KOMUT, İŞÇİK VE BELLEK DÜZEYİNDE PARALELLİĞE YÖNELİK DERLEYİCİ TEKNİKLERİ

Hazırlayan: Ali Üllenoğlu
No: 504061502

Ders: Bilgisayar Mimarisine Yeni Yaklaşımlar
Öğretim Üyesi: Prof. Dr. Bülent Örencik

1. Komut Düzeyinde Paralellik

Günümüzde tüm modern mikroişlemciler bir saat çevriminde birden fazla komut işleyebilme yeteneğine sahiptir. Derleyicinin görevi, komut düzeyinde paralellik kullanılarak bu özellikten maksimum derecede faydalanmaktır. Programlar genelde komutların işlemcilerde art arda yürütüleceği varsayılarak yazılırlar. Bu nedenle derleyicinin bu komutları işlemcide paralel şekilde en iyi nasıl yürüteceğini belirlemesi gerekir. Bu işlem için göz önüne alınması gereken parametreleri şöyle sıralayabiliriz:

- Programdaki maksimum paralellik potansiyeli.
- İşlemcideki maksimum paralellik potansiyeli.
- Derleyicinin verilen sıralı programı paralelleştirme yeteneği.
- Verilen komut sıralama kısıtlamaları dahilinde en iyi paralel sıralamayı bulma.

Paralelleştirme işlemi hem derleyici tarafında yazılımda, hem de donanımda yapılabilir veya bu ikisi birbirinin farkında olarak ortaklaşa paralelleştirmeye katkıda bulunabilirler. Sıralı çalışmak için yazılan kod parçaları, aynı anda birden fazla komut çalışacak şekilde yeniden düzenlenebilirler. Bu durumda bazı kod parçaları işlenmek için normal sırasına göre önceye veya sonraya atılabilir.

1.1 Komut Düzenleme (Code Scheduling)

Komut düzenleme, derleyicinin kod üretimi safhasından sonra gerçekleştirilen bir optimizasyondur. Paralelliği sağlayacak şekilde kodların yerlerini değiştirdiği için çeşitli kısıtlamalara tabidir. 3 tür kısıtı vardır:

- **Kontrol bağımlılığı**: Orjinal programdaki tüm işlemler optimize edilmiş programda da yapılmalı.
- **Veri bağımlılığı**: Orjinal programla optimize edilmiş program aynı sonuçları üretmeli. Ve yan etkiler olmamalı.
- **Kaynak bağımlılığı**: Optimize edilmiş program, orjinal programdan daha fazla kaynak tüketmemeli.

1.1.1 Veri Bağımlılığı

Eğer iki ayrı komut, aynı veri üzerinde işlem yapmıyorsa veya ikisi de sadece okuma işlemi yapıyorsa bu komutlar arasında veri bağımlılığı yoktur. Fakat bir komut verinin üzerine yazıyorsa, diğer komut ise veriyi okuyorsa aralarında veri bağımlılığı vardır ve yerlerini değiştirmek sorunlara yol açabilir. 3 tür veri bağımlılığından söz edilebilir.

- **Gerçek Bağımlılık:** Yazma işleminden sonra okuma işlemi yapılması durumudur. Çünkü okuma işlemi daha önce yazılan veriye bağımlıdır ve bu komutların yerleri değiştirilemez.

- **Ters Bağımlılık:** Okuma işleminden sonra yazma işlemi yapılması durumudur. Yazma işlemi, okuma işlemine bağımlı değildir ama okuma işleminden önce yazma yapılırsa okuma işlemi yapan komut yanlış bir değer okur.
- **Çıkış Bağımlılığı:** Yazma işleminden sonra yazma işlemi yapılması durumudur. Eğer bu bağımlılığa uyulmazsa komutların işletilmesi sonucunda yanlış bir değer elde edilir.

1.1.2 Bellek Erişimleri Arasındaki Bağımlılıkları Bulma

İki bellek erişimi adresi arasındaki veri bağımlılığını bulmak için iki komutun aynı bellek adresine erişmeleri gerektiğini bilmek yeterlidir. Hangi adrese eriştiklerini ayrıca bilmek gerekli değildir. Fakat derleyici, her zaman blok içerisindeki bellek adreslerinin birbirinin aynısı olup olmadığı bilinemez. Örneğin:

```
*p = 12;
*(p + 4) = 16;
```

işlemlerinde bellek bağlımlılığı olmadığı kolayca görülebilir. Fakat;

```
a =  12;
*p = 16;
```

işlemlerinde bağımlılık olup olmadığı her zaman kolayca bilinemeyebilir. p işaretçisi, a değişkeninin adresini gösteriyor olabilir. Bellek bağımlılığını çözmek için kullanılabilecek analiz yöntemleri aşağıdaki gibi özetlenebilir:

- **Dizi veri bağımlılığı analizi**: Dizi erişimleri sırasında indeks değerlerinin karşılaştırılarak aynı veriye erişilip erişilmediğinin kontrol edilmesidir. Örneğin:

  ```
  for(i = 0;iI < n; i++)
     array[2*i] = array[2*i+1]
  ```

  Döngüsünde dizinin çift indeksli elemanlarına kendisini takip eden tek indeksli elemanın değeri atanır. Bu analiz yapılarak dizi indekslerine erişim sırasında herhangi bir veri bağımlılığı olmadığı görülür.

- **Prosedürler arası analiz**: Bir değişken, farklı parametreler yoluyla aynı fonksiyona aktarılabilir. Bu şekilde fonksiyon parametreleri birbirinden farklı bellek adresinde zannedebilir fakat iki parametre de aynı adrestedir. Bu durumun çözümlenmesi işlemidir.

- **İşaretçi etiketi analizi**: Farklı işaretçiler, aynı bellek alanına işaret ediyor olabilir. Bu durumda oluşmuş olabilecek bağımlılığın çözülmesi için işaretçilerin analiz edilmesi gereklidir

1.1.3 Saklayıcı Kullanımı ve Parallellik Arasında Seçim Yapma

Derleyici ara kod üretme safhasında, program değişkenlerinin tutulması için sahte saklayıcılar kullanır. Bu saklayıcıların sayısı sınırsızdır ve herhangi bir mimariye uymaz. Son makine kodunun üretilmesi safhasında sahte saklayıcılar, gerçek saklayıcılarla eşleştirilir. Ara kodda kullanılan saklayıcı sayısı sınırsız olduğu için son kodda kullanılan saklayıcı sayısının azaltılması ihtiyacı doğar. Bu durum, komut düzeyinde paralelliğin maksimum derecede saklayıcı kullanma ihtiyacı ile çelişir.

Örneğin; `(a + b) + c + (d + e)` işleminin, minimum sayıda saklayıcı kullanılacak şekilde kodu oluşturulduğunda aşağıdaki kod elde edilir:

```
r1 = *a
r2 = *b
r1 = r1 + r2
r2 = *c
r1 = r1 + r2
r2 = *d
r3 = *e
r2 = r2 + r3
r1 = r1 + e2
```

Bu örnekte saklayıcı sayısının en aza indirilmesi, sıralı işleme sonucunu doğurmuştur. Bu şekilde herhangi bir paralellik elde edilemez. İşlemler bu yöntemle 9 adımda tamamlanır. Fakat 9 adet saklayıcı kullanılabilseydi veriler aşağıdaki gibi paralel işlenebilirdi ve 4 adımda işlem tamamlanabilirdi:

r1 = *a	r2 = *b	r3 = *c	r4 = *d	r5 = *e
r6 = r1+r2	r7 = r4+r5			
r8 = r6+r3				
r9 = r8+r7				

1.1.4 Kontrol Bağımlılığı

Bir x komutunun işlenmesi, bir y komutunun işlenmesi sonucuna bağlı ise, x'in, y'ye kontrol bağımlılığı vardır. Birbirine kontrol bağımlı olmayan işlemler aynı anda paralel olarak işlenebilirler. Kontrol bağımlı işlemlerin ise yan etkilere yol açmayacak kısmı spekülatif olarak işlenebilir. Eğer kontrol bağımlı olduğu işlem sonucunda komutun işlenmesine karar verilirse programın daha hızlı çalışması sağlanmış olur. Örneğin:

```
if (a > t)
     b = a * a
d = a + c
```

ifadelerinde, $b = a * a$ ifadesi ile $d = a + c$ ifadeleri arasında veri bağımlılığı yoktur. Fakat $b = a * a$ ifadesinin işlenmesi, $a > t$ ifadesinin doğru olmasına bağlıdır. $a * a$ işleminin yan etkilere neden olmayacağı varsayılırsa bu çarpım paralel olarak spekülatif

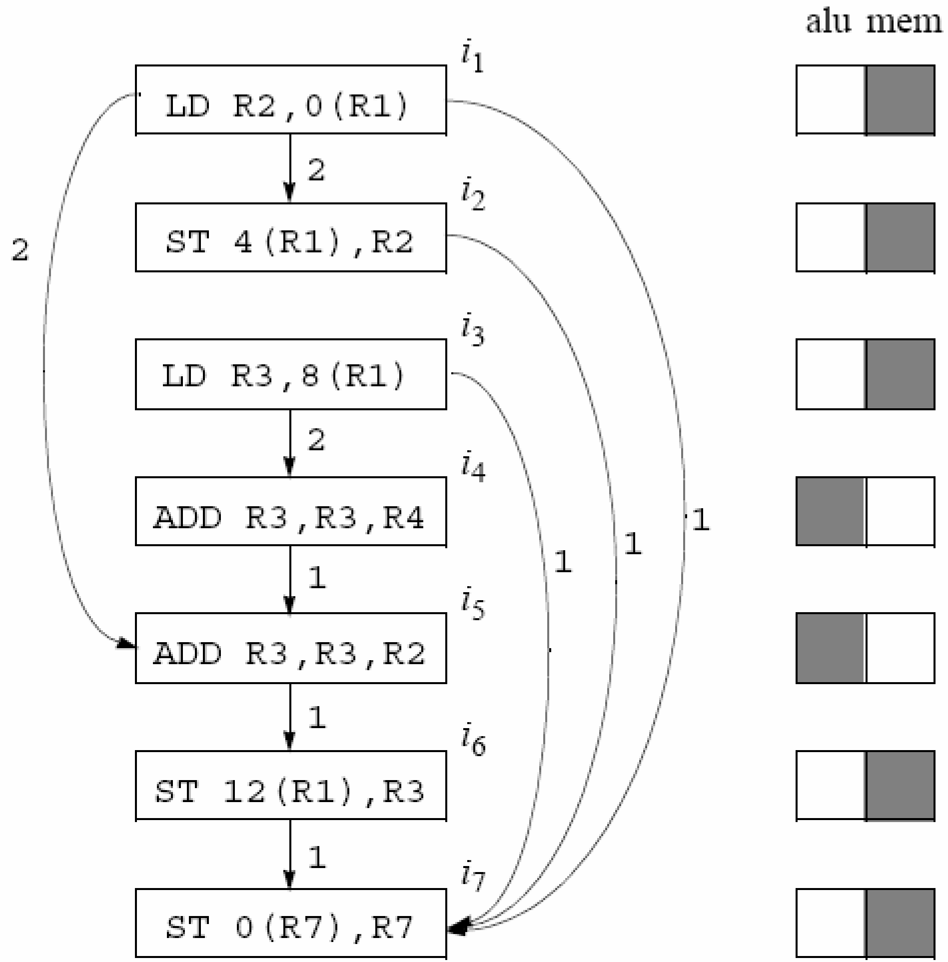
bir şekilde yapılır. Eğer $a > t$ doğru çıkarsa, $a * a$ değeri önceden hesaplandığı için hemen b'ye atanabilir.

1.1.5 Temel Kod Sıralama (Basic Code Scheduling)

Bir kod bloğu içerisindeki komutları sıralama işlemidir. Bir temel blok içerisinde birbirine çok sayıda bağımlı komut bulunur. Bu nedenle basit teknikler yeterlidir.

1.1.5.1 Veri Bağımlılık Grafı

Her temel blok, bir veri bağımlılık grafı ile temsil edilir. Veri bağımlılık grafı G = (N, E), N düğümler kümesi olarak komutları, E ise kenarlar kümesi olarak komutlar arasındaki veri bağımlılığını temsil eder.



Şekil 1 – Veri Bağımlılık Grafı

- Her komutun, kullandığı kaynakların sayısının bulunduğu bir kaynak tablosu bulunur.

- Her E kenarının, bitiştirdiği düğümler arasında olması gereken gecikmeyi belirten sayısal bir değeri vardır. Örneğin, A komutunda başlayıp B komutunda biten bir kenarın değeri 3 ise, B komutu A komutundan en az 3 saat çevrimi sonra çalışmalıdır.

1.1.5.2 Örnek Mimari

Örnek mimaride, işlemci bir saat çevriminde 2 işlem yapamaktadır. İlk işlem aşağıdaki şekildeki gibi bir dallanma veya ALU işlemi olmalıdır.

OP dst, src1, src2

İkinci işlem ise aşağıdaki şekilde bir yükleme (load) veya yazma (store) işlemi olmalıdır.

LD dst, addr
ST addr, src

Yükleme (LD) işlemi tamamen iş hattıyla gerçeklenmiştir ve 2 saat çevriminde tamamlanır. Fakat LD işlemi yapılan bellek alanı 1 saat çevrimi sonra ST ile yazılabilir çünkü değeri bellekten alınmıştır. Diğer işlemler 1 saat çevriminde tamamlanır. **Şekil 1'**de blok ve onun kaynak istekleri görülmektedir.

- Şekilde görüldüğü gibi, ilk işlem R2'ye değer yüklenmesidir. R2'ye yüklenen değeri kullanan i2 ve i5 işlemleri arasındaki kenar değerleri 2'dir. Diğer yükleme işlemleri için de aynı durum geçerlidir.

- R1 ve R7 saklayıcılarının değerlerlerinin birbiri ile ilişkisi hakkında herhangi bir şey söylenemez. Bu nedenle bu değerlerin aynı adresi gösterebileceği varsayımında bulunulacaktır.

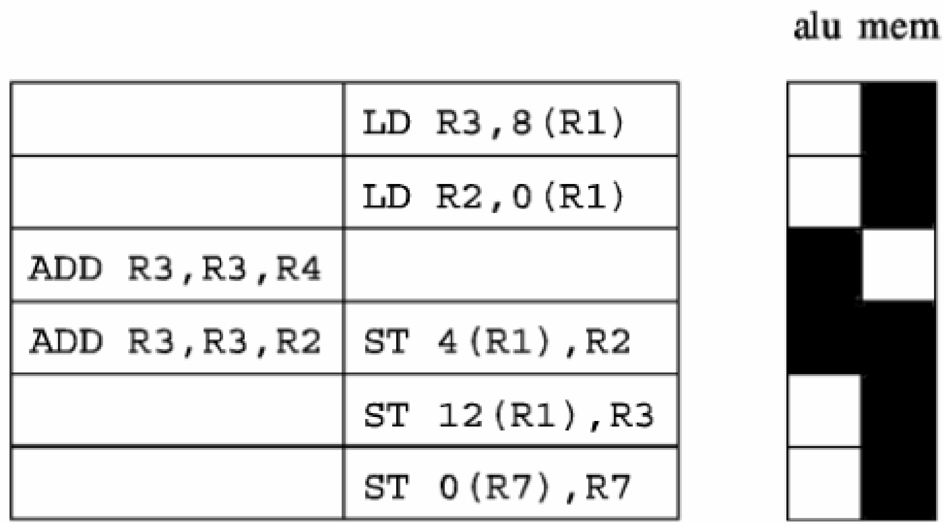
- Mimari, LD işleminden 1 saat çevrimi sonra aynı bellek alanına ST işleminin yapılmasına izin verir. Bu nedenle ilk ve son komutlar arasındaki kenarın değeri 1'dir.

- Diğer 1 değerli kenarlar, bağımlılık ile açıklanabilir.

1.1.5.2 Algoritma

```
HAZIR = önceleyen 0 düğüm olan düğüm var
HAZIR bos olana dek dön {
     HAZIR listesindeki en öncelikli düğümü n olarak seç
     n'i kaynak ve veri kısıtlamaları dahilinde en erken
     işlenebileceği yere yerleştir
     n'den sonra gelen düğümlerin önceleyenlerini 1 azalt
     HAZIR'i güncelle
}
```

Sonuçta elde edelen kod **Şekil 2'**deki gibidir.



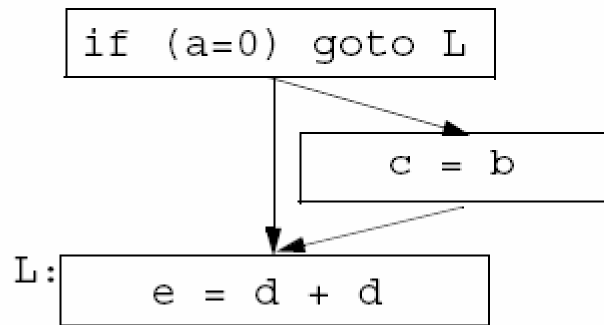
Şekil 2 – Algoritma Sonucu Oluşan Kod

1.1.6 Global Kod Sıralama

Komut düzeyinde paralellikten tam yararlanmak için sadece temel blokları sıralamak kaynakların etkin kullanılması bakımından etkin olmayabilir. Bu nedenle temel bloklar arasında da sıralama yapılmalıdır. Bu işlemde 2 adet kısıt vardır:

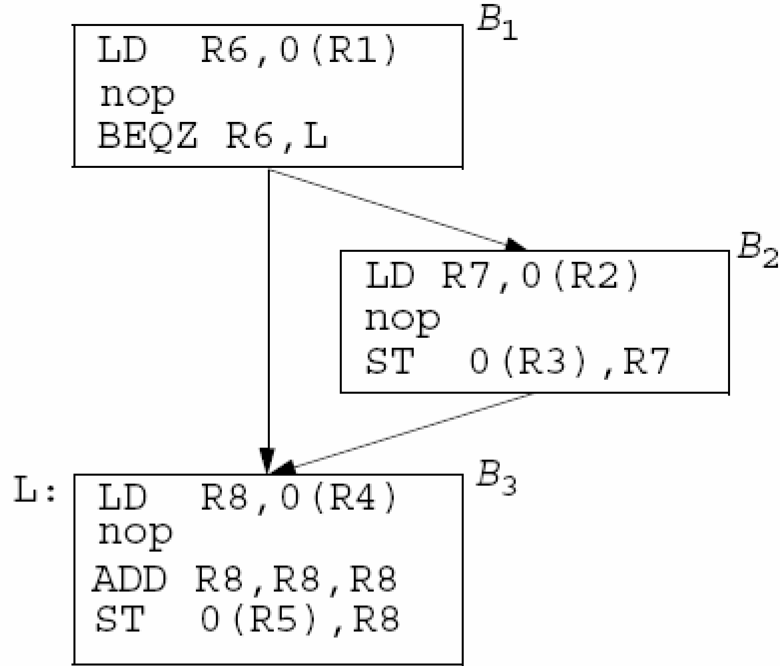
- Orjinal programdaki tüm komutlar, optimize edilmiş programda da bulunmalıdır.
- Optimize edilmiş programda yürütülen spekülatif komutlar, herhangi bir yan etkiye yol açmamalıdır.

Örneğin, basit kod sıralama tekniğinin gösterildiği mimarinin benzeri bir mimariye sahip bir işlemcide aşağıdaki şekilde bir kod bloğu yazılsın:



Şekil 3 – Global Kod Sırama (Örnek Kod)

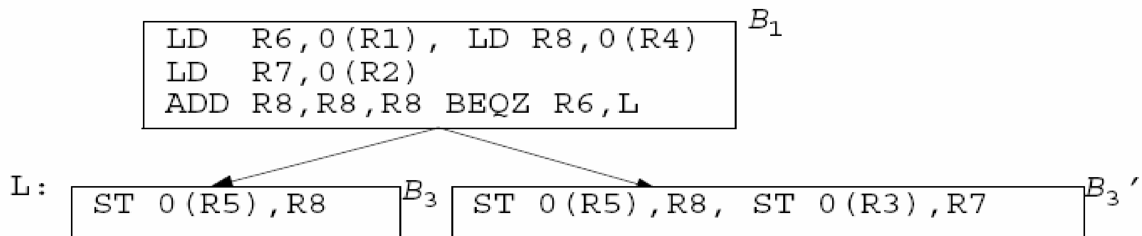
Program kodu, makine komutları olarak aşağıdaki şekle dönüştürülür:



Şekil 4 – Derleyici tarafından oluşturulmuş makine kodu

a, b, c, d, e değişkenlerinin sıra ile R1, R2, R3, R4, R5 değişkenlerine atandığını farz edelim. 3 temel blok arasında herhangi bir veri bağımlılığı yoktur. B_3 bloğundaki tüm işlemler, dallanma olsa da olmasa da yürütülürler. Bu nedenle B_1 bloğundaki işlemler ile birlikte paralel olarak yürütülebilir. B_1 bloğundaki işlemleri B_3'e taşıyamayız çünkü B_2 bloğunun işlenmesi, B_1 bloğunun sonucuna bağlıdır.

B2 bloğundaki LD işlemi spekülatif olarak yürütülebilir. Böylece, kaynak boş kaldığı durumda işlem yapılarak, eğer B_2'nin işletilmesine karar verilirse LD işlemi zaten yapılmış olacağı için hızlanma elde edilir. ST işlemi aşağı bloğa indirilemez çünkü B_2 dallanması gerçekleşmeyebilir. Bu durumda B_3 bloğundaki ST komutu kopyalanarak bir dallanma daha açılır. Bu dallanmalardan ikisinde de ST komutu işlenir ama sadece birisinde B_3 bloğuna dair ST komutu işlenir.



Şekil 5 – Global Kod Sıralama ile optimize edilmiş kod

1.1.6.1 Yukarı Kod Taşıma

Hedeften geçen ama kaynaktan geçmeyen bir kod yolu olması durumu: Bu durumda fazladan bir işlem yürütülür. İşlem yan etkisiz olmadığı sürece bu kod taşıma işlemi hatalıdır. Eğer kontrol akışı kaynaktan geçerse yararlıdır.

Kaynaktan geçen ama hedeften geçmeyen bir kod yolu olması durumu: Bu durumda hedeften geçmeyen yollara işletilecek kodun kopyası yerleştirilmelidir. Fakat bu işlemin aşağıdaki kısıtları vardır:

- Komutun operandları orjinal komutla aynı olmalı.
- Halen ihtiyaç duyulan bir verinin üzerine yazmamalı.
- Kaynak noktasına varmadan kendi sonucunun da üzerine yazılmamalıdır.

Hedeften geçen tüm yolların kaynaktan geçmesi ve kaynağa giden tüm kod yollarının hedeften geçmesi: Kod taşınması güvenlidir ve fazladan işlem yürütme gerekmez.

1.1.6.2 Aşağı Kod Taşıma

Hedefe ulaşan ama kaynaktan geçmeyen bir kod yolu olması durumu: Bu durumda bir eksta işlemin yürütülmesi gerekir. Hedefe giden kontrol akışındaki temel blokların kopyası alınır ve ekstra işlem bu ikinci yolun hedefine eklenir.

Kaynaktan geçen ama hedeften geçmeyen bir kod yolu olması durumu: Hedeften geçmeyen yollara, kod eklenmesi gerekir.

Hedeften geçen tüm yolların kaynaktan geçmesi ve kaynağa giden tüm kod yollarının hedeften geçmesi: Kod taşınması güvenlidir ve fazladan işlem yürütme gerekmez.

1.1.6.3 Global Kod Sıralama Algoritmaları

Alan Bazlı Kod Sıralama

Bu algoritma, kod sıralamasının iki çeşidinden yararlanır.

1. Birbirlerine kontrol – eşit temel bloklarda (hedeften geçen tüm yolların kaynaktan geçtiği ve kaynağa giden tüm yolların hedeften de geçmesi) işlemlerin taşınması.
2. İşlemlerin spekülatif olarak bir dal yukarıda kendisinden önce gelen ve kendisini domine eden (hedeften geçen tüm yolların kaynaktan da geçmesi) hedefe taşınması.

Algoritmada tüm temel bloklar topolojik sıralarına göre ziyaret edilirler. Bu şekilde her komutun yeri belirlenmeden önce, komutun bağımlı oldukları komutlar yerlerine yerleştirilmiş olur. Bir B temel bloğunda yer alacak komutlar ya B'ye kontrol – eşit olan bloklardaki kodlardan, ya da B'nin domine ettiği bloklardan seçilir.

Her temel bloğun sıralamasını oluşturmak için aday komutların bir listesi oluşturulur. Aday komutlar listesinde bulunan tüm komutların bağımlı oldukları komutların önceden sıralanmış olması gerekir. Saat saat tek tek komutlar yerine yerleştirilir.

Döngü Açma

Global kod sıralamada, döngüler kod taşıma konusunda engel teşkil edebilirler. Bu metotta döngüler bir miktar açılarak derleyicinin paralellikten daha fazla yararlanması sağlanır. Örneğin aşağıdaki gibi bir for döngüsü:

```
for (i = 0; i < n; i++)
     S(i);
```

Döngü 4 defa açılarak aşağıdaki halde daha fazla paralelleştirilebilir hale gelir:

```
for(i = 0; i + 4 < n; i += 4)
{
     S(i);
     S(i+1);
     S(i+2);
     S(i+3);
}
for(; i< n; i++)
     S(i);
```

1.2 Yazılım İş Hatları (Software Pipelining)

Yazılım iş hattı, donanımdaki iş hatlarını yazılım tarafında gerçekleyen bir derleyici optimizasyonudur. Komut düzenlenmesi derleyici tarafından otomatik olarak yapılır. Yazılım iş hattı yöntemi, genellikle döngü açma optimizayonu ile birlikte gerçeklenir.

1.2.1 Yazılım iş hattı için örnek bir mimari

Yazılım iş hattının gerçekleneceği mimaride aşağıdaki özelliklerin bulunduğu varsayılacaktır:

1. İşlemci bir saat çevriminde bir yükleme (load), bir yazma (store), bir aritmetik işlem ve bir dallanma işlemi yapabilir.
2. İşlemci döngünün başına aşağıdaki komutla döner. R saklayıcısının değerini 1 azaltır ve 0 olana dek L'ye dallanır.

BL R, L

3. Bellek işlemlerinin bir otomatik arttırarak adresleme modu vardır, saklayıcı değerinden sonra ++ getirerek gerçeklenir.

4. Aritmetik işlemler tamamen iş hattında yürütülürler, her saat çevriminde yeni aritmetik işlem verilebilir ve sonuçlar 2 saat çevriminde alınır. Diğer komutların sonuçları tek saat çevriminde alınabilir.

1.2.2 Yazılım İş Hattı Yöntemi

```
for(i = 0; i < n; i++)
   D[i] = A[i] * B[i] + c;
```

Şeklinde bir döngünün her bir çevrimi için üretilecek en iyi kod aşağıdaki gibidir:

```
L:   LD   R5, 0(R1++)                // R1 = &A
     LD   R6, 0(R2++)                // R2 = &B
     MUL  R7, R5, R6
     Nop
     ADD  R8, R7, R4                 // R4 = c
     Nop
     ST   0(R3++), R8  BL      R10, L    // R =&D, R10=n-1
```

Bu kodda paralellik az düzeyde olmasına rağmen, k adet döngü açma optimazyonu yapılarak paralel çalışma elde edilebilir. Her bir döngü 2k+5 saat çevriminde gerçekleşir. k değeri arttırılarak daha paralel çalışma sağlanabilir ama bu sefer de kod boyutu oldukça artar. Döngünün 4 kere açılmış şekilde çalışması aşağıdaki gibidir:

Saat	L:	LD				
1		LD				
2			LD			
3		MUL	LD			
4			MUL	LD		
5		ADD	LD			
6			ADD		LD	
7		ST		MUL	LD	
8			ST			
9				ADD	MUL	
10						
11				ST	ADD	
12						
13					ST	BL (L)

Yukarıdaki açılımda, her döngü 13 saat çevrimi sürer ve döngü başa döndüğünde iş hattı boş şekilde tekrar başlar. Yazılım iş hattı yöntemi ise döngünün başında iş hattını tekrar boş başlatmadan sürekli dolu devam ettirmektir. Bu yolla işlemcinin aynı anda işleyebileceği maksimum iş kapasitesinden faydalanılmış olur. Aşağıdaki şekilde aynı döngünün 5 defa açılmış haldeki kodları görülmektedir. Döngü 5 defa açılınca oluşan makine kodu aşağıda görüldüğü gibi olur.

Saat Çevrimi	J = 1	J = 2	J = 3	J = 4	J = 5
1	LD				
2	LD				
3	MUL	LD			
4		LD			
5		MUL	LD		
6	ADD		LD		
7			MUL	LD	
8	ST	ADD		LD	
9				MUL	LD
10		ST	ADD		LD
11					MUL
12			ST	ADD	
13					
14				ST	ADD
15					
16					ST

Kodda dikkat edilirse 7 ve 8. satırlar, 9 ve 10. satırlar ile aynıdır. 7 ve 8. satırlar j=1 ve j=4 arası döngülerin kodundaki işlemleri yaparken, 8 ve 9. satırlar j=2 ve j=5 arası döngülerin kodundaki işlemleri yaparlar. Aslında işlem sonsuza kadar devam ettirilirse, sürekli aynı işlemler bir iterasyon çıkarılıp bir iterasyon ekleniyormuş gibi sürekli tekrarlanır. Bu tekrarlanma işlemi döngü sonlanana kadar sürekli yapılırsa, iş hattı boşalmadan oldukça etkin bir çalışma sağlanabilir. Bu yönteme yazılım iş hattı yöntemi adı verilir. Yöntem uygulandıktan sonra oluşan kod aşağıdaki gibidir:

Saat						
1		LD				
2		LD				
3		MUL	LD			
4			LD			
5			MUL	LD		
6		ADD		LD		
7	L:			MUL	LD	
8		ST	ADD		LD	BL (L)
9					MUL	
10			ST	ADD		
11						
12				ST	ADD	
13						
14					ST	

7 numaralı saat çevrimine kadar olan kısma *prolog* adı verilir. Bu çevrime kadar olan kısımda iş hattı dolmaktadır.İş hattını bir iterasyon terkettiği zaman yeni bir iterasyon iş hattına verilir. Bu sayede iş hattı boşalmadan döngü sonlanana kadar 2 saat çevrimlik

döngüsüne devam eder. Döngü sonlandığı zaman iş hattının boşalma safhası başlar. Bu sonlanma safhasına da *epilog* adı verilir.

7 ve 8 numaralı işlemlerde X numaralı iterasyon LD işlemleri yapar. Ardından Y numaralı iterasyon iş hattında 7 ve 8 numaralı işlemleri yürüterek LD işlemleri yapar. Y, 7. saat çevriminde ilk LD işlemini yaparken X de MUL işlemini yapar. Ardından 2 saat çevrimi bekler. Normalde 1 saat çevrimi yapması yeterlidir. Fakat LD işlemlerinin 2 saat çevriminde bir iş hattına girebilmesi için 1 saat çevrimi fazladan bekleyebilir. Diğer iterasyonda, Y 7 numaralı çevrimde MUL işlemini yaptıktan sonra X 8 numaralı çevrimde ADD işlemini yapar. Sonraki iterasyonda da 8. saat çevriminde X ST işlemini yapar ve iterasyonunu sonlandırır. Y ise ADD işlemini yapar ve bir sonraki 8 numaralı saat çevriminde ST işlemini yapar ve iterasyonunu sonlandırır. Bu işlem, *prolog* ve *epilog* dışındaki tüm çevrimler için geçerlidir.

1.2.3 Yazılım İş Hatlarının Hedef ve Kısıtlamaları

Yazılım iş hattı yönteminin en önemli hedefi, döngüde birim zamanda yapılan iş sayısını maksimuma çıkarmaktır. İkinci hedef ise üretilen kod miktarını mantıklı derecede küçültmektir. Bu hedefe ulaşmak için yapılması gereken, prolog ve epilog kısımları arasındaki tekrarlanan kısmı minimumda tutmaktır.

Yazılım iş hatlarında en önemli 2 kısıt kaynak paylaşımı ve veri bağımlılığıdır.

Kaynak Kısıtlamaları

Bir iterasyon boyunca makinede var olan kaynaklardan daha fazlası kullanılamaz. Eğer açılan döngü sayısı, makinede var olan kaynaklardan daha azını kullanıyorsa, döngü bir kaç kez daha açılabilir. Fakat kaynak yoksa açılamaz.Örneğin, daha önce verdiğimiz örnekte işlemci 1 saat çevriminde 1 yükleme, 1 yazma, 1 aritmetik işlem, 1 dallanma işlemi yapabilir. Fakat 1 yükleme işlemi 2 saat çevrimi sürdüğü için yazılım iş hattı yöntemi kullanılarak oluşturulmuş bir döngü 2 saat çevriminden az süremez.

Veri Bağımlılığı Kısıtlamaları

Döngülerde veri bağımlılıkları daha önce gördüklerimizden farklıdır, çünkü döngü değerleri bir önceki döngüye bağımlı olabilecekleri için çevrimler oluşturabilir. Yazılım iş hattı optimizasyonu yapılmış bir döngünün çevrim sayısı, bağımlılığın çözülmesi için gerekli olan çevrim sayısından az olamaz. Örneğin aşağıdaki döngüyü ele alalım:

```
for (i = 0; i < n; i++)
      *(p++) = *(q++) + c
```

Bu döngüde p ve q işaretçilerinin aynı bellek alanını gösterme ihtimalleri olduğu için aralarında veri bağımlılığı vardır. Bu nedenle tüm yazma ve okuma işlemleri sırayla yapılmak zorundadır. Her bir döngü çevrimi, bu işlemlerin toplam yapılma süresinden az olamaz.

2. İşçik Düzeyinde Paralellik

İşçik seviyesinde paralellik, bir işlemcinin aynı anda birden çok ipçiği çalıştırabilme yeteneğidir. Aynı anda birden çok işçik sayesinde programlar giriş-çıkış ve bellek işlemleri gibi bloke olabilecekleri işlemlerde bloke olmadan, başka bir işçik sayesinde çalışmaya devam edebilirler. Çok çekirdekli işlemcilerin çıkması ile birlikte işçik seviyesinde paralellik daha çok önem kazanmıştır. İşçik seviyesinde paralellik maksimum derecede arttırılarak işçiklerin çekirdeklere yayılarak programların daha hızlı çalışması sağlanmak için yeni metodlar ortaya çıkmıştır.

2.1 Otomatik Döngü İşçikleri Oluşturma (Automatic Loop Multithreading)

Intel derleyicilerinde bulunan bu yöntem, Intel Pentium 4 ve Intel Xeon tabanlı işlemcilerde oldukça iyi performans kazancı sağlar. Bu yöntemle, program kodunun seri olarak düzenlenmiş kısımları otomatik olarak farklı işçiklerde çalıştırılabilecek duruma getirilir. Özelliklerde döngülerde etkin olan bir yöntemdir. Döngüler arasındaki veri bağımlılıkları kontrol edilerek, döngü birkaç döngü haline getirilir ve farklı işçiklerde çalışması sağlanır. Aşağıda orjinal program kodu ve paralelleştirilmiş program kodu görülebilir:

```
subroutine serial(a, b, c)
     integer, dimension(100) :: a, b, c
     do i=1,100
          a(i) = a(i) + b(i) * c(i)
     enddo
end subroutine serial
subroutine parallel(a, b, c)
     integer, dimension(100) :: a, b, c
     ! Thread 1
     do i=1,50
          a(i) = a(i) + b(i) * c(i)
     enddo
     ! Thread 2
     do i=51,100
          a(i) = a(i) + b(i) * c(i)
     enddo
end subroutine parallel
```

Bu dönüşümü yapabilmek için derleyici aşağıdaki adımları takip eder:

- Seri koddaki tüm döngüler belirlenir ve bir döngü hiyerarşi yapısı oluşturulur. Bu sayede döngüler arasındaki ilişkiler, döngü miktarı, döngünün başı ve sonu belirlenir.

- Döngüler içerisinde veri bağımlılığı analizi yapılır. Döngünün farklı iterasyonları arasında veri bağımlılığı bulunmayan döngüler, paralelleştirilebilir olarak işaretlenir.

- Paralelleştirilebilir olarak tanımlanan döngülerin, paralelleştirildikleri durumda getirecekleri performans kaybı ve kazancı hesaplanır. Eğer işçik oluşturma işlemi sisteme yarardan fazla yük getirecek ise paralelleştirmeden vazgeçilir.

2.2 İş Hattı Yöntemi

Bir çok döngüde, döngüler arasında veri bağımlılıkları bulunabilir. Bu veri bağımlılıklarını çözmek için kullanılabilecek bir yöntem iş hatlarının kullanımıdır. İş hatlarında, işlem farklı işçiklere farklı işlemcilerde işlenmesi için dağıtılabilir. İş hattının her adımında paylaşılan bir değişken bir işlemciden, hemen ondan sonra gelen diğer işlemciye geçer. Örneğin,

```
for(i = 1; i <= m; i++)
   for(j = 1; j <=n; j++)
      X[i] = X[i] + Y[i,j]
```

döngüsünü ele alalım. Döngünün her adımı, veri bağımlılığı dolayısıyla işlemci tarafından art arda sırayla işlenmelidir. Fakat toplama işlemi bir iş hattı kullanılarak farklı işçiklere ve işlemci çekirdeklerine aşağıdaki gibi dağıtılabilir. İş hattının her adımında, veri bağımlılığı olan değişkenin bağımlılığı çözülür ve diğer adımda diğer işlemcinin aynı değişken üzerinde işlem yapabilmesi sağlanır.

Time	Processors		
	1	2	3
1	$X[1]{+}{=}Y[1{,}1]$		
2	$X[2]{+}{=}Y[2{,}1]$	$X[1]{+}{=}Y[1{,}2]$	
3	$X[3]{+}{=}Y[3{,}1]$	$X[2]{+}{=}Y[2{,}2]$	$X[1]{+}{=}Y[1{,}3]$
4	$X[4]{+}{=}Y[4{,}1]$	$X[3]{+}{=}Y[3{,}2]$	$X[2]{+}{=}Y[2{,}3]$
5		$X[4]{+}{=}Y[4{,}2]$	$X[3]{+}{=}Y[3{,}3]$
6			$X[4]{+}{=}Y[4{,}3]$

Şekil 6 – İş Hattı Yönteminin Uygulanışı

2.3 Bellek Kullanma Optimizasyonları

- Derleyicinin, işlemi farklı işçiklere ayırırken bir ipçiğin mümkün olduğunca aynı bellek alanlarını kullanmasını sağlamasıdır. Böylece cep belleğin etkin bir şekilde kullanılması sağlanır. Ayrıca işlemciler arasında haberleşme masrafı da azalmış olur.

- Derleyici, bir ipçiğin işlediği verilerin bellekte mümkün olduğunca birbirine yakın olmasını sağlar. Bu şekilde yerellik artar ve kodun daha optimize bir şekilde çalışması sağlanır.

3. Bellek Düzeyinde Paralellik

Bellek düzeyinde paralellik, belli bir işlem anında cep bellek ıskası gibi beklemede olan birden fazla bellek işlemi bulunmasını ifade eder.

3.1 Okuma Iska Gruplaması (Read Miss Clustering)

Düzensiz sırada komut işleyen işlemciler komut pencerelerindelerinde komutlar işlerini tamamladıktan sonra komut penceresinden çıkarlar. Bu duruma bir istisna yazma işlemleridir çünkü yazma işlemi tamponlanarak daha sonra tamamlanabilir. İşlemci ve bellek arasındaki hız farkından dolayı, bir cep ıskası yüzlerce işlemci çevrimi sürebilir. Komut penceresinin sonuna ulaşan bir okuma işlemini ele alalım. Eğer okumadan sonra gelen komutlar yeterince hızlı işleniyorlarsa veya yazma işlemi iseler cep ıskasından dolayı meydana gelen gecikmeyi yeterince kapatamazlar ve işlemciyi yeterince meşgul edemezler. Daha sonraki işlemler bitmek için baştaki okuma işleminin bitmesini bekleyecekleri için komut penceresi dolar ve işlemci bloke olur.

Bu soruna getirebilecek bir çözüm, cep ıskasına neden olacak birden fazla okuma komutunu komut penceresine denk getirmektir. Bu şekilde okuma beklemelerinin üst üste binmesi sağlanarak program hızında yükselme meydana gelir.

Bu optimizasyonda göz önünde bulundurulması gereken kısıtlama, işlemcide bulunan cep belleğin büyüklüğüdür. Her bir okuma işlemi için cep belleğin bir gözünün kullanılması amaçlandığı için okuma işlemleri cep belleğin boyutu aşılacak şekilde gruplanırsa, programda yerellik kötü etkilenir.

Şekil 7'de gruplamadan önce ve sonra oluşan kodlar görülmektedir. Dış döngü, iç döngünün içerisine açılmıştır ve **Şekil 8**'deki gibi bir çalışma elde edilmiştir.

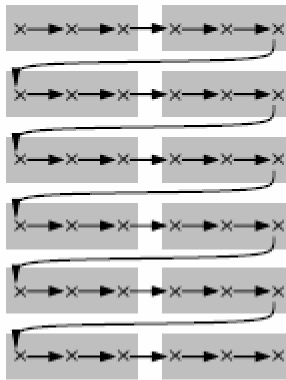
```
for(j=0;j<Io;j++)
 for(i=0;i<Ii;i++)
  ...A[j,i]
```

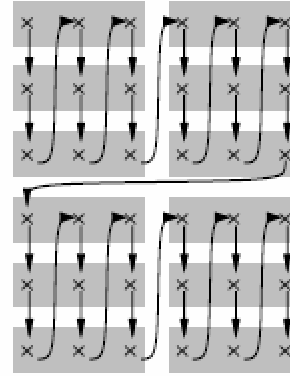
(a) Original code

```
for(j=0;j<Io;j+=N)
 for(i=0;i<Ii;i++)
  ...A[j,i]
  ...A[j+1,i]
  ......
  ...A[j+N-1,i]
```

(b) After clustering alone

Şekil 7 – (a) Gruplamadan Önce (b) Gruplamadan Sonra Kod





Şekil 8 – (a) Gruplamadan Önce (b) Gruplamadan Sonra Çalışma

3.2 Yazılımsal Önceden Getirme (Software Prefetching)

Derleyici tarafından, cep bellek ıskası olacağı tahmin edilen komutların önceden yürütülmesi işlemidir. İlk olarak, kaynak kodu analiz edilerek cep bellek ıskasına neden olacağı tahmin edilen (genellikle bir bellek dizisinin başlangıç referansları) komutlar belirlenir. Daha sonra, bu bellek adreslerini önceden getirmek için gerekli olan komutlar yerleştirilir.

4. Referanslar

1. *Compilers: Principles, Techniques and Tools 2nd Edition*. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. Addison-Wesley, 2006

2. “Instruction Level Parallelism”, http://en.wikipedia.org/wiki/Instruction_level_parallelism

3. “Software Pipelining”, http://en.wikipedia.org/wiki/Software_pipelining

4. *“Compiler Techniques for Concurrent Multithreading with Hardware Speculation Support”,* Zhiyuan Li,Jenn-yuan Tsai,Xin Wang,Pen-chung Yew,Bess Zheng, 1996

5. *“Exploiting Thread-Level and Instruction-Level. Parallelism for Hyper-Threading Technology”,* Xinmin Tian, Aart Bik, Milind Girkar, Paul Grey, 2003

6. *“Comparing and Combining Read Miss Clustering and Software Prefetching”*, Vijay S. Pai, Sarita V. Adve, Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on Volume , Issue , 2001 Page(s):292 – 303

7. “Memory Level Parallelism”, http://en.wikipedia.org/wiki/Memory_level_parallelism

8. *“Code Transformations to Improve Memory Parallelism”,* Vijay S. Pait, Sarita Advel