MINISTERUL EDUCAŢIEI ŞI CERCETĂRII AL REPUBLICII MOLDOVA
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică şi Microelectronică
Departamentul Ingineria Software și Automatică

Admis la susţinere
Şef departament:
FIODOROV Ion dr., conf.univ.

„___" _____ 2025

ANALIZA GESTIONĂRII MEMORIEI ÎN LIMBAJE DE PROGRAMARE MODERNE ȘI TRADIȚIONALE

Teza de master

Student: _____ **Trifan Denis, TI-231M**

Coordonator: _____ **Bîtca Ernest, asist. univ.**

Consultant: _____ **Cojocaru Svetlana, asist.univ.**

Chişinău, 2025

REZUMAT

În capitolul ”ANALIZA DOMENIULUI DE STUDIU” se efectuează un studiu asupra organizării memoriei în limbaje de programare tradiționale și moderne. În cadrul cercetării au fost selectate limbajele C++ și Java. Se definesc mecanismele de gestionare a memoriei în cadrul fiecărui limbaj. Importanța temei include necesitatea utilizării unui mecanism automatizat față de unul manual. Analizând cercetările similare existente, a fost realizată o comparare a acestora cu scopul de identificare a metodologiilor de realizare experimentată, metricilor necesare de comparare a performanței și alegerea testelor ce vor evidenția avantajele utilizării unui mecanism în favoarea altuia.

Capitolul ”METODOLOGIA EXPERIMENTALĂ” detaliază metodele de testare, realizarea testelor de performanță, explicarea funcționalității testelor și extragerea datelor din fișierele de tip log. În cazul limbajului C++, sunt selectate șapte metode de creare a obiectelor, folosind pointeri bruți, pointeri inteligenți unici, pointeri inteligenți partajați și algoritmul Boehm GC. Se stabilește metodologia de execuție și realizare a testelor, precum și modul de salvare a rezultatelor obținute. De asemenea, sunt definite compilatoarele utilizate, nivelurile de optimizare studiate, funcțiile de creare a obiectelor și procesul de instalare treptată a instrumentelor necesare. Pentru limbajul Java, sunt selectate trei suite de teste de performanță: SPECjvm2008, DaCapo și Renaissance. Fiecare test din aceste suite este descris în detaliu, evidențiindu-se funcționalitatea specifică. Procesul de selecție și filtrare a testelor este analizat, iar metoda de execuție a testelor prin terminal pentru fiecare suită este explicată. Sunt definiți parametrii de intrare și dimensiunile spațiului heap pentru fiecare test. Execuția aplicațiilor generează fișiere de tip log, care necesită prelucrare prin sisteme specializate în extragerea datelor. În acest sens, se realizează un studiu asupra soluțiilor existente pe piață și se selectează aplicația de tip desktop GCviewer pentru prelucrarea și analiza datelor din fișierele de log.

Capitolul ”ANALIZA REZULTATELOR” prezintă calculul statisticilor bazate pe rezultatele testelor de performanță pentru limbajele C++ și Java. Pentru limbajul C++, analiza rezultatelor se concentrează pe metrica timpului de execuție, comparând soluțiile oferite de biblioteca standard cu cele din biblioteca externă Boehm GC. Pentru limbajul Java, rezultatele testelor de GC sunt analizate pentru patru versiuni de JDK – 8, 11, 17 și 21. Performanța fiecărui algoritm de GC este evaluată și comparată pe baza a trei metrici esențiale: debitul, pauza maximă și timpul total de execuție. În cadrul suitelor de teste SPECjvm2008, DaCapo și Renaissance, sunt identificate și analizate separat acele teste care prezintă un debit scăzut, oferind o perspectivă detaliată asupra cazurilor care ar putea afecta performanța. Pe baza rezultatelor structurate, se trag concluzii privind eficiența fiecărui algoritm de GC pentru diferite versiuni de JDK, evidențiindu-se performanța fiecărui algoritm în funcție de versiunea platformei și metricele studiate.

ABSTRACT

In the chapter ”ANALYSIS OF THE STUDY DOMAIN”, a study is conducted on memory organization in traditional and modern programming languages. The research focuses on the languages C++ and Java. The mechanisms of memory management for each language are defined. The importance of the topic includes the necessity of using an automatic mechanism instead of a manual one. By analyzing existing similar research, a comparison was made to identify the methodologies for experimental implementation, the metrics necessary for performance comparison, and the selection of tests that will highlight the advantages of using one mechanism over another.

The chapter ”EXPERIMENTAL METHODOLOGY” details the testing methods, performance testing procedures, functionality of the tests, and data extraction from log files. For the C++ language, seven object creation methods are selected, using raw pointers, unique smart pointers, shared smart pointers, and the Boehm GC algorithm. The methodology for test execution and result recording is established, as well as the method of saving the obtained results. Additionally, the compilers used, optimization levels studied, object creation functions, and the step-by-step installation process for the required tools are defined. For the Java language, three performance testing suites are selected: SPECjvm2008, DaCapo, and Renaissance. Each test within these suites is described in detail, highlighting its specific functionality. The process of test selection and filtering is analyzed, and the method for executing tests via the terminal for each suite is explained. Input parameters and heap space sizes for each test are also defined. Application execution generates log files that require processing by specialized data extraction systems. In this regard, a study of the existing market solutions is conducted, and the GCviewer desktop application is selected for processing and analyzing the data from log files.

The chapter ”RESULT ANALYSIS” presents the calculation of statistics based on the performance test results for C++ and Java. For C++, the result analysis focuses on execution time metrics, comparing solutions provided by the standard library with those from the external Boehm GC library. For Java, GC test results are analyzed for four JDK versions – 8, 11, 17, and 21. The performance of each GC algorithm is evaluated and compared based on three essential metrics: throughput, maximum pause time, and total execution time. Within the SPECjvm2008, DaCapo, and Renaissance test suites, tests with low throughput are identified and analyzed separately, providing a detailed perspective on cases that may impact performance. Based on the structured results, conclusions are drawn regarding the efficiency of each GC algorithm for different JDK versions, highlighting the performance of each algorithm according to the platform version and the studied metrics.

CUPRINS

ABREVIERI, ACRONIME, DEFINIȚII

RAM – eng. Random Access Memory - memorie volatilă care stochează temporar datele și instrucțiunile utilizate de procesor pentru a accelera performanța sistemului în timpul funcționării;

Boilerplate - cod standardizat și repetitiv necesar pentru funcționalitatea de bază a unei aplicații;

Thread-safe - capacitatea unui cod, funcție sau obiect de a fi accesat simultan de mai multe fire de execuție fără a produce condiții de cursă sau comportamente neașteptate;

Memory management - procesul de alocare și eliberare a memoriei pentru aplicații, asigurându-se utilizarea eficientă a resurselor și prevenind scurgerile de memorie;

Memory leak - situație în care un program alocă memorie dar nu o eliberează corespunzător, ducând la pierderea treptată a memoriei disponibile și la potențiale probleme de performanță;

OutOfMemoryError - apare atunci când un program sau sistem de operare epuizează resursele de memorie disponibile, împiedicând alocarea suplimentară de memorie necesară pentru continuarea execuției;

Stack overflow - apare atunci când stiva de execuție a unui program depășește limita de memorie alocată, de obicei din cauza recursiei excesive sau a alocărilor mari de memorie pe stivă;

Dangling pointers - pointeri care continuă să facă referire la o locație de memorie care a fost eliberată provocând erori de acces la memorie sau comportamente neprevăzute ale programului;

Autorelease pool - mecanism de gestionare a memoriei în Objective-C care păstrează obiectele marcate pentru eliberare automată;

MRC – eng. Manual Reference Counting - metodă de gestionare a memoriei în Objective-C în care programatorul este responsabil pentru alocarea și eliberarea manuală a memoriei;

MRR – eng. Manual Retain Release - metoda de gestionare a memoriei în care programatorul este responsabil pentru gestionarea manuală a ciclului de viață al obiectelor;

ARC – eng. Automatic Reference Counting - mecanism de gestionare a memoriei în Objective-C care automatizează alocarea și eliberarea memoriei pentru obiecte;

JVM – eng. Java Virtual Machine - mediu de execuție care interpretează și rulează codul bytecode Java, oferind un nivel de abstractizare între aplicațiile Java și hardware-ul pe care rulează;

GC – eng. Garbage Collector - mecanism de gestionare a memoriei care automatizează procesul de eliberare a memoriei ocupate de obiecte care nu mai sunt utilizate;

STW – eng Stop The World - pauză în timpul căreia un GC oprește complet execuția aplicației pentru a colecta și a elibera memoria neutilizată;

Marking - etapă într-un algoritm GC în care sistemul identifică obiectele care sunt încă accesibile marcându-le pentru a le păstra, în timp ce restul obiectelor sunt considerate pentru deallocare;

Sweeping - etapă într-un algoritm GC în care sistemul eliberează memoria ocupată de obiectele care nu au fost marcate;

INTRODUCERE

Memoria este o resursă finită și esențială în cadrul oricărui sistem de calcul, fie calculator personal, un server, sau un dispozitiv mobil. Memoria unui sistem este împărțită în mai multe categorii, cum ar fi: RAM, cache, virtuală și secundară (persistența datelor fizice). *Memoria principală* (numită RAM) este cea mai accesibilă pentru procesor și reprezintă locul unde sunt stocate temporar datele și instrucțiunile în timpul execuției unui program. În cazul unor sisteme de calcul a căror resurse ale calculatorului pot fi limitate, sau se dorește de a implementa sisteme cu un timp de răspuns relativ mic sau care prelucrează volume mari de date, implică necesitatea de a controla procesele de alocare și dealocare a memoriei.

Gestionarea memoriei este un pilon central în funcționarea oricărui sistem informatic, având un impact semnificativ asupra eficienței și stabilității aplicațiilor. Într-un context tehnologic din ce în ce mai complex, unde resursele sunt limitate și trebuie gestionate cu precizie, mecanismele de alocare și eliberare a memoriei joacă un rol critic în funcționarea optimă a aplicațiilor software și a sistemelor de operare. În mod specific, acestea garantează că fiecare proces primește cantitatea de memorie necesară și că memoria este reutilizată eficient odată ce nu mai este necesară, prevenind astfel risipa de resurse și degradarea performanței sistemului.

Pe lângă înțelegerea acestor tipuri de memorie, este esențial de a cunoaște și mecanismele de gestionare care sunt folosite pentru a coordona accesul la aceste resurse. Una dintre cele mai comune tehnici este alocarea statică și dinamică a memoriei. Alocarea statică rezervă memorie pentru variabile și structuri de date în momentul compilării unui program, în timp ce alocarea dinamică permite gestionarea memoriei în timpul execuției, adaptându-se la nevoile fluctuante ale unui program. Acest mecanism dinamic este esențial pentru programele care au cerințe variabile și imprevizibile de memorie.

În limbajele de programare tradiționale, cum ar fi de exemplu C/C++, managementul memoriei era una din responsabilitățile de bază a dezvoltatorului de sisteme de calcul. Aceasta înseamnă că alocarea și dealocarea de memorie trebuia să fie executată manual și utilizând comenzi. Deși managementul static a memoriei prezintă o modalitate efectivă de control, aceasta implică adăugarea unui volum inpunător de logică business în cod, ce în cele din urmă încetinează procesul de elaborare a unui sistem informativ.

În cazul limbajelor de programare moderne, cum ar fi Java, gestionarea memoriei este realizată automat prin *garbage collection*. Acesta este un proces automatizat care monitorizează utilizarea memoriei și eliberează blocurile care nu mai sunt accesibile, prevenind astfel scurgerile de memorie. Deși această tehnică reduce responsabilitatea programatorului și transmite controlul spre o altă entitate/mecanism de management în ceea ce privește gestionarea memoriei, însă introduce o serie de provocări legate de performanță, deoarece procesul de garbage collection poate consuma resurse și poate afecta temporar viteza de execuție a programului. Compromisul acestui mecanism poate fi considerat justificat, deoarece, deși poate duce la o pierdere de performanță, contribuie semnificativ la reducerea potențialelor erori și la creșterea eficienței în dezvoltarea aplicațiilor.

BIBLIOGRAFIE

[1] M. van Putten and S. Kennedy, *Java Memory Management A Comprehensive Guide to Garbage Collection and JVM Tuning*, 1st edition. Birmingham: Packt Publishing, Limited, 2022.

[2] K. Bazoukis, "Garbage Collection vs. Manual Memory Management: Performance and Memory Consumption Issues," 2018, doi: 10.13140/RG.2.2.22961.28006.

[3] B. Andrist and V. Sehr, *C++ high performance: master the art of optimizing the functioning of your C++ code*, Second edition. in Expert insight. Birmingham Mumbai: Packt, 2020.

[4] F. Frantisek, *Memory as a Programming Concept in C and C++*, First edition. Cambridge University Press, 2004.

[5] B. Blunden, *Memory Management Algorithms And Implementation In C/C++*, First edition. Jones & Bartlett Learning, 2002.

[6] S. Meyers, *Effective Modern C++ : 42 Specific Ways to Improve Your Use of C++11 and C++14*, First edition. Sebastopol: O'Reilly Media, Inc., 2014.

[7] "A garbage collector for C and C++." Accessed: Sep. 13, 2024. [Online]. Available: https://hboehm.info/gc/

[8] H. Grgic, B. Mihaljevic, and A. Radovan, "Comparison of garbage collectors in Java programming language," in *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, Opatija: IEEE, May 2018, pp. 1539–1544. doi: 10.23919/MIPRO.2018.8400277.

[9] S. Oaks, *Java Performance : In-Depth Advice for Tuning and Programming Java 8, 11, and Beyond*, Second Edition. Sebastopol: O'Reilly Media, Inc., 2020.

[10] C. Hunt, M. Beckwith, P. Parhar, and B. Rutisson, *Java Performance Companion*. in Always learning. Boston Columbus New York Munich: Addison-Wesley, 2016.

[11] B. J. Evans, J. Gough, and C. Newland, *Optimizing Java : Practical techniques for imporving JVM application performance*, First Edition. Sebastopol: O'Reilly Media, Inc., 2018.

[12] M. Gupta, *Java 11 and 12 - New Features*. Birmingham, UK: Packt Publishing, 2019.

[13] M. Beckwith, *Jvm performance engineering: inside openjdk and the hotspot java virtual machine*, First edition. in Oracle press for java series. Hoboken: Addison-Wesley, 2024.

[14] R. Jones, A. Hosking, and E. Moss, *The Garbage Collection Handbook: The Art of Automatic Memory Management*, 2nd ed. New York: Chapman and Hall/CRC, 2023. Accessed: Sep. 12, 2024. [Online]. Available: https://www.taylorfrancis.com/books/9781003276142

[15] C. H. Flood, R. Kennke, A. Dinn, A. Haley, and R. Westrelin, "Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK," in *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, Lugano Switzerland: ACM, Aug. 2016, pp. 1–9. doi: 10.1145/2972206.2972210.

[16] S. Romanazzi, "From Manual Memory Management to Garbage Collection," 2018, doi: 10.13140/RG.2.2.22961.28006.

[17] X. Ma, J. Yan, Y. Li, J. Yan, and J. Zhang, "SPrinter: A Static Checker for Finding Smart Pointer Errors in C++ Programs," Nov. 2019, p. 1125. doi: 10.1109/ASE.2019.00117.

[18] I. Kopeć and J. Smołka, "A performance comparison of garbage collector algorithms in Java Virtual Machine," *J. Comput. Sci. Inst.*, vol. 13, pp. 359–365, Dec. 2019, doi: 10.35784/jcsi.1333.

[19] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro, "A study of the scalability of stop-the-world garbage collectors on multicores," in *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, Houston Texas USA: ACM, Mar. 2013, pp. 229–240. doi: 10.1145/2451116.2451142.

[20] R. Bruno and P. Ferreira, "A Study on Garbage Collection Algorithms for Big Data Environments," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 1–35, Jan. 2019, doi: 10.1145/3156818.

[21] Md. J. Islam, A. T. M. Rahman, and S. Rana, "Performance Analysis of Modern Garbage Collectors using Big Data Benchmarks in the JDK 20 Environment," Sep. 2023. doi: 10.1109/STI59863.2023.10464900.

[22] F. Mao, E. Zhang, and X. Shen, “Influence of program inputs on the selection of garbage collectors,” in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE’09*, Mar. 2009, p. 100. doi: 10.1145/1508293.1508307.
[23] P. Lengauer, V. Bitto, H. Mössenböck, and M. Weninger, “A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008,” in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, L’Aquila Italy: ACM, Apr. 2017, pp. 3–14. doi: 10.1145/3030207.3030211.
[24] M. Carpen-Amarie, P. Marlier, P. Felber, and G. Thomas, “A Performance Study of Java Garbage Collectors on Multicore Architectures,” Feb. 2015. doi: 10.1145/2712386.2712404.
[25] H. Li, M. Wu, and H. Chen, “Analysis and Optimizations of Java Full Garbage Collection,” in *Proceedings of the 9th Asia-Pacific Workshop on Systems*, Jeju Island Republic of Korea: ACM, Aug. 2018, pp. 1–7. doi: 10.1145/3265723.3265735.
[26] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro, “Assessing the scalability of garbage collectors on many cores,” *ACM SIGOPS Oper. Syst. Rev.*, vol. 45, no. 3, pp. 15–19, Jan. 2012, doi: 10.1145/2094091.2094096.
[27] K. Suo, J. Rao, H. Jiang, and W. Srisa-an, “Characterizing and optimizing hotspot parallel garbage collection on multicore systems,” in *Proceedings of the Thirteenth EuroSys Conference*, Porto Portugal: ACM, Apr. 2018, pp. 1–15. doi: 10.1145/3190508.3190512.
[28] R. Larsson and S. Pllana, “Evaluation of GraalVM Performance for Java Programs”.
[29] S. M. Blackburn and K. S. McKinley, “Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance”.
[30] P. J. P. Ferreira, “Information Systems and Computer Engineering”.
[31] Y. Yu, T. Lei, W. Zhang, H. Chen, and B. Zang, “Performance Analysis and Optimization of Full Garbage Collection in Memory-hungry Environments,” in *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, Atlanta Georgia USA: ACM, Mar. 2016, pp. 123–130. doi: 10.1145/2892242.2892251.
[32] A. Georges, D. Buytaert, and L. Eeckhout, “Statistically Rigorous Java Performance Evaluation,” in *ACM SIGPLAN Notices*, Oct. 2007. doi: 10.1145/1297027.1297033.
[33] X. Shen, F. Mao, K. Tian, and E. Zhang, “The Study and Handling of Program Inputs in the Selection of Garbage Collectors,” *Oper. Syst. Rev.*, vol. 43, pp. 48–61, Jul. 2009, doi: 10.1145/1618525.1618531.
[34] “DaCapo Benchmarks.” Accessed: Sep. 26, 2024. [Online]. Available: https://www.dacapobench.org/
[35] “Renaissance Suite, a benchmark suite for the JVM.” Accessed: Sep. 26, 2024. [Online]. Available: https://renaissance.dev/
[36] “SPECjvm2008.” Accessed: Sep. 26, 2024. [Online]. Available: https://www.spec.org/jvm2008/
[37] “Using Clang on Windows for C++.” Accessed: Oct. 29, 2024. [Online]. Available: https://wetmelon.github.io/clang-on-windows.html
[38] “2024 State of the Java Ecosystem Report | New Relic.” Accessed: Nov. 06, 2024. [Online]. Available: https://newrelic.com/resources/report/2024-state-of-the-java-ecosystem
[39] H. Oi, “A Comparative Study of JVM Implementations with SPECjvm2008,” in *2010 Second International Conference on Computer Engineering and Applications*, Bali Island, Indonesia: IEEE, 2010, pp. 351–357. doi: 10.1109/ICCEA.2010.75.
[40] “chewiebug/GCViewer: Fork of tagtraum industries’ GCViewer. Tagtraum stopped development in 2008, I aim to improve support for Sun’s / Oracle’s java 1.6+ garbage collector logs (including G1 collector).” Accessed: Nov. 13, 2024. [Online]. Available: https://github.com/chewiebug/GCViewer
[41] D. Trifan, *espada1317/gc-log-stats*. (Nov. 22, 2024). HTML. Accessed: Nov. 25, 2024. [Online]. Available: https://github.com/espada1317/gc-log-stats
[42] L. Xu, T. Guo, W. Dou, W. Wang, and J. Wei, “An experimental evaluation of garbage collectors on big data applications,” *Proc. VLDB Endow.*, vol. 12, no. 5, pp. 570–583, Jan. 2019, doi: 10.14778/3303753.3303762.