DÜŞÜK SEVİYELİ SANAL MAKİNE PROJESİ İLE DERLEYİCİ TASARIM ÖRNEĞİNİN GERÇEKLEMESİ

Yahya GEDİK
Kocaeli Üniversitesi, Teknoloji Fakültesi, Bilişim Sistemleri Mühendisliği
yahyagedikyg@gmail.com, ORCID ID: 0000-0001-9014-251X

Serdar SOLAK
Kocaeli Üniversitesi, Teknoloji Fakültesi, Bilişim Sistemleri Mühendisliği
serdars@kocaeli.edu.tr, ORCID ID: 0000-0003-1081-1598

Özet

Bilgi ve iletişim teknolojilerinin yaygınlaşmasıyla beraber kullanılan akıllı cihaz ve kullanıcı sayısı her geçen gün artmaktadır. Bu cihazları akıllı yapan üzerinde kullanılan yazılımlardır. Yazılımlar bilgisayar programcıları tarafından genellikle konuşma diline yakın ifadelerle yazılmaktadır. Yazılan bu programların bilgisayarlar veya akıllı cihazlar tarafından anlaşılması için düşük seviyeli bilgisayar koduna çevirmek amacıyla Derleyiciler kullanılmaktadır. Derleyiciler, geçmişten günümüze kadar çok fazla güncelleme geçirmiş ve yeni özellikler eklenmiştir. Bu özellikler sayesinde, insan hatalarını düzeltebilen ve ilgili platforma ait bilgisayar kodlarını optimize ederek hızlandıran uygulamalar haline gelmiştir. Derleyiciler, geliştirildiği programlama diline ait sınırları zorladığından çok fazla soyutlama katmanı gerektiren uygulamalardır. Bu katmanlar derleyicinin parçalara ayrılmasını sağlayarak karmaşayı azaltmayı hedeflemektedir. Bu kapsamda, derleyici tasarımında oluşacak sorunları çözmeyi amaçlayan düşük seviyeli sanal makine (Low Level Virtual Machine - LLVM) derleyici alt yapı projesi geliştirilmiştir. LLVM projesi sayesinde, geliştirilen derleyicilerin her platform ve işletim sistemi için tekrar yazılma gereksinimi ortadan kaldırılmıştır. LLVM projesi kullanılarak gerçekleştirilen derleyicilerde, ara derleme sunucusu bulunmaktadır. Derleyici, bu ara derleme sunucusu sayesinde istenilen işletim sistemi ve platformda çalıştırılabilir ve yüksek seviyeli programlama dillerine göre analizi kolay olduğundan çok daha fazla optimizasyon işlemi gerçekleştirmektedir. LLVM projesi ile gerçekleştirilen derleyiciler, JIT (Just in Time) mantığı ile çalıştığından hızlı çalışabilen programlar oluşturmayı sağlamaktadır. Çalışmada, LLVM projesi ile derleyici tasarım örneği gerçekleştirilmiş olup elde edilen deneyimler sunulmaktadır. Sonuç olarak, derleyici tasarımı için hızlı, soyutlama olanağı iyi olan ve yüksek seviyeli, C++, C# veya Java gibi bir dil kullanmak gerekmektedir. Ayrıca, söz dizimi analizinde Flex, Bison ve ANTLR gibi gelişmiş araçlar kullanmak derleyicinin davranışını tahmin edebilmek için önem arz etmektedir.

Anahtar Kelimeler: Yazılım, Programlama Dilleri, Derleyici Tasarımı, LLVM Projesi

THE EXAMPLE OF COMPILER DESIGN WITH LOW LEVEL VIRTUAL MACHINE PROJECT

Abstract

The number of smart devices and users used is increasing day by day with the spread of information and communication technologies. It is the software used on these devices that make them smart. Software are computer programs that enable us to use the hardware features of computers or smart devices effectively. Programs are usually made by computer programmers with expressions close to spoken language. The compilers are used to convert these written programs into low-level computer code so that they can be understood by computers or smart devices. The compilers have had a lot of updates and new features in recent years. Thanks to these features, they have become applications that can correct programmers' errors and speed up the computer codes of the relevant platform by optimizing them. The compilers are applications that contain a lot of abstraction layers that aim to reduce clutter by allowing the compiler to be divided into parts. In this context, The Low Level Virtual Machine (LLVM) compiler infrastructure project has been developed, which aims to solve problems that may arise in compiler design. Thanks to this project, the developed compilers don't need to be rewritten for each platform and operating system. The compilers implemented using the LLVM project have an intermediate compiler

server. The compiler can be run on any operating system and platform thanks to this intermediate compiler server, and it performs much more optimization processes as it is easy to analyze compared to high-level programming languages. The compilers realized with the LLVM project allow to create programs that can run quickly because they work as Just in Time (JIT). In this paper, the example of compiler design has been realized with the LLVM project and experimental results are presented. As a result, it is necessary to use a fast, abstractionable and high-level language such as C ++, C # or Java for compiler design. It is also important to use advanced tools such as Flex, Bison, and ANTLR to predict compiler behavior in syntax analysis.

Keywords: Software, Programming Language, Compiler Design, LLVM Project

1. GİRİŞ

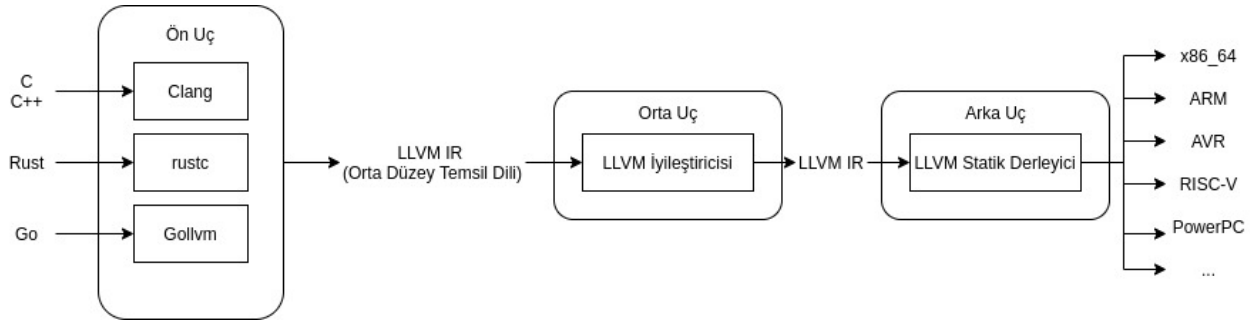
Son yıllarda yüksek seviyeli programlama dillerinde bir artış görülmektedir. Bu artışın sebebi farklı işler için farklı kapasiteye sahip programlama dillerine olan ihtiyaçtan kaynaklanmaktadır. Bu diller genelde bir takım derleyici içermektedirler. Yüksek seviyeli diller kullanılarak yazılan bu uygulamaların kararlı ve yüksek performanslı çalışması derleyicinin kaynak kod üzerine uygulayacağı iyileştirmeler ve hata düzeltmelerine bağlıdır. Bu yüzden yüksek performanslı, kararlı ve optimize edilebilen derleyiciler geliştirmek önemlidir.

Derleyici tasarlanmasına ait çalışmalar yıllar öncesine dayanmasına karşın son yıllarda da literatürde çok sayıda çalışmaların yapıldığı görülmektedir [1-8]. 2018 yılında Simon Moll ve arkadaşları tarafından yapılan bir çalışmada, SX-Aurora yüksek performanslı vektör işlemcisinin var olan kapalı kaynak kodlu derleyicisi yerine düşük seviyeli sanal makine (Low Level Virtual Machine, LLVM) kullanılarak geliştirilmiş bir derleyici fizibilitesi test edilmiştir [1]. 2018 yılında Nikita Kataev tarafından yapılan çalışmada LLVM ile program yapısının incelenmesi gerçekleştirilmiş ve paralelleştirme işlemini uygulamaya çalışılmıştır [2]. 2020 yılında Andrew Litteken ve arkadaşları tarafından yapılan bir başka çalışmada ise Kuantum bilgisayarlarda kullanılan kaynak tahmini programları oluşturmak için LLVM tabanlı bir derleyici oluşturulmuştur. Yapmış oldukları çalışma da LLVM altyapısının esnekliği hakkında kanıtlar ve örnekler sunulmaktadır [3].

Bildiri kapsamında gerçekleştirilen çalışmada, LLVM projesi kullanarak yüksek performanslı, kullanıcı hatalarını düzelten ve iyileştirmeler gerçekleştirebilen bir soyut derleyici altyapısı önerilmektedir. Ayrıca bu derleyici altyapısı çapraz platform derleme olanağı sunarak farklı işletim sistemleri ve platformları üzerinde çalışabilmektedir. Makalenin ikinci bölümünde kullanılan geliştirme yöntemi, üçüncü bölümde deneysel çalışmalar ve son bölümde elde edilen sonuçlar sunulmaktadır.

2. YÖNTEM

Bildiri kapsamında gerçekleştirilen derleyici örneğini oluşturmak için .NET Core derleyicisi ve C# programlama dili tercih edilmiştir. Bu ikili, derleyicinin farklı işletim sistemleri üstünde aynı şekilde çalışacağını garanti etmektedir. Derleyicinin kendisi için ANTLR ve LLVM tercih edilmiştir. ANTLR sözcüksel ve söz dizimsel analiz için LL(*) gramer notasyonunu kullanan modern bir araçtır. Bu aracı kullanmak obje kodu derleme aşamasından önce söz dizimine bulunan kullanıcı hatalarının fark edilmesini ve gerekliyse derlemenin iptal edilmesini sağlayan aşamadır. LLVM, derleyicinin esas kod oluşturmak için kullanılan aracıdır. Bu araç, 2003 yılından bu yana geliştirilmekte olan bir derleyici altyapısı olarak ön plana çıkmaktadır. LLVM kullanmak derleyiciye doğal olarak çapraz platform desteği ve kod iyileştirme desteği sunmaktadır. Şekil 1'de LLVM projesi kullanan derleyiciler için çalışma diyagramı sunulmaktadır.



Şekil 1. LLVM Tabanlı Derleyici Çalışma Diyagramı

LLVM projesi kullanarak derleyici tasarım örneği gerçekleştirirken aşağıda alt bölümler halinde sunulan işlemler sırayla yapılmıştır.

2.1. Proje Hazırlığı

Derleyiciler çok büyük uygulamalar olduğundan, katmanlı mimari yapısı kurallarına uyarak parçalara ayırmak derleyicinin karmaşıklığını azaltmak gerekmektedir. Bu kapsamda, derleyici tasarımı maddeler halinde sunulan yedi ana projeden oluşmaktadır.

1. Gama: Derleyici giriş noktası
2. Gama.Preprocess: Ön işleyici
3. Gama.Antlr: Sözcüksel ve Söz dizimsel Analizci
4. Gama.Compiler: Obje kodu oluşturucu
5. Gama.Types: Standard türler
6. Gama.Error: Derleyici hataları
7. Gama.Interop: LLVM iletişim aracısı

2.1.1. ANTLR

Derleyicide kullanılacak olan sözcüksel ve söz dizimsel analiz için kullanılacak olan Antlr aracının kurulumu Nuget yada ‘dotnet’ sistem aracıyla gerçekleştirilir:

$ dotnet add package Antlr4.Runtime.Standard

Bu, projede kullanacağımız analiz fonksiyonlarını içeren kütüphaneyi kuracaktır. Antlr normalde Java ile yazılmış bir araçtır. Bu araç Java haricinde C++, C#, Python gibi programlama dilleri içinde çıktı sağlamaktadır.

2.1.2. LLVM

LLVM projesi normalde C++ için yazılmış bir kütüphanedir. Bu kütüphaneyi C# ile kullanabilmek için LLVMSharp adlı kütüphanenin kullanılması gereklidir. Bu kütüphanenin kurulumu Nuget yada ‘dotnet’ sistem aracıyla gerçekleştirilir:

$ dotnet add package LLVMSharp --version 10.0.0-beta

Bu kütüphanenin 10.0 beta sürümü kullanılmıştır. Bu sürüm her ne kadar daha az stabil olsa da performans iyileştirmeleri ve hata düzeltmeleri içerir.

LLVM normalde bir C++ projesi olduğundan dolayı fonksiyon ağırlıklı programlama metodları içerir. Bunun önüne geçip C fonksiyonlarına olan yüklenmeyi azaltmak için Gama.Interop projesi oluşturulmuştur.

2.2. Sözcüksel ve Söz dizimsel Analiz

Antlr aracı ile LL(*) notasyonuna sahip gramerler derlenip hedef dil için analizciler oluşturulabilir. LL(*) notasyonunun avantajları:

- Yazımı & anlaması kolay
- Soldan sağa tarama gerçekleştirir
- Sonsuz ileri ve geri bakma (lookahead, lookbehind)
- Modüler

```
s : ID
  | ID '=' expr
  | 'unsigned'* 'int' ID
  | 'unsigned'* ID ID
  ;
```

Şekil 2. Basit bir LL(*) Kuralı

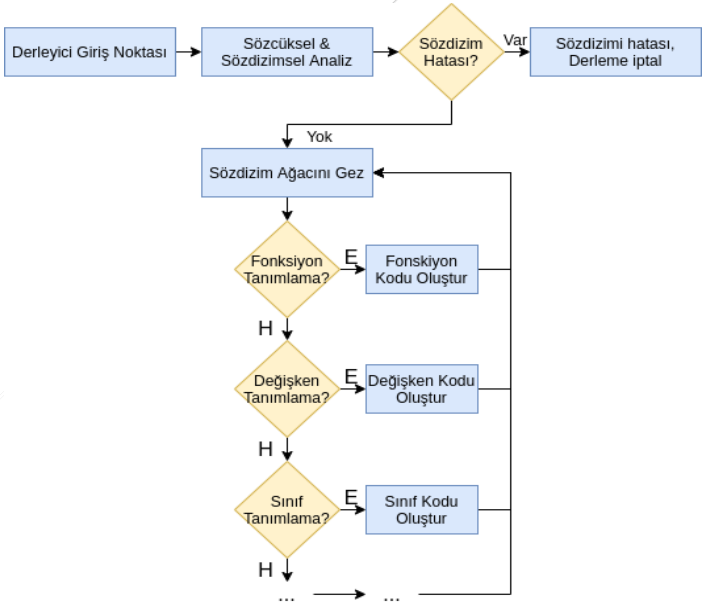
Şekil 2'de LL(*) kuralına ait basit bir örnek sunulmaktadır. Hazırlanan gramerin hedef dile derlenebilmesi için bir komut betiği kullanılması gereklidir:

$ java -jar antlr.jar Grammar.g4 -visitor

Bu betik Antlr java aracını kullanarak hedef dile analizcileri oluşturur, ek olarak misafir desenleri için dosyalar oluşturur.

2.3. Derleyici Akışı

Derleyici yüksek seviyeli programlama dilinde yazılmış olan kaynak kodlarını girdi olarak alıp sözcüksel analiz aşamasından geçirmektedir. Şekil 3'te derleyici akışını gösteren, akış şeması sunulmaktadır.



Şekil 3. Derleyici Akış Şeması

Bu aşamada analizci girdiyi kelimelere, noktalama işaretlerine, sayılara vb. küçük parçalara ayırır. Bu parçalara 'Token' denilir. Bu tokenler daha sonra söz dizimsel analiz birimine gönderilir. Bu birimde tokenler dizimlerine bağlı olarak gruplandırılır, dizim hataları kontrol edilir ve gerekliyse derleme durdurulmaktadır. Bu gruplanan tokenler daha sonra AST (Abstract Syntax Tree, Sanal Söz dizim Ağacı) denilen bir yapıya dönüştürülür. Derleyici bu yapı üstünde baştan aşağıya doğru tarama gerçekleştirir ve karşılaştığı derleme işlemlerini gerçekleştirir. Bu işlemler 'for' betikleri, 'if' betikleri, değişken tanımlamaları veya sınıf tanımlamaları olabilir. Obje kodu oluşturma aşaması

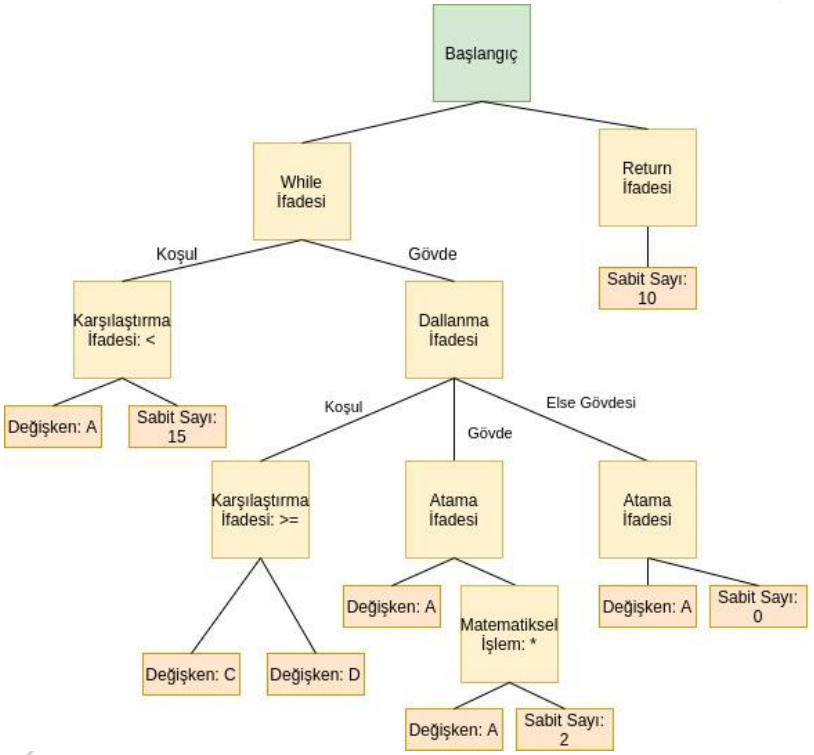
tamamlandığında oluşturulan çıktı dosya istenilen bağlayıcı ile sistem kütüphanelerine bağlanarak çalıştırılabilir dosya oluşturulmaktadır.

2.4. Sanal Söz dizim Ağacı

Sanal söz dizim ağaçları gramer bakımından incelenmiş kaynak dosyasındaki grupların bulunduğu bir ağaç yapısıdır. Bu yapı üzerinde çalışması kolay ve hızlı olduğundan genelde derleyiciler ve betik dilleri tarafından kullanılmaktadır. Derleyicinin bu ağaç üstünde gezebilmesi için başlangıç düğümünün altındaki düğümleri soldan sağa doğru inceleyerek derlemesi gerekmektedir. Eğer bir düğümün alt düğümleri bulunuyorsa öncelikle alt düğümlerin derlenmesi gereklidir.

Antlr bu ağaç yapısını gezebilmek için Misafir (Visitor) deseni kullanmaktadır. Bu desende Antlr o anki düğümü derleyici yerine gezerek gerekli olan soyut ve sanal çağrıları gerçekleştirir.

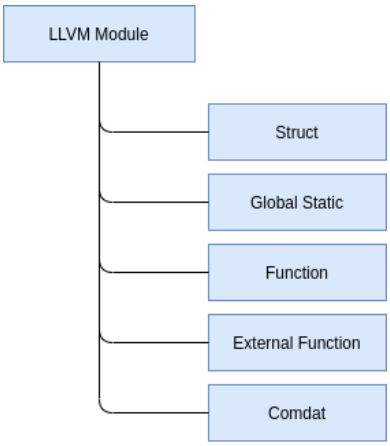
Derleyici Antlr tarafından oluşturulmuş 'GrammarBaseVisitor' sınıfını devralarak sanal çağrıları değiştirebilmektedir. Antlr bu değişmiş metodlara çağrılar göndermekte ve içeriğe bağlı olarak derlemeyi gerçekleştirebilmektedir. Şekil 4'te sanal söz dizim ağacı örneği sunulmaktadır.



Şekil 4. Sanal Söz dizim Ağacı (AST) Örneği

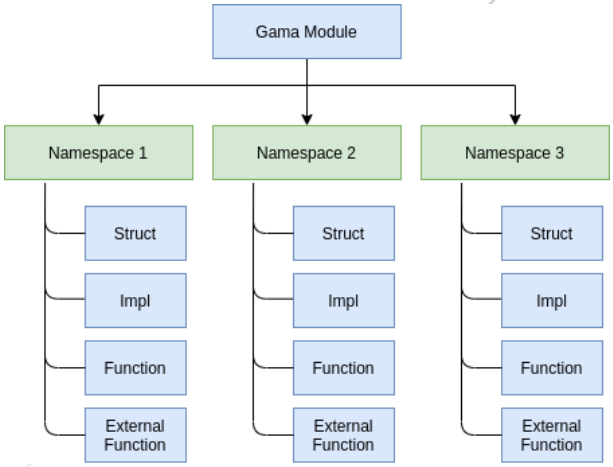
2.5. Obje Kodu Oluşturma

Çalışmada LLVM kullanarak obje kodu oluşturmak için LLVM C bağlantılarını kullanmak gereklidir. Bu kütüphane fonksiyon ağırlıklı olduğu için çok fazla çağrı gerçekleştirecek ve derleyicinin performansını azaltacaktır. Bunun yerine bir aracı kütüphane (Gama.Interop) oluşturup çağrıları bu kütüphane üstünden gerçekleştirmek avantajlı olamaktadır. Örneğin bir kod bloğunun ismini almak için `llvm::BasicBlock::getName(...)` metodunu kullanmak gerekmektedir. Her ne kadar C++ derleyicisi bu metodu optimize edip kaynak içine entegre etse de C#'ta bu özelliği kullanmak mümkün değildir. Bunun yerine 'GamaBlock' isimli bir sınıfı aracı proje içinde oluşturup bloğun ismini hafızada tutarak daha sonra fonksiyon kullanmadan elde etmek mümkün olmaktadır.



Şekil 5. LLVM Modül Yapısı

Şekil 5'te LLVM modül yapısı sunulmaktadır. LLVM tüm obje kodunu 'Module' adlı bir yapının altında birleştirir. Bu yapıda hiyerarşi yoktur, tüm fonksiyonlar, global değişkenler ve sınıflar aynı bölümde bulunur ve yönetmek zordur. Bunun yerine namespace (Alan Adı) tabanlı hiyerarşik bir sistem oluşturmak daha sonra derleyici içinde oluşabilecek karmaşayı azaltacaktır.



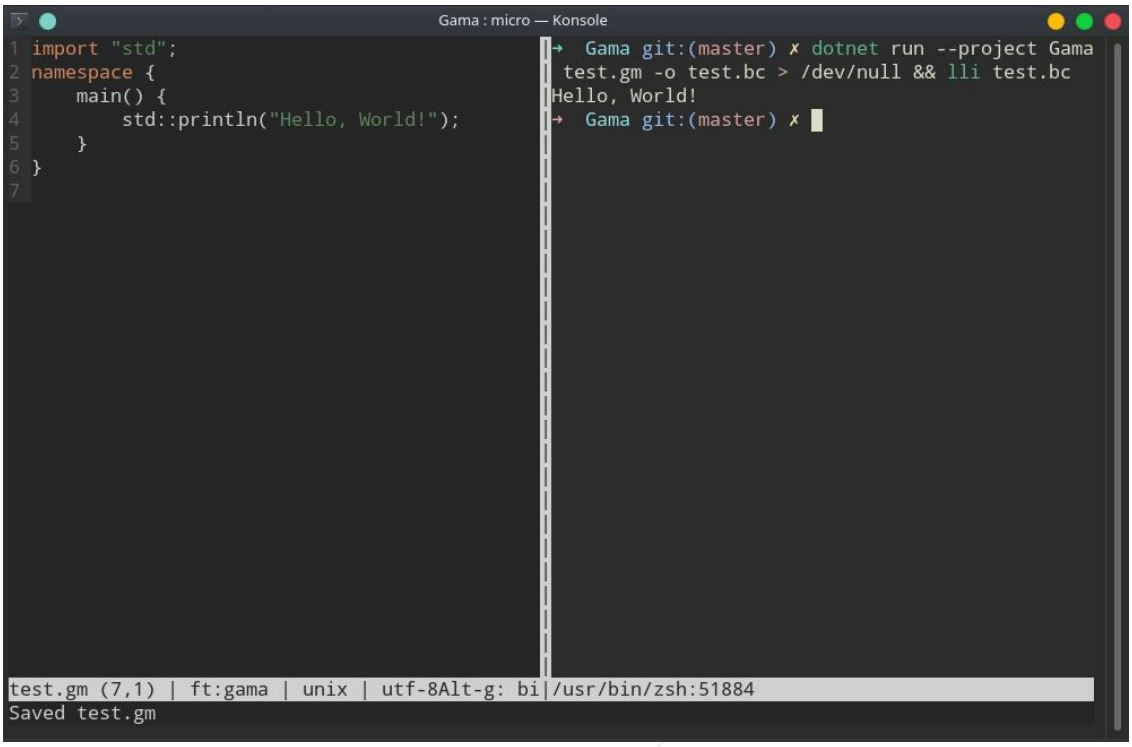
Şekil 6. Hiyerarşik Modül Yapısı

Şekil 6'da gösterilen hiyerarşik modül yapısı çalışma kapsamında gerçekleştirilen derleyici örneğinde kullanılmıştır. Bu yapı, katı alan adları, ayrı sınıf tanımlamaları ve açıklamaları, global statiklerin olmaması ve comdatların olmaması özelliklerini içermektedir. Bu özellikler hem performans hem de yönetilebilirlik açısından önem arz etmektedir. Katı alan adı sistemi, program tek fonksiyon olacaksa bile her tanımlamanın kesinlikle bir alan adının altında olmasını zorunlu kılmaktadır. Bu zorunluluk sayesinde aynı isimli fonksiyon, sınıf tanımlamaları ve dış fonksiyon tanımlamaları farklı alan adları altında çakışmadan bulunabilmektedir. Ayrı sınıf tanımlamaları ve açıklamaları sayesinde var olan sınıflara eklemeler yapılabilir. Örneğin var olan 'Vector' sınıfına başka bir kütüphane tarafından yeni fonksiyonlar eklenebilmektedir. Global statiklerin olmaması çoklu işleç kullanan uygulamalar için avantaj sağlamaktadır. İşleçler global değişkenler yerine içerik sınıflarını Mutexler ile paylaşıp birbirlerinden haberdar olmaktadır. Bu işlem hem daha yüksek performans hem de daha fazla güvenlik ve pratiklik sağlamaktadır.

3. DENEYSEL ÇALIŞMA

Çalışma kapsamında oluşturulan derleyicinin kararlılığını ve performansını belirlemek için birtakım testler gerçekleştirilmiştir. Tüm testler Linux 5.9 çekirdeği içinde ve aynı komut betiği kullanılarak

derlenmiştir. Derleme işlemi için; "$ gamac -c input.gm -o output.bc && lli output.bc" kullanılmaktadır. Geliştirilen derleyiciye ait örnek derleme işlemi Şekil 7'de sunulmaktadır



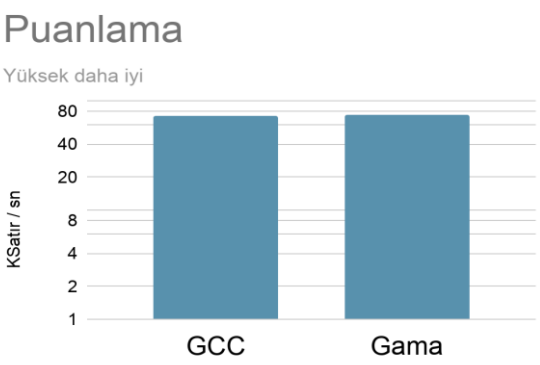
Şekil 7. Geliştirilen derleyici ile örnek kod derleme işlemi

Geliştirilen derleyici kullanılarak, örnek kod yazımı ve sınıf kullanımı aşağıda verilen örnekte sunulmaktadır.

Örnek Kodlar:

```
import "std";
import "math";
namespace {
    using namespace std;
    using namespace std::math::spatial;
    main() {
        let v1 : vector2f;
        let v2 : vector2f;
        v1.x = 10.0;
        v2.y = 20.0;
        printf("%f\n", v1.x + v2.y);
    }
}
```

Derleyici performans testi için, GCC (GNU C Compiler, GNU C Derleyicisi) ve geliştirilen derleyici karşılaştırılmıştır. Derleyiciler saniyede derledikleri satır başına puanlandırılmıştır. Şekil 8' iki derleyicini performans testini göstermektedir. Testlerde iki derleyicide saniyede yaklaşık 74.000 satır derlemeyi başarmıştır. Geliştirilen derleyici için bu hız daha fazla optimizasyon ile artırılabilmektedir.



Şekil 8. Derleyici Performans Karşılaştırması

4. SONUÇLAR VE TARTIŞMA

Yüksek seviyeli programlama dilleri kullanılarak yazılan dillerin kararlı ve hızlı çalışabilmesi için derleyici tasarımı oldukça dikkat edilmesi gereken konular arasındadır. Ayrıca derleyici sürelerini azaltmak üretimde devamlılığı sağlayan en büyük faktörlerden biri olarak karşılaşılmaktadır. Literatürde, derleyici tasarımlarına ait performans iyileştirme çalışmaları yaygın olarak devam etmektedir. Bildiri çalışması kapsamında düşük seviyeli sanal makine projesi kullanılarak örnek bir derleyici tasarımı gerçekleştirilmiştir. Geliştirilen derleyici performans testleri ve kararlılık bakımından oldukça yeterli sonuçlar vermektedir. Antlr araç takımı sayesinde derleyici, söz dizimini doğru bir şekilde kontrol etmekte ve kullanıcıya yararlı hata mesajları hızlı bir şekilde göstermektedir. LLVM projesi sayesinde kod iyileştirmeleri bulunmaktadır. Gerçekleştirilen soyut altyapı sayesinde derleyiciye istenildiği zaman özellikler eklenip çıkarılabilmektedir. Çapraz platform desteği sayesinde, fiziksel bir hedef platform bulunmasa da, ilgili platformun standart kütüphaneleri ile derleme yapılabilmektedir.

KAYNAKLAR

[1] Moll, S., Kurtenacker, M., & Hack, S. (2020). Potential of LLVM for SX-Aurora. In Sustained Simulation Performance 2018 and 2019 (pp. 111-121). Springer, Cham.
[2] Kataev, N. (2018, September). Application of the LLVM Compiler Infrastructure to the Program Analysis in SAPFOR. In Russian Supercomputing Days (pp. 487-499). Springer, Cham.
[3] Litteken, A., Fan, Y. C., Singh, D., Martonosi, M. R., & Chong, F. (2020). An updated LLVM-based quantum research compiler with further OpenQASM support. Quantum Science and Technology.
[4] Leary, C. D., & Springer IV, R. C. (2017). LLVM FUSING KERNEL COMPILER DESIGN.
[5] Nguyen, A. (2019). Compiler design and implementation in OCaml with LLVM framework.
[6] Miletić, V., Páll, S., & Gräter, F. (2017). LLVM AMDGPU for High Performance Computing: are we competitive yet?
[7] Özen, G., Atzeni, S., Wolfe, M., Southwell, A., & Klimowicz, G. (2018, November). OpenMP GPU offload in Flang and LLVM. In 2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) (pp. 1-9). IEEE.
[8] Fornaia, A., Scafiti, S., & Tramontana, E. (2019, June). JSCAN: Designing an easy to use LLVM-based Static Analysis Framework. In 2019 IEEE 28th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE) (pp. 237-242). IEEE.