# EAFT: Evolutionary Algorithms for GCC Flag Tuning

Burak Tağtekin
*Machine Learning*
*Engineer Mef University,*
*INSIDER* Istanbul, Türkiye
tagtekinb@mef.edu.tr

Tuna Çakar
*Comp. Eng. Dept.*
*Mef University, TAM Finans A.Ş.*
Istanbul, Türkiye
cakart@mef.edu.tr

2022 7th International Conference on Computer Science and Engineering (UBMK) | 978-1-6654-7010-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/UBMK55850.2022.9919557

***Abstract*—Due to limited resources, some methods come to the fore in finding and applying the factors that affect the working time of the code. The most common one is choosing the correct GCC flags using heuristic algorithms. For the codes compiled with GCC, the selection of optimization flags directly affects the speed of the processing, however, choosing the right one among hundreds of markers during this process is a resource consuming problem. This article explains how to solve the GCC flag optimization problem with EAFT. Rather than other autotuner tools such as Opentuner, EAFT is an optimized tool for GCC marker selection. Search infrastructure has been developed with particle swarm optimization and genetic algorithm with diffent submodels rather than using only Genetic Algorithm like FOGA.**

***Keywords*—Compiler optimization, Compiler flags, Autotuning, Optimization, Genetic algorithm, Particle Swarm Algorithm**

## I. Giriş

GCC gibi derleyicilerin en temel işlevi C, C++ gibi yüksek seviyeli dillerde yazılmış olan programın çalıştırılabilir ikili dosyalara dönüşmesini sağlamaktır [1]. Derlenmiş kodun çalışma süresinde iyileşme sağlamak için, derleyici içerisinde birçok optimizasyon işaretleyicisi bulunmaktadır. Bu işaretleyicilerin sayısı GCC versiyonuna bağlı olarak değişmekle birlikte, GCC versiyon 9.3 içerisinde kullanıma hazır olan tam 114 adet işaretleyici mevcuttur [2]. Daha yeni versiyonlarında ise örneğin GCC 11'de tam 232 adet işaretleyici bulunmaktadır [3]. Bunlardan birini örnek vermek gerekirse `unroll-all-loops`, döngü içerisindeki kontrol yapılarını azaltarak ya da ortadan kaldırarak kodun çalışma zamanını azaltmayı amaçlar [4]. Optimizasyonun etkisi kodun yapısına bağlı olduğundan aynı optimizasyon işaretleyicileri farklı kodlar üzerinde aynı oranda çalışma zamanında iyileşme sağlamayabilmektedir [5]. Örneğin yukarıda belirtilen `unroll-all-loops`, döngü içermeyen bir koda uygulandığında herhangi bir optimizasyon sağlamayacaktır. İşaretleyici seçimi kodu derleyecek olan kullanıcı tarafından yapılmaktadır [6]. Tüm işaretleyicilerin açık ve kapalı olarak iki seçeneği olduğu düşünüldüğünde tüm seçimlerin sayısı $2^{232}$ olmaktadır ve kullanıcıların bütün işaretçilerin anlamlarını bilip, tüm kombinasyonların etkisini anlaması ve bunların içerisinde de doğru olanı seçmesi pek mümkün olamayabilmektedir [7].

İşaretleyicilerin kullanımını kolaylaştırmak için GCC geliştiricileri `O1, O2, O3` gibi optimizasyon seviyeleri tanımlamışlardır. Bu seviyeler `unroll-all-loops` gibi sayıları 100'den fazla olan işaretçilerin bazılarını açıp bazılarını kapatarak kodu belirli düzeyde optimize etmeyi amaçlar. Bölüm III içerisinde bu seviyelerin amaçlarından bahsedilmiştir. Bu seviyeler ile kodun, kullanıcı tarafından belirli bir düzeyde optimize edilmesi sağlanır ancak bütün C, C++ geliştiricilerinin yazdıkları kodların birbirinden ne kadar farklı olduğu düşünüldüğünde, her kod için istenilen çalışma zamanı performansını elde etmeyi garanti edebilecek bir optimizasyon seviyesi yoktur [8]. Bu sebeple önceden tanımlanmış optimizasyon seviyelerinin kullanılmasından ziyade, istenilen çalışma zamanı performansını sağlamak için her bir koda özgü optimizasyon işaretleyicileri seçilmelidir. EAFT açık kaynak kodlu GCC optimizasyon işaretleyicileri için özelleştirilmiş bir otomatik ayarlayıcı (autotuning) programıdır [9]. Kullanıcının geliştirdiği ve optimize etmek istediği kodu alıp ona en uygun, çalışma zamanını en uygun şekilde optimize edecek işaretçilerin hangileri olduğunu gösterir. Bunu yaparken içerisinde bulunan arama algoritmalarından yararlanır. Bu algoritmaların detaylarına IV içerisinde bahsedilmiştir. Kodunu optimize etmek isteyen kullanıcı, işaretleyicileri tek tek seçmek yerine sadece hangi arama algoritmasının onun için çalışmasını istediğini belirtir. EAFT içerisinde kullanılan bir diğer seçenek ise genetik algoritma içerisindeki modellerdir.Geliştirilen kullanıcı arayüzü ile de terminalden EAFT'ın hangi aşamada olduğu adım adım takip edilebilmektedir. Arayüzü ile ilgili detaylar bölüm 5'de yer almaktadır.

## II. Literatür Taraması

En iyi optimizasyon işaretleyicilerini bulmak birçok araştırmanın konusu olmuş, farklı yaklaşımlar ile sorunlar çözülmeye çalışılmıştır. Bazı çalışmalar EAFT gibi sezgisel arama algoritmaları kullanarak çözüm kümesinde arama yaparken, bazı çalışmalar da kodun yapısını analiz edip oradan çıkardığı sonuç ile en uygun işaretleyici setini bulmaya çalışmaktadır.

Opentuner [10], verilen problem özelinde, içerisindeki arama algoritmalarını kullanarak en uygun çözümü bulmaya çalışan bir otomatik ayarlayıcı (autotuning) programıdır. Sa-

978-1-6654-7010-0/22/$31.00 ©2022 IEEE

dece GCC işaretleyici optimizasyonu değil, genel arama problemlerine hitap ettiğinden içerisindeki bazı arama algoritmaları, parçacık sürü optimizasyonu gibi, ikili arama problemleri için özelleştirilmemiştir. Mevcut problemlerin geneline uygun olacak şekilde uygulama içerisine dahil edilmiş bu algoritmalardan ziyade EAFT GCC işaretleyici optimizasyonuna uygun ikili arama algoritmalarını kullanılmaktadır. Genetik algoritmanın sadece nesil modelini kullanan Opentuner'a kıyasla EAFT içerisine 5 farklı model dahil edilmiştir. Python ile yazılmış olan Opentuner'a kiyasla EAFT GOLANG ile yazılmıştır. Go dilinin Python'a göre paralel çalışma ve eşzamanlılık gibi konularda daha hızlı olması ile birlikte, arama algoritmasinin çalışma süresinde kısalma sağlanmıştır [11]. Benzer bir çalışma Tağtekin ve diğerlerinin yaptığı FOGA [12]dır. Bu çalışmada Opentuner'dan farklı olarak yalnızca genetik algoritma kullanmakta olup çaprazlama yöntemleri, mutasyon yöntemleri ve seçim yöntemleri doğrultusunda problem özelinde çeşitlendirme sağlanmıştır. EAFT ise kullanıcıya genetik algoritma ile birlikte parçacık sürü optimizasyonu algoritması da sunar. EAFT kapsamında genetik algoritma içerisinde 5 farklı model bulundurken FOGA yalnızca 1 adet model bulundurmaktadır. Yukarıda bahsi geçen Golang ile yazılan kodun Python'a kıyasla daha avantajlı olma durumu EAFT vee FOGA için de geçerlidir. FOGA da Opentuner gibi Python ile yazılmıştır.

Genetik algoritma ile optimizasyon işaretleyicisi seçiminin en eski örneklerinden biri Cooper ve diğerleri tarafından yapılmıştır [13]. Yazarların amacı bu yazının amacından farklı olarak kod boyutunu optimize etmektir. Bu yöntem ile SPEC benchmarkında %40 daha az kod boyutuna optimize edilmiş kodlar elde edebilmişlerdir. Knijnenburg ve diğerleri [14] ise döngü derinliği konusunda kodu optimize etmek isterken, uygun işaretçi seçimini genetik algoritmaya bırakmıştır. Yazarlar genetik algoritmanın sadece nesil modelini kullanarak, Fortran77 ve g77 derleyicileri kullanılarak matris çarpımı, matris vektör çarpımı gibi kodlarla benchmark yapmışlardır. Leather ve diğerleri [15] kodun gramer yapısını inceleyerek modele feature oluşturma konusunda genetik algoritmadan destek almış ve optimizasyon işaretleyici seçimi konusunda geliştirdikleri yöntem ile %50'den fazla çalışma zamanında iyileşme yakalamışlardır. Gramer yapısı incelenerek kodun içerisinden 3 derinliğe sahip birleştirme (assembly) bloğu sayısı, toplam assembly basic block sayısı gibi özellikler oluşturmuşlardır. [16] Kodun statik özelliklerini kullanarak hazırladıkları Markov modeli ve genetik algoritma ile %22'lik bir çalışma zamanı iyileştirmesi yakalamışlardır. Bu statik özellikler üzerinden çok fazla çalışma yapılmıştır. [17] Fursin ve diğerleri yukarıda bahsedilen çalışmalardan farklı olarak MilepostGCC adı verilen bir araç geliştirmişlerdir. Bu araç kodun üzerinde statik analiz yaparak derleyici optimizasyon problemini çözmeye çalışmaktadır. Kodu çalıştırılabilir dosya haline getirmeden önce birleştiriciye (assembly) dönüştürülerek üzerinden bazı statik analizler yapmaktadır. Kaç adet temel blok (basic block) içerdiği, kaç adet kontrol yapısı ya da kaç adet döngü içerdiği gibi toplam 55 adet özelliği çıkarıp kodun yapısını analiz etmektedirler. Daha sonra da bu çerçevede kod için en uygun optimizasyon işaretleyicileri bunu tahminlemeye çalışmaktadırlar. Projenin yapım yılı ve kullanılan GCC versiyonunun eskiliği nedeniyle günümüzde pek kullanılmamaktadır [18]. Cavazos ve diğerlerinin yaptığı çalışma ise MilepostGCC gibi kodun statik değerleri üzerinden ilerleme yapmaktadır ancak bu kodun yapısından ziyade, kodun çalıştığı bilgisayarın kaynaklarını kullanımından yola çıkarak bir model eğitir. Modeli kodun kaç CPU döngüsünde (cycle) yürüttüğü, önbellek vuruş (cache hit) sayıları, ADD, MUL gibi aritmetik birleştirici işlemlerini kaçar kez yaptığı gibi 42 adet özellik ile eğitmişlerdir. Daha sonra bu modeli, kullanıcının vereceği yeni bir C kodu ile kullanarak ona en uygun optimizasyon işaretçilerini bulmaya çalışmışlardır [19]. Eunjung ve diğerleri ise yukarıda bahsi geçen iki çalışmayı birleştirerek yeni bir araç geliştirmişlerdir. Hercules adını verdikleri bu araç, kodun kaynak kullanım bilgilerini ve Milepost özelliklerini bir araya getirmektedir. Tüm bunlara ilave olarak Hercules, verilen kodun içerisindeki döngüleri de analiz edip, onların da türlerini kendi tanımladıkları 35 farklı kalıba göre derecelendirmektedir. LLVM-MCA [20] yukarıdaki statik analiz yapan programlara bir alternatiftir. Kodun çalışma süresi üzerinde tahminleme yapıp bunu en iyi optimizasyon işaretleyicisi bulmakta kullanan Ithemal [21] gibi çalışmalar da akademik literatürde mevcuttur. Bu araçlar kullanılarak kodun çalıştırılmadan çalışma süresi tahminlemesi yapılabilir ancak kodun birleştiricinin yapısının çalışma zamanında son haline gelmesinden ötürü gerçek çalışma süresi ile tahminlenen arasında korelasyon çoğu zaman olmamaktadır [22]. Ballal ve diğerleri [23] tüm GCC işaretleyicileri yerine çok daha küçük bir küme üzerinde çalışma yapmıştır. Klasik genetik algoritma modeli kullanıp sadece 36 adet işaretleyici özelinde genetik algoritma modeli denemişlerdir. Yukarıda da bahsi geçtiği üzere EAFT ile GCC versiyonumuz doğrultusunda toplam 232 adet işaretleyici üzerinden en iyileri ararken, toplam 5 adet genetik algoritma modeli, 2 adet evrimsel algoritma ve 4 adet çaprazlama yöntemi kullanılmıştır.

## III. GCC VE OPTİMİZASYON

Bölüm I içerisinde ön tanımlı optimizasyon seviyeleri açıklanmıştır. Bunların ne olduğundan detaylı olarak sunulacak olursa:

- **O0** : Kodun optimize edilmeden derlenmesini sağlayan komuttur. Hata ayıklama işleminin düzgün çalışmasını kontrol eder.
- **O1** : Bazı optimizasyon işaretleyicilerinin açılmasından sorumludur. GCC bu aşamada kod boyutunu ve çalışma süresini düşürmeye çalışır.
- **O2** : O1'den daha fazla optimize etmeye çalışır. Daha fazla işaretleyici kullanımından ötürü bu aşamada kodun derleme süresinde uzama görmek mümkündür. Ancak GCC burada da nihai amacımız olan çalışma süresinde kısaltma yapmamızı sağlamaya çalışır. Bunun için feda etmemiz gereken alanlardan birisi de hafıza kullanımıdır.
- **O3** : O2'den daha fazla işaretleyici açmaya çalışarak çalışma süresinin kısalmasını sağlar. O2 için bahsettiğimiz yan etkiler bu işaretleyici için de mevcuttur.

## IV. YÖNTEM

## V. EVRİMSEL ALGORİTMALAR

### A. Genetik Algoritma

EAFT içerisinde arama probleminin çözümünde metasezgisel bir yöntem olan genetik algoritma kullanılmıştır. Doğadan esinlenilerek oluşturulmuş genetik algoritma G.A. canlıların evrimsel süreçlerini örnek alır. Güçlü türlerin genlerini bir sonraki nesile aktarabilmesi, zayıf olanlarınsa zaman içerisinde silinmesi bu yaklaşımın en temel noktasıdır [24]. Sadece güçlü genlere sahip bireylerin eşleşmesinden ziyade, dış etkenler de göz önüne alınmış ve bu sebeple seçilme kriteri, mutasyon gibi değerler de algoritma içerisine katılmıştır [25]. G.A. içerisindeki tüm bireyler aynı canlılarda olduğu gibi kromozom gösterimine sahiptir ve bu şekilde temsil edilirler. Kromozom değerleri kullanılarak amaç fonksiyonu hesaplanmaktadır. Kromozom içerisindeki herbir gen sadece iki değer alabilir: $\in 0, 1$. Kromozom üzerindeki her bir gen bir optimizasyon işaretleyicisini temsil eder. Örneğin bölüm I'de GCC 9.3 içerisindeki toplam 114 adet işaretleyici olduğundan bahsedilmişti. Bu bilgi ile kromozom uzunluğunun 114 olacağını söyleyebiliriz. Genler ise buradaki her bir işaretleyiciyi temsil eder. Gen içerisindeki değer 0 ise işaretleyici kapalı, 1 ise açık anlamına gelmektedir. Amaç fonksiyonu ise her bir kromozomun içerisindeki işaretleyici bilgilerini alıp, kodu derleyip çalışma süresini kaydedecek. EAFT içerisine dahil edilmiş bir diğer algoritma olan parçacık sürü optimizasyonu ile G.A. kıyaslandığında Polybench datasetinin çoğunda genetik algoritmadan daha iyi sonuçlar vermiştir. Farklı kodlar üzerinde farklı sonuçlar almak mümkün olduğu gibi sonuçlardan ilerleyen aşamalarda bahsedilecektir.

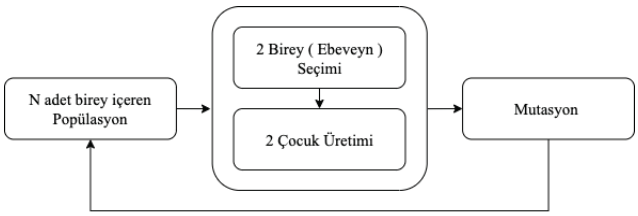
*1) Çaprazlama Yöntemleri:* Birey seçimlerinden sonra yavru üretimi için bir sonraki adım çaprazlamadır. Bu aşama yeni bir yavru üretimi için anne ve babadan alınacak olan kromozomların farklı yöntemler ile aktarılması sonucu oluşur [26]. EAFT içerisinde 4 farklı çaprazlama yöntemi kullanılmıştır:

- Partially Mapped Crossover
- Ordered Crossover
- Generalized N-point Crossover
- Cross Uniform Crossover

*2) Seçim Yöntemleri:* Seçim yöntemleri bir sonraki üretilecek olan popülasyonun aslında bir öncekinden daha iyi olması için yapılması umulan doğru ya da umut vaad edenleri bulmak üzerine olan bir seçim stratejisidir [27]. EAFT içerisinde iki adet seçim metodu kullanılmıştır:

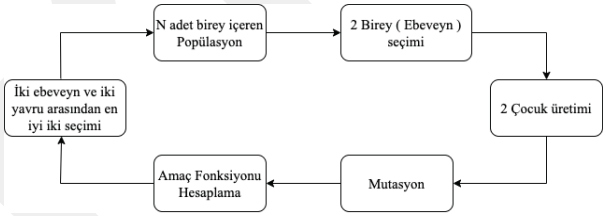
- Rulette Selection
- Tournament Selection

*3) Nesil Modeli:* Bu model en yaygın kullanılan genetik algoritma modellerinden birisidir. Başlangıç olarak n farklı yavru üretip n farklı birey içeren popülasyonu bu yavrular ile değiştirmeyi temel alır [28]. Yavrular, popülasyon içerisinden seçilecek olan 2 farklı bireyin çaprazlanması sonucu üretilmektedir. Bu işlem n adet yavru üretilene kadar devam etmektedir. Üretilen yavrular üzerinde etkili olacak olan mutasyon kullanıcının isteğine bağlı olarak olasılıksal olarak belirli bir oranda yapılacaktır. İşlem gücü açısından bir önceki modelden daha az kaynak tüketmekle birlikte, yeni üretilen yavruların potansiyelinin ortaya çıkartılması beklenir [29].



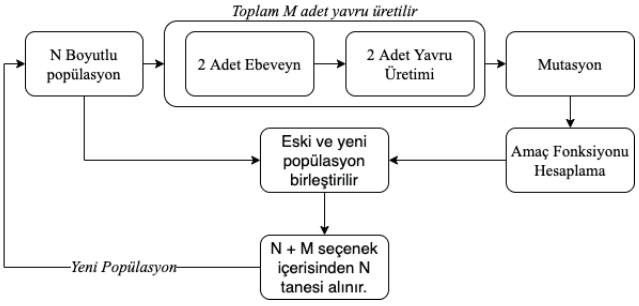

Şekil 1. Kuşak Modeli

*4) Denge Durumu Modeli:* Kuşak modelinden farklı olaran bu modelde üretilen yavrular direkt olarak popülasyonun yerini almaz. Ebeveynlerin üretilen yavrularını bir sonraki nesle eklemek yerine, iki ebeveyn ve iki yavru arasından en iyi iki birey popülasyona geri sürülür [28]. Bu şekilde popülasyon sayısı aynı kalmış olur. Bu işlem yapılırken popülasyona geri verilecek iki birey amaç fonksiyonuna göre seçilir. İyi ebeveynlerin kötü yavruları olması durumunda yavrular bir sonraki nesle aktarılamayacağı gibi bu durum yazılım içerisinde değiştirilerek yukarıda bahsedilen, yavruların potanyiselini ortaya durumunu çıkarması beklenebilir [29] [30].




Şekil 2. Denge Durumu Modeli

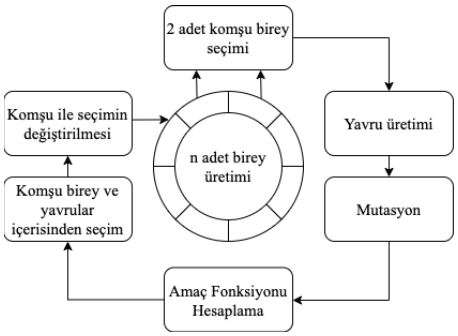
*5) Popülasyon Küçültme Modeli:* Bir önceki model ile kıyasladığımızda bu modelde popülasyon sayının başlangıçta üretilecek olan M adet yavru kadar büyümesi sağlanır ve daha sonrasında belirli bir seçim kriteri uygulanarak başlangıçtaki popülasyon sayısına geri düşmesi sağlanır. Burada uygulanacak olan seçim kriteri de önem kazanmaktadır [31].




Şekil 3. Popülasyon Küçültme Modeli

*6) Halka Modeli:* Bu modelde ise popülasyona yeni katılacak bireylerin üretilmesi popülasyondaki bireylerin komşuları seçilerek yapılmaktadır. Halka modeli üzerinde ilerleme tek yönülüdür. Üretilen yavrular ile ebeveynler karşılaştırıldıktan sonra en iyi sonucu verenler popülasyondaki yerlerini alır [32].

*7) Mutasyon Modeli:* Bu model ise diğerlerinden farklı olarak herhangi bir çaprazmala işlemi yapmaz. Çaprazlamanın yanı sıra sadece mutasyon ile bireylerin değişmesi sağlanır. Mutasyon olasılığının çok arttığı durumlarda rastsallık çok arttığından işaretleyici optimizasyonunda iyileşme yavaşlarken, düşük mutasyon olasılığında ise değişim olmadığından belirli bir aralıkta popülasyon kalmaktadır.




Şekil 4. Halka Modeli

### B. Parçacık Sürü Optimizasyonu

Parçacık sürü optimizasyonu (PSO) yaygın olarak kullanılan bir optimizasyon algoritmasıdır. Çıkış noktası olarak kuşların, arılar ve balıkların sosyal çevreleri incelenerek hareketlerinden esinlenilmiştir ve simüle edilmeye çalışılmıştır [33]. Genetik algoritmada olduğu gibi her birey bir ikili vektör şeklinde temsil edilmiştir. Parçacık sürü optimizasyonu sürekli problemler içerisinde kullanılsa da ikili değer problemleri için de çok sayıda araştırma yapılmıştır. İşaretleyici optimizasyonu problemi de bir ikili problem olduğu için yazılımının geliştirilmesinde genelden daha farklı bir yol izlemek gerekmektedir. Genetik algoritmadan farklı olarak burada bireylerin global ve kişisel en iyi değerleri de önem kazanmaktadır. Kennedy ve Eberhart'ın formülasyonuna göre [33] herbir parçacığın d boyutlu bir vektör içerisinde $X_i$=($x_{i_1}$, $x_{i_2}$, $x_{i_3}$ ..., $x_{i_d}$) konumunda bulunduğunu hızının da $V_i$=($v_{i_1}$, $v_{i_2}$, $v_{i_3}$ ..., $v_{i_d}$) olduğunu belirtelim. Bu durumda parçacıkların en iyi değerleri de $P_{i_{best}}$=($p_{i_1}, p_{i_2}, p_{i_3}$ ..., $p_{i_d}$) ve global olarak en iyi değerleri $P_{g_{best}}$=($p_{g_1}$, $p_{g_2}$, $p_{g_3}$ ..., $p_{g_d}$) bu şekilde gösterebiliriz. Sürüdeki bireyin hızının ve konumunun hesaplanmasında ise aşağıdaki formül kullanılmaktadır.

$$v_i(t+1) = w.v_i(t) + c_1\varphi_1(p_i - x_i(t)) + c_2\varphi_2(p_g - x_i(t)) \quad (1)$$

$$x_i(t+1) = x_i(t) + v_i(t+1) \quad (2)$$

$c_1$ ve $c_2$ pozitif sabitler olup, $\varphi_1$ ve $\varphi_2$ 0 ile 1 arasında rastsal değerlerdir. Denklem (1)'e ilave olarak denklem (3) kullanılarak olasılıksal olarak parçacığın hızı hesaplanır.

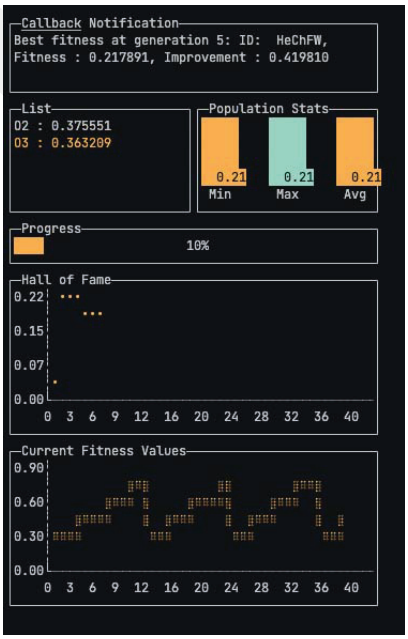
$$V_{ij}(t) = sig(v_{ij}(t)) = \frac{1}{1 + e^{-v_{ij}(t)}} \quad (3)$$

$$x_{ij}(t+1) = \begin{cases} 1, & \text{eğer } r_{ij} < sig(v_{ij}(t)) \\ 0, & \text{aksi halde.} \end{cases} \quad (4)$$

$r_{ij}$ $[0, 1]$ aralığında rastsal bir değerdir. Bu değişimler göz önünde bulundurulduğunda (2) yerine (4) kullanılarak parçacığın konumu belirlenecektir.

## VI. Kullanici arayüzü

EAFT için diğer araçlardan farklı olarak terminalden izlenebilen bir arayüz bulunmaktadır. Arayüzün bir örneği şekil 5'de görülebilir. Callback Notification k ısmında herbir popülasyon için değerler bulumaktadır. Improvement değeri O2 optimizasyon seviyesi ile karşılaştırılarak hesaplanmaktadır. List kısmında O2 ve O3 öntanımlı işaretleyicileri ile çalıştırılma hızları, Popülasyon Stats kısmında her bir popülasyonun minimum, maksimum ve ortalama amac¸ fonksiyonu görülebilmektedir. Progress kısmı ilerleme çubuğu olup Hall Of Fame her bir popülasyonun en iyi değerini göstermektedir. En alttaki çizelge ise anlık olarak o an çalışan bireylerin hesaplanan fitness değerlerini göstermektedir. Genel olarak gidişatı görmek için ve tüm genetik algoritma bitmeden takip edebilmek için bu kullanıcı arayüzü¨ geliştirilmiştir. EAFT aracının kullanılabilmesi için komut satırından bazı seçenekler verilmelidir. Örnek çalıştırma biçimi şu şekilde olmaktadır `gcc 2mm.c GA HM`. Burada `gcc` kullanıcının komut satırından GCC'i çağırdığı şekilde yazılcaktır. `2mm.c` optimize edilmesini istediği kod parc¸asının tam uzantısı. `GA` ya da `PSO` verilerek hangi ana algoritmayı kullanmak istediğini belirtecektir ve `HM` ile de genetik algoritma kullanıldığı durumda hangi modelin kullanılacağını belirtir.




Şekil 5. CLI

## VII. Veri seti ve çalışma ortamı

EAFT'ın performans ölçümü ve içerisindeki farklı metotların denenmesi Polybench kullanılarak yapılmıştır. Polybench içerisinde matris çarpımı, korelasyon ve kovaryans hesaplanması, gauss filtreleme, 1D-2D Jacobi hesaplama, LU faktörizasyonu ve vektör çarpımı gibi çok temel matematik problemlerinin C kodu ile yazılmış versiyonları vardır. Benchmarkta small-medium-large-extralarge olarak toplam 4 farklı input bulunmaktadır. Benchmark içerisindeki tüm kodlar bu ortak inputları kullanarak çalışmaktadır. Başlangıç olarak MEDIUM değeri atansa da kullanıcı hangi inputu kullanmak isteser değişim yapabilir. Deneyler Ubuntu 20.04'te çalışan 32 çekirdekli Intel Xeon işlemcili bir bilgisayarda yapılmıştır. Veri toplama aşamasında Polybench kodları çalıştırılmış olup en doğru sonucu almak için her kod toplam 5 kez çalıştırılmış, en düşük ve en yüksek değerler çıkarıldıktan sonra kalan 3 çalışma süresinin ortalaması alınmıştır. Bu sayede kodların çalışma süreleri üzerindeki hata payları düşürülmüştür. Genetik algoritma kütüphanesi olarak [31] örnek alınmıştır.

## VIII. Katki

GCC işaretleyici optimizasyonu alanında çok farklı çözümler yer almaktadır. Bunların detaylarından Bölüm II içerisinde bahsedilmiştir. EAFT olarak diğerlerinde olmayan ve katkı olarak sağlanılan bazı geliştirmelere aşağıda listelenmiştir. EAFT ile işaretleyici optimizasyonu alanında yapılan katkıları aşağıdaki çerçevede özetleyebiliriz:

- GCC derleyicisi çatısı altındaki işaretleyici optimizasyonu için direkt kullanıma hazır açık kaynaklı kod.
- Popülasyon istatistikleri ve genetik algoritmanın ilerleyişi ile ilgili çıktıların canlı verildiği komut satırı arayüzü.
- Farklı çaprazlama yöntemleri, seçim methodları ve genetik algoritma modelleri ile çeşitli çözüm yöntemleri.
- İkili kromozom temsiline uygun şekilde literatür araması yapılarak ile koda dökülmüş parçacık sürü optimizasyonu.

## IX. Sonuçlar

Tabloda Nesil Modeli NM, Denge Durumu Modeli DMM, Popülasyon Küçültme Modeli PKM, Halka Modeli HM ve Parçacık Sürü Optimizasyonu PSO olarak isimlendirilmiştir. O3 ise yukarıda bahsedilen ön tanımlı seçeneklerden birisidir. Ölçümler yapılırken Polybench veri setinin MEDIUM input seçeneği kullanılmıştır. 2mm kodu çalışma ortamımızda yaklaşık 3 saniye sürerken gramschimdt 40 saniye kadar sürebilmektedir. Bu durumun önüne geçip tablodaki okunaklılığı arttırmak için verisetindeki her problemin, 2mm ya da deriche vs gibi, O2 işaretleyicisi ile olan çalışma süresi 1 olacak şekilde oranlanmıştır. 2mm probleminin Halka Modeli ile olan sonucu 0.5237 olduğu tablodan görülebilir ve buradan şu sonuca varılmıştır: HM modeli O2'e kıyasla neredeyse yarı sürede çalışmıştır. Aynı şekilde floyd-warshall'ın O3 ile olan sonucu 1'den büyük bir değer olduğu için bu sonucun O2 ile kıyaslandığında, O2'den daha uzun sürdüğünü belirtir. O3 optimizasyon seviyesi O2'e kıyasla daha çok seçenek içerse de bazı problemler için O3 çalışma sürelerinin 1'in üstüne çıkarak

| | MODELLER | | | | | |
|---|---|---|---|---|---|---|
| *PROBLEM* | O3 | NM | HDM | PKM | HM | PSO |
| 2mm | 0.9979 | 0.6028 | 0.9867 | 0.9796 | **0.5237** | 0.7441 |
| 3mm | 1.0002 | 0.3709 | 0.9743 | 0.9756 | **0.3351** | 0.3875 |
| deriche | 0.9658 | 0.9779 | 0.9912 | 0.9931 | **0.9255** | 0.9706 |
| cholesky | 0.8242 | 0.8077 | 0.8458 | 0.8016 | **0.7994** | 0.8140 |
| jacobi-2d | 0.6272 | 0.2553 | 0.6297 | 0.6197 | **0.5347** | 0.6281 |
| durbin | 1.0056 | 0.8839 | 0.9380 | 0.8906 | **0.8320** | 0.9019 |
| mvt | 1.0088 | 0.9519 | 0.9624 | 0.9735 | **0.9028** | 0.9746 |
| floyd-warshall | 1.0862 | 0.8920 | 0.9573 | 0.9515 | **0.8423** | 0.9171 |
| correlation | 0.9900 | 0.9405 | 0.9897 | 0.9832 | **0.8204** | 0.9124 |
| gesummv | 1.0397 | 0.9668 | 0.9690 | 0.9676 | **0.8767** | 0.9611 |
| bicg | 0.9671 | 0.8289 | 0.9013 | 0.9353 | **0.7590** | 0.8304 |
| doitgen | 0.9646 | 0.5688 | 0.9809 | 0.4122 | **0.3835** | 0.8203 |
| atax | 1.0018 | 0.9671 | 0.9823 | 0.9628 | **0.9372** | 0.9817 |
| heat-3d | 0.6013 | 0.4912 | 0.5189 | 0.5074 | **0.4427** | 0.4607 |
| trisolv | 0.9640 | 0.9078 | 0.9517 | 0.9532 | **0.8833** | 0.8912 |
| syr2k | 0.7944 | 0.7324 | 0.9216 | 0.8030 | 0.7183 | **0.6775** |
| jacobi-1d | 0.9494 | 0.9241 | 0.9680 | 0.9467 | **0.9228** | 0.9308 |
| trmm | 0.9895 | 0.9672 | 1.0254 | 0.9641 | **0.9541** | 0.9766 |
| gemver | 1.0040 | 0.9434 | 0.9694 | 0.9569 | **0.8677** | 0.9273 |
| nussinov | 0.9894 | 0.8669 | 0.9292 | 0.9186 | **0.8554** | 0.8763 |
| covariance | 0.9851 | 0.7938 | 0.9884 | 0.9792 | **0.6933** | 0.9428 |
| ludcmp | 0.8371 | 0.6923 | 0.8302 | 0.7295 | **0.6777** | 0.6924 |
| adi | 0.9400 | 0.8040 | 0.9435 | 0.8542 | **0.7972** | 0.8110 |
| syrk | 0.8276 | 0.6804 | 0.6723 | 0.7015 | **0.6530** | 0.6686 |
| seidel-2d | 0.8834 | 0.2201 | 0.3932 | 0.4296 | **0.1999** | 0.2279 |
| symm | 0.9445 | **0.8459** | 0.9729 | 0.8659 | 0.8627 | 0.8602 |
| fdtd-2d | 0.6852 | 0.6279 | 0.6870 | 0.6464 | **0.6252** | 0.6295 |
| lu | 0.8536 | 0.8374 | 0.8681 | 0.8479 | **0.7731** | 0.8495 |
| gramschmidt | 0.9734 | 0.9473 | 0.9712 | 0.9600 | **0.9366** | 0.9387 |
| gemm | 0.6142 | 0.5180 | 0.5470 | 0.5151 | **0.4739** | 0.5096 |

O2'den daha yavaş çalıştığını görmekteyiz. Her problem için modeller, tamamı için olmasa da, O3 çalışma süresinden daha iyi sonuç bularak iyileşme sağlamışlardır. Tüm modeller çalışma süresi kısalmasını sağlayabilmişken, literatürde kullanılan nesil modeline kıyasla, EAFT içerisine dahil edilmiş halka modeli daha iyi sonuç göstermiştir. PSO algoritmasi simetrik matrisler üzerinde işlemlerin yapıldığı symm, syrk ve syr2k algoritmalarında Halka modelinden daha iyi ya da çok yakın sonuçlar üretmiştir. PSO algoritmasını da klasik nesil modeli ile kıyasladığımızda EAFT içerisinde kullanıcının hangi modeli kullanıcığı tercihine kalmış durumdadır. Burada gelecek çalışma olarak kodun başlangıçta bir statik analizinin yapılarak, hangisi arama algoritması ile daha iyi çalışacağı konusunda bir geliştirme yapılması planlanmaktadır.

## Kaynakça


[1] M. Hall, D. Padua, and K. Pingali, "Compiler research: the next 50 years," *Communications of the ACM*, vol. 52, no. 2, pp. 60–67, 2009.

[2] G. Fursin, "Collective tuning initiative: automating and accelerating development and optimization of computing systems," in *GCC Developers' Summit*, 2009.

[3] M. T. Jones, "Optimization in gcc," *Linux journal*, vol. 2005, no. 131, p. 11, 2005.

[4] J. W. Davidson and S. Jinturkar, "An aggressive approach to loop unrolling," tech. rep., Citeseer, 1995.

[5] Y. Chen, S. Fang, Y. Huang, L. Eeckhout, G. Fursin, O. Temam, and C. Wu, "Deconstructing iterative optimization," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 3, pp. 1–30, 2012.

[6] K. Hoste and L. Eeckhout, "Cole: compiler optimization level exploration," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pp. 165–174, 2008.

[7] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, pp. 1–42, 2018.

[8] M. Püschel, "Compiler flag optimization beyond offline learning," 2021.

[9] B Tagtekin, "EAFT: Evolutionary Algoritms for GCC Flag Tuning." https://github.com/ghisloine/EAFT, 2022.

[10] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pp. 303–316, 2014.

[11] P. Shukla, S. Thakur, S. Arora, and A. Wadhwa, "Nature-inspired algorithms analysis on various benchmark functions using python and golang," in *2022 1st International Conference on Informatics (ICI)*, pp. 226–228, IEEE, 2022.

[12] B. Tağtekin, B. Höke, M. K. Sezer, and M. U. Öztürk, "Foga: Flag optimization with genetic algorithm," in *2021 International Conference on INnovations in Intelligent SysTems and Applications (INISTA)*, pp. 1–6, IEEE, 2021.

[13] K. D. Cooper, P. J. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms," in *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, pp. 1–9, 1999.

[14] T. Kisuki, P. M. Knijnenburg, and M. F. O'Boyle, "Combined selection of tile sizes and unroll factors using iterative compilation," in *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. PR00622)*, pp. 237–246, IEEE, 2000.

[15] H. Leather, E. Bonilla, and M. O'boyle, "Automatic feature generation for machine learning–based optimising compilation," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 1, pp. 1–32, 2014.

[16] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, J. Thomson, M. Toussaint, and C. K. Williams, "Using machine learning to focus iterative optimization," in *International Symposium on Code Generation and Optimization (CGO'06)*, pp. 11–pp, IEEE, 2006.

[17] G. Fursin, C. Miranda, O. Temam, M. Namolaru, A. Zaks, B. Mendelson, E. Bonilla, J. Thomson, H. Leather, C. Williams, *et al.*, "Milepost gcc: machine learning based research compiler," in *GCC summit*, 2008.

[18] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. O'Boyle, and O. Temam, "Rapidly selecting good compiler optimizations using performance counters," in *International Symposium on Code Generation and Optimization (CGO'07)*, pp. 185–197, IEEE, 2007.

[19] E. Park, C. Kartsaklis, and J. Cavazos, "Hercules: Strong patterns towards more intelligent predictive modeling," in *2014 43rd international conference on parallel processing*, pp. 172–181, IEEE, 2014.

[20] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86, IEEE, 2004.

[21] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin, "Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks," in *International Conference on machine learning*, pp. 4505–4515, PMLR, 2019.

[22] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, pp. 1–53, 2008.

[23] P. A. Ballal, H. Sarojadevi, and P. Harsha, "Compiler optimization: A genetic algorithm approach," *International Journal of Computer Applications*, vol. 112, no. 10, 2015.

[24] S. Mirjalili, "Genetic algorithm," in *Evolutionary algorithms and neural networks*, pp. 43–55, Springer, 2019.

[25] H. Zang, S. Zhang, and K. Hapeshi, "A review of nature-inspired algorithms," *Journal of Bionic Engineering*, vol. 7, no. 4, pp. S232–S237, 2010.

[26] M. Srinivas and L. M. Patnaik, "Genetic algorithms: A survey," *computer*, vol. 27, no. 6, pp. 17–26, 1994.

[27] N. M. Razali, J. Geraghty, *et al.*, "Genetic algorithm performance with different selection strategies in solving tsp," in *Proceedings of the world congress on engineering*, vol. 2, pp. 1–6, International Association of Engineers Hong Kong, China, 2011.

[28] G. Syswerda, "A study of reproduction in generational and steady-state genetic algorithms," in *Foundations of genetic algorithms*, vol. 1, pp. 94–101, Elsevier, 1991.

[29] F. Vavak and T. C. Fogarty, "Comparison of steady state and generational genetic algorithms for use in nonstationary environments," in *Proceedings of IEEE International Conference on Evolutionary Computation*, pp. 192–195, IEEE, 1996.

[30] D. Noever and S. Baskaran, "Steady state vs. generational genetic algorithms: A comparison of time complexity and convergence properties," *Preprint series*, pp. 92–07, 1992.

[31] M Halford, "EAOPT." https://github.com/MaxHalford/eaopt, 2017.

[32] A. Hassanat and E. Alkafaween, "On enhancing genetic algorithms using new crossovers," *arXiv preprint arXiv:1801.02335*, 2018.

[33] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95-international conference on neural networks*, vol. 4, pp. 1942–1948, IEEE, 1995.