Gradus Vol 11, No 1 (2024)
ISSN 2064-8014

MÓDSZEREK A JAVA PROGRAMOK TELJESÍTMÉNYÉNEK JAVÍTÁSÁRA

METHODS FOR IMPROVING THE PERFORMANCE OF JAVA PROGRAMS

Subecz Zoltán[ORCID0000-0001-5036-4679 1*]

[1] Informatika Tanszék, GAMF Műszaki és Informatikai Kar, Neumann János Egyetem, Magyarország
https://doi.org/10.47833/2024.1.CSC.007

Kulcsszavak:
Java programozás
program teljesítmény
Just-In-Time fordító
Java processzor
Java Virtuális Gép

Keywords:
Java programming
program performance
Just-In-Time compiler
Java processor
Java Virtual Machine

Cikktörténet:
Beérkezett 2023. július 4.
Átdolgozva 2024. január 5.
Elfogadva 2024. január 15.

Összefoglalás

A Java az egyik legnépszerűbb programozási nyelv, ezért fontos a lehető legjobb teljesítmény elérése. Szabványos végrehajtási technikája az interpretáció, ahol az értelmező közvetlenül hajtja végre a forráskódot, ez biztosítja a programok hordozhatóságát, ami a Java nyelv egyik előnye. De ez az értelmezés meglehetősen lassú, azonban a Java programok futási teljesítménye fordítással javítható. A közvetlen fordítók átalakítják a Java forráskódot a célprocesszor nyelvére. Ez az összeállítás jelentősen javítja a Java programok teljesítményét, de csökken a hordozhatóság, mivel a generált kód csak a célprocesszoron futtatható. A cikk olyan módszereket mutat be, amelyek növelik a sebességet, miközben megőrzik az alkalmazás platformfüggetlenségét.

Abstract

Java is one of the most popular programming languages, so it's important to get the best possible performance. Its standard execution technique is interpretation, where the interpreter directly executes the source code. This ensures the portability of the programs, which is one of the advantages of the Java language. But this interpretation is quite slow, however, the runtime performance of Java programs can be improved by compilation. Direct compilers convert Java source code into the language of the target processor. This assembly significantly improves the performance of Java programs but reduces portability because the generated code can only be run on the target processor. This article shows methods to increase speed while keeping the application platform independent.

1. Bevezetés

A *Java nyelvnek* sok olyan tulajdonsága van, ami népszerűvé teszi a programfejlesztők körében. Ilyenek a platformfüggetlenség, az objektum-orientált modell, a többszálú és az elosztott programozás támogatása és az automatikus szemétgyűjtés.

* Kapcsolattartó szerző.
E-mail cím: subecz.zoltan@nje.hu

A Java programok végrehajtásának fő módszere az *értelmezés*, ami biztosítja a programok *hordozhatóságát*. A Java „írd meg egyszer és futtasd bárhol” egy olyan tulajdonság, ami megkönnyíti a fejlesztők munkáját, mert segíti a hordozhatóságot. Ennek a hordozhatóságnak, rugalmasságnak az ára azonban a lassabb futás.

A *hordozhatóságot* úgy valósítják meg, hogy a Java forráskódot architektúra független *bájtkódra* fordítják, amit végre lehet hajtani bármilyen platformon, ami támogatja a *Java Virtuális Gépet (Java Virtual Machine, JVM)*. A legtöbb JVM a bájtkódot futási idejű értelmezéssel hajtja végre, ami lassú végrehajtási időt eredményez. Természetesen a teljesítmény jelentősen javul, ha a forrásprogramot közvetlenül fordítjuk gépi kódra (direkt fordító), de így elveszik a hordozhatóság.

Ez a cikk olyan technikákat mutat be, amelyekkel a Java nyelven írt programok *teljesítményét lehet javítani*. Egyik ilyen módszer a Just-In-Time (JIT) fordító, ami a bájtkódot futási időben fordítja gépi kódra. Vannak más módszerek is a Java programok teljesítményének növelésére a hordozhatóság megőrzése mellett, mint például a bájtkód optimalizáció, vagy a dinamikus fordítás, vagy a párhuzamos programvégrehajtás. Egy másik módszer: a JVM megvalósítása közvetlenül a hardver szintjén.

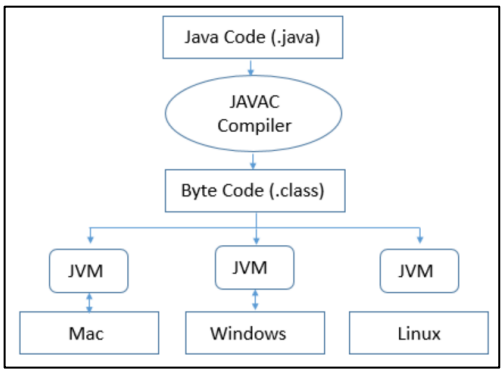
Ebben a cikkben bemutatom ezen technikák tulajdonságait, előnyeit és hátrányait.

2. A Java alap végrehajtás

A Java nyelven írt programok alap végrehajtása a Java *forráskóddal* (.java fájlok) kezdődik. A Java forráskódot a Java fordító *bájtkódra* (.class fájlok) fordítja. A bájtkód a JVM utasításkészletét használja.

2.1. A Java Virtuális Gép (Java Virtual Machine, JVM)

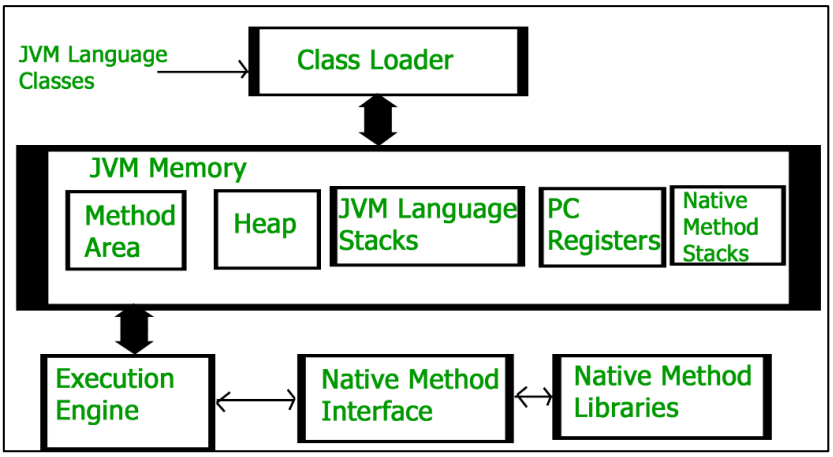
A JVM végrehajtja a Java *bájtkódot* [2]. A JVM már figyelembe veszi az aktuális hardver platformot, ezért a *JVM-et először implementálni* kell a célplatformon, mielőtt bármilyen Java programot végrehajthatnánk azon a platformon. Mivel a JVM-et számos platformon lehet implementálni, ezért ez segíti a hordozhatóságot. (1.ábra)



1. ábra. A platformfüggetlenség megvalósítása

forrás: https://net-informations.com/java/intro/jvm.htm

A JVM egy *interfészt* jelent a lefordított Java programok és a célplatformok között. Hagyományosan a JVM értelmezi a bájtkódok folyamát, mint utasítások sorozatát. A regisztereket, a stack-et, a heap-et implementálni kell a JVM-ben. (2.ábra)



2. ábra. A JVM felépítése és működése

forrás: https://www.geeksforgeeks.org/jvm-works-jvm-architecture

A bájtkódok a *Method Area*-ban vannak tárolva. A *Program counter* a regiszterek között található és a következő bájtra mutat, amit a JVM-nek végre kell hajtania. Az utasítások paramétereit és azok eredményét a *stack*-ben tárolják. A JVM egy stack alapú rendszer, minden művelet és adat a stack-en megy keresztül. A JVM magja az *Execution engine*, ami egy virtuális processzor, ami végrehajtja a bájtkódot

3. Java programok alternatív végrehajtási technikái

Az alap JVM működéshez képest számos végrehajtási technikát lehet alkalmazni a Java programok *teljesítményének növelésére*.

3.1. Java fordítók

3.1.1. Just-In-Time (JIT) fordító

A JIT a Java bájtkódot fordítja gépi utasításokra. Ez a *fordítás futási időben* történik, a metódus hívása után. Egy metódus meghívásakor a kód értelmezése helyett a JIT fordító lefordítja a metódus bájtkódját gépi utasítások sorozatára. A lefordított bájtkódot a cache-be másolja, így kiküszöbölve a bájtkódok többszörös fordítását.

A legtöbb esetben a JIT fordító által generált *gépi kód hatékonyabb*, mint a bájtkód értelmezése, mivel az értelmező azonosítja és értelmezi a bájtkódot annak minden előfordulásánál a végrehajtás során. A JIT azonban csak egyszer hajtja végre az azonosítást és a fordítást – amikor a metódus először kerül meghívásra. A nagy ciklusokat vagy rekurzív metódusok esetén a JIT fordítás és a keletkezett gépi kód végrehajtás sokkal gyorsabb az értelmezésnél.

Mivel egy Java program tényleges végrehajtási ideje a fordítási időnek és a végrehajtási időnek a kombinációja, ezért a JIT fordítónak mérlegelnie kell, hogy van-e annyi nyereség a készítendő gépi kód futtatásán, hogy érdemes legyen elvégezni a fordítást.

Míg a JIT egy egész metódust fordít egy lépésben, addig az értelmező csak az aktuális utasítást értelmezi. Ha egy metódust csak ritkán hajtunk végre, akkor nem érdemes a JIT-et alkalmazni, mert a fordítási idő hosszabb, mint amit a végrehajtáson nyerünk.

Készítettek JIT fordítókat számos platformra, amik közül több optimalizációs technikákat is alkalmaz [7].

3.1.2. Direkt fordítók

A *direkt fordító* vagy a Java forráskódot, vagy a bájtkódot fordítja le gépi kódra, ami közvetlenül végrehajtható a célprocesszoron. A fő különbség a közvetlen fordító és a JIT fordító között az, hogy a direkt fordító által generált kódot a Java program későbbi végrehajtásakor is fel lehet használni. Mivel a JIT fordítás futási időben történik, ezért a fordítási időkövetelmények korlátozzák a későbbi felhasználási lehetőségeket.

A direkt fordítás *statikusan* történik, ezért lehet alkalmazni hagyományos, időigényes optimalizációs technikákat, amivel a fordított kód teljesítményét lehet javítani. Mivel statikus fordítást alkalmaz, ezért nem támogatja a dinamikus osztálybetöltést. A direkt fordítás a hordozhatóság csökkenését eredményezi, mivel a generált kódot csak a célprocesszoron lehet végrehajtani. A hordozhatóság nem veszik el teljesen, ha az eredeti forráskód vagy bájtkód továbbra is felhasználható.

3.1.3. Bájtkódról forráskódra fordítók

Ezek olyan statikus fordítók, amelyek egy közbülső *magas szintű nyelvi kódot* (például a C nyelv) generálnak a Java bájtkódból. Ebből egy már megszokott fordító program készíti el a futtatható gépi kódot. Egy magas szintű nyelvet választva köztes kódnak, mint például a C, lehetővé teszi már meglévő fordító technológia használatát. Ez hasznos, mert sok mai fordító alkalmaz kifinomult optimalizációs technikákat szinte az összes platformra.

Ennek a módszernek az előnye a JIT fordítóval szemben, hogy mivel *statikus fordító*, így jóval a programfuttatás előtt elvégezhető a fordítás, így a fordítás nem csökkenti a végrehajtás idejét, és így van idő sok optimalizációs módszer alkalmazására. Ezzel szemben a dinamikus JIT fordító közvetlenük a végrehajtáskor fut le. Hátránya viszont a JIT-el szemben, hogy a dinamikus környezeti információkat nem tudja figyelembe venni, míg a JIT igen.

3.2. Java processzorok

Ha közvetlenül végre tudjuk hajtani a bájtkódot egy *processzoron*, akkor az értelmező memóriahatékonysága és a JIT fordító gyorsítása kombinálható. Egy ilyen processzornak támogatnia kell a JVM szerkezeti jellemzőit. A Java processzor a *JVM-et implementálja hardver szinten* és közvetlenül végrehajtja a Java bájtkódot.

Egy Java processzort a Java környezet alapján készítik, hardveres támogatást biztosítva olyan jellemzőknek, mint a stack feldolgozása, többszálú végrehajtás és szemétgyűjtés. Így egy *Java processzorral* sokkal jobb teljesítményt tudunk elérni, mint egy általános célú processzorral.

Vannak olyan Java processzorok, amelyek nem valósítják meg az összes utasítást, csak a leggyakoribbakat, a többi utasítást szoftveres úton valósítják meg. Így a hardver tervezése jelentősen leegyszerűsödik, mert a bonyolultabb de ritkább utasításokat nem implementáljuk a hardverben.

Vannak processzorok, amiben az alap JVM utasításkészletet kiegészítenek újabb utasításokkal. Erre egy példa: több olyan utasítást, ami gyakran egymás után jön, egy utasításként hajtanak végre. Például adat másolása lokális változóból a stack tetejére, majd egy utasítás, ami ezzel az adattal végez művelete. Ez a módszer felgyorsítja a Java bájtkód végrehajtását [4].

4. Egyéb Java végrehajtást gyorsító technikák

Számos olyan végrehajtási módszert dolgoztak ki, ami növeli a Java teljesítményét a hordozhatóság megtartása mellett.

4.1. Bájtkód optimalizálás

A *bájtkód optimalizálásra*, amivel javítjuk az alkalmazás teljesítményét, számos módszer létezik. Az egyik módszer újra alkotja a forrásprogramot a bájtkódból, majd a forráskódon számos optimalizációs módszert alkalmaz és végül lefordítja azt újra bájtkódra.

Ezen *optimalizációs módszerek* egyike az összetartozó változókat a memóriában közel helyezi el egymáshoz, ezzel felgyorsítva azok elérését. Egy másik módszer a tömböket újraszervezi a memóriában. Ha egy tömb egy adott területének elemeit gyakran használjuk együtt, akkor érdemes ezeket az elemeket közel elhelyezni a memóriában egymáshoz, ami szintén gyorsítja azok elérését. Egy másik módszer próbálja csökkenteni a futtatható fájl méretét a következő technikákkal:

- csökkenti az osztályok, metódusok, változók nevének méretét,
- nemhasznált vagy redundáns osztályok, metódusok, változók kiiktatása.
- Osztályok szétdarabolása gyakran használt és ritkábban használt részekre. Először a gyakrabban használt részeket olvassa be a memóriába.
- Összetartozó osztályok egyesítése egy osztályba

4.2. Párhuzamos és elosztott végrehajtási technikák

A *párhuzamosítás* egy nagyobb problémát kisebb részekre oszt, amit egymástól függetlenül párhuzamosan végre lehet hajtani egy többprocesszoros vagy többszálas környezetben. A módszer növeli a rendelkezésre álló számítási teljesítményt a gyorsabb alkalmazás eléréséhez.

Számos technika létezik a Java forráskód vagy bájtkód párhuzamosítására, amivel az alkalmazások futási idejű teljesítményét javítani lehet. Ennek egyik módja a többszálú programozás, amit a Java nyelv támogat. Így ezek a technikák megtartják a párhuzamosított Java programok hordozhatóságát, ezért a párhuzamosított kód végrehajtható minden JVM-en, ami támogatja a többszálú programvégrehajtást.

Ezen párhuzamosító módszerek egyike, az alap Java többszálú képességeket felhasználva a ciklusok helyett generál párhuzamos kódot. Egy másik újrastrukturáló fordító már meglévő algoritmusok alapján a soros Java kódból készít párhuzamosított kódot. Egy következő módszer az egy processzorra írt kódot oszt meg elosztott memóriájú többprocesszoros rendszerben. Egy további párhuzamosító módszer kiterjeszti a Java alapképességeit, amivel egy új, Java alapú nyelvet alkot meg. Ez a nyelv párhuzamosítást támogató osztályokat definiál és a metódusokat nagyszámú processzoron hajtja végre. Ezek a processzorok lehetnek virtuálisak vagy elosztott rendszerek processzorai.

Adatstruktúra párhuzamosítás: Adatbázisokat, mátrixokat, vektorokat és más adattípusokat feloszt kisebb részekre, amelyeket több processzoron dolgoz fel.

Feladat párhuzamosítás: Szétosztja a feladatokat több processzoron, így sok különböző feladat végrehajtható párhuzamosan. A feladatok párhuzamosításának egyik gyakori típusa a folyamatkezelés, ahol egy adathalmazt egy sor diszkrét feladaton keresztül mozgatják, és minden feladatot a többitől függetlenül hajtanak végre.

Szál szinkronizáció: A többszálú programvégrehajtás mindig megkívánja a szálak optimális végrehajtását az optimális sebesség eléréséhez. A szálak monitorozásával próbálják csökkenteni a végrehajtási időt és a memóriahasználatot.

4.3. Dinamikus fordítás

A JIT fordításnál a fordítási idő növeli az alkalmazás teljes végrehajtási idejét, a módszer minőségét. A *dinamikus fordítás* ezt a problémát célozza meg és a kódnak csak azokat a részeit optimalizálja (*program hotspot-ok*), amelyeket gyakran végrehajtanak. Általában a programok az idejük nagy részét a kódjuknak csak egy kis részénél töltik el, ezért a *hotspot metódusok optimalizálása* nagy teljesítménynövelést eredményezhet, miközben a fordítási idő viszonylag gyors marad.

A metódusok első végrehajtásakor egy értelmező futási idejű információt generál a metódusról. Majd ezt az információt felhasználja hotspot-ok detektálására a programban. A hotspot metódusok azonosítása után ezeket dinamikusan lefordítja, ami egy optimalizált gépi kódot ad. A ritkán végrehajtott kódot továbbra is értelmezéssel hajtja végre, így csökkentve a kód generálásra fordított időt és memóriát. A hotspot-ok azonosításánál, ha a teljesítményjavulás elér egy adott határértéket, akkor a dinamikus optimalizáló fordítót elindítja, ami újra fordítja a hotspot metódusokat. Végül a nem optimalizált kódot helyettesíti az optimalizált kóddal.

A módszernek egyik alesete, amikor az első végrehajtásnál lefordítja a kódot optimalizáció nélkül, majd egy dinamikus optimalizáló fordító a hotspot-okat optimalizált végrehajtható gépi kódra fordítja futási idejű információkat felhasználva [1].

4.4. Szemétgyűjtés

Folyamatosan figyeli a változókat a memóriában és azokat, amelyeket a jövőben már nem használ fel, *felszabadítja a helyét* a memóriában. Ez memóriatöredezettséget eredményez, ezért időnként *memóriaátrendezést* hajt végre a töredezettség javítására.

4.5. Halott kód kiküszöbölés

Ha a fordító találkozik *szükségtelen utasításokkal*, akkor eltávolítja az utasítások közül. Ez jelenti azokat az utasításokat, amelyeket soha nem hajt végre, vagy pl. olyan változókba írást, amelyeket később nem olvasunk.

4.6. Inlining módszerek

A módszer helyettesíti a gyakran végrehajtott metódusok meghívását a metódus tartalmával. Ezzel *elkerüljük a metódushívások többletköltségét*. A metódushívás a gépi kódban egy ugró utasítással jár, ami lassabb, mint a többi alaputasítás. Bár a metódusok hívása gyorsan történik, de ha azt több ezerszer kell végrehajtani, akkor a módszerrel jelentős javítást lehet elérni.

Megszámolja, hogy egy adott metódust hányszor hívunk meg. Ha többször, mint egy határérték, akkor alkalmazza a módszert. Ez a határérték módosítható egy JVM beállítással induláskor. Minden metódusra nem akarjuk alkalmazni a módszert, hiszen ez időigényes és nagy bájtkódot eredményez, ezért elsősorban csak kis méretű metódusokra alkalmazzák.

4.7. Ciklus optimalizáció

Általában a legtöbb sebességprobléma a *ciklusokkal* kapcsolatos. Minél több lépésből áll a ciklus és minél több ideig tart egy lépés végrehajtása, annál nagyobb lesz a ciklus hatása a sebességre. Ha csökkentjük az utasítások számát egy ciklusban, azzal a program futási ideje javítható, még akkor is, ha közben növeltük a külső utasítások számát. Ha egy soklépéses ciklusban véletlenül bekerül egy olyan utasítás, ami a cikluson kívül is végrehajtható lenne, akkor az jelentősen csökkenti a végrehajtás sebességét. Ezért kiviszik a ciklusból azokat az utasításokat, amik kívül is végrehajthatóak, a ciklusban a csak a feltétlenül szükségesek maradnak.

4.8. Null vizsgálat kiküszöbölés

A Java nyelvben is gyakran ellenőrizzük egy változóról, vagy objektumról, hogy az értéke null-e. Azonban ez a vizsgálat általában egy ugrást jelent az assembly kódba, ami jelentősen lassítja a végrehajtást. Helyette a JIT először ellenőrzi, hogy a változó felveszi-e valamikor a null értéket, ha nem, akkor kihagyja ezt a vizsgálatot a kódból.

4.9. Néhány további, a programozó által alkalmazható technika

- **Kevesebb reflekció alkalmazása**
  A Java reflekció (reflection) lehetővé teszi osztályok, interfészek, mezők és metódusok futás-idejű attribútumainak vizsgálatát és/vagy módosítását. Ez különösen akkor hasznos, ha nem tudjuk a nevüket a fordításkor.
- **Rekurziók kerülése**
- **Szál szinkronizáció megfontolt alkalmazása**
  A szál szinkronizáció csökkenti a teljesítményt, így csak nagyon indokolt esetben és csak a megfelelő helyen alkalmazzuk
- **Esetenként a publikus változok használata indokolt.**

5. Összefoglalás

A Java programozási nyelv lehetőséget biztosít a platformfüggetlenségre és ezzel kombinálva a nyelvi tulajdonságok széles körét, mint például az objektum-orientált modell, többszálú programozás, automatikus szemétgyűjtés stb. Ezek a tulajdonságok a Java-t vonzóvá eszik, azonban csökkentik a programok sebességét. A Java programok hordozhatóságának biztosításához a forráskódot először platform-független bájtkódra fordítják a CPU ismerete nélkül, amin a kódot majd végrehajtják. Ezért néhány fordítás és értelmezés szükséges futási időben, ami növeli a program végrehajtási idejét. A cikk olyan módszereket mutatott be, amelyek növelik a sebességet, miközben megőrzik az alkalmazás platformfüggetlenségét.

Irodalomjegyzék

[1] Burke, M. Choi, J.: The Jalapeno dynamic optimizing compiler for Java. In Proceedings of the ACM 1999 Conference on Java Grande, 129–141
[2] Lindholm, T and Yellin, F 1997. The Java Virtual Machine Specification. Addison-Wesley, Reading, MA.
[3] Maldikar, P. , Li, S. and Chow K.: Java Performance Mysteries. ITM Web Conf. Volume 7, 2016. 3rd Annual International Conference on Information Technology and Applications (ITA 2016), eISSN: 2271-2097, https://doi.org/10.1051/itmconf/20160709015

[4] McGham, H. and O'Connor M.: PicoJava: A direct execution engine for Java bytecode. IEEE Computer 1998., Oct., 22–30.
[5] Sabah A. Abdulkareem1 and Ali J. Abboud: Evaluating Python, C++, JavaScript and Java Programming Languages Based on Software Complexity Calculator. OP Conference Series: Materials Science and Engineering, 2nd International Scientific Conference of Engineering Sciences (ISCES 2020) DOI 10.1088/1757-899X/1076/1/012046
[6] Sonoyama, A.; Kamiyama, T.: Performance Study of Kotlin and Java Program Considering Bytecode Instructions and JVM JIT Compiler. 2021 Ninth International Symposium on Computing and Networking Workshops. IEEE Xplore. INSPEC: 21528076 https://doi.org/10.1109/CANDARW53999.2021.00028
[7] Suganuma, T. Olasawara, T.: Overview of the IBM Java just-in-time compiler, IBM Systems Journal 2000, vol. 39, no. 1, 175–193.
[8] Zhang, Y; Shao, S. and Liu H: Refactoring Java Programs for Customizable Locks Based on Bytecode Transformation. IEEE Access (Volume: 7), INSPEC: 18715192, ISSN: 2169-3536, https://doi.org/10.1109/ACCESS.2019.2919203