Efikasan obilazak grafova kontrole toka programa

Ivan Ristović
Oracle Labs, Beograd, Serbia
Matematički fakultet
Univerzitet u Beogradu
ivan.ristovic@oracle.com
0000-0002-1679-3848

Milan Čugurović
Matematički fakultet
Univerzitet u Beogradu
Oracle Labs, Beograd, Srbija
milan.cugurovic@matf.bg.ac.rs
0009-0003-4149-5820

Strahinja Stanojević
Matematički fakultet
Univerzitet u Beogradu
Oracle Labs, Beograd, Srbija
strahinja.stanojevic@matf.bg.ac.rs
0009-0007-6076-3586

Marko Spasić
Matematički fakultet
Univerzitet u Beogradu
Oracle Labs, Beograd, Srbija
marko.spasic@matf.bg.ac.rs
0009-0000-0392-0935

Vesna Marinković
Matematički fakultet
Univerzitet u Beogradu
vesna.marinkovic@matf.bg.ac.rs
0000-0003-0526-899X

Milena Vujošević Janičić
Matematički fakultet
Univerzitet u Beogradu
Oracle Labs, Beograd, Srbija
milena@matf.bg.ac.rs
0000-0001-5396-0644

***Apstrakt*—Prilikom prevođenja programa, kompilatori obilaze veliki broj grafova kontrole toka, stoga brzina individualnih obilazaka može u značajnoj meri uticati na ukupno vreme kompilacije. Iako standardni algoritmi obilaska grafa u širinu i u dubinu rade u linearnoj vremenskoj i prostornoj složenosti u funkciji broja čvorova i grana u grafu, njihovo vreme izvršavanja i upotreba memorijskog prostora variraju u zavisnosti od oblika grafa i strukture podataka koja se koristi u implementaciji algoritma.**

U ovom radu analiziramo vremensku i prostornu efikasnost izvršavanja obilaska grafova kontrole toka dobijenih kompiliranjem Java i Scala programa kompilatorom Graal. Rezultati analize pokazuju da je obilazak grafova kontrole toka u širinu i do 1.6 puta brži od obilaska u dubinu i da ima manji utrošak memorije na svim referentnim programima. Pokazujemo, takođe, da odabir strukture podataka koja se koristi u implementaciji algoritma obilaska ima uticaj na njegovu brzinu, pri čemu se dvostruko ulančana lista pokazala kao najefikasnija na svim referentnim programima.

Ključne reči — Obilazak u dubinu, obilazak u širinu, kompilatori, graf kontrole toka, Graal

I. Uvod

U modernim aplikacijama grafovi su sveprisutna struktura podataka koja osim samih podataka opisuje i odnose između njih. Mnogi realni problemi se mogu modelovati grafovski: grafovima se modeluju društvene mreže, biološki podaci, mape gradova i ulica i slično [1]. U praksi se često susrećemo sa veoma velikim grafovima. Na primer, analiza velikih podataka (eng. *big data analytics*) u kontekstu društvenih mreža može zahtevati obilaske grafova koji imaju milijarde čvorova [2, 3].

Kompajleri u internoj reprezentaciji (eng. *intermediate representation, skr. IR*) koriste grafove [4, 5] za predstavljanje kontrole toka delova koda. Ovakvih grafova kontrole toka ima veliki broj, te je stoga efikasnost algoritama obilaska grafa veoma važna. Efikasan obilazak grafova kontrole toka programa veoma je značajan za performanse aplikacija koje koriste modele mašinskog učenja. Takvi modeli vrše predviđanja na osnovu atributa izvučenih iz grafova kontrole toka programa. Na primer, modeli mašinskog učenja mogu iz grafa interne reprezentacije kompajlera ekstrahovati atribute kojima se opisuju delovi koda od značaja [6]. U ovom slučaju je efikasnost ekstrakcije atributa direktno korelisana sa efikasnošću obilaska grafa interne reprezentacije.

Implementacije obilaska grafova su mnogobrojne [7], ali se uglavnom svode na modifikacije dobro poznatih algoritama obilaska u dubinu (eng. *depth first search, DFS*) i u širinu (eng. *breadth first search, BFS*). Algoritam obilaska grafa mora posetiti sve čvorove grafa i razmotriti sve njegove grane, pa je njegova vremenska složenost linearna u funkciji broja čvorova i broja grana grafa. Vreme obilaska grafa i maksimalno zauzeće radne memorije računara zavise od svojstava grafa i strukture podataka koja se koristi u implementaciji algoritma obilaska.

U ovom radu su evaluirane performanse algoritama obilaska grafova i strukture podataka korišćene u njihovim implementacijama. Metrike na koje smo se fokusirali jesu vremenska i prostorna složenost ovih algoritma.

Kao skup podataka za evaluaciju koristili smo grafove kontrole toka programa koje kompilator Graal (eng. *Graal compiler*) [5] kreira u procesu kompilacije. Da bismo performanse evaluirali na grafovima modernog koda, kao programe za evaluaciju odabrali smo programe iz skupa *Renaissance* [8], kog čine raznovrsni programi pisani u jezicima Java i Scala. Merenje performansi izvršili smo unutar reda kompilacije kompilatora Graal na nivou pojedinačnih metoda. Rezultati evaluacije pokazuju da je, na osnovu odlika grafova kontrole toka programa, algoritam BFS u kombinaciji sa dvostruko ulančanom listom kao implementacijom reda obilaska brži i memorijski efikasniji na svim referentnim programima od drugih verzija istog algoritma i od algoritma obilaska u dubinu.

U sekciji II opisujemo kompilator Graal i njegovu grafovsku internu reprezentaciju. U sekciji III diskutujemo algoritme obilaska grafova, dok u sekciji IV opisujemo sprovedene eksperimente. U sekciji V predstavljamo rezultate evaluacije.

U sekciji VI definišemo zaključke i diskutujemo moguće pravce budućeg rada.

II. Kompilator Graal

Kompilatori za programski jezik Java dele se na JIT (eng. *Just-In-Time*) kompilatore i AOT (eng. *Ahead-Of-Time*) kompilatore. JIT kompilatori prevode Java bajtkod u mašinski kod tokom interpretacije programa. Najpoznatiji JIT kompilator je HotSpot [9]. AOT kompilatori prevode bajtkod u mašinski kod pre nego što se program pokrene. AOT kompilaciju odlikuje brže pokretanje aplikacija u odnosu na JIT kompilaciju (eng. *fast startup*), jer se kompilacija dešava pre pokretanja programa, dok JIT kompilatori prevode program prilikom njegovog izvršavanja.

Kompilator Graal [5] je moderni kompilator koji pruža značajna poboljšanja performansi u odnosu na tradicionalne kompilatore. Kompilator Graal koristi napredne optimizacije poput agresivnog umetanja (eng. *aggresive inlining*) [10], duplikacije koda zasnovanoj na simulaciji (eng. *simulation-based code duplication)* [11, 12], i analize delimičnog izlaska (eng. *partial escape analysis*) [13] da bi kreirao efikasne programe.

Kompilator Graal podržava AOT i JIT režime kompilacije. Komponenta kompilatora Graal koja omogućava AOT kompilaciju naziva se *Native Image*. *Native Image* koristi statičku analizu da prevede na mašinski jezik sve funkcije koje program koji se kompilira može pozvati. Na taj način se kompilirana aplikacija može izvršiti na računaru kao samostalna izvršiva datoteka (eng. *standalone executable*).

U procesu kompilacije komponenta *Native image* koristi internu grafovsku reprezentaciju Graal IR (eng. *Graal IR*) [14, 15] da predstavi naredbe Java bajtkoda, izvrši optimizacije i kreira mašinski kod. Graal IR je grafovska reprezentacija koja naredbe bajtkoda predstavlja kao čvorove u grafu dok njihove međusobne odnose (prethodnik i sledbenik) modeluje granama grafa. Graf programa sastoji se od grafa fiksiranih čvorova (eng. *fixed nodes*) kojim se opisuje kontrola toka u programu i grafa pokretnih čvorova (eng. *floating nodes*) kojima se opisuje tok podataka u programu (eng. *data flow*).

Na osnovu međureprezentacije koda koju koristi kompilator Graal kreira se graf kontrole toka programa koji se sastoji od blokova. Blokovi grafa kontrole toka sastoje se od čvorova Graal IR grafa, gde se naredbe grananja iz Graal IR grafa bijektivno preslikavaju u naredbe grananja grafa kontrole toka. Graal IR omogućava efikasnu analizu i izvođenje optimizacija na visokom nivou. Kao međureprezentacija koja se parsira iz Java bajtkoda, Graal IR takođe omogućava i vešejezičnu kompilaciju, sa mogućnostima predstavljanja svih jezika koji se kompiliraju na Java bajtkod, npr. Java, Scala, Kotlin, itd.

III. Optimizacije obilaska grafova

Dva osnovna algoritma obilaska grafa su obilazak grafa u širinu i dubinu. Obilazak grafa u širinu podrazumeva da se graf obilazi po nivoima pri čemu se vodi računa da svaki čvor poseti najviše jednom. Najpre se obilazi početni čvor, zatim svi njegovi susedi, nakon toga neposećeni susedi suseda itd. Ovaj algoritam u svojoj implementaciji koristi strukturu podataka red (eng. *queue*), iz koga u svakom koraku uzima element sa početka, a zatim sve njegove dotad neposećene susede označava kao posećene i dodaje ih na kraj reda. Za implementaciju reda je moguće koristiti različite strukture podataka kao što su lista ili niz. Algoritam pretrage grafa u dubinu pokušava da maksimalno ekspolatiše jednu putanju u grafu. Kada nije moguće nastaviti obilazak nekom putanjom (čvor nema suseda ili su svi susedi posećeni), algoritam se vraća unazad do prvog čvora od koga je moguće krenuti nekom novom do tada neposećenom putanjom, i njome ide u dubinu dokle god je to moguće. Algoritam je rekurzivan pa u svojoj implementaciji koristi strukutru podataka stek (eng. *stack*) koji može biti implementiran koristeći različite strukture podataka kao što su lista ili niz.

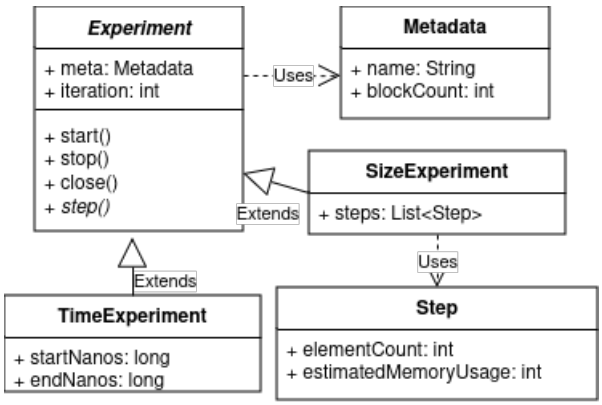
Tokom evaulacija performansi modfikacija algortama BFS za domenski-specifične grafove rađene su i analize efikasnosti obilaska ovih grafova. *Beamer i dr.* [16] posmatraju grafove malog dijametra i predlažu hibridni algoritam boljih performansi koji kombinuje tradicionalni BFS algoritam sa novim pristupima koji uključuju specifičan izbor čvorova od kojih se započinje pretraga. *Xia* i *Prasanna* [17] modeluju veze između topologije grafa i performansi algoritma BFS i menjaju algoritam BFS tokom obilaska u zavisnosti od topoloških svojstava grafa.

Izmene u strukturi podataka koja se koristi takođe mogu imati efekta u varijacijama algoritama obilaska. Na primer, *Leiserson* i *Schardl* [18] kombinuju paralelni BFS sa multiskupom kao strukturom podataka umesto tradicionalnog reda.

Današnji procesorski sistemi mogu da vrše paralelna izračunavanja i stoga se izučavaju i modifikacije algoritama obilaska grafa za višenitne, višeprocesorske i distribuirane sisteme [19]. Bader i dr. [20] prikazuju skaliranje performansi grafovskih algoritama na simetričnim multiprocesorskim klasterima i višenitnim arhitekturama. Modifikacije algoritma BFS za višeprocesorske sisteme [21, 22, 23, 24] dodaju paralelno procesiranje algoritmu radi unapređivanja performansi i smanjenja memorijskog zauzeća. Grafičke jedinice se takođe koriste za paralelna izračunavanja, i izučavane su modifikacije algoritama pretrage pogodne za izvršavanje na arhitekturi CUDA [25, 26, 27, 28].

Prilikom analize velikih podataka koriste se distribuirani sistemi za obradu grafovskih podataka, pri čemu distribuirano izvršavanje može otežati i usporiti izvršavanje nekih algoritama i stoga zahteva nove tehnike obilaska grafova [29, 30, 31]. U ekstremnim slučajevima, ukoliko se posmatraju veoma veliki grafovi, kao što je slučaj sa grafom Interneta, veličina grafa vremenom eksponencijalno raste [32], stoga merenja efikasnosti obilazaka grafova u ovakvim sistemima uključuju domenski-specifične metrike.

Tokom našeg istraživanja nismo pronašli radove koji analiziraju uticaj odabira algoritama obilaska na performanse u domenu kompilatora. Međutim, postoje istraživanja koja analiziraju odabir algoritma obilaska u drugim domenima [33].



Slika 1: UML dijagram hijerarhije klase `Experiment`, koja predstavlja osnovu sistema za merenje utrošenog vremena i memorije algoritma obilaska grafova kontrole toka.

IV. Precizno merenje vremena i memorijskog zauzeća prilikom obilaska grafova

Sistem za merenje utrošenog vremena i memorije algoritma obilaska grafova kontrole toka implementiran je u okviru infrastrukture kompilatora Graal. Radi merodavne simulacije obilazaka grafova kontrole toka prilikom procesa prevođenja, merenje je implementirano kao faza u okviru reda kompilacije (eng. *compile queue*) metoda. Konfiguracija sistema za merenje uključuje: tip eksperimenta, broj iteracija algoritma nad istim grafom i kratak naziv strukture podataka koja će biti korišćena u implementaciji algoritma obilaska grafa.

Eksperimenti za merenje vremena i utrošene memorije tokom obilaska grafa su implementirani kao potklase bazne klase `Experiment` sa slike 1. Bazna klasa `Experiment` uključuje deljena polja i metode zajedničke za sve algoritme — metapodatke o metodi, postavke algoritma i merenja. Klasa `TimeExperiment` sadrži i polja u kojima se čuva vreme u nanosekundama početka i kraja obilaska grafa jedne metode, a klasa `SizeExperiment` veličinu strukture podataka u svakom koraku algoritma obilaska.

Prikupljena merenja se ispisuju u datoteke u struktuiranom obliku koristeći format *JSON*. Svaka datoteka sadrži prikupljene podatke tokom eksperimenta za svaki metod koji je kompiliran. Pošto se eksperimenti u kojima se meri vreme izvršavanja programa mogu vršiti u više iteracija radi pronalaska prosečne vrednosti i eliminisanja šuma tokom merenja, uz proseke su prisutne i sve zabeležene informacije svakog merenja. Te datoteke se zatim grupišu po tipu i konfiguraciji eksperimenta u zasebne direktorijume u formatu `naziv-referentnog-programa/tip-eksperimenta-/konfiguracija` čiji sadržaj analiziramo koristeći biblioteke *matplotlib*[34] i *pandas*[35].

V. Evaluacija

Evaluaciju vršimo na skupu relevantnih programa kako bi dobijeni rezultati bili relevantni za moderne Java i Scala aplikacije. Radi minimizacije uticaja hardverskog i sistemskog šuma na rezultate merenja obezbedili smo adekvatnu hardversku i sistemsku konfiguraciju.

A. Priprema okruženja za evaluaciju

Jedan od najvećih problema koji se javlja prilikom preciznih merenja je hardverski šum. Zbog toga je mašina na kojoj su sprovedeni eksperimenti imala sledeću konfiguraciju:

- Ugašen *Intel Turbo Boost* — frekvencija procesorskih jezgara se ne skalira dinamički prilikom izvršavanja složenih izračunavanja.
- Ugašen *Intel Hyper-Threading* — na svakom fizičkom jezgru se može izvršavati samo po jedna nit.
- Procesorsko C stanje (eng. *processor C-state*) postavljeno na 0 — rad procesora u potpuno operativnom modu (eng. *fully operational mode*).
- Korišćen *real-time kernel* — fiksirano maksimalno kašnjenje između prekida izvršavanja procesa i ponovnog početka izvršavanja.
- Fiksirana frekvencija procesora — u svakom trenutku procesor može da izvrši jednak broj instrukcija bez skaliranja frekvencije.
- Proces vezan za jedno procesorsko jezgro — proces u kom merimo performanse se izvršava uvek na istom procesorskom jezgru. Dodatno, to je jedini proces koji se izvršava na tom jezgru.
- Postavljen maksimalan prioritet procesu — procesu u kom vršimo merenja se dodeljuje maksimalan prioritet.

Mašina na kojoj su vršeni eksperimenti poseduje procesor Intel i7 frekvencije 2.9GHz sa 8 procesorkih jezgara, veličinu RAM memorije od 32GB i operativni sistem Ubuntu 20.04. Korišćena je GraalVM verzija 21 [36] i OpenJDK 21 [37].

B. Programi korišćeni u evaluaciji

Za evaluaciju algoritama obilaska grafova koristili smo programe iz skupa programa *Renaissance* [8]. Skup programa *Renaissance* sastoji se od modernih Java i Scala aplikacija i uključuje veliki broj programa pisanih različitim stilovima i u različitim programskim paradigmama. Programi skupa *Renaissance* uključuju konkurente aplikacije, aplikacije za rad sa bazama podataka, programe pisane u funkcionalnom stilu, veb aplikacije, *Apache Spark* aplikacije [38] i algoritme mašinskog učenja. Tabela I prikazuje programe korišćene u evaluaciji. Uz naziv svakog od progama dat je kratak opis.

C. Dobijeni rezultati

Na skupu referentnih programa P relativno vreme izvršavanja r_p algoritma BFS u odnosu na algoritam DFS izračunava se po formuli

$$r_p = \frac{1}{n_p} \sum_{i}^{n_p} \frac{T_{BFS}(m_i(p))}{T_{DFS}(m_i(p))}$$

gde je $p \in P$, n_p broj metoda u referentnom programu p, a $T_{DFS}(m_i(p))$ i $T_{BFS}(m_i(p))$ prosečna vremena izvršavanja algoritma DFS i BFS, redom, nad i-tom metodom m_i referentnog programa p.

Tabela I: Programi iz skupa programa *Renaissance* korišćeni za evaluaciju

Ime programa	Opis
scrabble	Rešava slagalicu *Scrabble* koristeći JDK Stream API.
scala-stm-bench7	Pokreće *stmbench7* test [39] koristeći radni okvir *ScalaSTM* [40].
scala-kmeans	Pokreće algoritam klasterovanje K-sredina (eng. *K-Means*) [41] koristeći Scala kolekcije.
scala-doku	Rešava sudoku zagonetke koristeći Scala kolekcije.
rx-scrabble	Rešava slagalicu *Scrabble* koristeći *Rx* tokove (eng. *Rx-Streams*) [42].
reactors	Pokreće testove inspirisane referentnim programima *Savina* korišćenjem radnog okvira *Reactors.IO*.
philosophers	Implementira algoritam filozofa koji večeraju koristeći radni okvir *ScalaSTM* .
par-mnemonics	Rešava problem memorisanja telefonskog imenika koristeći paralelne JDK strimove.
mnemonics	Rešava problem memorisanja telefonskog imenika koristeći JDK strimove.
fj-kmeans	Implementira algoritam klasterovanja K sredina koristeći *fork/join* pristup komponentnog programiranja.
db-shootout	Izvršava *shootout* test nad bazom podataka koristeći nekoliko *in-memory* baza podataka.
akka-uct	Pokreće i ocenjuje performanse opterećenja *Unbalanced Cobwebbed Tree* testa u radnom okviru *Akka* [43].

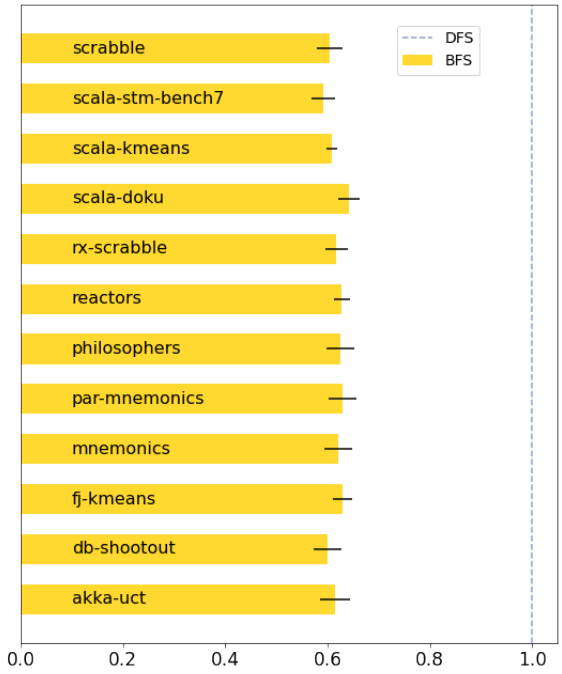
Za evaluaciju vremena obilaska grafova kontrole toka oba algoritma koristi se implementacija dvostruko ulančane liste standardne biblioteke programskog jezika Java [37] u kojoj se čuvaju tekući čvorovi grafa kontrole toka za obradu kako bi se smanjio uticaj strukture podataka na vreme izvršavanja. Evaluacija vremena obilaska grafova kontrole toka prikazana je na slici 2 i ukazuje da je algoritam BFS brži od algoritma DFS u proseku 1.6 puta na skupu programa korišćenim za merenje.

Dodatno je evaluiran uticaj izbora strukture podataka na vreme obilaska grafa kontrole toka i upoređene su strukture podataka:

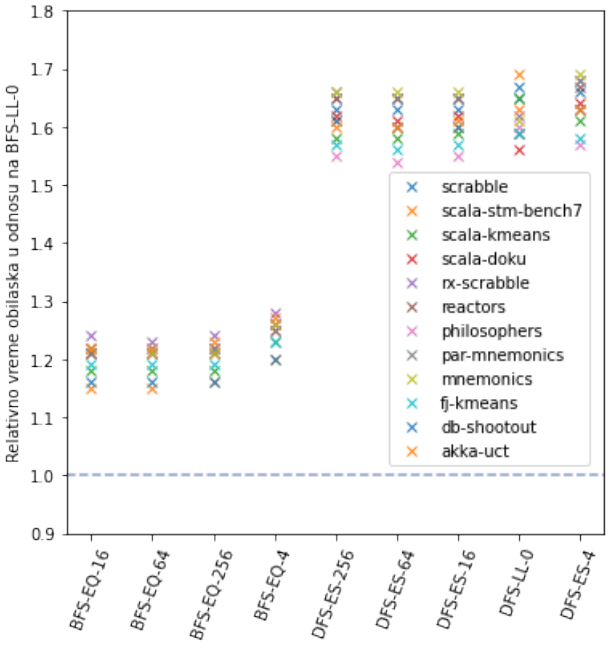
- `java.util.LinkedList (LL)` — Implementacija dvostruko ulančane liste standardne biblioteke programskog jezika Java.
- `EconomicQueue (EQ)` — Naša implementacija niza sa dinamički proširivim kapacitetom sa semantikom cirkularne `FIFO` strukture podataka.
- `EconomicStack (ES)` — Naša implementacija niza sa dinamički proširivim kapacitetom sa semantikom `LIFO` strukture podataka.

Za strukture podataka `ES` i `EQ` sprovedena su merenja sa rezervisanjem kapaciteta od $4, 16, 64$ i 256 elemenata radi minimizacije memorijskih alokacija potrebnih za proširivanje strukture podataka usled maksimalne popunjenosti. Rezultati evaluacije uticaja izbora strukture podataka na vreme izvršavanja obilaska grafa kontrole toka prikazani su na slici 3. Može se zaključiti da je BFS obilazak najbrži kada se koristi struktura podataka `LL`.

Prosečna utrošenost memorije algoritma BFS je 1.5 puta manja od algoritma DFS zbog činjenice da su grafovi kontrole toka duboki i da svaki čvor ima najčešće jedan ili dva potomka.



Slika 2: Relativno vreme izvršavanja algoritma BFS (žuta) u odnosu na algoritam DFS (plava) nad skupom programa *Renaissance* [8].



Slika 3: Relativno vreme obilaska grafova kontrole toka u širinu i dubinu, korišćenjem struktura podataka LL, EQ, ES u implementaciji, u odnosu na obiazak BFS-LL-0 (plava linija).

VI. Zaključak

U ovom radu je evaluiran efekat odabira algoritma obilaska grafova kontrole toka na vreme kompilacije programa. Ispitani su algoritmi obilaska grafa u dubinu i širinu sa različitim strukturama podataka u njihovoj implementaciji. Evaluacija je urađena na grafovima kontrole toka kompilatora Graal dobijenim kompilacijom skupa referentnih programa *Renaissance*. Dobijeni rezultati ukazuju da je algoritam BFS u kombinaciji sa dvostruko ulančanom listom za implementaciju reda obilaska i do 1.6 puta vremenski i do 1.5 puta memorijski efikasniji na svim referentnim programima.

Budući rad uključuje evaluaciju algoritama obilaska grafova kontrole toka na širem skupu referentnih programa i na osnovu toga optimizaciju ekstrakcije atributa modela mašinskog učenja koji predviđa profile u Graal kompajleru.

Zahvalnica

Ovaj rad je podržan od strane Ministarstva za nauku, tehnološki razvoj i inovacije Republike Srbije, ugovor 451-03-47/2023-01/200104, kao i putem istraživačkog projekta koji je obezbedio Oracle America, Inc.

Literatura

[1] O. Ore, R. Wilson, Graphs And Their Uses, 1990.

[2] M. Kim, J. Leskovec, Modeling social networks with node attributes using the multiplicative attribute graph model, arXiv preprint arXiv:1106.5053 (2011).

[3] M. Dürr, V. Protschky, C. Linnhoff-Popien, Modeling social network interaction graphs, in: 2012 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining, IEEE, 2012, pp. 660–667.

[4] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, M. F. O'Boyle, H. Leather, Programl: A graph-based program representation for data flow analysis and compiler optimizations, in: International Conference on Machine Learning, PMLR, 2021, pp. 2244–2253.

[5] C. Wimmer, C. Stancu, P. Hofer, V. Jovanovic, P. Wögerer, P. B. Kessler, O. Pliss, T. Würthinger, Initialize once, start fast: application initialization at build time, Proceedings of the ACM on Programming Languages 3 (OOPSLA) (2019) 1–29.

[6] M. Čugurović, M. Vujosević Janičić, V. Jovanović, T. Würthinger, Graalsp: Polyglot, efficient, and robust machine learning-based static profiler, Journal of Systems and Software (2024).
URL https://doi.org/10.1016/j.jss.2024.112058

[7] D. C. Kozen, D. C. Kozen, Depth-first and breadth-first search, The design and analysis of algorithms (1992) 19–24.

[8] A. Prokopec, A. Rosa, D. Leopoldseder, G. Duboscq, P. Tuuma, M. Studener, L. Bulej, Y. Zheng, A. Villazon, D. Simon, et al., Renaissance: Benchmarking suite for parallel applications on the jvm, in: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2019, pp. 31–47.

[9] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, D. Cox, Design of the java hotspot™ client compiler for java 6, ACM Transactions on Architecture and Code Optimization (TACO) 5 (1) (2008) 1–32.

[10] M. Vukasovic, A. Prokopec, Exploiting partially context-sensitive profiles to improve performance of hot code, ACM Transactions on Programming Languages and Systems 45 (4) (2023) 1–64.

[11] D. Leopoldseder, L. Stadler, T. Würthinger, J. Eisl, D. Simon, H. Mössenböck, Dominance-based duplication simulation (dbds): code duplication to enable compiler optimizations, in: Proceedings of the 2018 International Symposium on Code Generation and Optimization, 2018, pp. 126–137.

[12] D. Leopoldseder, Simulation-based code duplication for enhancing compiler optimizations, in: Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, 2017, pp. 10–12.

[13] L. Stadler, T. Würthinger, H. Mössenböck, Partial escape analysis and scalar replacement for java, in: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, 2014, pp. 165–174.

[14] G. Duboscq, L. Stadler, T. Würthinger, D. Simon, C. Wimmer, H. Mössenböck, Graal ir: An extensible declarative intermediate representation, in: Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop, 2013, pp. 1–9.

[15] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, H. Mössenböck, An intermediate representation for speculative optimizations in a dynamic compiler, in: Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages, 2013, pp. 1–10.

[16] S. Beamer, K. Asanovic, D. Patterson, Direction-optimizing breadth-first search, in: SC'12: Proceedings of the International Conference on High-Performance Computing, Networking, Storage and Analysis, IEEE, 2012, pp. 1–10.

[17] Y. Xia, V. K. Prasanna, Topologically adaptive parallel breadth-first search on multicore processors, in: Proc. 21st Int'l. Conf. on Parallel and Distributed Computing Systems (PDCS'09), Citeseer, 2009.

[18] C. E. Leiserson, T. B. Schardl, A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers), in: Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures, 2010, pp. 303–314.

[19] J. Chhugani, N. Satish, C. Kim, J. Sewall, P. Dubey, Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency, in: 2012 IEEE 26th International Parallel and Distributed Processing Symposium, 2012, pp. 378–389. doi:10.1109/IPDPS.2012.43.

[20] D. A. Bader, G. Cong, J. Feo, On the architectural requirements for efficient execution of graph algorithms, in: 2005 International Conference on Parallel Processing (ICPP'05), IEEE, 2005, pp. 547–556.

[21] R. E. Korf, P. Schultze, Large-scale parallel breadth-first search, in: AAAI, Vol. 5, 2005, pp. 1380–1385.

[22] V. Agarwal, F. Petrini, D. Pasetto, D. A. Bader, Scalable graph exploration on multicore processors, in: SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2010, pp. 1–11.

[23] D. A. Bader, K. Madduri, Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2, in: 2006 International Conference on Parallel Processing (ICPP'06), IEEE, 2006, pp. 523–530.

[24] D. P. Scarpazza, O. Villa, F. Petrini, Efficient breadth-first search on the cell/be processor, IEEE Transactions on Parallel and Distributed Systems 19 (10) (2008) 1381–1395.

[25] S. Hong, T. Oguntebi, K. Olukotun, Efficient parallel graph exploration on multi-core cpu and gpu, in: 2011 International Conference on Parallel Architectures and Compilation Techniques, IEEE, 2011, pp. 78–88.

[26] D. Merrill, M. Garland, A. Grimshaw, High-performance and scalable gpu graph traversal, ACM Transactions on Parallel Computing (TOPC) 1 (2) (2015) 1–30.

[27] P. Harish, P. J. Narayanan, Accelerating large graph algorithms on the gpu using cuda, in: International conference on high-performance computing, Springer, 2007, pp. 197–208.

[28] S. Hong, S. K. Kim, T. Oguntebi, K. Olukotun, Accelerating cuda graph algorithms at maximum warp, Acm Sigplan Notices 46 (8) (2011) 267–276.

[29] T.-Y. Cheung, Graph traversal techniques and the maximum flow problem in distributed computation, IEEE Transactions on Software Engineering (4) (1983) 504–512.

[30] M. Then, M. Kaufmann, F. Chirigati, T.-A. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, H. T. Vo, The more the merrier: Efficient multi-source graph traversal, Proceedings of the VLDB Endowment 8 (4) (2014) 449–460.

[31] C. Zhang, A new perspective of graph data and a generic and efficient method for large scale graph data traversal, arXiv preprint arXiv:2009.07463 (2020).

[32] J. M. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, A. S. Tomkins, The web as a graph: Measurements, models, and methods, in: Computing and Combinatorics: 5th Annual International Conference, COCOON'99 Tokyo, Japan, July 26–28, 1999 Proceedings 5, Springer, 1999, pp. 1–17.

[33] R. Mercado, E. J. Bjerrum, O. Engkvist, Exploring graph traversal algorithms in graph-based molecular generation, Journal of Chemical Information and Modeling 62 (9) (2021) 2093–2100.

[34] J. D. Hunter, Matplotlib: A 2d graphics environment, Computing in Science & Engineering 9 (3) (2007) 90–95. doi:10.1109/MCSE.2007.55.
[35] T. pandas development team, pandas-dev/pandas: Pandas (Feb. 2020). doi:10.5281/zenodo.3509134.
URL https://doi.org/10.5281/zenodo.3509134
[36] Oracle, GraalVM 21 (2023).
URL https://www.graalvm.org/release-notes/JDK_21/
[37] OpenJDK, OpenJDK 21 (2023).
URL https://github.com/openjdk/jdk21
[38] A. Spark, Apache spark, Retrieved January 17 (1) (2018) 2018.
[39] R. Guerraoui, M. Kapalka, J. Vitek, Stmbench7: a benchmark for software transactional memory (2006).
[40] M. Odersky, L. Spoon, B. Venners, Programming in scala, Artima Inc, 2008.
[41] J. Burkardt, K-means clustering, Virginia Tech, Advanced Research Computing, Interdisciplinary Center for Applied Mathematics (2009).
[42] S. Khare, S. Tambe, K. An, A. Gokhale, P. Pazandak, Scalable reactive stream processing using dds and rx, ISIS 14 (2014) 103.
[43] S. N. Srirama, F. M. S. Dick, M. Adhikari, Akka framework based on the actor model for executing distributed fog computing applications, Future Generation Computer Systems 117 (2021) 439–452.

Efficient control-flow graph traversal

Milan Čugurović, Ivan Ristović, Strahinja Stanojević, Marko Spasić, Vesna Marinković, Milena Vujošević Janičić

ABSTRACT

In the process of program translation, compilers traverse a large number of control flow graphs, and therefore the speed of individual traversals can significantly impact overall compilation time. While standard algorithms for graph traversal such as Breadth-First Search (BFS) and Depth-First Search (DFS) operate in linear time and space complexity concerning the number of nodes and edges in the graph, their execution time and memory usage vary depending on the shape of the graph and the data structure used in the algorithm implementation.

We analyze the time and space efficiency of executing control flow graph traversals obtained by compiling Java and Scala programs using the Graal compiler. The results analysis shows that breadth-first traversal of control flow graphs is up to 1.6 times faster than depth-first traversal and incurs lower memory overhead across all benchmark programs. We also demonstrate that the choice of data structure used in the algorithm implementation affects its speed, with a doubly linked list proving to be the most efficient across all benchmark programs.

Key words — Depth-first search, Breadth-first search, Compilers, Control flow graph, Graal