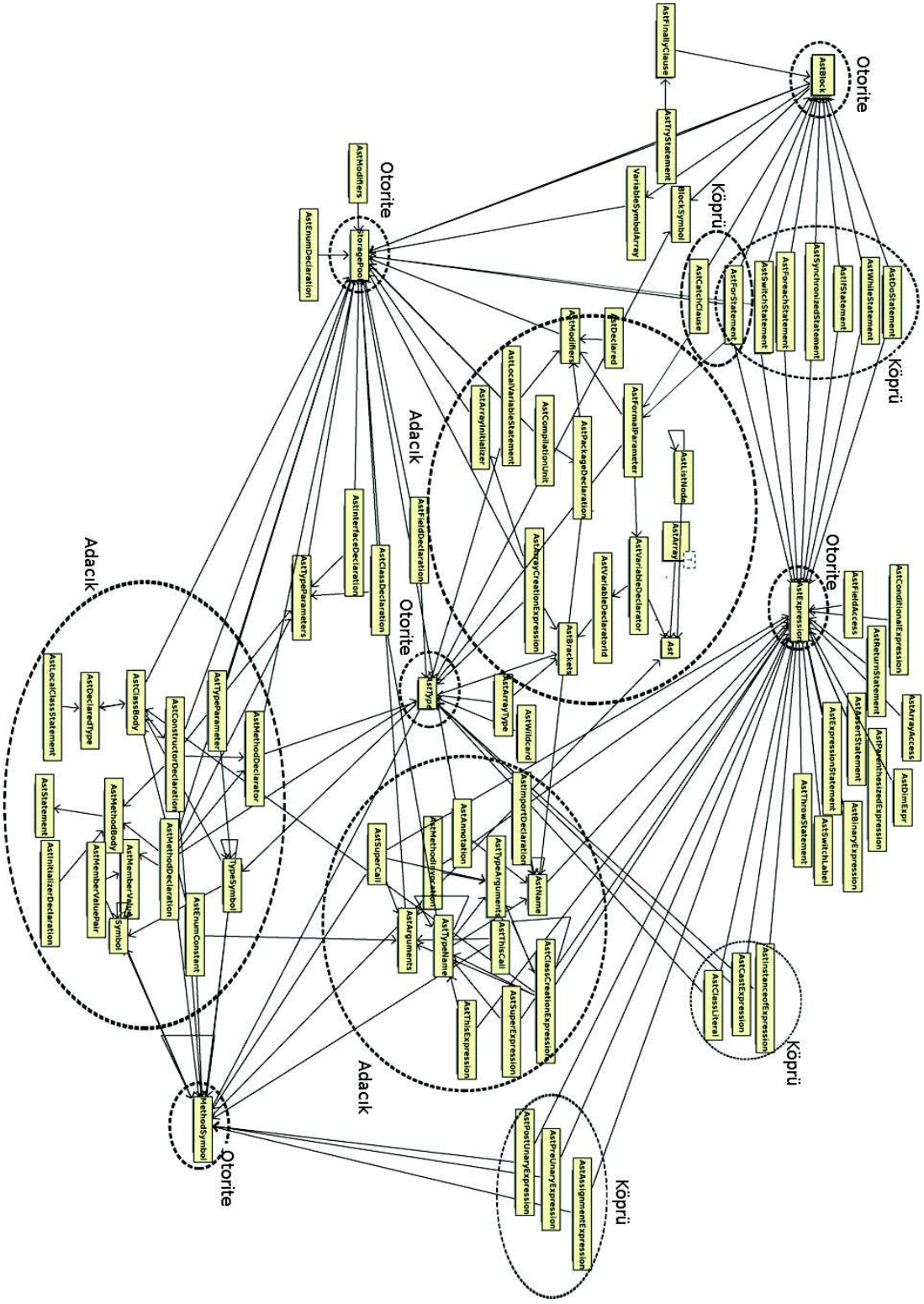
[5] Li, X., Meng, C., Yu, P., Jianhua, Z., Guoliang, Z., Timing analysis of UML activity diagrams. In: *Proceedings* of the Fourth International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools. Springer-Verlag, London, UK, sf. 62–75.

[6] Engels, G., Küster, J., Groenwegen, L., Consistent interaction of software components. *Journal of Integrated Design and Process Science,* **6,** 4, 2–22.

[7] Giese, H., Klein, F., Burmester, S., Pattern synthesis from multiple scenarios for parameterized real-time UML models. In: *Scenarios: Models, Transformations and Tools*. **Vol. 3466** of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, sf. 193–211.

[8] Zangerl, T., Optimisation: Operating system scheduling on multi-core architectures. http://tzangerl.net/doc/MulticoreScheduling.pdf
[9] Siddha, S., Multi-core and linux kernel. http://software.intel.com/sites/oss/pdfs/mclinux.ppd
[10] MSDN section on windows scheduling. http://msdn.microsoft.com/en-us/library/ms685096%28VS.85%29.aspx
[11] Solaris 11 programming interfaces guide. http://download.oracle.com/docs/cd/E19963-01/821-1602/psched-23069/index.html
[12] Kim S., Ch D., ve Solihin Y., Fair cache sharing and partitioning in a chip multiprocessor architecture, in *Proceedings* IEEE PACT, sf. 111–122.
[13] Tam D., Azimi R., Soares L., ve Stumm M., Managing shared L2 caches on multicore systems in software, in Workshop on the Interaction between Operating Systems and Computer Architecture, 2007.
[14] Merkel A. ve Bellosa F., "Memory-aware scheduling for energy efficiency on multicore processors," in *Proceedings* of HotPower.
[15] Ha J., Arnold M., Blackburn S.M., ve McKinley K.S., A concurrent dynamic analysis framework for multicore hardware, in *Proceedings* of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications, (NY, USA), sf. 155–174, ACM.
[16] Zhou B., Qiao J., ve Lin S. kuan, Research on dynamic cache distribution scheduling algorithm on multi-core processors, in *Proceedings* of E-Business and Information System Security, 2009. EBISS '09. sf. 1-4, 23-2.
[17] Chatzigeorgiou, A., Tsantalis, N., Stephanides, G., Application of graph theory to OO software engineering. In: *Proceedings* of the International Workshop on interdisciplinary software engineering research. WISER'06. ACM, New York, NY, USA, sf. 29–36.
[18] Ovatman T., Weigert, T., Buzluca, F., Applying enhanced graph clustering to software dependency analysis. In: *Proceedings* 19th International Conference on Software Engineering and Data Engineering. sf. 1 –7.
[19] Ovatman T., Weigert, T., Buzluca, F., Exploring implicit parallelism in class diagrams. *Journal of Systems and Software*. Vol.84, no. 5, sf. 821-834.
[20] Gamma B., Helm R., Johnson R., ve Vlissides J., *Design Patterns: Elements of Reusable Object- Oriented Software*. Addison-Wesley Professional, 1994.
[21] Ovatman T., Buzluca F., Model Driven Cache-Aware Scheduling of Object Oriented Software for Chip Multiprocessors, *14th Euromicro Conference on Digital System Design Architectures, Methods and Tools*, 2011 Oulu,Finland, pp.719-726.

Ek A – Örnek Bir Bağımlılık Çizeneği



Şekil 12. Jikes AST modülünün bağımlılık çizeneği ve içerisindeki bağımlılık örüntüleri